UNIVERSITE BRETAGNE LOIRE MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITE DE RENNES 1
COMUE **U**NIVERSITE **B**RETAGNE **L**OIRE

Ecole Doctorale N° 601
Mathématique et Sciences et Technologies de l'Information et de la Communication
Spécialité : *(voir liste des spécialités)*

Par

Paul TEMPLE

Investigate the Matrix : Leveraging Variability to Specialize Software and Test Suites

Thèse présentée et soutenue à RENNES , le 7 décembre 2018
Unité de recherche : Equipe DiverSE, IRISA
Thèse N° :

Rapporteurs avant soutenance :

Philippe COLLET, Professeur, Université Nice Sophia Antipolis / Université Côtes d'Azur, Nice, FRANCE

Myra B. COHEN, Professeur, Iowa State University, Ames, USA

Composition du jury :

Attention, en cas d'absence d'un des membres du Jury le jour de la soutenance, la composition ne comprend que les membres présents

Président : Philippe COLLET, Professeur, Université Nice Sophia Antipolis / Université Côtes d'Azur, Nice, FRANCE

Examinateurs : Philippe COLLET, Professeur, Université Nice Sophia Antipolis / Université Côtes d'Azur, Nice, FRANCE

Myra B. COHEN, Professeur, Iowa State University, Ames, USA

Patrick PÉREZ, Directeur Scientifique, Valeo.ai, Paris, FRANCE

Yves LE TRAON, Professeur, Université de Luxembourg, Luxembourg, LUXEMBOURG

Mathieu ACHER, Maître de conférences, Université Rennes 1, Rennes, FRANCE

Dir. de thèse : Jean-Marc JÉZÉQUEL, Professeur, Université Rennes 1, Rennes, FRANCE



Résumé en français

Contexte

Aujourd'hui, les logiciels sont présents partout autour de nous. On les trouve dans nos téléphones, ordinateurs et toutes sortes d'objets high-tech ; plus récemment, ils se sont introduits directement dans les télévisions, les véhicules, *etc.* Dans le même temps, ils permettent également de traiter de nouvelles tâches. Leur omniprésence a pour conséquence une hausse des attentes des consommateurs en terme d'efficacité, de performances, *etc.* En marge de cela, puisque personne n'a les mêmes attentes ni les mêmes demandes, le besoin de personnaliser les logiciels s'est développé.

Malgré tout cela, de nouveaux logiciels continuent d'être développés chaque jour pour traiter des problèmes qui sont parfois similaires. Par exemple, suivre le ballon durant un match de football ou une voiture de sport lors d'une course ou bien même des personnes piétonnes dans la rue sont autant de tâches exprimées de manière différente. Mais, d'un certain point de vue, en utilisant les bonnes abstractions, nous pouvons voir toutes ces tâches sous le même œil : repérer et suivre des entités. Les tâches précédemment citées sont seulement des instances spécifiques de la plus abstraite. Cela veut dire que, d'une certaine manière, ces tâches, individuellement, ne prennent qu'une partie de l'ensemble des exigences possibles (par exemple, en ne considérant qu'un champ visuel fixe, avec une caméra qui ne peut pas bouger). Dans le cas plus général des logiciels, cela se traduit par une optimisation particulière : une consommation d'énergie, de mémoire ou de processeur plus basse, un temps d'exécution minimisé ou encore essayer de produire les résultats les plus précis possibles en sont quelques exemples.

L'exemple potentiellement le plus représentatif de logiciels personnalisables (ou configurables) est le noyau Linux qui compte à peu près 13000 options de configuration. Si l'on suppose que chacune de ces options ne peuvent être qu'activée ou désactivée, le nombre de combinaisons possibles est de 2^{13000} soit à approximativement 10^{3250} possibilités. Dû à ce nombre titanesque, essayer d'exécuter chacune de ces combinaisons dans le but d'évaluer laquelle est la plus appropriée à des besoins définis par l'utilisateur est impossible. Il est donc difficile pour un utilisateur de réussir à trouver au moins une configuration (*i.e.,* une combinaison d'options) qui satisfasse tous ses besoins. De plus, la relation existante entre les options de configuration et les besoins est souvent mal définie ou mal documentée ce qui ajoute de la difficulté à

cette phase de configuration pour l'utilisateur.

Même si l'on met ce problème de côté et que l'on suppose que toutes les configurations peuvent être générées, le temps et les ressources allouées à l'activité de test sont souvent limités, ne permettant d'en exécuter qu'un sous-ensemble et d'en analyser les résultats. Ces résultats sont souvent utilisés pour trouver des bugs dans les programmes mais les besoins exprimés par l'utilisateur peuvent aussi contenir des objectifs de performance (par exemple, le programme doit s'exécuter en un temps inférieur à un certain seuil, la consommation de mémoire ne doit pas excéder un certain seuil, *etc.*). Pour vérifier qu'un programme respecte les besoins de l'utilisateur (que ce soit en terme de performances ou non), plusieurs tests sont souvent nécessaires pour pouvoir observer le comportement du système dans différents contextes d'utilisation. En plus de cela, différents apsects peuvent venir influencer les performances d'un programme. L'utilisation d'oracles (définissant les résultats attendus d'une exécution) devient alors difficile rendant, de fait, l'évaluation de performances délicate. Prenons par exemple le cas d'un encodeur vidéo, ses perfromances (*e.g.,* le temps d'exécution ou la qualité de la vidéo en sortie) peuvent dépendre de la vidéo d'entrée elle-même : la vidéo peut être de mauvaise qualité avec du bruit, ce qui la rend difficile à encoder correctement (diminuant la qualité en sortie) ou bien un algorithme pour enlever le bruit peut être ajouté dans le processus de traitement mais cela augmentera le temps d'exécution comparé à d'autres exécutions qui ne nécessitent pas l'ajout de cet algorithme.

Au final, il faut prendre en compte deux dimensions distinctes : tout d'abord, la phase de configuration du système qui consiste à sélectionner des options et à leur donner des valeurs ; et également sélectionner des cas de tests, pertinents pour une tâche, qui permettent d'observer les comportements des programmes, générés par la dimension précédente, dans des conditions différentes rendant possible de définir si un système respectent les besoins utilisateurs.

Dans cette thèse, nous représentons ces deux dimensions comme une matrice. Une dimension représente les systèmes que l'on génère grâce à la phase de configuration tandis que l'autre dimension représente l'ensemble des cas de tests qui seront exécutés sur chacun des systèmes. De ce fait, chaque cellule représente l'exécution d'un programme particulier sur un cas de test donné et l'on y reporte les performances observées. Dans le cas général, plusieurs performances peuvent être observées lors d'une seule exécution afin de décider si le système respecte bien tous les besoins de l'utilisateur, la celllule peut alors être représentée sous la forme d'un vecteur.

	Programmes				
Cas de tests		Programme 1	Programme 2	...	Programme N
	Test 1	12	1	...	5
	Test 2	1	348	...	10
	...				
	Test M	50	101	...	260

FIGURE 1 – Un exemple de matrice de performance exploitée dans cette thèse. Chaque cellule est le résultat d'une exécution d'un programme (colonne) sur un cas de test (ligne). Dans cet exemple le temps d'exécution a été mesuré et reporté dans les cellules de la matrice exprimé en secondes.

La figure 1 donne un exemple de représentation de cette matrice. Dans cet exemple, les colonnes représentent différents programmes (réalisant tous la même tâche mais avec des paramètres différents) à comparer pour que l'utilisateur final puisse choisir celui qui lui convient le mieux alors que les lignes représentent les différents cas de test à exécuter qui sont représentatifs de l'environnement dans lequel le programme sera plongé pour réaliser sa tâche. Pour chaque exécution, le temps d'exécution est mesuré (en secondes) et reporté dans la cellule adéquate de la matrice.

Nous nous intéressons à cette matrice car une simple analyse des valeurs reportés dans les cellules montre déjà un certain intérêt :

— Le programme 1 a l'air plutôt stable sur l'ensemble des cas de tests car il produit des temps d'exécution qui ont l'air relativement bas (moins d'une minute) par rapport à d'autres exécution ;
— Le premier cas de test a l'air relativement simple à traiter puisque les valeurs rapportées sur la première ligne sont très basses (quelques secondes uniquement) ;
— Le second cas de test semble difficile pour le second programme la valeur correspondante étant élevée (plusieurs minutes).

On a donc une analyse qui prend en compte les deux dimensions (soit de manière indépendante soit les deux à la fois). Cette première analyse est donc intéressante que ce soit du point de vue des logiciels (les paramétrisations des logiciels peuvent être mises en relation avec les performances ce qui peut aider du point de vue de la sélection du programme adéquat) ou du testeur (puisque les variations de performance des différents programmes peuvent être observées pour un cas de test donné ce qui peut soulever par la suite des interrogations et des analyses plus poussées).

Contributions

Cette thèse prétend que cette matrice de performance est un concept fondamental qu'il faut exploiter car elle apporte des informations essentielles que ce soit du point de vue de la pertinence des cas de tests utilisés ou du point de vue de la performance des programmes à comparer. Dans les mains d'ingénieurs du logiciel, cette matrice devrait permettre de réaliser différentes tâches utiles (que ce soit pour améliorer des logiciels existants mais aussi dans les phases de test avant production *etc.*).

Dans cette thèse, deux contributions principales sont mises en avant.

Tout d'abord, le problème de configuration et de sélection d'une configuration est difficile dû au nombre gigantesque de possibilités que les logiciels modernes proposent grâce aux options de configuration. Malheureusement, pour l'utilisateur final, c'est souvent un sous-ensemble de cet immense espace de configuration qu'il est nécessaire d'analyser afin de trouver une paramétrisation suffisante pour réaliser une tâche spécifique sous certaines conditions. Une pratique commune est alors de tester une paramétrisation (aussi appeler configuration) d'un programme et de voir si elle convient ou pas, si ce n'est pas le cas, il faut changer la valeur des paramètres et recommencer jusqu'à réussir. L'utilisateur se retrouve donc un processus de sélection basé sur l'essai-erreur. Dans le meilleur des cas, l'utilisateur va trouver en quelques essais une configuration qui lui convient, dans le cas contraire, il faudra plus d'essais. Un autre aspect a prendre en compte est que la paramétrisation peut prendre un certain temps (par exemple, dans le cas où le système doit être recompilé), dans ce cas, même quelques essais peuvent être chronophage et énergivore ce qui n'est pas désirable. Notre but est donc de réussir à réduire cet espace de configuration en amont afin que l'utilisateur est un nombre moindre de programmes à prendre en compte. Nous proposons d'utiliser une technique d'apprentissage automatique afin de réaliser cette réduction. A partir d'un sous-ensemble de programmes, d'exécutions sur des cas de test ainsi qu'une fonction oracle qui réfère aux besoins de l'utilisateur. Le principe est alors d'utiliser l'oracle sur les résultats des exécutions afin d'apposer un label définissant si le programme (ou plus précisemment sa paramétrisation) doit être gardée dans l'ensemble des programmes qui peuvent être utilisés par l'utilisateur final ou non. Une fois toutes ces exécutions annotées, l'algorithme d'apprentissage automatique peut créer une fonction séparatrice entre les paramétrisations à garder et les autres afin de réduire automatiquement l'espace de configuration. Grâce à l'approche statistique et au pouvoir de généralisation de l'algorithme d'apprentissage automatique, cette approche

permet d'écarter un certain nombre de configurations tout en gardant les programmes avec un fort potentiel d'adéquation aux critères de l'utilisateur. Un problème subsistant est alors de ne pas faire trop d'erreurs de classification ce qui pourrait réduire les possibilités de configuration de l'utilisateur plus que nécessaire ou au contraire ne pas être capable d'écarter assez de configurations ce qui reviendrait au processus d'essais-erreurs initial. Cette première contribution vise donc à réduire la première dimension de notre matrice de performance. Nous validons cette approche sur différents systèmes, notamment un générateur de séquence vidéos.

La seconde contribution majeure de cette thèse est une nouvelle méthode qui vise à évaluer la capacité de suites de tests à révéler des différences significatives dans les performances des différents programmes qui réalisent une même tâche. En effet, le temps et les ressources alloués à l'activité de tests étant limités, il est nécessaire de réduire au maximum le nombre d'exécutions à réaliser et donc il faut pouvoir proposer un nombre de cas de tests suffisant mais minimal afin d'optimiser cette activité. Le problème est que le choix des cas de tests à utiliser reste un problème ardu. Par exemple, les banques de données d'images utilisés dans les compétitions d'algorithmes de reconnaissance d'objets sont de plus en plus grandes ce qui allongent le temps de calcul et qui désavantages les compétiteurs qui ne peuvent pas se permettre d'avoir de grosses puissances de calcul ; mais est-ce que toutes ces images sont vraiment nécessaires ? Ne peut-on pas réduire ces données de tests tout en conservant la capacité de l'ensemble de tests à discriminer les compétiteurs ? Dans ce contexte, une suite de tests nous paraît intéressante si elle est capable de donner une vue d'ensemble des performances des programmes que l'utilisateur pensent pouvoir utiliser. Ainsi, il est nécessaire de garder dans cette suite de tests des cas de tests qui sont capables de montrer que, par exemple, ils ont été traîtés de manière beaucoup plus longues que d'autres par certains programmes, ce qui peut être décisif dans le cas où le programme est plongé dans un environnement où il nécessite de répondre en temps réel. Dans cette contribution, nous introduisons la notion de "couverture de performance". Pour mesurer cette couverture, nous proposons d'utiliser le score de dispersion. Il se construit sur la base d'un histogramme qui va permettre de séparer le domaine de définition d'une performance donnée en plusieurs sous-domaines disjoints. Le but est qu'au moins une exécution permette de peupler chaque sous-domaine. Plus le nombre de sous-domaine représenté est élevé, plus la suite de tests est considérée comme étant intéressante et devrait être conservée puisqu'elle permet d'observer différents

comportements provenant de différents programmes. Cette approche a été évaluée sur différents domaines d'applications et nos résultats montrent d'une part l'efficacité du score de dispersion tel que nous l'avons défini pour conserver des suites de tests qui semblent plus intéressantes que d'autres et d'autre part la possibilité d'utiliser cette approche pour différentes tâches comme par exemple réduire un ensemble de suite de tests ou encore mettre en avant des comportements de programmes qui semblent "bizarres" (*i.e.,* déviants par rapport au reste de l'ensemble des programmes considérés) ce qui permet de pousser l'analyse plus loin pour peut-être découvrir des bugs dans le code. Cette deuxième contribution s'attaque donc tout d'abord la deuxième dimension de notre matrice de performances. En évaluant la qualité des suites de tests, il est possible par la suite d'optimiser la phase d'exécution en réduisant le nombre de tests à exécuter ou en donnant un ordre (par exemple, ceux qui ont un score de dispersion plus élevé d'abord).

Au final, nos contributions visent à réduire l'une ou l'autre des deux dimensions présentées par la matrice de performances permettant par la suite d'avoir un choix restreint mais toujours pertinent pour la suite du processus de génération de programmes ou de tests.

Abstract

Nowadays, software have to be efficient, fast to execute, *etc.* They can be configured in one way or another to adapt to specific needs. Each configuration leads to a different system and usually it is hard (if not impossible) to generate them all. Thus, the exhaustive evaluation of their performance is impossible. However, a single user has specific requirements and needs to find an appropriate configuration of a system. To ensure the adequacy between performances and requirements, several executions under different conditions are needed adding computation time to the daunting task of selecting a proper configuration.

Two dimensions emerge from this description of performance testing : the selection of relevant system configurations that influence the behavior of associated system and the selection of test cases allowing to observe performances of systems under different conditions.

We propose to represent those two dimensions as a (performance) matrix : one dimension represents selected systems for which performances can be observed while the other dimension represents the set of test cases that will be executed on each of these systems. Each cell is the execution of a program variant regarding a test.

The contributions of this thesis are as follows :

First, we leverage Machine Learning techniques in order to specialize a Software Product Line (in this case a video generator) helping the selection of a configuration that is likely to meet requirements. End users must be able to express their requirements such that it results in a binary decision problem (*i.e.,* configurations that are acceptable and those that are not). Machine Learning techniques are then used to retrieve partial configurations that specialize a Software Product Line to guide end users and reduce the configuration space. In the end, this work aims at diminishing the first dimension of the matrix that deals with systems and programs.

Second, we propose a new method assessing the ability of test suites to reveal significant performance differences of a set of configurations tackling the same task. This method can be used to assess whether a new test case is worth adding to a test suite or to select an optimal test set with respect to a property of interest. In the end, it may help structuring the execution of tests. For instance, it can create an order of execution resulting in using less test cases that are presented in the second dimension of the matrix. We evaluated our approach on several systems from different domains

such as OpenCV or Haxe.



ACKNOWLEDGEMENTS

First, I would like to thank very warmly the members of the jury for accepting to review the work I have conducted during the past three and a half years. This PhD was a big step in my life and having such encouraging comments and relevant questions coming from such a panel of researchers was way out of my consideration for a long time. I would like to thank Prof. Philippe Collet for having taken the chair of this jury. Having you, a mentor of one of my supervisor, in this jury was very challenging for me considering that I knew very little about configurable systems a few years ago. Thank you to Prof. Myra Cohen for accepting to listen to me despite it was very early over there. Your comments and enthusiasm were very heart-warming. Thank you Prof. Yves Le Traon for the every now and then talks we had during these three years and for showing your interest in my work. Thank you very much to Patrick Pérez for being the only representative (in this jury) of the image processing community. We have met several times during this PhD, you were always careful about what I was presenting and you were of great advice despite being so much busy. Furthermore, you were constantly reminding me how interesting the ideas we were talking about could benefit to industries giving me full of energy every time we met.

I would like to thank my former teachers who taught me everything I know about video and image processing but also machine learning, computer graphics and much more. From IUT to the ESIR, thank you all Pierre, Philippe, Adib, Sébastien, Kadi, Rémi, Fabrice and the others.

Ewa and Laurent you were excellent advisers during my internship at TEXMEX some years ago. You taught me so much during these few months about research in general but also the level of expectation that I needed to reach to be an excellent researcher. In addition, you introduced me to the security problems in Machine Learning and put me in touch with Battista. I am very grateful to both of you for all of that.

Guillaume, you told me about configurable systems more than 5 years ago now, almost immediately we talked about how similar this domain and research in machine learning can be. This talk has driven my research and you were also the one telling me about this PhD opening in DiverSE. I want to thank you very warmly for this.

While it was very hard for me to get anything from the first talks during breaks or meetings, the atmosphere inside DiverSE always remained excellent. I want to say thank you DiverSE (present and past members) for being so kind, curious (about my knowledge) and understanding about the fact that not everybody knows software engineering (even if you think that this is a big mistake) and to Olivier and Benoît for leading the team. José, you were the person that put me on track and I am very pleased I was able to work with you ; I must have been a pain in the butt during this first year when I hardly knew anything about the stack behind variability, configurations, *etc.* You have always been there to help me being very calm and willing to share your knowledge. Thank you Pierre, Kévin, Fabien, Marcellino, Alejandro and Oscar for the all the moment we shared (about work but also the others). Special thanks to Johann with whom I could always talk about tennis (even if I was disturbing you). Tifenn, we arrived at the same time in the team, we were two costarmoricans in this foreign country that is Rennes. You were also there to talk about anything but work, I really appreciated all these moments with you. Caroline, DiverSE is lucky to have you, you are kind of the "mom" of the team, thank you for everything.

Of course, Jean-Marc and Mathieu ! Thank you very much for giving me the opportunity to work with you. You were tremendous supervisors despite your very busy schedules. Of course, I learnt a lot from your scientific guidance. Brainstorming sessions were exhausting to me, yet, a lot of ideas, new directions long terms vision came out of these. I am glad to know that the work I have begun will continue in the team under your supervision and that somehow we started a new focus in the team. Apart from that, you were always there when I needed explanations, when I was feeling uncomfortable or when I did not know where to go ; you always took one hour or more to talk about all these problems and I cannot emphasize enough how much it was important to me.

I am very proud of what I did at work, but I could not have accomplished that without all the people I met outside the lab. The TCTF is the best tennis club I have ever seen. I met a lot of people willing to play tennis just as much as I do. Thank you to all the partners and teammates. Olivier B. you are the best trainer ever ! From the first hour of training (in groups) to the last (in individual), you made far better than I could have ever imagined. All those hours with you on the court improved my focus and my endurance which were beneficial in my work. Thank you very much for everything. I hope you will keep giving this good energy that you give to people and that you will be able to share

your passion with people for a long time.

M'man and Pierre... When I started my PhD, things were a bit complicated between us : "why do you work that much ? Why do you work that late ?" were common questions I heard every now and then. I think you know now why I did this. However, as the thesis kept going, you were there to cheer me up, to comfort me and support me constantly. This is also thank to both of you that we have a doctor in the family now.

Finally, to you, the person I met 8 years ago in Lannion. The three past years were not easy for us. Despite all of this, you were the person that kept me alive, pushing me to take vacations when I was exhausted and could not think straight a few weeks before deadlines. You were there no matter what, even when you were just as exhausted as I was. I think you lived this PhD like you were actually doing a PhD. I am sorry for the hard time I made you live (and also for the numerous upcoming ones as long as we stay together). I love you Marine.

REMERCIEMENTS

Tout d'abord, je voudrais remercier très chaleureusement les membres du jury qui ont accepté de rapporter et d'évaluer les travaux que j'ai mené durant ces trois dernières années et six mois. Cette thèse a été un moment très important dans ma vie et le fait d'avoir reçu ces commentaires encourageants et des questions aussi pertinentes provenant de ces personnes m'a toujours paru hors de portée. Je souhaiterai remercier Philippe Collet pour avoir accepter de présider ce jury. Vous avoir vous, un mentor d'un de mes propres encadrants, dans ce jury a été très particulier pour moi surtout en sachant que je ne connaissais pratiquement rien aux systèmes configurables il y a quelques années de cela. Merci à vous, Myra Cohen pour avoir accepter de m'écouter alors qu'il était très tôt aux Etats-Unis. Vos commentaires et votre enthousiasme m'ont été très réconfortant. Merci à Yves Le Traon pour avoir montré autant d'intérêt à mes travaux et pour les discussions que nous avons eu pendant ces trois ans. Patrick Pérez, merci beaucoup d'avoir été le seul membre de ce jury a être le représentant de la communauté du traitement de l'image. Nous nous sommes rencontrés régulièrement durant cette thèse, vous avez toujours su vous montrer d'une écoute particulièrement attentive, vous m'avez été de précieux conseils et tout ça même en étant extrêmement débordé. En plus de cela, vous m'avez toujours rappelé à quel point les idées que j'avais pouvais être intéressantes pour le monde de l'industrie, me redonnant à chaque fois de l'énergie pour continuer.

Je voudrais également remercier mes enseignants qui m'ont tout appris au sujet du traitement de l'image et de la vidéo mais également tout ce qui traite du machine learning, la synthèse d'images et plein d'autres choses. Depuis l'IUT jusqu'à l'ESIR, merci à Pierre, Philippe, Adib, Sébastien, Kadi, Rémi, Fabrice et tous les autres.

Ewa et Laurent, vous avez été d'excellents maitres de stage pendant mon séjour à TEXMEX, j'ai été extrêmement fier d'avoir pu travailler avec vous. Vous m'avez appris tellement, pendant ces quelques mois, sur la recherche en général mais également sur le niveau d'éxigence que je devais atteindre si je voulais être bon dans ce que je fais. En plus de ça, vous m'avez aussi fait connaitre les problèmes de sécurité liés au machine learning et j'ai pu, grâce à vous, être en contact avec Battista, je vous suis

très reconnaissant pour ça.

Guillaume, tu as commencé à me parler des systèmes configurables il y a 5 ans à peu près, très rapidement nous avons parlé des similitudes qu'il existait entre ce domaine et celui du machine learning. Cette discussion m'a guidé régulièrement pendant cette thèse et tu m'as également mis au courant que DiverSE recherchait un candidat pour une thèse au parfait moment. Je veux te remercier énormément pour ça.

Même si il a été très difficile pour moi de comprendre quoi que ce soit aux premières pauses et premiers meetings, l'atmosphère au sein de DiverSE a toujours été excellente. Je remercie beaucoup DiverSE (membres présents et passés) pour avoir été si gentil, curieux (sur mon parcours et ce que je pouvais apporter à l'équipe) et compréhensif sur le fait que tout le monde ne connaisse pas vraiment le génie logiciel (même si, à priori, c'est une énorme bêtise) et à Olivier et Benoît pour organiser cette équipe. José, tu m'as guidé au tout début et j'ai été très heureux d'avoir pu travailler avec toi ; j'ai pourtant dû être une vraie plaie pendant ma première année, quand je ne connaissais pratiquement rien à tout ce qui touchait à la variabilité, aux configurations et tout ça. Tu as toujours été présent pour m'aider et tu as su toujours partager tes connaissances. Merci à Pierre, Kévin, Fabien, Marcellino, Oscar et Alejandro pour tous les moments partagés (boulot et autres). JMerci également à Johann avec qui je pouvais toujours parler tennis (même si je sentais que je le dérangeais). Tifenn, on est arrivé au même moment dans l'équipe, nous étions les deux costarmoricains dans ce pays très lointain qui est Rennes. Tu étais toujours là pour parler de tout sauf du travail, ça m'a fait énormément de bien. Caroline, DiverSE peut être vraiment content de t'avoir, tu es un peu la "maman" de l'équipe, merci pour tout.

Bien sûr, il y a aussi Jean-Marc et Mathieu ! Merci énormément pour m'avoir permi de travailler avec vous. Vous avez été tout simplement exceptionnels en tant que directeur et encadrant bien que vous aviez un emploi du temps bien chargé à côté. J'ai appris tellement de choses grâce à vous et votre sens de la recherche. Les sessions de brainstorming ont été très éprouvantes pour moi au début, mais, il en ressortait toujours tout un tas d'idées, des nouvelles directions à explorer et des visions à long-terme. Je suis très fier de savoir que le travail que j'ai commencé va se poursuivre au sein de l'équipe à vos côtés, mais je suis également fier du fait que l'on est pu commencé, en quelque sorte, un nouvel axe de recherche. En plus de tout ça, vous avez toujours su être là quand j'avais besoin d'explications ou de conseils, quand je me posais tout un tas de questions ; vous avez toujours pris du temps pour parler de

tout ça et je n'ai pas les mots pour vous dire à quel point ça a été important pour moi.

Je suis très fier du travail que j'ai accompli, mais je n'aurais pas pu terminer tout ça sans les personnes qui étaient là en dehors du labo. Le TCTF est le meilleur club que j'ai connu. J'y ai rencontré des tas de personnes qui ont toujours au moins tout autant motivées que moi pour jouer. Merci à tous mes partenaires de jeu et aux co-équipiers. Olivier B., tu es au top ! Du premier entrainement (en groupe) jusqu'au dernier (en indiv), tu m'auras fait devenir bien meilleur que ce que je n'aurais pu imaginer. Toutes ces heures passées avec toi auront également amélioré ma concentration et mon endurance ce qui aura été très bénéfice pour le travail. Merci énormément ; j'espère que tu réussiras à garder l'énergie que tu transmets et que tu pourras continuer à partager ta passion pendant encore longtemps.

M'man et Pierre... Quand j'ai commencé ma thèse, ça n'a pas toujours été facile pour nous : "Pourquoi tu travailles autant ? Pourquoi tu restes aussi tard ?" ont été des questions récurrentes. Je pense que maintenant vous comprenez pourquoi. Au fur et à mesure que la thèse avançcait, vous avez toujours été là pour moi. C'est aussi grâce à vous 2 que nous avons un docteur dans la famille.

Pour terminer, à toi la personne que j'ai rencontré il y a 8 ans à Lannion. Les trois dernières années n'ont pas été faciles pour nous. Malgré tout cela, tu as été la personne qui m'a fait survivre, en me forçant à prendre des vacances quand j'étais totalement exténué et que je n'arrivais plus à réfléchir alors que des deadlines arrivaient. Tu étais là, tous les jours, même quand toi-même tu n'en pouvais plus. Au final, je pense que tu as vécu cette thèse tout autant que moi. Désolé pour tous les mauvais moments que je t'ai fait vivre (et aussi pour les prochains à venir). Je t'aime Marine.

TABLE OF CONTENTS

Résumé en français **3**

Abstract **9**

1 Introduction **20**

2 Background **25**

2.1 Software Product Lines 25

2.1.1 Perks of reuse 25

2.1.2 SPL development process 26

2.2 Feature Models 29

2.2.1 Fundamentals of feature models 30

2.3 Machine Learning 34

2.3.1 Stages to use a machine learning algorithm 36

2.3.2 The training phase 37

2.3.3 Evaluating prediction performances 38

2.3.4 Overfitting and underfitting 39

2.3.5 Hyperparameters and validation set 40

2.4 Summary 41

3 State of the Art **43**

3.1 Software product lines and Testing 44

3.1.1 Configuration sampling 44

3.1.2 Fault Detection in software product lines 46

3.1.3 Metamorphic Testing 48

3.2 Tests quality 49

3.2.1 Traditional metrics 49

3.2.2 Mutation Testing 50

3.2.3 Quality of performance tests 51

3.3 Machine learning and software product lines 51

3.3.1 Performance prediction 51
3.3.2 Testing machine learning techniques 53
3.4 Summary 54

4 Automatic Specialization of software product lines using Machine Learning 57
4.1 Introduction 57
4.2 Method 60
4.3 Case Study 64
4.3.1 Case and Problem 64
4.3.2 Solution for Inferring Constraints 65
4.3.3 Generating a training set out of the variability model 66
4.3.4 Oracle 67
4.3.5 Machine learning 67
4.3.6 Extracting constraints 68
4.4 Experiments 71
4.4.1 Experimental Setup 71
4.4.2 Results 71
4.4.3 Threats to validity 77
4.5 Discussions 79
4.6 Conclusion 81

5 Learning-based Performance Specialization of Configurable Systems 83
5.1 Introduction 83
5.2 Motivation and Problem Statement 85
5.2.1 Motivating scenario 85
5.2.2 Approach 87
5.2.3 Novel problems 88
5.3 Discussions 90
5.3.1 Impacts of performance objectives on the learning problem . . . 90
5.3.2 Measures to assess the prediction power of machine learning models 91
5.4 Experiments 94
5.4.1 Subject systems and configuration performances 94
5.4.2 Experimental setup 94

5.4.3 Presentation of results . . . 96
5.4.4 RQ1) Does our method allow to accurately classify configurations ? 96
5.4.5 RQ2) Does our method allow to maintain flexibility while being safe ? . . . 103
5.5 Conclusion . . . 110

6 Multimorphic Testing **113**
6.1 Introduction . . . 113
6.2 Multimorphic Testing . . . 116
6.2.1 Motivation . . . 116
6.2.2 The principle of Multimorphic Testing . . . 117
6.2.3 Properties of a measure . . . 118
6.2.4 Design of dispersion measures . . . 119
6.3 Empirical Evaluation . . . 123
6.3.1 Research questions . . . 123
6.3.2 Evaluation settings . . . 123
6.3.3 RQ1 : Is the dispersion measure right ? . . . 128
6.3.4 RQ2 : Is the dispersion score a right measure ? . . . 132
6.3.5 Concluding remarks over the method . . . 137
6.3.6 Reproducibility of experiments . . . 138
6.4 Discussions and Threats to Validity . . . 139
6.4.1 Internal threats . . . 139
6.4.2 External threats . . . 139
6.5 Conclusion . . . 142

7 Conclusion and Future Work **143**
7.1 Conclusion . . . 143
7.2 Perspectives . . . 144
7.2.1 Machine Learning, Variability and Software Product Lines . . . 144
7.2.2 Developing an appropriate sampling method . . . 146
7.2.3 Adversarial Machine Learning and Software Product Lines . . . 147
7.2.4 Taking into account the surrounding context . . . 148

Bibliography **158**

CHAPITRE 1

INTRODUCTION

People are nowadays more and more demanding regarding the characteristics of their software. They want software to be efficient, fast to execute and sometimes even able to optimize several different aspects at once. In the meantime, software are taking evermore importance in our daily activities : they are in our computers and smartphones ; for a few years now, they are helping us driving cars, *etc.* As the number of programs increases, they tackle new problems that are more and more complex. Yet, different software continue to be independently created to tackle similar tasks. For instance, tracking a ball in soccer games or cars in races or even tracking people in the street might all seem different (because it does not aim to track the same entity) however, at a certain level of abstraction, all of them *track entities*. Their differences come from the fact that they take into account only a subset of users' requirements but not all at once. Some are optimized to consume less memory, others are tuned to produce the most accurate results, *etc.*

One representative example of software trying to cope with different users' requirements is the Linux Kernel which contains about $13,000$ options. Assuming that all options can only be activated or deactivated, the number of combinations of options is up to $2^{13,000}$ or about $10^{3,250}$ possibilities. This number is so big that it is impossible to review them all exhaustively in order to find which ones suit pre-defined requirements. Thus, it is hard for users to choose and find a proper configuration (*i.e.,* combination of options) that complies with their requirements. Finding such a configuration is usually difficult as the mapping between options and requirements is not straightforward and the documentation might not reflect properly on how an option affects the behavior of the program.

Even if all configurations of a system can be generated, time and resource budgets conferred to the testing activity permit to generate only a few of them, restricting the observation of performances to resulting programs. Besides the task of finding bugs in systems, requirements might express some performance goals (*e.g.,* run under a cer-

tain amount of time, use at most a certain amount of memory). Usually, to assess that a given program complies with requirements (being related to performances or not), several tests are needed to observe the system under different conditions. Furthermore, because different aspects can influence performances, the definition of oracles (or expected results) can be difficult, making the assessment of performances tricky. For instance, considering a video encoding algorithm, its performances (*e.g.,* its execution time or the quality of the output video) might depend on the input itself : if the video is of poor quality, with dynamic noise, it might be hard to encode correctly or a denoising algorithm can be used in the pipeline of the encoder resulting in an increase in its execution time compared to other executions without this additional step.

In the end, two dimensions emerge : first, configuring a system which consists in selecting options and assigning them a value ; and, second, selecting relevant test cases that will allow to observe the behavior of generated systems under various conditions in order to know whether they will meet requirements. We propose to represent those two dimensions as a (performance) matrix : one dimension represents selected systems for which performances can be observed while the other dimension represents the set of test cases that will be executed on each of these systems. Each cell is the execution of a program variant regarding a test. In the general case, a cell might be a vector as multiple measures being observed at the same time and needed to decide whether a system meets requirements. Figure 1.1 illustrates how we represent this matrix. In this example, columns represent program variants and rows represent test cases of a test suite to execute. Let us consider that for each execution, the execution time is measured (in seconds) and reported in cells of the matrix. Based on this matrix, we can say a few things :

— Program 1 seems to be rather stable, providing rather low execution times (less than a minute) ;
— Test Case 1 seems to be easily handled by most of the Program Variants ;
— Test Case 2 seems to be difficult for Program 2 which shows a high value.

All of these analyses are interesting either from the software system point of view (as we can map configurations to performances) or from the test suite point of view (as we can observe the diversity of performance results with regards to a test case).

This thesis claims that performance matrices are a fundamental concept that brings interesting information regarding execution of tests and program variants and thus should be leveraged by software engineers for several very useful tasks.

	Program Variants				
Test cases		Program 1	Program 2	...	Program N
	Test 1	12	1	...	5
	Test 2	1	348	...	10
	...				
	Test M	50	101	...	260

FIGURE 1.1 – An example of the performance matrix we exploit in this thesis. Each cell is the result of the execution of a Program Variant (columns) with a Test case (rows). Let us consider that execution time is measured and expressed in seconds.

Contributions :

First, we leverage machine learning techniques in order to specialize a configurable system. The goal is to help selecting a configuration that is likely to meet requirements. In this context, machine learning will use a set of available configurations to predict whether a specific configuration is likely to meet user-defined requirements. End users must be able to express their requirements such that it results in a binary decision problem (*i.e.,* configurations that are acceptable and those that are not). Machine learning techniques are then used to retrieve partial configurations that specialize a configurable system to guide end users and reduce the configuration space. In the end, it aims at diminishing the first dimension of the matrix that deals with systems and programs. We validate this approach with a case study (a video generator) and answer the following research questions : *i)* can we extract constraints from the machine learning technique that actually make sense ? ; *ii)* are machine learning techniques accurate in their prediction regarding the fact that a product is able to meet users' requirements ? ; *iii)* analyzing pros and cons of the proposed approach.

Second, we propose a new method assessing the ability of test suites to reveal significant performance differences of a set of configurations tackling the same task. More precisely, we propose a framework defining and evaluating the coverage of a test set with respect to a quantitative property of interest, such as the execution time or the memory usage. This framework can be used to assess whether a new test case is worth adding to a test suite or to select an optimal test set with respect to the property of interest. In addition, this technique might help structuring the execution of tests. For instance, it can create an order of execution resulting in using less test cases that are presented in the second dimension of the matrix. We validate this new method on three different case studies and answering the following research questions : *i)* is our new

measure used to evaluate test suites right ? Meaning that, does it reflect on the fact that test suites can be discriminated ? And is the measure stable ? ; *ii)* are new test suites (created by optimizing our measure) efficient ? In other words, is a test suite with a higher score better than an other one with a lesser score ?

Figure 1.2 shows rather intuitively how these contributions interact with the performance matrix. The second contribution does not appear on Figure 1.2 as we presented

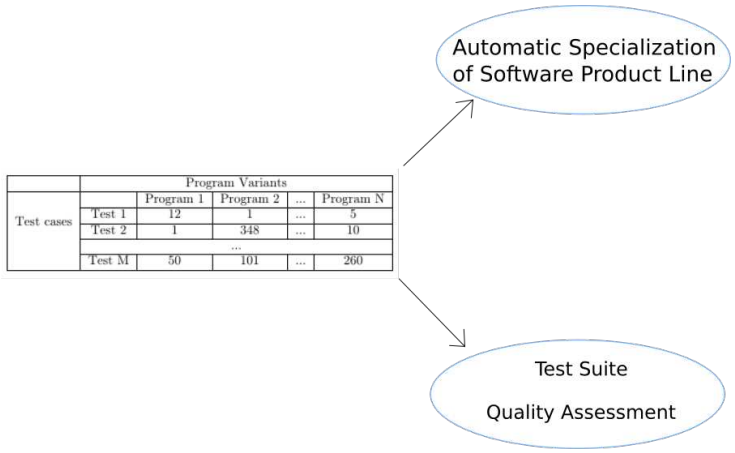


FIGURE 1.2 – Our two contributions in perspective of the performance matrix

it as an extension of the first contribution focusing on a specific aspect (*i.e.,* the definition of users' requirements) and their impact on the performances of machine learning.

The remaining of this thesis is structured as follows : Chapter 2 gives the main concepts related to software product lines, variability models and machine learning. In particular, we give the main motivation to use software product lines and software reuse, we focus on a specific kind of variability models called feature models and we finally present basics concepts behind machine learning.
Chapter 3 gives an overview of previous works that have been conducted in the field of testing program variants, test suite optimization, performance evaluation and quality of tests.
Chapters 4 to 6 detail the contributions of this thesis.
Chapter 7 concludes and discusses future works.

CHAPITRE 2

BACKGROUND

2.1 Software Product Lines

With today's mass customization industry, the traditional software engineering development process has changed [54, 55, 74]. From building one single piece of software answering requirements of a single user, it has come to the point where multiple similar software systems are developed from a common base of code [53, 61].
Clements *et al.* [21] gives the following definition of a software product line :

Definition 1 (Software Product Lines) *A **software product line** is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.*

Definition 1 shows two aspects of the code of software product lines : a common part and a "variable" part that answers specific needs. Common parts by definition are shared by all products while "variable" parts are present only in certain products.

2.1.1 Perks of reuse

Today's software are getting bigger and more complex in terms of customization possibilities. As an example of customizable software, the Linux Kernel [21, 60, 91] is probably the most complex piece of configurable software ever created with more than $13,000$ configuration options. With such a number of options, it is difficult to keep a clear view of the structure of the system.

Pohl *et al.* [74] state the following benefits from reusing as many pieces of code as possible :

— reducing development cost ;
— reducing time-to-market ;

— improving code quality.

Since reuse is at the heart of software product lines, developers have to think carefully about the structure of their code. Capitalizing on code reuse is a way to reduce the amount of code to develop by integrating already existing pieces in new software. With less functionalities to be developed, new products (*e.g.,* software or systems) can be ready and accessible to customers more quickly. Finally, with variability, the structure of code changes (*e.g.,* with *parameters* and *options* activated at run-time or *ifdef* instructions at compile time). Since links are made between code and functionalities, it can be easier to target specific parts of code in which a bug have been detected. Also it results in fixing the bug at one place while being applied to every products that share the modified piece of code. Since different functionalities can be split among several *options*, *ifdefs* and *if conditions* at different places in the code, code become harder to read and its flow might be broken making it harder to follow and understand.

In the end, commonalities are conceived only once ; remaining parts of the code (*i.e.,* variable or optional parts being specific to some requirements) are decoupled and built such that they can be combined to generate the desired software.

Nowadays, software embed so much variable aspects that they are depicted as variability intensive systems [29, 70].

Svahnberg *et al.* [95] defines software variability as follows :

Definition 2 (Variability) ***Software variability*** *is the ability of a software system or artifact to be efficiently extended, changed, customized or configured for use in a particular context.*

As software variability becomes omnipresent (*e.g.,* in video encoding, machine learning techniques, operating systems, code generators, *etc.*), the number of products to manage increases quickly and becomes out-of-hand.

Hence, it can be hard to keep track of implemented functionalities and where can be found in the code or, the other way around, what are the parts of the code that are impacted by a certain functionality. Thus, there is a need to model and to document code, functionalities, requirements, *etc.*

2.1.2 SPL development process

Figure 2.1 [1] shows the typical development process of a software product line.

1. inspired from [85]

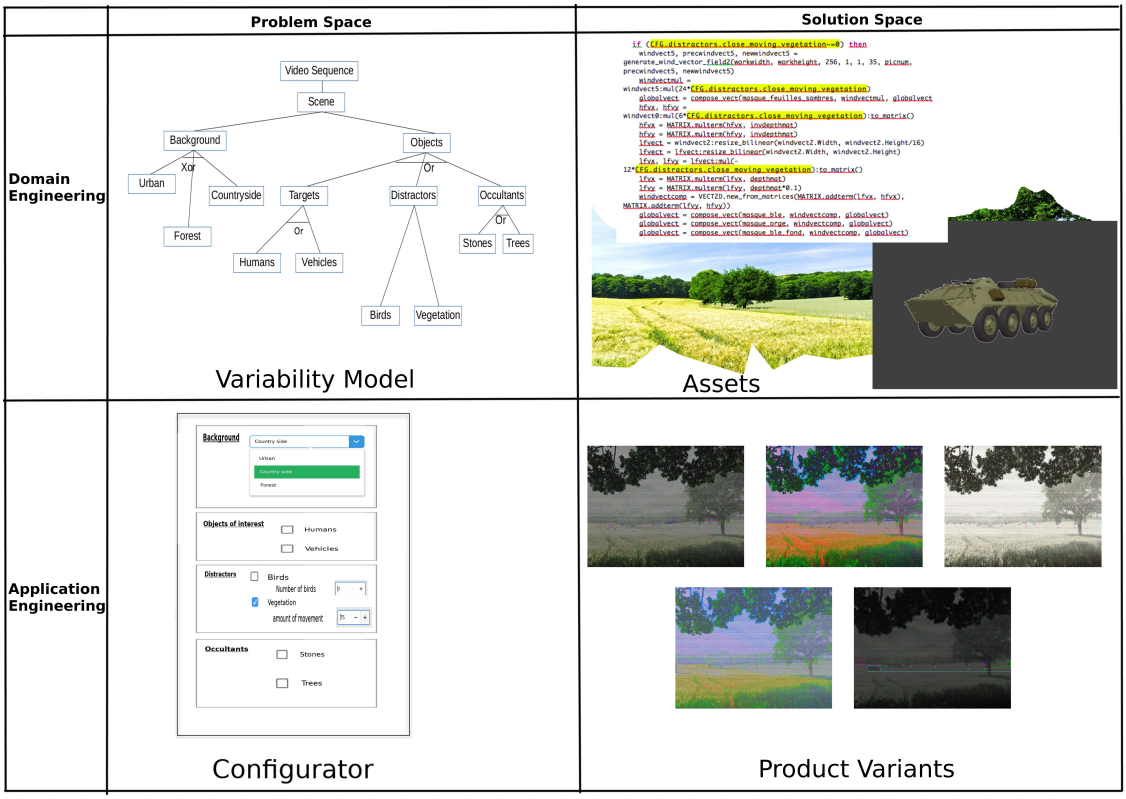


FIGURE 2.1 – Common view of the software product line development process.

Different entities appear in this table. First, columns decouple the *Problem Space* and the *Solution Space*. According to Génova *et al.* [35], *Problem* and *Solution* refer to the contrast between the system under study (*i.e.,* to be modeled) and its application domain. Thus, the *Problem Space* describes the system (using high-level abstractions) in terms of requirements, specifications, *etc.* On the other hand, *Solution Space* tries to implement or at least define artifacts in order to address those requirements. Artifacts are mainly written by and for developers.
Moving from *Problem Space* to *Solution Space* is going from requirements and specifications expressed in natural language to code artifacts written in programming languages. There might be difficulties to link requirements to artifacts as they are not expressed in the same language. One way to go from the first space to the other is the explicit mapping between requirements and specifications via artifacts on one hand and the establishment of features on the other.

Second, rows differentiate *Domain Engineering* and *Application Engineering*. *Domain engineering* refers to a rather abstract activity in which people try to define the general/abstract scope for customers' needs in term of software. *Domain engineering* is about development for reuse since developed pieces of code will be used in a maximum of products created out of the software product line. It is usually composed of four activities. First, the domain analysis in which commonalities and variable aspects are identified. Then assets are developed in order to create the software product line resulting in three more activities : domain design, domain coding and domain testing. On the other hand, *Application Engineering* is centered around development for reuse. Products are built by composing or assembling different assets developed in the previous engineering stage. Again, four activities compose the *Application Engineering* : application requirements engineering, application design, application coding and application testing. They are the pendant of the activities from *Domain engineering*. While application design tries to understand the needs of end users, application coding builds the final product and the last activity tests it. In other words, moving from *Domain* to *Application* is moving from a global point of view of what can be done with the system that is being developed to a user specific point of view in which it will have specific expectations regarding the system they will use.

In any case, there is a need to describe and model variability and possibilities. This is done by the use of variability models (on the top left corner of Figure 2.1). Variability models offer a formalism to visualize variable aspects of a system, which of the features

are related or independent, *etc.*

Definition 3 (Variability Models) ***Variability modeling*** *is the process of representing commonalities and variabilities of a software product line. These aspects can be modeled in different ways depending on the adopted viewpoint.*

A ***variability model*** *is a representation (*i.e., *a model) of commonalities and variabilities of a software product line specific to a certain viewpoint. It documents variable aspects but also it documents which combination of variables(*i.e., *both common and variable aspects) are forbidden.*

According to the previous definitions, we give the following definition to a configuration of a variability model.

Definition 4 (Configurations) *A* ***configuration*** *is an assignment of values for all variables of a variability model.*

As said before, information given by variability models can be mapped directly to pieces of source code (*i.e.,* going from the problem space to the solution space in Figure 2.1). Based on possibilities and constraints provided by the variability model, a configurator can be set up in order to give feedback to users and help configure a product (*i.e.,* going from Domain Engineering to the Application Engineering in Figure 2.1).

Now that source code have been developed and end-users have selected requirements that should be addressed by the system (or modules that should be included in the final product), pieces of source code can be assembled accordingly to the configuration in order to provide the expected system. This is done by associating the selection made in the configurator (*i.e.,* bottom left part of Figure 2.1) with corresponding assets that were previously developed (*i.e.,* top right part of Figure 2.1). Once the system is assembled, it is called a variant and the process of delivering a system is called Product Derivation.

2.2 Feature Models

Feature models are a specific kind of variability models that focus on the representation of variability via features. Even if different languages exist to express variability [5, 84], focusing on different aspects of it [74], feature models are the *de facto* most popular approach to represent variability to date [8, 13, 14, 44].

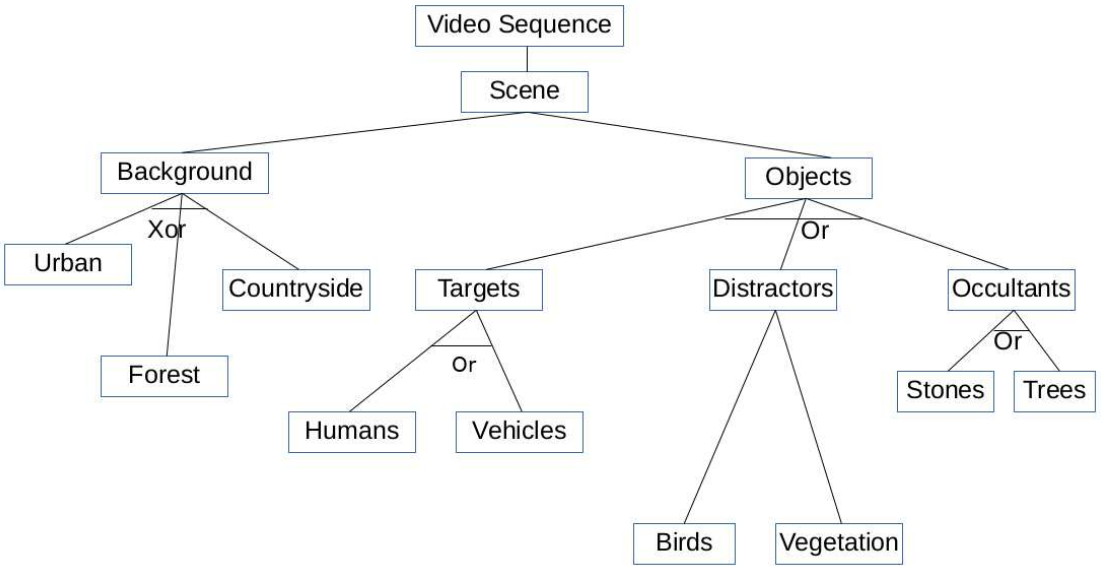


FIGURE 2.2 – A feature model representing how to create a Video Sequence

Feature models provide information about how features can be assembled and which choices can be made regarding the use of specific features or sub-features.

A feature model being an instance of a variability model, the definition given by Def. 4 still hold as options become features.

The representation of a feature model is a combination of a graphical representation and a textual description.

2.2.1 Fundamentals of feature models

Graphical representation

Different graphical representations exist, all representing the same information but with different graphical code.

Feature models represent systems as a feature hierarchy. Hierarchy enables to organize a large number of concepts (or features) into increasing levels of detail. The hierarchy usually is represented as a tree with the system under study placed at the root of the tree since it is the most general concept. Features are represented as nodes while edges representing parent-child relationships between features. These relations allow for specializing concepts (*e.g.,* a feature is a sub-feature of an other feature or, the other way around, a specific instance of a super-feature) or aggregate features (*e.g.,* a

feature can require other sub-features to be selected). Other graphical elements can be added to describe variability. For instance, a feature can be optional, meaning that they can be used in certain products but not necessary all of them. In this thesis, we consider that features are mandatory (*i.e.,* have to be used in all products) if they are not marked as optional.

Alternative groups can also be specified : *Or* can be defined to force one or several choices among a set of possible features. *Xor*-group is the exclusive version of an *Or* group meaning that only one feature can be selected at once.

In fact, feature models not only define how features can be combined and assembled but they also describe which associations are forbidden via the use of constraints.

Implies and *excludes* constraints can be stated. These constraints are more complex as they can put different features in relation that are not at the same level in the hierarchy (or even not under the same parent feature). Even though *implies* and *excludes* constraints can be represented graphically, it is usual to write them down in a textual form. Usually, constraints are written in propositional logic which allows a powerful expressiveness using disjunction($\vee$), conjunction($\wedge$), negation($\neg$), implication($\Rightarrow$) and bi-implication($\Leftrightarrow$). However, sometimes it is not enough. Some constraints might be hard to express in propositional logic or might involve a large number of propositions (and features) and thus it may become clearer too simply right them in a different form aside of the feature model. Then, the problem remains to connect these constraints with the feature model and the whole automatic configuration and derivation process.

Configuring feature models

Assigning a value to each feature in the model also serves to select and discriminate the desired product from the ones encoded by the feature model. However, when assigning values to features, constraints need to be checked otherwise it could result in a product that cannot be created. In addition to expressed constraints (in propositional logic in the feature model), the following rules must be followed :

rule 1 : if a feature is selected, its parents are also selected (the edge between the two features not only defines a conceptual relationship but also a logical dependency) ;

rule 2 : if a parent is selected, all mandatory sub-features must be selected ; exactly one sub-feature in each of its Xor-group must be selected ; at least one sub-feature in each Or group must be selected ;

rule 3 : constraints must hold (*e.g.,* implication and exclusion constraints)

To illustrate those rules, let us consider Figure 2.2. Rule 1 states that, for instance, if the feature called "Urban" is selected in a configuration, then the feature "Background" also have to be included in the configuration. The same logic applies to "Background" and "Scene", *etc.*

Rule 2 says if feature "Scene" is selected in a configuration, sub-features "Background" and "Objects" should also be selected in the configuration as they are both mandatory.

Finally, Rule 3 specifies that constraints in propositional logic should not be violated. For instance, if the following constrain was specified : Feature "Birds" implies Feature "Forest", then every time the Feature "Birds" is selected, the associated Background should be "Forest".

If all constraints and rules are respected, the configuration and resulting product are said to be valid.

Definition 5 (valid configurations) *A configuration is valid if values conform to constraints. A variability model* VM *characterizes a set of valid configurations denoted* $[\![VM]\!]$*.*

Expressivity of feature models

Different kind of feature models exist bringing their own languages and own expressivity [5]. Maybe the most simple form of feature model is the boolean one. In such feature model, the only possibility for each feature is to be selected or deselected.

Figure 2.2 shows an example of a boolean feature model. It represents how tracking algorithms [2] can be built out of different techniques.

As we said before, features (*i.e.,* nodes or rectangles in Figure 2.2) not marked as optional are mandatory. Thus, each tracking algorithm derived from this feature model will embed a "Recognition" sub-system (which can be either a "Template Matching" algorithm or a "Pyramidal" technique). Both "Detect" and "Tracking" can be selected or deselected at the same time or independently. At the bottom of Figure 2.2, cross-tree constraints are expressed further limiting the combination of features.

On top of that, more complex feature models can allow features to take real values or values in a given domain (*i.e.,* a set of values). Attributes can also be associated to

2. algorithms designed to recognize objects of interest and follow their paths in videos

features. They add even more complexity in the reasoning as it increases expressiveness.

Automated Reasoning

As the expressiveness of the model increases, the number of possible configurations becomes too large and more constraints might be necessary. With more constraints, it becomes harder to have a clear view and a clear mind map of allowed combinations of features. Still, the modeling part via variability models is crucial since it is the starting point of the configuration process as shown in Table 2.1. Unnecessarily constraining feautures or missing some of the constraints can lead to undesired behaviors in the next steps of the configuration process. Examples of undesired behaviors are : dead-feature (*i.e.,* features that are never used in any products), empty set of valid configurations (*i.e.,* no product can be derived), under-constrained features (*i.e.,* features can be activated in products that do not use them), *etc.* There is a need to ensure that configurations can be derived for real and that the variability model is able to provide the right configurations, no more, no less. Providing such insurances is not trivial as configurations of feature models are expressed in propositional logic. That is, for each configuration, all features are present in the formula separated by conjunctions or disjunctions. These kind of formulas are hard to follow for human beings especially when they involve a large number of features.

Because of that, automated reasoning is needed as shown in [12, 13, 23, 63, 64, 112]. Automated reasoning tools have been developed and explored according to the nature of the feature model. They usually take as input a formula expressed in propositional logic (representing the set of constraints expressed by the feature model) and a partial configuration (*i.e.,* not all values of a configuration have been specified) and answer whether the configuration can be completed such that it satisfies all constraints. Among explored solutions, SAT (satisfaisability) solvers or Binary Decision Diagram seem to be viable solutions when dealing with Boolean feature models while, for more complex feature models, Constraints Satisfaction Problem (CSP) solvers or Satisfiability Modulo Theories (SMT) solvers are a better choice. Note that the goal of solvers is to complete the partial configuration such that constraints are met and return a configuration (or return an empty set is no solutions can be found).

Other tools have been developed in order to reason about software product lines. FeatureIDE [48, 106] is a tool that aims at improving software product lines by analyzing

features and structures of the feature model and proposing fixes (*e.g.,* removing dead-features).

Familiar [2] is another tool that proposes a domain-specific language supporting the separation of concerns in feature modeling. It provides, among other things, automatic reasoning facilities about the structure of the feature model.

Being able to establish a proper variability model remains challenging and important as it determines the configuration space in which configurations are selected. Work presented in this section tackle the problem of verifying the structure of the variability model. They also allow to automatically draw configurations that satisfy constraints stated in the model which is important in our case as the performance matrix presented in Figure 1.1 relies on an available set of product variants (and their configurations).

2.3 Machine Learning

Machine learning is a part of artificial intelligence gathering methods that try to model reality based on data, experience and statistics. Being experienced-driven, these kind of algorithms are supposed to perform better as more and more data are provided. Via training, it aims to automatically induce models, such as rules or patterns, that predict a value associated to an observation. Formally, it is noted as

$$y = f(\mathbf{x}) \tag{2.1}$$

where y is the value [3] to predict and **x** a vector representing observations.

Usually, machine learning is associated to decision problems : "According to what have been seen previously, which value is the most probable for this particular observation ?" Depending on the nature of the values to predict, the decision problem can be either a classification problem (if values are discrete) or a regression problem (in the case of continuous values).

To illustrate what has been said, we take as an example one of the oldest data sets studied by machine learning researchers : the iris data set [33]. This data set gathers 150 examples of irises. The goal is to categorize these examples into one of three classes, namely : *iris versicolor*, *iris setosa*, *iris virginica*. Figure 2.3 shows categories'

3. it can also be represented as a vector, for instance, to represent the confidence in belonging to a particular category



FIGURE 2.3 – 3 categories of irises with a representative of each category

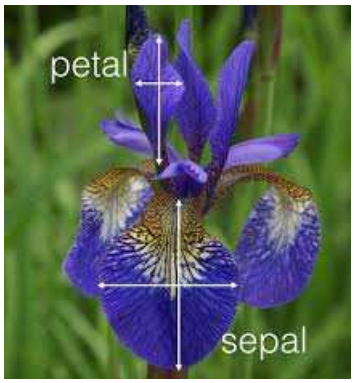


FIGURE 2.4 – Irises are described by the length and width of their petals and sepals.

names as well as a representative of each category.

In this data set, examples are not raw data or images, examples consist of observations of 4 characteristics (also called features[4]) that describe the plants. These 4 features are : length of the petal, width of the petal, length of the sepal and width of the sepal as shown in Fig. 2.4.

Describing data with fewer information as it is done with irises can be viewed as applying a first transformation (called feature representation) to data. This feature representation results in a vector that projects data in a feature space that is supposed to be more interesting to perform the task at end (*e.g.,* classifying irises into categories) as illustrated in Fig. 2.5.

Note that the plan shown in this figure separating points into two areas (*i.e.,* above represented by blue points and below the plan represented by red points) is only one solution of the separation problem and other solutions may exist. Taking this into account, in the case of algorithms looking for linear solutions, equation (2.1) changes and

4. feature is an other word to talk about descriptors that are used to describe data and examples in a more concise way. Note that, in the software product line world, features can have a different meaning.

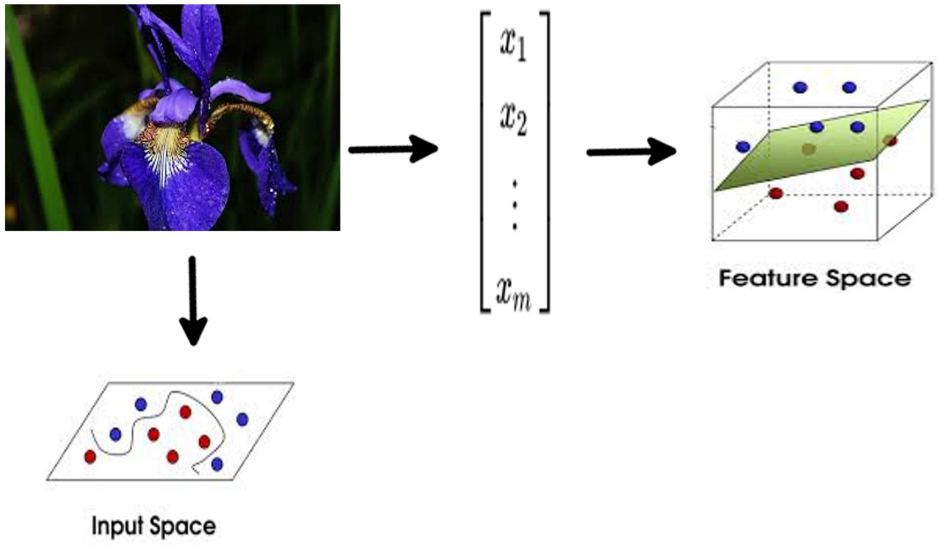


FIGURE 2.5 – the process which is used to learn how to separate configurations

becomes :

$$y = f(\mathbf{x}, \mathbf{w}) \tag{2.2}$$

In this equation, **w** can be seen as a vector of weights that is applied to **x**. It can give more importance to certain features than others which will, in the end, modify the equation of the model.

Now, we detail the process to compute a model.

2.3.1 Stages to use a machine learning algorithm

Traditional stages to use machine learning algorithms are as follows :

— collect examples that would be given as the training set to the algorithm ;
— determine the feature space and transform data to retrieve a feature vector [5] ;
— choose a learning algorithm corresponding to the kind of problem to tackle as well as the nature of data ;
— train the predictive model ;
— evaluate the performance of the algorithm's decisions on new data

The first step consists in collecting data that are representative of the task at hand. As machine learning algorithms are about statistics, collected data must also be representative of the variability that can be encountered in the nature. Said differently,

5. Deep Learning algorithms includes this step in the training phase (at least for multimedia data)

they should follow the underlying probability distribution (*e.g.,* outliers should not be over-represented and modeling the distribution of data should globally fit the original distribution).

Then, as we said with the iris data set, a feature space must be determined (which is supposed to provide a better representation for the task to perform) and features are computed over raw data.

The third step is also important since some machine learning algorithms are more or less efficient to deal with the different nature of descriptors (*i.e.,* discrete/continuous, ordered, *etc.* .) and their homogeneity (*i.e.,* are all dimensions of the feature space of the same nature ?).

The training phase will fix parameters of the model (*i.e.,* vector of weights **w** in equation (2.2)).

Finally, the evaluation of the performances assesses whether the model is general enough to perform well on previously unseen observations.

Hereafter, we detail these two last steps which are at the heart of machine learning techniques.

2.3.2 The training phase

This phase uses a part of collected examples[6] in order to build the function $f(.)$ from (2.2).

The goal is to set **w** such that the model is able to, ideally, assign the correct value y for each given example **x**.

Machine learning algorithms are usually divided into two big families : *unsupervised* and *supervised*[7]. Those two families rely on different information to build their model.

Unsupervised vs Supervised

The main difference between unsupervised and supervised techniques appears in the training phase of the algorithm. In the *unsupervised* family, only the description of data are given to the algorithm. Defined features are supposed to structure the feature space such that interesting properties of examples be grouped together. Usually, unsupervised algorithms try to learn the probability distribution that generated the data

6. the remaining examples are used to evaluate the performances of the model
7. however, with the growing interest in the field, the frontier between the two families become fuzzier

set. The learning algorithm uses the proximity of examples and heuristics to determine a model that separates two distributions (or more). These methods are typically used to tackle clustering problems. For instance, using the 4 features of the iris data set, we hope that the feature space will create 3 clusters that gather irises from the same category while separating each category from the two others.

In the *supervised* family, the expected labels (or values) are given along with examples. The algorithm can then try to discover relations between features and expected labels. Typical applications of *supervised* algorithms are *classification* or *regression* problems. The difference between the two applications is that *classification* tries to find categories or labels while *regression* associates continuous values with data.

Since setting the best values for parameters **w** on the first try is unlikely, the training process might result in an iterative process increasing performing prediction of the model at each step. To do so, evaluating the prediction performance is necessary to assess whether the model is "good enough" [8].

2.3.3 Evaluating prediction performances

Evaluating the prediction performances of a machine learning algorithm consists in assessing the generalization capability of the model (*i.e.,* whether the built model is able to assign the correct value to previously unseen data). Data used in the evaluation must not have been encountered before, however, their distribution should follow the same distribution as the examples used to train the model. This set of data is usually called *test set*.

In the *regression* case, assessing the generalization capability of a machine learning technique consists in computing the difference between the value given by the algorithm with the expected one. Several methods can be used to compute these differences. The most common being the Mean Squared Error (MSE) and the Mean Absolute Error (MAE).

When talking about *classification*, the evaluation compares the output given by the model on new data to the expected one (called *ground truth*). The ground truth is usually defined by "experts" (or an oracle) that "knows the true nature of data" (*i.e.,* their exact distribution). Note that with the use of Mechanical Turks over the recent past

8. the notion of error acceptance is usually correlated to the context in which a machine learning algorithm is used and to the task to tackle

		Ground truth	
		+1	-1
ML decision	+1	True Positive (TP)	False Positive (FP)
	-1	False Negative (FN)	True Negative (TN)

TABLE 2.1 – An example of a confusion matrix for a 2-class problem (*i.e.,* class +1 and -1).

years, the term "experts" does not necessary mean that people have great knowledge about the field, *etc.* [77, 17].

Comparing decisions from the oracle and the machine learning model is usually sum up into a confusion matrix.

Table 2.1 depicts elements of a confusion matrix of a machine learning algorithm confronted to a ground truth over a 2-class problem. The two classes are represented as +1 and -1. In total, four possibilities are shown in this matrix (*i.e.,* True Positive, False Positive, False Negative and True Negative). True Positive, False Positive, False Negative and True Negative refer to quantities that can be summed up into two situations. Either the machine learning decisions agree with the ground truth either they do not. At every decision performed by the machine learning classifier, the amount in the corresponding cell is increased. When the decision and the ground truth agree, only the main diagonal can be impacted (*i.e.,* True Positive or True Negative). When they disagree, the other diagonal is impacted (*i.e.,* False Positive or False Negative) and it is, in fact, an error from the machine learning algorithm.

Based on the four quantities depicted by the cells of the matrix, ratios can be derived in order to give an idea on how well a machine learning algorithm is able to take the right decision. Probably the most commonly used ratios are :

— *Accuracy* : $\frac{TP+TN}{TP+FP+FN+TN}$
— *Precision* : $\frac{TP}{TP+FP}$
— *Recall* : $\frac{TP}{TP+FN}$

but others exist [31] and can be useful depending on what needs to be analyzed.

2.3.4 Overfitting and underfitting

As we said, combining training and evaluation steps can lead to an iterative process.

A straightforward approach to make this process iterative is to inject into the training set examples from the test set that were wrongly predict. Nonetheless, by doing

so, there are no guarantees that the impact of adding such examples to the training set will be positive (*i.e.,* reducing prediction errors in the training set and/or test set). Sometimes it can have negative impacts since adding examples to the training set might reduce the generalization capability of the trained model. Meaning that it will perform worse after retraining than before.

When a model is performing well on the training set but has little generalization capability (*i.e.,* producing a lot of prediction errors on the test set), it is said to **overfit** the training set. On the contrary, if it does not even succeed in producing a few errors on the training set, it is said to **underfit** the training set.

In the end, the performance of a machine learning algorithm can be determined by its ability to : produce few training errors and make the gap between training and test errors small (*i.e.,* having a "good" generalization capability).

2.3.5 Hyperparameters and validation set

For now, we supposed that the learning step produced only one model, however, some machine learning algorithms can have hyperparameters that can influence the behavior of the algorithm. An example that is used by Goodfellow *et al.* [37] is finding a polynomial that is able to fit some data points. The degree of the polynomial model will affect its ability to fit or not those points. In this example, a quadratic function is sufficient to fit data and increasing the degree of the polynomial degree might lead to overfitting.

To avoid overfitting, when several models can be produced, a third data set can be used. It is called the *validation set* and is usually a subset of the training set that was set apart and not used in the training process. Thus, data from the validation set follows the same distribution that the training set and contain examples that were never-seen before by the algorithm.

The set is used to fix hyperparameters of the model and might help selecting a model that will perform well on both the training and validation set.

A typical method to set hyperparameters is the **cross-validation**. It is really useful in the case that the training set contains a rather small number of observations. Cross-validation repeats the training and validation process on differently randomly chosen subsets of the training set. K-fold cross-validation is the most common form of cross-validation in which k disjoint subsets are created. $k-1$ folds are used for training and the

last fold is used for validation. The process is repeated k times, changing the validation fold every time, and the average of the errors made over the k folds is computed. It gives an idea of the generalization capability of the model.

Setting hyperparameters have influences on the performances of machine learning. They can affect different aspects of a machine learning technique, for instance, considering neural networks, hyperparameters can set up the number of layers of the network but also the activation functions of neurons, *etc.* Some parameters can even impact algorithms in making more or less prediction errors during the training step to enhance generalization capabilities afterwards.

2.4 Summary

In this chapter, we introduced basics of software product lines and variability models and machine learning.

Software product lines are centered around customers' requirements and code reuse. Nowadays, there are plenty of techniques that tackle the same problem but focusing on different aspects caused by the fact that customers have different needs (*e.g.,* minimal resources consumption, fast to execute, *etc.*). This results in changes in the way software are conceived : because software tackle the same problem, it is reasonable to think that a common part exists and should be developed once and reuse for all program variants.

As a result of entering an area of mass software customization, software are becoming more and more complex and customers are evermore demanding regarding their requirements. There is a need to model which parts of the code are common to all products and which are variable (*i.e.,* particular to a subset of software). Variability models have been created to that end and feature models are the *de facto* standard to reason about software variability. Using this formalism, a product (*i.e.,* a specific piece of software) is described by a configuration (*i.e.,* a set of features selected from the feature model).

Some bridges exist between the use of variability models and machine learning. For instance the notions of valid and invalid products that remind a classification problem usually associated to machine learning techniques.

Due to the increasing number of products that can be produced by software product lines, trying to derive all products becomes unfeasible. But, users might be interesting in

only a sub-space of products that can be represented. Machine learning can be used to restrict automatically large configuration spaces according to some user-defined requirements. Since machine learning are data-based techniques, leveraging a sample of previously used configurations and their configurations can help narrowing the scope of products to cover.

CHAPITRE 3

STATE OF THE ART

This part gives an overview of previous work that are related to the problem we tackle in this thesis.

From the performance matrix we have presented in Chapter 1 and illustrated in Figure 1.2, we can understand that two different aspects are important. They are presented in the following figure, providing a complete overview of how work presented in this thesis are positioned :

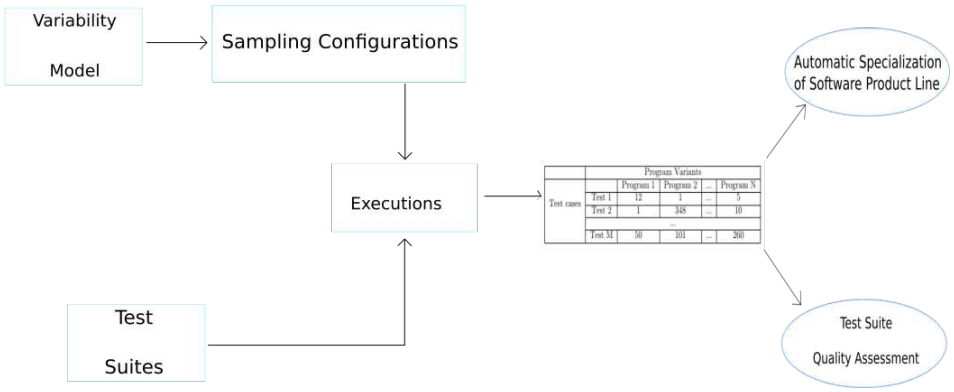


FIGURE 3.1 – The studied state of the art in relation with the performance matrix

Before being able to build the performance matrix, we can understand that variants need to be derived and available which necessitates to test the underlying software product line. Testing a software product line consists in two steps : checking the associated variability model and testing resulting products. Checking a variability model corresponds to explore the configuration space in order to see whether problems can occur (*e.g.,* unauthorized combination of options) while testing derived products requires to execute them and assess that they provide expected results including fulfilling performance concerns.

The execution of variants relies on the use of test cases, that corresponds to the other dimension of the matrix. From this perspective, the problem of building a test

suite of good quality is important. Nonetheless, evaluating the quality of a test suite is difficult and the definition of quality is dependent of what has to be assessed.

In the end, because of the sheer size of the matrix, the Cartesian product of both dimensions (*i.e.,* the number of executions to be made to fill the matrix) is too large to be computed extensively. In the following, we discuss works that are related to software product line testing, test quality assessment and machine learning techniques applied to predict performances of software product lines.

3.1 Software product lines and Testing

As software product lines are becoming more complex, proposing evermore customization possibilities, it becomes difficult to know if bugs exist in the derivation process. It is also difficult to know whether the model provided by the variability model is correct or whether some parts of the model are useless, *etc.*

Testing a software product line consists in selecting a set of products that are derived and then executed on test cases in order to assess their behavior (*e.g.,* do not crash, compile, do not take too long to be executed nor an excessive memory consumption, *etc.*). The goal of the selection of products is to cover as many different and diverse configurations as possible in order to use all features at least once and check that the derivation process is well-coded. The second part is to measure performances and, thus, compare measures with respect to requirements. These two levels of testing correspond to the dimensions of the matrix we presented in the introduction of this thesis : the first dimension represents products while the other dimension represents test cases.

This section is dedicated to work tackling the first aspect of testing a software product line. In particular, we focus on the following problems : configuration sampling and verifying software product lines.

3.1.1 Configuration sampling

Since configurable systems become more customizable, being able to properly sample configurations out of software product lines is important. It gives the opportunity to appreciate the panel of performances that products are able to reach.

Efficiently sampling configurations is a problem that has caught the interest of researchers for several years. Empirical studies conducted by Sarkar *et al.* [79] and Medeiros *et al.* [62] showed that sampling strategies can influence the number of faults that have been detected. For instance, increasing the size of sample sets can have positive effects on the fault-detection capability, but it also has a cost. Choosing the right sampling strategy is still an open problem and many different methodologies have been already proposed as shown by Medeiros *et al.* [62].

Different work [22, 42, 56]) point out that generating valid configurations by random choices is extremely difficult in presence of constraints. Constraints may considerably restrict the configuration space and may lead to inconsistent (or invalid) configurations if not taken into account, in turn producing incorrect systems. Furthermore, constraints might be complex as they can bring several features into the equation. These work try to efficiently create configurations that cover combinations of features while taking into account constraints of a variability model.

Other works have proposed to use a different approach in the derivation process. Guo *et al.* [40] adapts genetic algorithms to derive products that satisfy constraints producing only configurations that are valid *w.r.t.* the feature model. However, it does not tackle the problem of covering the configuration space but rather to quickly produce configurations that are valid thus reducing the try-and-error process that might occur in previous works and helping debugging the variable-intensive system.

Al Hajjaji *et al.* [3] have proposed an approach which assumes that bugs occur via feature interactions. This approach tries to find feature interactions and try as much as possible to diversify configurations that are covered by the test set. Following a similar idea, Johansen *et al.* [46] propose to focus on covering arrays that will produce a set of configurations in order to cover all combinations of a fixed number of features (called t). For instance, if $t = 2$, all pairs of features should be covered. Different work [113, 75], in the context of quality assurance, use a sophisticated framework encompassing covering arrays along with classification trees in order to characterize the configuration options that are responsible for failures. Galindo *et al.* [34] present an approach that optimizes a test suite following a t-wise criterion coupled with a multi-objective optimization function. This work considers a video generator automatically producing video sequences in order to benchmark Computer Vision algorithms. The video generator is modeled by an attributed feature model which contains both attributes and features that can take real-values. The heterogeneous nature of the feature model makes previous

work hard to use as they were designed for boolean feature models only. Attributes and real-values are important as users can express an intention in the product they want to create (*e.g.,* create a dense environment with a lot of moving objects that can partially occult each other) or a specific atmosphere (*e.g.,* a scene recorded at night). The multi-objective optimization is brought by the fact that users might be interested in deriving variants with predefined value for some features. For example, regarding video sequences, users might want to produce video sequences that take place during the day and thus, they would like to produce only this kind of videos which limit values that can be set to some features (*e.g.,* features controlling light sources and their brightness on the scene).

All those works aim at sampling the configuration space, trying to pick configurations to cartography this space. Recently, Varshosaz *et al.* [109] proposed a survey classifying different sampling techniques in order to provide a new, clearer structure and trying to pinpoint new fundamental problems in the field. Being able to appropriately sample a space is an important topic that remains crucial in the world of software product line. However, presented work do not take into account the variability of performance that can be reached. That is, despite knowing that configurations are linked to performances of systems, they only consider covering the configuration space in order to test the software product line. Being able to sample both diverse configurations that also show diverse behaviors is important from our point of view since both are important to end-users. Furthermore, we exploit both aspects in our performance matrix (see Fig. 1.1) : diverse configurations represent the program variants (*i.e.,* a first dimension) which, hopefully, leads to diverse behaviors regarding various test cases (*i.e.,* the second).

3.1.2 Fault Detection in software product lines

Detecting faults and testing software product lines can be decomposed into two activities : checking the structure of the underlying variability model and testing products derived from this model.

Thüm *et al.* [104] proposed a survey analyzing a number of techniques and tools that have been developed in order to verify software product lines. They are mostly based on testing, type checking, model checking or theorem proving. Kim *et al.* [49] applied static program analysis techniques to find irrelevant features for a test. SPLif,

proposed by Souto *et al.* [93], aims to detect bugs of software product lines with incomplete feature models.

Once the structure has been checked, testing products require computations to test their behaviors. It might still take a long time. Thus, there is also a need to order the execution of test cases. The goal of ordering their execution is to execute, first, test cases that are the most important (*i.e.,* that provide the most information about the behavior of variants). A typical application is to ensure that variants behave as expected on common use cases. Besides ordering test cases, it can also be used to reduce the test suite such that executions never exceed a given testing time budget.

A body of work [3, 26, 49, 50, 51, 66, 93, 104] has been developed focusing on which tests should be executed first according to the time budget that is allocated to the test activity. The problem here is the increasing number of products that can be derived from a product line. As this number gets bigger, testing all of them becomes increasingly time consuming up to a point that it is not feasible anymore (*e.g.,* the linux kernel can provide up to $2^{13,000}$ different configurations which is more than the estimated number of particles in the universe). Halin *et al.* [41] reported that an industrial project (*i.e.,* JHipster) only consider the most commonly used configurations in order to run tests over the product line. Due to limited resources, developers were able to run tests on only 12 configurations while the product line can provide more than $26,000$ configurations. Halin *et al.* showed that deriving and analyzing the behavior of all products of a product line can provide useful information. In particular, in their study, they found that more than $\frac{1}{3}$ of the configurations did not provide a product able to compile caused by a feature interaction that was not documented. In addition, authors used different sampling techniques in order to check whether one of those strategies might fit into the test budget but all of them failed, meaning that there is still room for improvement in this area.

In previously discussed work, the order of execution of variants remains to be determined by the tester. They provide a way to reduce a test suite by selecting a subset of the test suite but no orders are applied on elements of the resulting set. Meaning that no other guidance has been given to testers such that it could help them to decide which test cases of the subset should be executed first. Retrieving an order of execution may give several benefits. First, it could avoid exceeding testing time budget while minimizing the impact of not executing a bunch of test cases. Second, it could further reduce the size of the test suite since the last test cases may not matter that much and

thus could be discarded. Mustafa *et al.* [3] propose to use a similarity-based approach to address this problem. That is, by analyzing configurations of systems, products can be clustered as being "far" or "close" one to an other. Thus, the idea is to select first the product that maximizes feature interactions (*i.e.,* the product that activates the maximum of features), then to take the most dissimilar product, and so on with respect to this set of products and a distance measure [1]. The order in which products are drawn corresponds to the order of execution that should be applied. This approach seems to work very well for binary feature models but might be more difficult to use when features can take real-values since similarity may rely on heuristics and the number of products might quickly increase due to multiple interesting values taken by such features. Furthermore, a threat that has been identified is the scalability of the approach. This technique might take too long to identify products that maximize the hamming distance with the set of products under construction as constraints might be complex and the set of valid products might be very large to explore.

3.1.3 Metamorphic Testing

Metamorphic testing is widely used when the definition of an oracle for tests is hard to define. Metamorphic testing relies on the definition of so called *Metamorphic Relations* which are necessary properties stipulating the expected behavior of the software under test. The goal is to check these *Metamorphic Relations* via multiple executions of the software. If a single execution breaks one of the *Metamorphic Relations* then bugs have been introduced into the code of the software.

A typical example is the implementation of the *sinus* function. One possible *Metamorphic Relations* regarding the *sinus* function is $sin(\pi - x) = sin(x)$ stating that computing $sin(x)$ should return the same value as computing $sin(\pi - x)$.

Twisting a bit the idea, instead of considering different executions differing in only the input (as it is the case for the previous example), one could consider different implementations of the same functions. Since these programs are supposed to provide the same function, results should be equal. Results being different is abnormal and, in some sense, a bug has been introduced in one of the implementations. Further investigations might be needed to target more precisely which ones of the implementations or which part of the code are faulty, meaning the potential use of more executions, in

1. authors proposed to use the Hamming distance

turn, involving more implementations and/or inputs.

The goal is thus to compare different programs using the same input and determine whether some relations are kept from one execution to another [10]. For instance, Donaldson *et al.* [27] applied metamorphic testing on computer graphics renderer, a domain in which programs can be compiled with different compilers, using different librairies, *etc.* However, in such a context, assessing that the rendered image and the reference image are identical can be difficult (*e.g.,* due to different optimization, memory representations or color representations).

Furthermore, Liu *et al.* [59] conducted a study to assess the fact that metamorphic testing is able to catch bugs when oracles are hard to define. Since then, several surveys have been conducted [10, 82].

Non-functional properties remain out of the scope of all discussed work. Non-functional properties or performance properties are usually hard to assess with the definition of a strict oracle (as it could be done with the sinus function) since the environment might affect these properties. For instance, assessing execution time might depend on the processor of the machine or the workload of the processor might also impact the execution time. Recently, some work discuss the use of metamorphic testing [15, 83] to alleviate the test oracle problem in the context of evaluating non-functional properties.

3.2 Tests quality

Until now, we have talked about testing software code through the selection of configurations but different techniques have also tried to assess the performance of tests in discovering bugs. We provide an overview of those techniques in the following.

3.2.1 Traditional metrics

Probably the first methods to test programs were the so-called "structural" software testing [43]. The ultimate goal of structural testing is to find bugs in a program. Tests must be executed in a white-box context (*i.e.,* the code must be accessible in order to assess how much have been covered by a test suite). Further works [67, 98] have discussed benefits and limits of such approaches. Structural testing can be used as a complementary approach to other testing techniques to report which portions of code

have not been executed at all. Such parts of the code might be a problem as bugs can loom there. This piece of information can be used to put more effort in producing new test cases that will cover these parts. This kind of analysis can be done at different scales, for instance : statement, branch, *etc.* The most simple scale is the statement-level. In this case, a test suite will try to cover a maximum of lines of code. An associate coverage measure is used and can be declined for every scale. In the case of statement coverage, it is defined as the ratio of executed lines of code to the total number of lines in the system under test. Based on this score, test suites are supposed to be comparable.

In any case, structural testing does not focus on functional aspects of the code nor other non-functional aspects (*e.g.,* the execution time is "normal" *w.r.t.* requirements or any other specifications, *etc.*).

3.2.2 Mutation Testing

In the software engineering community, *Mutation analysis* is a well-known technique to assess the effectiveness of test suites or to support test generation [6, 7, 11, 36, 69, 78].

Mutation testing [7, 78] tries to assess how well a test suite can detect bugs in programs by running it against synthetic program variants (called mutants) in which faults are injected. Different fault models can be used such as the removal of function calls, changing operators in numerical operations, removing a set of instructions, *etc.* Test effectiveness is measured based on the number of “killed” mutants (*i.e.,* mutants failing the test). Baudry *et al.* [11] proposed to adapt genetic algorithms to produce a bacteriological algorithm in order to produce a set of test cases. While genetic algorithms tend to produce sets converging towards similar individuals, bacteriological algorithms, on the contrary, try to provide a diverse set of individuals by starting with a common "bacteria" that will mutate in different ways and create new individuals as new generations are computed. The goal is to build new test cases that are likely to find new bugs in a system.

3.2.3 Quality of performance tests

In the end, based on the body of work we presented, it seems that no approaches exist in assessing the quality of performance tests. Structural testing does not care about performances at all, it only reports on parts of the code that have been executed at least once, helping to find bugs or to create new test cases that will focus on uncovered parts of the code.

Mutation testing assesses that tests are able to catch bugs in pieces of code. Once again, fault models used to inject bugs only care about the structure of the code (*e.g.,* removing some of the pieces) but they do not focus on performance aspects, and therefore are not able nor designed to detect such problems.

In the end, there is a lack of interest in the testing community to assess performance aspects of systems. Because of that, the problem of building efficient test suites assessing performances is also left apart. Probably a representative example is machine learning algorithms. Every year, different competitions are organized proposing their own data sets to evaluate competitors' techniques. Proposed data sets have grown over the years, up to a point that millions of images are given to competitors but nobody knows whether every image in the data set are essential to the evaluation.

3.3 Machine learning and software product lines

Artificial intelligence is already present in the world of software product lines. Solvers have been used, for many years, in configurators to help users know whether solutions exist in the configuration space based on choices that they have already made. However, nowadays, artificial intelligence is evermore assimilated to machine learning and deep learning.

In the following, we propose an overview of synergies existing between machine learning and software product lines.

3.3.1 Performance prediction

With the amount of products that can be derived from modern systems, trying to predict performance of individual products is an important topic of research. Chen *et al.* [20] emphasize the fact that for component-based systems, it is difficult to properly

understand the influence of the use of components over the performances of systems. It is even more difficult since interactions might occur between two components. In the same vein, Tawhid *et al.* [97] remind that predicting performances of product variants is difficult since products produced from a same software product line may be used under different settings, in different contexts. To avoid that, generic performance behaviors are built via the use of benchmarks. Chen *et al.* [20] proposed a method to build more fine-grained models for components by observing their behaviors on a benchmark while Tawhid *et al.* [97] proposed a succession of two model transformations starting from the software product line model to a performance annotated product model and finally to a performance model. Sincero *et al.* [92] proposed a framework in which (non-functional) performances of previously derived products are kept into a database. As the database grows larger and larger, more information are gathered and allow the analysis of the influence of features over performances of the system. However, this requires to select a configuration, derive the corresponding product and assess its performances over a benchmark. These are systematic approaches that are unlikely to scale to modern variability-intensive systems. Instead, the proximity between configurations should be exploited to predict the behavior of new configurations. The underlying assumption is that similar configurations (*i.e.,* presenting only a few differences in the activation of options) should behave similarly since most options are the same, thus, using similar pieces of code, *etc.*

With prediction models coming from machine learning, the promise is to avoid deriving and measuring all products but to use a small number of initial configurations and exploit similarities between configurations of this set and new configurations in order to infer performances of the new product. In this case, regression techniques are widely used as they are designed to predict continuous values.

[39, 79, 87, 88, 89, 108, 115] create a performance-influence model out of a few configurations in order to predict performances of systems that will be derived in the future. First of all, this model learns the influence of feature interactions on the performances of a system. Then, it exploits discovered influences and interactions to predict the performances of new configurations based on their activated features.

The effectiveness of statistical learning techniques and regression methods has been empirically studied. Siegmund *et al.* [87] combined machine-learning and sampling heuristics to compute the individual influences of configuration options and their interactions. The approach has applications in performance-bug detection or configu-

ration optimization. Because of these applications, the model needs to be understandable by human beings. Siegmund *et al.* [39, 79, 89] have focused on using CART (Classification And Regression Trees) in order to build models that can be easily interpreted. Produced models are binary trees which make a decision over a feature value at every level of the tree.

However, these works tackle a regression problem and try, in the end, to predict the exact performance (or at least a value as close as possible to the real one) of a system. Providing user-guidance in order to specialize a software product line does not need to predict the exact performance value of a configuration but rather to be close enough in order to help users assessing whether performances of a given configuration are good enough with respect to their requirements. In spite of being similar, these two problems are rather different since they aim for two different objectives.

3.3.2 Testing machine learning techniques

Machine learning techniques have a strong mathematical background which should ensure strong guarantees about convergence and the correspondence between the output of the technique and the expected result of the task at hand. Despite this background, empirical evaluation is performed following the process we have presented in Section 2.3.3.

Nonetheless, some efforts have been made in the world of Computer Vision [2] in order to bring some validation methods from the Software Testing community to this particular domain. Machine learning and deep learning based systems are rather different from "traditional" software systems. Indeed, results of computations are defined by data that are fed in entry of the process (*i.e.,* in the training phase) which affects the behavior of the system later. As an example, we can imagine that the same machine learning based system initially fed with two different data sets (following the same data distribution) will produce separating functions that differ in some aspects, in turn producing differences in the classification of some inputs and finally activating different parts of the code. Since the definition of separating functions is determined by the training set (and sometimes even by the order in which data are fed to the algorithm), the final behavior of the system is very hard to predict. Therefore, structural software testing

2. A domain initially trying to model and understand how human beings perceive, recognize, *etc.* objects surrounding them and thus relying a lot on machine learning techniques.

techniques are unusable as testing the underlying code would not tell anything about the behavior of the system.

Still, neural networks and deep learning become evermore used in computer vision. Recently, DeepXplore [73] proposed a measure similar to structural testing applied for deep learning algorithms. In this work, they adapt the underlying assumption of structural tests from code to the structure of neural networks such that it becomes : test suites should activate at least once every neuron of a network. DeepXplore remains a white-box testing approach that assumes to have access to internal details of the neural network under study, which might not always be the case. Based on this work, DeepTest [107] is a tool automatically building test cases that activate as many neurons as possible.

After that, Zhang *et al.* [114] proposed to use Generative Adversarial Networks to create test cases automatically. Generative adversarial networks are a recent trend in machine learning that breaks the fundamental assumption of machine learning stating that training and test sets must be drawn following the same distribution function. They validate their approach by using metamorphic testing presented in section 3.1.3.

3.4 Summary

This chapter overviewed different work conducted in research fields related to this thesis. We have discussed techniques to test software product lines and other software testing approaches before moving to the use of machine learning in the software product lines field and vice-versa.

It appears that performance evaluation of software systems remains a difficult task. However, this thesis tackles such problem. Long terms objective of this thesis would be to guide users in future configuration selection of a system.

Most of research efforts have been put in efficiently detecting and fixing bugs in software code while performances aspect are mostly left aside. For instance, testing a software product line consists in first testing that the variability model is correct *w.r.t.* the requirements of end-users. Meaning that, first, configurations are sampled before executing them to check that products are not buggy. No performance aspects are present in this process.

Choosing appropriate test cases is also a difficult problem. In particular, assessing the quality of test suites is a difficult problem due to the potential large number of test

cases to consider. And again, those test cases do not focus on performance aspects.

In the end, it is difficult to properly assess performances of products derived from a software product line and it is difficult to guide users in the selection of a product that is able to meet users' performance requirements.

Existing work on performance prediction are interesting but they target the selection of the configuration that optimizes a certain criterion without letting possibilities for users to choose a configuration among a set of potential interesting configurations. Because these works use machine learning prediction power, they do not assess that test suites used to evaluate performance criteria allow to observe a variety of different behaviors (*i.e.,* the quality of performance test suites are not evaluated).

The contributions of this thesis are addressing these two aspects. The first contribution provides a method to restrain the number of configurations that needs to be considered when trying to configure a product such that it meets specific goals. The second contribution evaluates the quality of performance test suites in terms of their ability to exhibit various behaviors of different products coming from a same software product line.

The next chapters present and detail these contributions.

CHAPITRE 4

AUTOMATIC SPECIALIZATION OF SOFTWARE PRODUCT LINES USING MACHINE LEARNING

4.1 Introduction

This work focuses on the generation of *program variants* that will populate the first dimension of the matrix presented in Figure 1.1. We remind that one goal of this matrix (from the products perspective) is to assess performances of program variants coming from a software product line. It helps to get an idea of the range of performances that can be reached by products. This is especially important when trying to choose a configuration that have to fit specific needs. In certain cases, for instance when requirements are too strong, users will have poor chances to find a configuration that can be derived into a product that meet requirements. Therefore, before trying to estimate the range of performances, it is smart to reduce the number of configurations such that available products have high probabilities to meet performance goals. Still after scoping configurations, the number of remaining configurations can remain too big to be manageable manually.

As presented in Section 3.3.1, a body of work is dedicated to predict performances of system variants of a software product line. However, these works tackle the problem of predicting accurately performance values for a given configuration of a software product line. Being more accurate : considering a configuration, performance prediction methods try to infer [1] performance values of the derived product associated with the configuration. Work discussed in Section 3.3.1 consider a problem (*e.g.,* optimizing a specific performance measure) allowing to return a single configuration that meets pre-

1. meaning that performances are not actually measured

defined requirements without taking into account that users might have unexpressed preferences (*e.g.,* requirements that are difficult to state. For instance, putting in relations different modules, requirements that are hard to check, *etc.*). In this situation, users fall back into a try-and-error process in which they will ask for configurations, check that preferences are reached, if not, selected configurations are added to the constrain set and thus forbidden to be selected again. To tackle this problem, we ask users (or any automated routine) to decide whether a configuration is interesting and should be kept in the set of available configurations.

Keeping the idea of predicting performances of configurations (without having to actually measures performances), we tackle the problem of specializing a software product line which restrains the space of valid configurations such that they meet a certain performance goal. Therefore, the space of actually *interesting* products for a specific goal (defined by users) can be seen as a subspace of the initial space of possible products given by a variability model. A typical way to go from the initial space to the desired subspace is to add constraints to the variability model further restricting the initial space down to the desired subspace. It can be equally seen as a way to constrain the acceptable inputs of the variables of an automatic product derivator.

As a result, we aim at restricting configurations of a software product line such that remaining valid configurations as defined by Def. 5 meet a certain goal defined by users (*e.g.,* an upper bound for execution time). Doing so, we aim to ease the selection of an interesting configuration. Nonetheless, due to the number of configurations to check, this kind of restriction cannot be done manually.

Except for the beginning of this section and the conclusion, the remaining of this chapter is mainly copied from our paper published at the Software Product Line Conference 2016 [102]. We changed the terminology of faulty (resp. non-faulty) configurations/products to not interesting (resp. interesting) configurations/products as it seems more appropriate.

The challenge addressed in this work is to automatically synthesize a set of constraints that would be both precise (allow all interesting configurations) and complete (never allow a configuration that do not respect performance goals). First, we assume that goals can be stated as a routine that can be called automatically. It can take the form of a function that tests configurations of a system and assesses that they can meet a performance goal. First, this routine is used on randomly generated products from the product line, keeping for each of them its resolution model (*i.e.,* its configuration). The

routine classifies those products into two categories (*i.e.,* interesting or not interesting). Based on the configurations and the decisions taken from the routine, we propose to use a machine learning approach to discover combinations of features and/or range of values producing a model that is able to categorize new configurations. Finally, based on the classification model, we build new cross-tree constraints and add them to the variability model in order to guide further generation of products.

We validate our approach on a product-line video generator developed in the industry [1, 34], using a simple computer vision algorithm as an oracle.

The rest of this part is organized as follows. Section 4.2 presents the overall principle of our approach and discusses the kind of product-lines it is suited for. Section 4.3 applies our method on an existing video generator. Section 4.4 presents the experimental results in terms of meaningfulness, precision and recall and discusses threats to validity. Section 4.5 indicates possible directions of improvements. Section 4.6 concludes with perspectives open by this work.

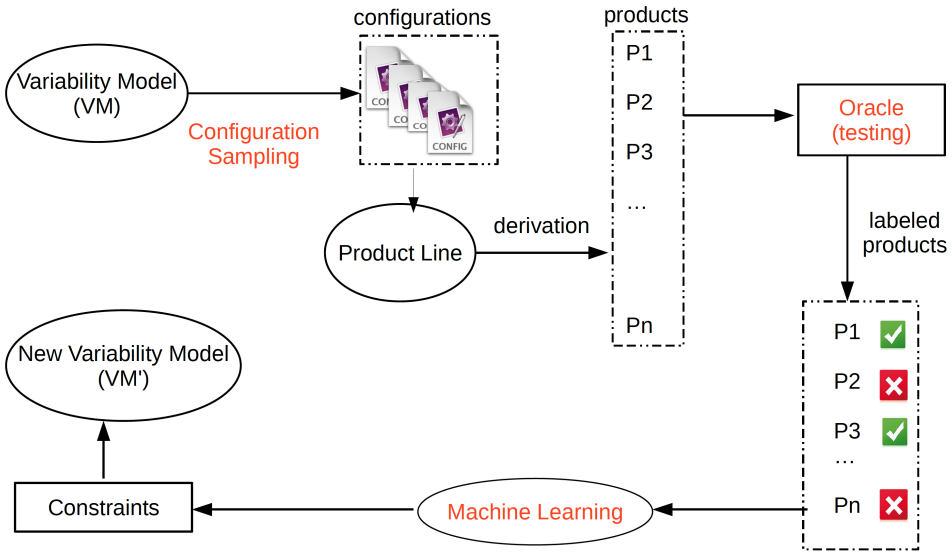


FIGURE 4.1 – Sampling, testing, learning : process for inferring constraints of product lines

4.2 Method

Figure 4.1 describes our approach. We assume that there is a variability model that documents the configuration options of the product line under consideration. In such variability model, options can be Boolean features or numerical attributes. From a configuration (see Def. 5), product line artifacts are assembled and parameters are set to derive a product. The variability model may already contain some *constraints* that restrict the possible values of options (*e.g.,* the inclusion of a Boolean feature implies the setting of a numerical value).

A first step in the process is *configuration sampling*. It consists in producing valid configurations (see Def. 5) of the original variability model VM. The set of sampled configurations is a subset of $[\![VM]\!]$. Numerous strategies can be considered such as the generation of T-wise configurations, random configurations, *etc.* [3, 22, 34, 42, 46, 56, 62, 79]

Second, an *oracle* is reused or developed. It tests the fact that a product (derived from a configuration) meets users' requirements. It may refer to the fact the product does compile, does not crash at run-time, passes the test suite, and/or does meet a particular quality of service (*e.g.,* compute under a fix amount of time). The previously mentioned oracle is used to create two classes of configurations. Labels are either

interesting or *not interesting* as defined in Def. 6.

Definition 6 (Oracle and interesting configurations) *An oracle is a **function** that takes a derived product as input and returns* true *or* false*. An **interesting** configuration is a configuration for which the oracle returns* true *when taking as input the corresponding derived product. Conversely, configuration are said to **not interesting** if the oracle returns* false*.*

A third step is to use a *machine learning* procedure that operates on the previously labeled configurations and automatically infers constraints. A new variability model VM' is created by adding the newly identified constraints to the original variability model. Therefore, the new variability model VM' is a specialization [105] of VM and $[\![VM']\!] \subset [\![VM]\!]$. In other words, VM' forbids not interesting configurations that were initially considered as interesting in VM.

Instead of using machine learning, an alternate and sound approach is to remove not "manually" interesting configurations from the original variability model. It consists in negating all not interesting configurations and then making their conjunctions (see Definition 7). However, this approach is very limited since (1) it only removes a typically small number of configurations – only those that have been sampled and tested ; (2) it does not identify which configuration options and values are involved and the root causes of the fault.

Definition 7 (Sound approach) *Given a set of not interesting configurations* $\{fc_1, fc_2, ..., fc_n\}$*, a sound approach computes a new variability model* VM' *such that* $VM' = VM \wedge \epsilon$ *where* $\epsilon = \bigwedge_{i=1..n} \neg fc_n$

Therefore, a simple removal of not interesting configurations is not a viable solution for product lines exhibiting a large number of configuration options or numerical values. As an oversimplified example, let say we have configurations $\{fc_1, fc_2, fc_3, \ldots, fc_n\}$ that are not interesting. Among values of attributes and features, the attribute A varies as follows : $A = 0.265$ in fc_1, $A = 0.26$ in fc_2, $A = 0.275$ in fc_3, ..., $A = 0.29$ in fc_n. With a basic case by case extraction, we cannot infer that (perhaps) A must not be in the range $[0.26; 0.3]$. The use of machine learning can improve the inference of constraints through the prediction of ranges of values that make configurations not interesting.

The expected benefit is to discard much more not interesting configurations with the inference of constraints : Figure 4.2 summarizes the potential of machine learning.

A rectangle is used to represent $[\![VM]\!]$. The set of configurations that can be detected as not interesting is represented as red clouds/rectangles in Figure 4.2. This set of not interesting configurations is a priori unknown (some configurations might be known but not all of them) which is precisely the problem.

Configurations detected by an oracle as not interesting are included in this set (see crosses in Figure 4.2). The enumeration and testing of configurations for covering the whole set of not interesting configurations would take a large amount of resources and time. In response, machine learning infers a set of constraints delimiting *sets* of not interesting configurations (instead of only forbidding individual configurations). Thanks to learning and prediction, we expect the capture of additional configurations that are not interesting *without the cost of testing those configurations*(*i.e.,* without actually generating variants and make them run against test suites).

In the ideal case, machine learning determines perfectly the contours of a set of not interesting configurations. That is, this sub-space will contain only configurations that are not interesting. We represented such behavior as the dashed rectangle exactly corresponding to the red rectangle (see left-hand side Figure 4.2). However, machine learning might produce false positives. That is, some configurations are classified as not interesting whereas they are actually valid from the oracle's perspective. An example is given in Figure 4.2 with the red cloud and dashed rectangle at the bottom : some configurations are included outside the red cloud and in the blank area. Another kind of misclassification can happen when the dashed rectangle is included in the red cloud (see right-hand side of Figure 4.2). In this case, machine learning failed to classify some configurations as not interesting. The set is incomplete. Despite specific cases in which machine learning can be unsound and incomplete, we expect that, in general, learning constraints enables to capture more not interesting configurations than a simple conjunction of negated configurations.

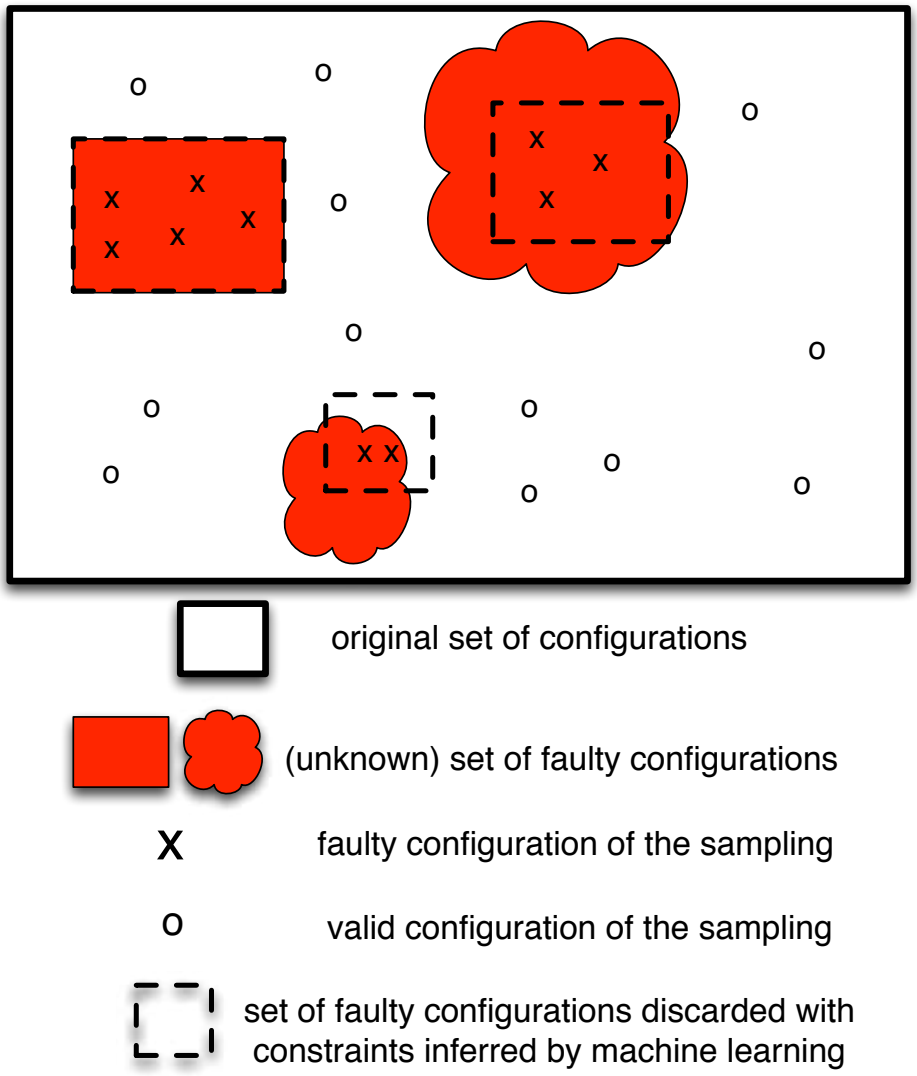


FIGURE 4.2 – Constraining the configuration space

4.3 Case Study

We apply and evaluate the proposed method with a real-world product line called MOTIV which is a highly-configurable video generator developed in the industry [1]. Our objective is to address the following research questions :
RQ1) *Do extracted constraints make sense for a computer vision expert ?*
RQ2) *What is the precision and recall of our learning approach ?*
RQ3) *What are the strengths and weaknesses of our approach compared to existing techniques ?*

4.3.1 Case and Problem

Given a configuration file, the MOTIV generator synthesizes a video variant with specific properties (luminance, trajectory of vehicles, noises, distractors, *etc.*). The software generator is written in Lua and implements numerous complex, parameterized transformations for synthesizing variants of videos [1, 34].

The motivation behind the generator is that the current practice for benchmarking tracking algorithms (*e.g.,* algorithms that track objects of interests in a scene) is limited to a *manual* collection of video sequences. Such manual effort is error-prone and time-consuming since it requires to (1) shoot video sequences in real environments and (2) annotate videos with a *"ground truth"*. As a result, benchmarks with limited size and diversity are usually employed. In response, this MOTIV generator has been developed to *automatically* produce a large and diverse set of video sequences.

The generator comes with a variability model (an excerpt is shown in Fig. 4.3 similar to the feature model presented in Fig. 2.2). The model organizes features and attributes in a hierarchical tree. Some features are Boolean (*i.e.,* included or not) while others, called attributes, are defined over continuous ranges of numerical values (see bottom of Figure 4.3). A given configuration is obtained by choosing a particular value for all these features and attributes. The software generator finally exploits the selected values to produce a video sequence corresponding to a configuration. Users can control the generation process to cover a large diversity of video variants and thus challenge tracking algorithms under different setups.

Problem. Users quickly noticed that the specification of constraints in the variability model is crucial for the video generator. Without constraints, many configurations lead

to the generation of unrealistic video variants, due to the incompatibility between features and attributes' values. Generating such kind of videos is an issue for two major reasons. First, the production of videos has a cost (about half an hour of computation on average per video variant). As a result, the synthesis of large datasets with thousands of video variants (as originally planned by industrials) would create a lot of irrelevant videos, thus wasting computation power as they would not be used. Second, running tracking algorithms on videos are computationally expensive. Forcing them to run on irrelevant videos (*e.g.,* with too much noise, or no illumination) would be, again, a waste of time and resources. Prior efforts [1, 34] were made to properly formalize the variability but were not sufficient. As a consequence, we need to enforce the generator with constraints to make it usable and useful for practitioners.

Although some basic constraints have been manually specified, the generator still produced irrelevant video variants. In order to capture additional constraints and gather some knowledge, we have made several iterations with the developers of the video generator through meetings and mail exchanges. Finally, we came to the conclusion that an analysis of the Lua source code or further efforts to manually specify constraints present strong limitations. It is mainly due to the fact that (1) the configuration space is extremely large (see hereafter for more details) ; (2) there is not enough knowledge to comprehensively capture constraints over features and attributes' values.

A manual exploration of the configuration space requires substantial efforts for both setting configuration values, assessing the quality of the output (videos) and learning from defects. We thus propose to use the method we described in the previous section to automate all these tasks, including the inference of constraints via machine learning.

We now detail how we instantiated every part of our method (sampling, testing, learning) within our case study.

4.3.2 Solution for Inferring Constraints

Figure 4.4 presents the entire process of extracting constraints of the video generator. First, a set of configuration files is sampled from the variability model ; the Lua generator derives a video variant for each configuration. This set of video variants acts as a training set for the machine learning technique to come. An oracle is then used to label these videos as interesting or not by computing the quality of each video.

As explained before, the goal is to test tracking algorithms, thus, a sufficient quality (*i.e.,* for which objects can be distinguished) results in the label "interesting" and "not interesting" otherwise. Finally, a machine learning process is executed to extract the constraints and re-inject them into the original variability model. We now detail each step of the process.

4.3.3 Generating a training set out of the variability model

In the MOTIV case study, the variability model exhibits numerous features and attributes whose range of values are reals and continuous. Figure 4.3 presents an excerpt of its hierarchical structure and possible values for features and attributes. In total, the variability model contains : 42 real attributes, 46 integer attributes, 20 Boolean features, and 140 constraints (mainly constraints specified by the experts). Ranges vary between 0 and 6 for the integers domains, and in average between 0.0 and 27.64 for the real domains with a precision of 10^{-5}. This would end up in approximately a total of 10^{103} configurations (not considering constraints) : 2^{20} (because of the boolean variables) $\times 6^{46}$ (because of the integer variables and) $\times 2764000^{42}$ (because of the real variables). Overall the variability model presents the particularities of encoding a large configuration set with lots of non-Boolean values.

The nature of this model has encouraged us to develop a new version of the FAMILIAR tool suite [2, 34]. We implemented a solution capable of natively coping with real attributes which rely on the Ibea solver [2] provided with Choco 3 [76].

To generate a training set for the machine learning process, we need to produce a set of valid configurations (gathering both interesting and not interesting products). Different sampling techniques [3, 22, 34, 42, 46, 56, 62, 79]can be considered but some of them are not applicable to our case since we have to deal with real and integer values. We implemented the following procedure. First we randomly pick a value for each attribute within the boundaries of its domain. Then, we propagate the attribute values to other values with a solver ; the goal is to avoid invalid configurations (see Def. 5). We continue until having a complete and valid configuration. We reiterate the process for collecting a sample of configurations.

2. `http://www.ibex-lib.org/`

4.3.4 Oracle

Some videos of the generator are not interesting for computer vision algorithms and humans. Typically, these are videos in which the vision system cannot perceive anything or cannot distinguish moving objects that should be tracked from other that should not be (*e.g.,* distractors). Image quality assessment tries to understand the conditions under which the vision system is likely to fail this kind of distinction. We implemented an image quality assessment oracle, presented in [28], that can automatically assess whether a video is faulty. The principle is to perform a Fourier transformation and to reason about the resulting distribution of Fourier frequencies. Such a technique evaluates the quality of a single image while we produce entire videos. To avoid calling the oracle on every frame of a video and to save time, we sample a video into a smaller set of images (*i.e.,* taking one frame regularly out of the video stream). After applying the image quality assessment method on sampled images, we aggregate results to decide whether a video is interesting or not. We empirically set a threshold : if at least half of the images are considered not interesting, then the whole video sequence is also considered not interesting.

4.3.5 Machine learning

Using our oracle, we assigned a label (*i.e.,* interesting or not interesting) to every video of the sample. We use a *machine learning* algorithm to understand the relationship between videos being interesting and features/attributes' values. Different machine learning methods exist. Some of them are sophisticated and perform only a few classification errors while others are less advanced but are much more understandable when visualizing the output model. In our case, we wanted to obtain a high level of understandability when extracting constraints. Specifically, we employed *Binary Decision Trees*.

Figure 4.5 presents an excerpt of the decision tree obtained from the Weka [3] software. In this tree :

— *ovals* represent features (written inside) on which decisions will have to be taken ;

— *labels over edges* represent threshold value to decide which path to take ;

3. http ://www.cs.waikato.ac.nz/ml/weka/

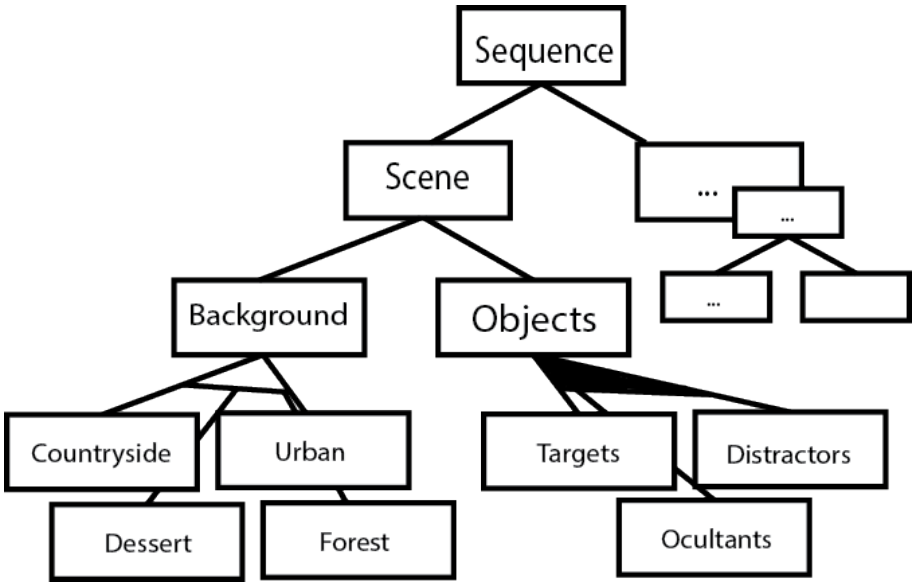
— *rectangles* represent leaves of the tree and groups of configurations that are mainly of the same class.

Leaves present several pieces of information. First, there is the label of the most represented class. In our case it is either ’1’ (not interesting) or ’0’ (interesting). Second, the number of configurations associated to the label. Finally, the number of configurations of the other class. As previously discussed in Section 2.3, due to the nature of machine learning algorithms, classification errors can be made (even at training time). This is the case, for instance in this figure, considering the top right corner leaf in which one configuration associated with the label "not interesting" have been grouped with 110 configurations associated with the label "interesting". These are classification errors, typically configurations that are at the boundary of two classes.

4.3.6 Extracting constraints

Once the decision model has been created, we traverse the tree and reach every leaf. If a leaf is labeled ’1’, its path is extracted and remembered. We consider that a path is a set of decisions, where each decision has to be taken regarding the value of a single feature. We create new constraints by building the negation of the conjunction of the different decisions to make along the path to reach a leaf labeled not interesting. If features appear repeatedly (in different constraints for instance), some simplifications are performed. In the example of Figure 4.5, we can extract the two following simplified constraints :

```
!(signal_quality.luminance_dev > 1.01561 && signal_quality.
   luminance_dev <= 18.1437)
!(signal_quality.luminance_dev <= 21.3521 && signal_quality.
   luminance_dev > 18.1437 && capture.local_light_change_level
   <= 0.481449)
```




```
//Distractors
real distractors.butterfly_level [0.0 .. 1.0]
real distractors.bird_level [0.0 .. 1.0]
real distractors.far_moving_vegetation [0.0 .. 1.0]
real distractors.close_moving_vegetation [0.0 .. 1.0]
real distractors.light_reflection [0.0 .. 1.0]
real distractors.blinking_light [0.0 .. 1.0]
//Capturing conditions
real camera.vibration [0.0 .. 1.0]
real camera.focal_change [0.0 .. 1.0]
real camera.pan_motion [0.0 .. 1.0]
real camera.tilt_motion [0.0 .. 1.0]
real camera.altitude [0.0 .. 5.0]
real capture.illumination_level [0.0 .. 1.0]
// Signal quality
int signal_quality.force_balance [0 .. 1]
real signal_quality.luminance_mean [0.0 .. 255.0]
real signal_quality.luminance_dev [0.0 .. 255.0]
```

FIGURE 4.3 – Variability model excerpt of the generator

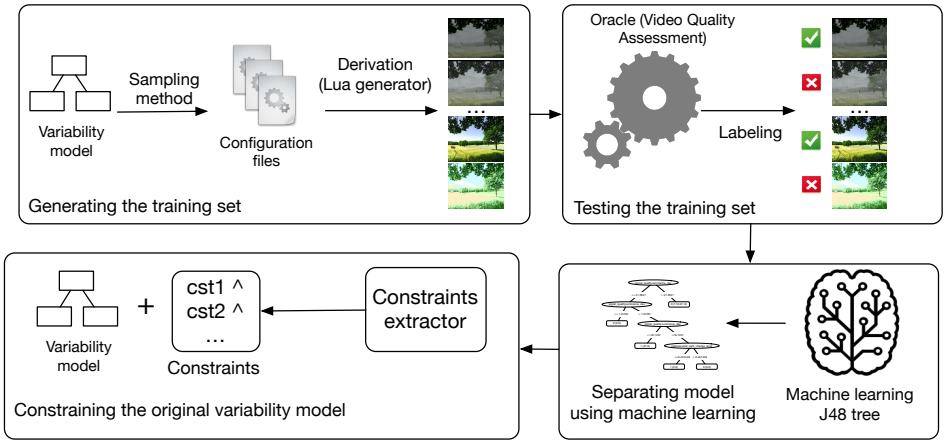


FIGURE 4.4 – Learning method on the video generator

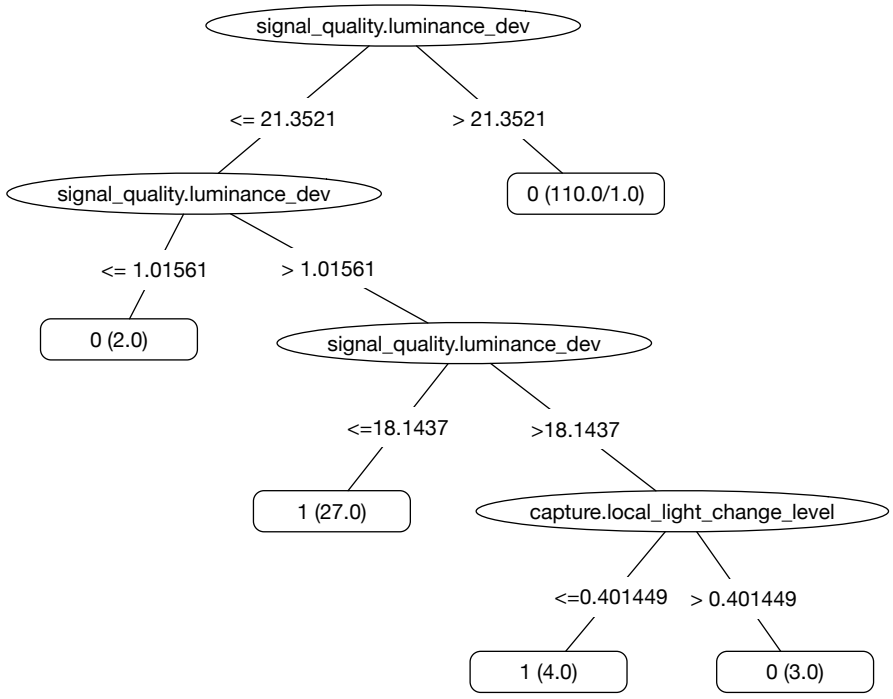


FIGURE 4.5 – An excerpt of the decision tree built from a sample of 500 configurations/-videos

4.4 Experiments

4.4.1 Experimental Setup

To generate a training set, we sampled 500 configuration files from the MOTIV variability model. All of them are given to the video generator to create 20 seconds long videos. The process of deriving associated video variants takes about 30 minutes in average per video. As generating all those videos require time, we used a cloud-based architecture for distributing the computations. To decrease the influence of randomly creating training set (which could result in advantageous or disadvantageous settings), we run the experiment of learning and validating results 20 times (see Section 4.4.2 for more details). After the synthesis of videos, the oracle we presented in Section 4.3.4 assigned "interesting" or "not interesting" labels to videos. In total, the oracle labeled 53 videos as not interesting on average, *i.e.,* roughly 10% of the videos.

Regarding the implementation of the machine learning algorithm, we used the Weka framework. Weka offers different implementations of binary decision trees (and other various machine learning techniques). We used J48 since it is the most widely used. Various options can be tuned to increase the classification performances. This process of selecting the best set of parameters is application-dependent, so we used the default ones proposed by Weka.

In order to facilitate reproducing experiments, all experimentation data are available at `http://learningconstraints.github.io`.

4.4.2 Results

RQ1) Do extracted constraints make sense for a computer vision expert ?

The first research question focuses on the readability and comprehensibility of extracted constraints from an expert point of view. To be able to answer this question, we have asked a computer vision expert and advanced user of the video generator whether the extracted constraints did make sense or not. The expert told us that constraints are globally understandable and actually help understanding interactions that can occur between features/attributes. Importantly, he noted that constraints are in line with the definition of the oracle : Some combinations of values can indeed participate to the degradation of the perception of video contents.

```
!(signal_quality.blur_level > 0 && signal_quality.
   static_noise_level <=0.135519)
```

(a)

```
!(signal_quality.blur_level < 0 && signal_quality.
   compression_artefact_level > 0.363438 &&  capture.
   illumination_level <= 0.609669 && signal_quality.
   compression_artefact_level>0.436673 &&  vehicle5.trajectory
   >6 && vehicle1.identifier <=11)
```

(b)

```
!(signal_quality.blur_level <= 0 && signal_quality.
   compression_artefact_level <= 0.363438 && signal_quality.
   dynamic_noise_level<=0.428141 && signal_quality.
   force_balance=0 && capture.illumination_level <= 0.12151)
```

(c)

FIGURE 4.6 – three constraint extracted from our case study

Specifically, Figure 4.6a shows a short constraint only constituted by two terms. This constraint puts together two images quality criterion (blur and static noise) that can indeed degrade the quality of videos. In Figure 4.6b, the constraint involves other features that have an effect on the quality of videos : compression and illumination. Interestingly, blur is present again. In Figure 4.6c, blur, compression, and dynamic noise are features that are related to quality criterion of videos as well. Overall, all features/attributes previously mentioned make sense with regards to the oracle we implemented (see Section 4.3.4). Too much noise, poor illumination and blurs : all these factors can indeed degrade the quality of videos and produce the videos that our oracle detects as not interesting.

In general, extracted constraints make sense and the answer to RQ1 is positive. However there is room for improvement (see also Section 4.5). Specifically the expert complains about the presence of features/attributes that are not relevant and disturb the reading. For instance, in Figure 4.6b, $vehicle1.identifier <= 11$ does not make much sense. Indeed, the kind of vehicles should not have any influence on the definition of interesting videos. The expert has to somehow ignore this kind of information.

RQ2) What is the precision and recall of our machine learning approach ?

To answer this question, we consider that the constraints found by the decision tree are added to the variability model, thus, resulting in a new variability model denoted VM' such that $VM' = VM \wedge \Delta$ where Δ is a conjunction of inferred constraints. Then, we used the performance evaluation process described in Section 2.3.

The evaluation protocol is decomposed into two steps. We first evaluate classification performances on the training set to ensure that 500 configurations are enough to learn a classification model that is sufficiently good. In this context, we expect at least performances to be higher than 50% (equivalent to random assignments) but the higher, the better. Second, we perform an evaluation on the prediction performances of the model (and thus VM') to ensure that learnt constraints do not overfit the training set.

The overall classification performance of machine learning is not perfect, *i.e.,* 93.6% on average after performing a cross-validation with the training set on 500 configurations/videos. Since the resulting tree is not large (*i.e.,* containing only 8 leaves with a maximum depths of 4 meaning that a maximum of 4 decisions have to be taken before reaching a leaf), it is unlikely to overfit the training set. Errors might rather come from the fact that a proper separation is hard to find between videos of "good" and "bad " quality using our oracle. In this case, some configurations might be similar (*e.g.,* with only one feature value differing) but should be assigned to different labels which make the problem of classification of machine learning techniques harder and the result error-prone.

Now, to perform the second part of the evaluation (*i.e.,* evaluate the prediction performances of the solution), we generated another set of 4000 configurations and videos. We used again a cloud-based infrastructure to synthesize these variants. Our oracle labeled every video of this new data set. It resulted in 370 not interesting videos on average. Then, we compared the decision of the oracle with the decision of the variability model augmented with extracted constraints. We run the experiment 20 times by randomly picking configurations out of the 4500 [4] we generated in order to establish the training set, remaining configurations constituting the test set.

With this experiment, we are interested to know whether a configuration labeled as not interesting by the oracle is now forbidden by VM'. This comparison between

4. $4000 + 500$ from the first part of the evaluation

		Oracle	
		Not interesting	Interesting
variability model (VM')	Not interesting (invalid)	234 (± 57.899)	69.5 (± 26.973)
	Interesting (valid)	141.1 (± 60.440)	3566.2 (± 25.804)

TABLE 4.1 – Confusion matrix of our experiment

the two decisions is usually performed through a confusion matrix. Our results are presented in Table 4.1.

In this table, columns are the labels given by the oracle and rows are labels given by our variability model. Cells present the average of classification over 20 runs as well as standard deviation (under brackets). The main diagonal of this matrix tells us where the two labels agree. The other diagonal provides classification errors of our variability model compared to the oracle :

False Positives (FP) are configurations considered as "not interesting"[5] in VM' whereas they are classified as interesting by the oracle. The machine learning approach has inferred too restrictive constraints that now uselessly forbid "interesting" configurations.

False Negatives (FN) are "interesting" configurations in VM' whereas they are classified as "not interesting" by the oracle. The machine learning approach fails to infer constraints that could have forbidden "not interesting" configurations.

Precision is the measurement assessing the number of correct classifications performed for a label (main diagonal) regarding the total number of classification made for this label (sum of a row). Over the 20 runs, the mean precision for the label "interesting" is : $P_{mean-interesting} = \frac{3566.2}{3566.2+141.1} \simeq 0.96$. Similarly, precision for the label "not interesting" is $P_{mean-not-itneresting} \simeq 0.77$.

The overall precision is the mean of the two values : $P_{overall} = \frac{0.96+0.77}{2} = 0.865$, *i.e.,* the classification will roughly perform well 9 times out of 10.

Recall is the measurement assessing the number of correct classification performed for a label regarding the total number of configurations declared by the oracle for this label (sum of a column). Similarly to the precision, recall can be computed for

5. Strictly speaking, configurations are invalid (resp. valid) in VM'. We use the term "not interesting" (resp. "interesting") for keeping an unified terminology with the oracle.

each label and then combined into an overall measure. This gives : $R_{mean-interesting} = \frac{3566.2}{3566.2+69.5} \simeq 0.98$; $R_{mean-not-interesting} \simeq 0.62$ and $R_{overall} = \frac{0.98+0.62}{2} = 0.80$. Here, recall is lower for the "not interesting" label which gives us an idea of how difficult it is to understand the distribution of this class. This difficulty can come from the fact that there are fewer examples assigned with the label "not interesting" than with the label "interesting".

RQ3) What are the strengths and weaknesses of our approach ?

We now compare the properties of a sound and a machine learning approach in line with results of RQ1 and RQ2. We recall that a sound approach (see Def. 7) takes the output of the oracle and built constraints out of "not interesting" configurations/videos. It means that constraints will be very specific to the configurations given to the oracle, involving every feature and attributes with their values.

Meaningfulness of constraints. A consequence is that constraints are typically difficult to read with a case-by-case extraction. A sound approach would have produced the conjunction of 53 constraints, each constraint being constituted by the number of features/attributes' values of a configuration. As a configuration corresponds to 80+ values in our case, experts would have severe difficulties to review and understand what are the precise features and attributes involved in the fault. Our proposed approach computed 5 constraints on average with only a few features/attributes. This drastic reduction in size helps a video expert to better understand the constraints.

One can argue that techniques for reducing constraints (*e.g.,* [110]) can be adapted to numerical values and perhaps improve a sound approach. However, since the configurations do not necessarily share common values, especially in continuous domains, adaptations would not be straightforward. In fact, there is a more fundamental issue : constraints of a sound approach are so precise they cannot be able to capture other not interesting configurations even in their close neighborhood. Hence, the value of a machine learning approach resides in the ability to produce more general and thus meaningful constraints. The model deciding which label to assign to previously unseen configurations is an approximation of the real nature of data. It can be seen as a convex hull in the space of configuration. However, machine learning allow the hull to be complex and capture asperities to reduce the number of classification errors provided that the model can equally express this complexity.

Precision and recall. The strict strategy of a sound approach has another practical

implication. In our case, the sound approach would only remove 53 configurations out of 500 (no more no less). On the other hand, our machine learning method removes 234 more configurations than this first approach (see top left cell in Table 4.1). Indeed, when validating our classification models with 4000 new configurations, the machine learning model was able to recognize 234 (as a mean over 20 runs according to Table 4.1) configurations as not interesting – without having to produce and test the video variants. The sound approach was unable to detect them because these 234 configurations were simply not in the original sample. Hence, the prediction of not interesting configurations with machine learning gives a factor improvement of $(48 + 234)/53 = 5,3.$ 48 referring to the number of videos classified as not interesting by the machine learning model during the training process. 234 corresponds to the average number of videos classified as not interesting when using the same model on 4000 new configurations. Finally, 53 is the number of not interesting videos that have been actually produced and fed to the oracle which corresponds to what would be done by a sound approach.

What we try to highlight here is that even if initially (during the training phase), both approaches seem equal (the sound approach performs even better), the real additional value of using machine learning techniques resides in the exploitation of the output model which allows to classify configurations that were never encountered before and at almost no cost (in this case, going through the tree which is very quick to the low number of decision to perform on a very low number of features) without ever having to generate associated video sequences.

In order to scale (*i.e.,* capture a similar set of not interesting configurations than a machine learning approach), a sound approach has to sample a significant number of additional configurations. In our case study, there are two major drawbacks. First, covering a large percentage of the configuration space is simply not possible, mainly due to numerical values. Second, the cost of testing a configuration is very high : half an hour to generate a video and a few seconds to process it with the oracle. Concretely, the cost in time and resources for 4000 configurations is $4000 * 30 = 12000 \approx 2000$ hours for one machine. Hence, the use of a sound approach can be very costly since we envision to generate even more videos in the future. Overall, the major strength of machine learning resides in its ability to reduce the testing cost through the prediction of not interesting configurations.

The prediction capability of machine learning is also a weakness since it induces some errors. In our case, out of 4000 (see Table 4.1), in average 141 configurations were

classified as interesting by the learnt model (despite being actually not interesting). A sound approach is also unable to forbid such configurations in the first place since they are not part of the sample. That is, a sound approach would have suffered from similar issues. Finally, in average, 69.5 configurations out of 4000 were classified by machine learning as not interesting (despite being actually interesting). In this case, a sound approach would have kept these configurations and, thus, is superior to a machine learning technique.

As a summary there is a trade-off to find. On the one hand, the ability of machine learning to be one step ahead can reduce testing effort, produce meaningful constraints, and forbid more configurations that are not interesting (as in our case study). On the other hand, a sound approach can be of interest when product line practitioners do not want to unnecessarily lose some configurations.

4.4.3 Threats to validity

External validity. There are conditions that limit our ability to generalize the results. A first threat is that we only used one product line. Thus, the results of our experiment might not be extensible to other product lines. We selected an industrial product line that has been used for more than two years now and that has passed several testing phases. We consider the variability model as realistic since numerous experts were involved in its design and there is a concrete connection with a product line. The product line generates data (videos) that differ from more traditional artifacts like code or models. However the implementation follows general principles of product lines with a variability model and a software derivation engine.

The proposed method relies on supervised machine learning technique which assumes that oracles are available. In our case we have to develop a specific oracle based a computer vision algorithm. In other domains, oracles might be harder to develop and only able to catch a few configurations that are not interesting. On the other hand, traditional oracles relying on compilers or test suites can be used as well.

Internal validity. Since our earliest efforts [1, 34], the variability model has been subject to several iterations mainly due to the evolution of configuration files handled by the Lua code. Before applying the method, we considered the model as stable. Yet the model may exhibit errors (*e.g.,* wrong domains for the attributes). As part of our testing experiments, we did not observe such inconsistency of configurations' values.

In fact, it could be the role of our method to detect such errors (if any).

A threat concerns the sampling of valid and not interesting configurations. We trained our machine learning algorithm with 500 videos and then verified the extracted constraints in front of 4000 videos. To enhance internal validity, we run the experiment of learning and validating results 20 times.

One could argue that it could have been better to compare inferred constraints with existing ones. However we do not have any ground truth to rely on. We rather place ourselves in a realistic situation : inferring constraints not already specified or simply a priori unknown by developers and experts. Thus, we measure whether added constraints help to narrow the space of possible configuration *w.r.t.* a set of valid products (*i.e.,* in our case : video variants that are not blurry nor noisy). On the qualitative side, we also seek to understand whether constraints make sense for a video expert.

Another threat is that the oracle can be badly implemented, leading to incorrect judgment of the quality of some videos. First it should be noted that the machine learning method we used does not seek to undermine what 'not interesting' means ; we rather learn from a set of authoritative decisions imposed by the oracle. Hence the precision and recall of our machine learning method (see RQ2) is not affected. However an incorrect oracle can be problematic for the expert point of view and induces a threat to RQ1. When implementing the oracle, we review a sample of dozens of interesting and not interesting videos. Yet we cannot review all videos and there is a risk that the oracle is still not correct. What is reassuring is that the expert considered that constraints are in line with the oracle, hence giving confidence on the quality of the oracle. In general a co-design of oracle and constraints seems needed in case there are some uncertainties in the oracle's implementation.

4.5 Discussions

Based on our experience, we now highlight several points that could improve the results of our general method.

Sampling techniques. In our case study, we used a fairly simple sampling technique that relies on randomly picking values of features/attributes. There are other sampling techniques and strategies to address various kinds of problems. In validation & verification, for instance, T-wise sampling is used to produce interactions between T activated features [34, 42, 62]. The assumption is that a T-wise criterion can increase the ability to detect faults. On the other hand, our basic strategy allows us to capture some diversity and to explore different areas in the configuration space. Sophisticated methods for *uniform* generation of configurations [19] can even be considered, but deserve to scale for our cases in which we have numerous numerical options.

Another possible direction is to use expert knowledge to eventually guide the sampling. The sampling can be done iteratively as well.

In general, other sampling methods could allow getting fewer configurations while having a good configuration space coverage and a good intuitions over the classes' distribution functions. Ensuring such properties can, in the end, increase the classification performance of the machine learning algorithm.

Definition of oracles. We reused a method proposed in [28] to assess the quality of images in terms of distribution's frequencies in the Fourier domain. In our case, we used the method on a sample of images constituting a video sequence. The oracle is a non-trivial software procedure that may fail to detect some videos as not interesting. One way to improve our oracle is to use humans (typically computer vision experts) for better reasoning about perceptual details. The counterpart is that reviewing videos can be time-consuming and even error-prone due to fatigue. Our oracle has the advantage of being an automated procedure so that we can consider the analysis of much more videos. A possible solution is to combine different oracles for mitigating the limits of some oracles. As a summary, the choice of an oracle depends on (1) the quality of its judgment and (2) its cost. Both factors can have an impact on our learning phase and perhaps be in conflict (*e.g.,* good quality but very high cost). It should be noted, however, that the selection of an oracle can be much simpler in other settings (*i.e.,* there is no trade-off to find as oracle's definition might be trivial). For instance, a compiler is usually a fast and reliable procedure that can serve as an oracle for testing family of

programs.

Machine learning algorithm.

One could wonder : why using decision trees and not other techniques ? Or even, can other machine learning algorithms do better regarding classification performances ? We chose to use decision trees as we knew the output would be very simple to understand compared to other techniques such as artificial neural network or support vector machines. Moreover, decision trees are built according to the following rule : features exposing more information entropy should be placed in the higher levels of the tree which ease the readability and understandability of extracted constraints. Nonetheless, using other machine learning techniques might be worth since they could be more discriminating. It would result in retrieving more constraints with higher precision (*i.e.,* without introducing errors in further prediction).

Although, we reported that our decision tree makes classification errors ; in our case, miss-classifying interesting configurations as not interesting (false positives) is not that problematic since we can produce other videos. We are aware that, in other contexts, this kind of errors can be a problem. Overall different machine learning strategies can be employed to either increase precision or recall.

Readability of constraints.

Computer vision experts highlighted the fact that some features appear in constraints whereas they should not be discriminant in the decision of interesting/not interesting configurations. For instance, the feature "vehicle1.identifier" in the constraint of Figure 4.6b. This behavior clearly shows a need to reduce the numbers of features taken into account when generating constraints. By lowering the number of features in the representation of data, the impact of the so-called "curse of dimensionality" will be reduced and thus, it could help building more concise and meaningful constraints. Domain knowledge can be explicitly employed for removing unnecessary features. The sampling of additional configurations is yet another possibility to further refine constraints. From this respect, an expert can incrementally guide the sampling strategy, based on her understanding of constraints.

4.6 Conclusion

We addressed the problem of inferring constraints for a large, complex variability model in order to restrict the space of possible configurations. We proposed a method based on sampling, testing, and machine learning to identify which combinations of features/attributes (and their values) are likely to produce products that are not interesting. From the model produced by machine learning, constraints can be automatically extracted and injected into the variability model, typically to enforce a product line.

We instantiated our method on a video generator developed in the industry that originally produced irrelevant videos. Results showed that our method can classify with a good precision and recall products (videos) that were never derived and tested before. Furthermore extracted constraints express some interactions between features while remaining readable and general enough. Thanks to constraints, we can now consider the synthesis of very large datasets of truly relevant videos.

We believe the method is general enough to be applicable to product lines in other contexts and domains. In the following chapter, we address this perspective by applying this technique to both new systems and considering new applications in which the classification is defined based on observed performance measures.

CHAPITRE 5

LEARNING-BASED PERFORMANCE SPECIALIZATION OF CONFIGURABLE SYSTEMS

5.1 Introduction

In this work, we extend the previous chapter and propose a generalization of the specialization method. To this end, the new approach takes as input a performance objective that has to be turned into an oracle function in the process. We evaluate it on 16 different systems that present different characteristics (*e.g.,* size, nature of features, observed performances).

In addition to these additional results, we introduce the notion of *safety* and *flexibility* in the specialization process which might help assessing the adequacy of computed prediction models with users' performance requirements (represented by the oracle function).

Even if we present this work as an extension of the previous one, it is part of the main contributions of this thesis. The inclusion of human requirements (both fuzzy and exact) to guide the specialization of a product line is novel as well as discussions provided about the fact that traditional performance measures in machine learning are not enough in our context significantly differentiate this work from the previous chapter.

This chapter is inspired from our technical report [101]. The report is available to everyone and provides a more detailed view of this work, including discussions on different points that we think being important.

This chapter is organized as follows : first we will present the motivation and problem statement of this work in Section 5.2 which are rather similar to the ones from the previous chapter but emphasis differences. Then, in Section 5.3, we discuss two im-

portant points : the impact of the performance objective given by users on the learning process and the inadequacy of precision and recall, the two well-known measures to evaluate the capability of the machine learning algorithms to classify correctly data, to assess that the specialized system is suitable for users in our context and the need to find other measures. Section 5.4 provides experimental setup and global conclusions that we find when conducting our experiments before concluding in Section 5.5.

5.2 Motivation and Problem Statement

Similarly to Chapter 4, we start from the observation that the configuration space of configurable systems is huge and finding a configuration being both valid and meet specific requirements is time-consuming, energy demanding and usually takes place in a try-and-error process. If users are able to declare their expectations in terms of performance properties (*e.g.,* running under X seconds), then it should be easy to take this piece of information into account and incorporate it in our previous framework. The difficulty here is that performances are usually defined in continuous domains while our solution considers a binary classification problem. The role of the oracle function is now to take as input the performance objective defined by users and incorporate it in a function such that the result is boolean allowing, in the end, to come back to a binary classification problem.

5.2.1 Motivating scenario

Let us consider the x264 system. It is a tool to encode and transcode videos into a specific format (usually H.264). In the world of video encoding, the quality of the video is related to the notion of size and time needed to send the video to a different computer (which is at the heart of todays videos on demand services). To adapt to different devices with different computational power and network bandwidth, x264 offers configuration options such as output size, frames per second or encoding heuristics (*e.g.,* parallel encoding on multiple processors or encoding using specific data structure to minimize introduced errors). Users may configure x264 via command-line parameters or through a graphical user interface. Figure 5.1 shows an excerpt of options : no_mbtree, qp, no_asm, and no_cabac are Boolean options (true or false values) and crfRatio, ipratio, and b_bias are numerical values with different ranges.

The mix between Boolean features and numerical values makes the possible number of configurations very high (*i.e.,* 10^{27} configurations) similarly to the MOTIV video generator used in Chapter 4. Even if x264 provides a comprehensive documentation of options [1], the description is in natural language and does not necessary capture all interactions between options. Nonetheless, these interactions can have severe impacts on performances (such as execution time or the quality of the video in the output).

1. `http://www.chaneru.com/Roku/HLS/X264_Settings.htm`

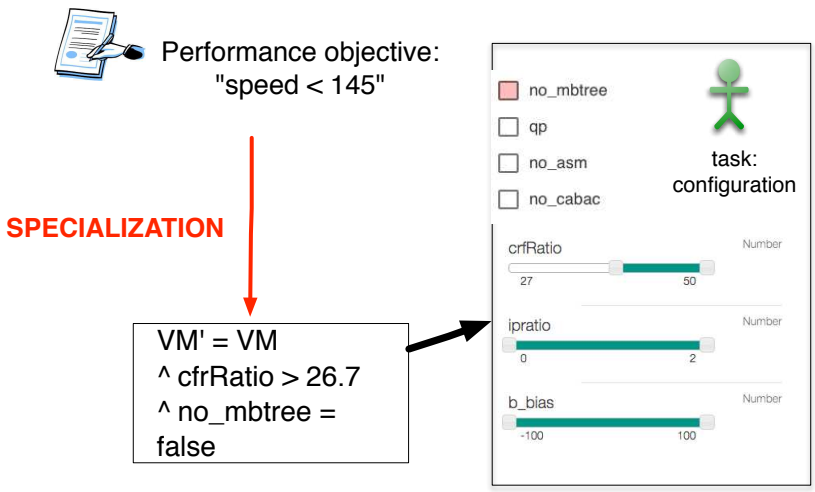


FIGURE 5.1 – Configuration of a specialized x264. Given a performance objective, the specialization method infers and fixes some options values (no_mbtree and crfRatio) ; users still have some flexibility to configure other options.

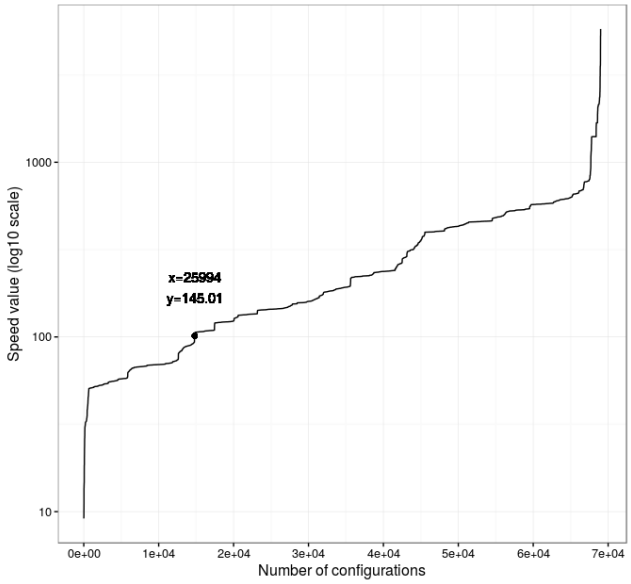


FIGURE 5.2 – Number of x264 configurations running under a certain time : X-axis represents a number of configurations ; Y-axis represents the execution speed (in seconds) to encode a video benchmark ; *e.g.,* about 25994 configurations can encode the video in less than 145.01 seconds.

Figure 5.2 shows the distribution of the execution time *w.r.t.* configurations. Please note that, as we said, x264 can provide about 10^{27} configurations but only about 70k are presented on the figure. These configurations come from other studies [39, 79, 87, 88] which have sampled a subset of the configuration space to conduct their work.

5.2.2 Approach

Following the same approach as in the previous chapter, we want to assist users in reaching configurations that meet both functional requirements and performance requirements. To do so, we want to both : (1) *restrict* the space of valid configurations (*w.r.t.* the Variability Model) ; (2) let users *decide* among a set of *interesting* configurations (as defined in Chapter 4 via Def. 5). In Figure 5.1, x264 is specialized in such a way that valid configurations has an execution time less than 145 seconds.

Figure 5.2 shows that a large portion of sampled configurations are above this threshold, meaning that users want to configure x264 such that it is rather fast to execute. In the meantime, we do not want to impose users a unique configuration but rather we want to let them choose among the set of configurations that meet their requirements (in this case, we want to let them choose among the $\approx 26,000$ configurations).

Being a little more specific, once the threshold value of the performance objective is defined, the system should be specialized such that valid configurations have high probabilities to meet this objective. For instance, in Figure 5.1, the system has been specialized such that the feature *no_mbtree* is deselected and the feature *crfRatio* can only take values higher than 26.7 which is in adequacy with the documentation stating that the quality of video tends to be lower as *crfRatio* takes higher values. As the quality is lower, it takes less time to encode and thus these configurations can run faster than others.

Again, we rely on the same process than given before and that is reminded in Figure 5.3. First, users define their objectives which will be turned into an automatic procedure that will serve as an oracle to decide whether configurations are interesting or not. Note that, as we presented until now, the objective is in fact a threshold regarding a performance value that acts as an upper or lower bound depending on the application (*i.e.,* we talked about the execution time which is usually an upper bound, but regarding object detection algorithms, users could define a lower bound to reach regarding the precision of the algorithm to retrieve objects of interest). Then, configu-

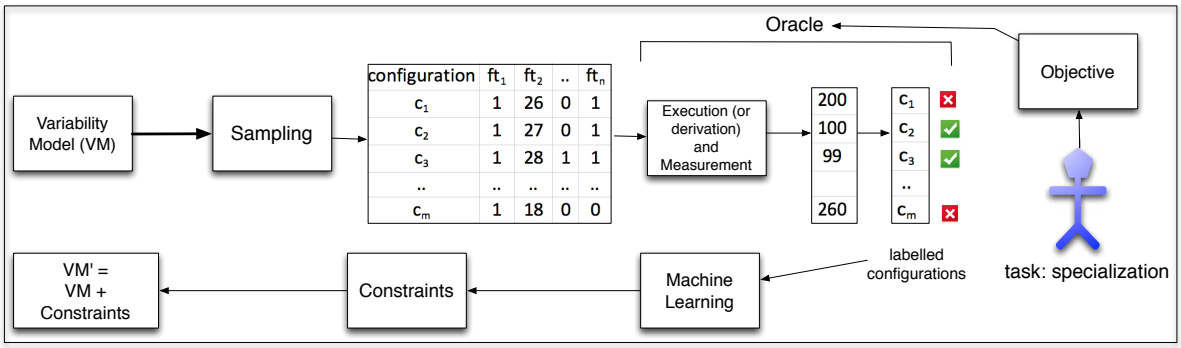


FIGURE 5.3 – Sampling, measuring, learning : given a performance objective, there is an automated method for specializing a configurable system

rations are sampled and the oracle is applied on each of them in order to create two classes. And the learning process begins, allowing the extraction of constraints that will be injected into the original variability model.

The challenge in this situation remains to be able to learn a classification model that is as close as possible from the real one [2] but considering only a small portions of the configurations.

5.2.3 Novel problems

In Chapter 4, we have used an oracle defined as an approximation of the quality of a video sequence. As it is an approximation, it may induce errors in the sense that the oracle itself might state that a video sequence is not interesting while actually being interesting when visually assessed and vice versa. It comes from the fact that the quality of a video is not easy to quantify. In this chapter, we rather want to use our approach on factual aspects. Users have to specify a performance objective (*e.g.,* the execution time must not exceed X seconds) that will be used by an oracle function in order to define a separation between interesting and not interesting configurations. Keeping the example of video sequences, their visual quality are related to the amount of noise (or high frequency) which is hard to encode producing bigger video files. Instead of asking users to define a "fuzzy" threshold (a user might think "I want my video to be of good quality", whatever "good quality" means), we ask to be more factual and quantify a

2. We refer to the sound approach we depicted in the previous chapter (see Def. 7) which is usually unfeasible on a practical way.

related property (*e.g.,* "the resulting video file must not exceed X Mb").

Compared to state of the art approaches (see Section 3.3.1), this work is also quite novel as it does not tackle the problem of assigning a performance value to a configuration and then decide whether it is interesting or not. Instead, we measure the performance of a sample of configurations and turn those measures into data used to solve a binary classification problem. Related works tackle the problem of : "what is the performance value of this configuration ?" which implies to choose a configuration first. Regarding this particular aspect, it is rather different from our approach which tries to help users in selecting a configuration that has high probability to meet their goals.

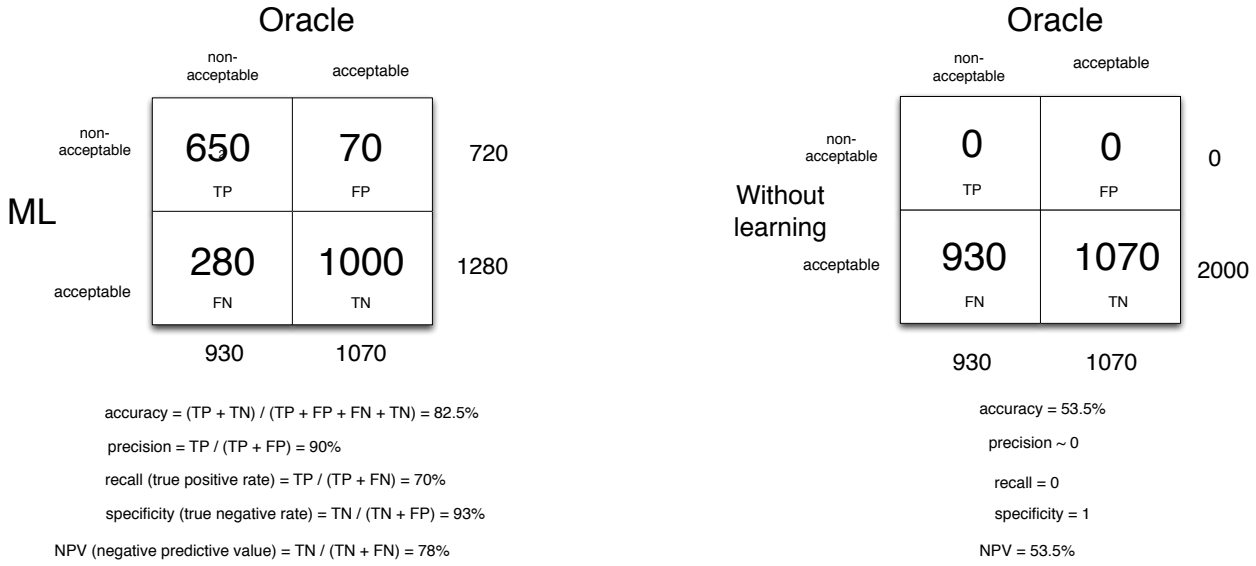
5.3 Discussions

So far, we have talked about very important concepts in the context of this study without naming them. First, the definition of the performance goal has an influence over the number of configurations that are considered as interesting which in turn affects the learning process making it easier or more difficult. Second, previously used performance metrics to evaluate how machine learning techniques perform in predicting the class of configurations (*i.e.,* precision and recall) might not always be suited.

5.3.1 Impacts of performance objectives on the learning problem

Machine learning needs two data sets : the training set to learn a model and the test set on which the model will be applied when predictions have to be made. In the training set, both classes should be represented and the model might be better if both classes are balanced (*i.e.,* equally represented). Consider a training set in which only 10 examples are from one class and $10,000,000$ examples from the other class. The machine learning algorithm might consider that, since only 10 examples are from the first class, they are rare events and most of data that will be encountered in the "nature" (*i.e.,* the test set) should be assigned to the second one. Thus, from the machine learning algorithm's point of view, it is easier to classify everything as being part of the second class since the number of errors is very low, meaning that predictions will be accurate most of the time.

In our case, users might fall in this situation. Considering the distribution of the execution time from x264 configurations given in Figure 5.2, if users want to push the approach to the limit, they will define a very low threshold as a performance goal (*e.g.,* 50 seconds) which might correspond to only 100 configurations that are able to reach such goal. In this situation, it is very unlikely that the sampling process (that comes prior to the learning phase) is able to find enough configurations to balance classes. As the majority of configurations leads to exceeding the performance goal, the machine learning algorithm will classify all configurations as *not interesting*. Retrieved constraints will over-constrain the variability model up to a point that no configurations will be valid anymore resulting in a product line on which cannot derive products. Similarly, the other way around, users that define a performance goal of 950 seconds might build a classification model that states that all configurations are interesting and thus do not



(a) With machine learning

(b) Without learning

FIGURE 5.4 – Confusion matrix and classification metrics : with machine learning *vs* without learning (example)

restrain anything.

Since the distribution of performances of a system is not known a priori, it might be difficult to find a threshold that suits the machine learning algorithm on the first try. Because of that, users should not have strong goals that cannot be changed but rather be a bit flexible on the value to reach as it might help the training process.

Maybe users will have to try different threshold values or perform some analysis on the system before establishing a goal value such that the prediction model can be as precise as possible.

5.3.2 Measures to assess the prediction power of machine learning models

Another problem which is related to the balance of classes is the metrics used to evaluate machine learning models.

We remind that performance evaluation of machine learning algorithms compare predictions given by the algorithms to expected results given by the oracle. It gives a confusion matrix as the ones shown in Figure 5.4. Several measures can be computed on such matrix including *accuracy*, *precision* and *recall* which are the most common.

We have used precision and recall before but these measures only quantify the classification power of the model and it can be a problem in the case of imbalanced classes. Besides, from the user's point of view, we do not solely want to assess the ability of the model to discard not interesting configurations but also its ability to keep interesting configurations. We want to assess, on one hand, chances to pick a configuration that meets performance goals and, on the other, the portion of configurations meeting performance goals that can still be chosen **after constraints have been added** to the variability model. These quantities are correlated to the notion of *Flexibility* and *Safety* of a variability model for a user.

To be a little more precise, we consider a variability model VM and VM', its specialized version following our approach. However, we can use the number of interesting and not interesting configurations in the test set (and their distributions in the resulting confusion matrix) to assess those quantities. We define Safety and Flexibility as follows :

Definition 8 (Flexibility and Plasticity) ***Flexibility*** *is the notion of being able to choose a configuration complying with users' performance goal. To quantify this notion, we define the* ***Plasticity*** *of a specialized variability model regarding a set of configurations s as :* $Plasticity_s(VM') = \frac{N_s}{M_s}$*, with N_s the number of interesting configurations observed in s and that are valid in VM' (represented by TN, in the bottom right cell of the confusion matrix in Figure 5.4a) and M_s the number of interesting configurations in s (*i.e., *the number of configurations labeled as interesting by the oracle which is represented by the second column in Figure 5.4a). It is a ratio defined in the range $[0; 1]$ and the closer to 1 it gets, the better is the flexibility.*

Definition 9 (Safety and Sureness) ***Safety*** *is the notion of being able to pick an interesting configuration* ***after specialization****. Again, to quantify this notion, we define the* ***Sureness*** *of a specialized variability model regarding a set of configurations s as :* $Sureness_s(VM') = \frac{N_s}{[\![VM']\!]_s}$*, with N_s the number of interesting configurations observed in s and that are valid in VM' and $[\![VM']\!]_s$ the number of valid configurations from s according to VM' (represented as the row "ML acceptable" in Figure 5.4a). Similarly to plasticity, sureness is a ratio defined over the range $[0; 1]$ and, again, the safety of a variability model increases as this ratio gets closer to 1.*

Plasticity and sureness are defined according to a confusion matrix and respectively correspond to specificity and NPV explicited at the bottom of Figure 5.4a. To give an

intuition on how to interpret these notions and quantities, we give an example based on confusion matrices given in Figure 5.4.

In this example, 2000 configurations have been sampled from a system and an oracle has been created taking into account some user's performance goal. Figure 5.4a refers to the specialized version of the system following our approach while Figure 5.4b refers to the original system without any specialization. Labels "non-acceptable" and "acceptable" respectively refer to labels "not interesting" and "interesting" we have used before. Several measures can be used to assess the capability of a model learnt via machine learning to take the same decisions as the corresponding oracle. The most common are given at the bottom of Figure 5.4a. Since no specialization has been applied on the system associated with Figure 5.4b, the system cannot discarded not interesting configurations leading to poor precision and recall measures. However, because the system does not discard any configurations, choices left to users are large corresponding to a high flexibility of the system. This is showed by a plasticity or (specificity) of 1. Regarding safety, since no specialization have been applied and all configurations have been generated from this system, the sureness (or NPV) is exactly the number of interesting configurations over 2000, leading to about 0.53. The system is considered as not safe as almost one configuration out of two is likely to *not* meet the performance goal.

On the other hand, applying the specialization process, Figure 5.4a shows that the machine learning process helps improving the *safety* of the algorithm by discarding some of the configurations that do not meet performance goal (650 configurations over 930 to be precise). It improves the sureness of the specialized system compared to the previous one. However, because the machine learning makes some classification errors, 70 configurations are unnecessarily forbidden which makes the system less *flexible* which is highlighted by a lower plasticity measure than in the previously discussed case.

5.4 Experiments

We want to apply our method on real-life systems and assess that it is able too accurately classify configurations as interesting or not *w.r.t.* any performance objective and with a reasonably small training set. But as we said, classification performances are not the only parameters to take care of, we want to avoid loosing to much *flexibility* in the configuration process (*i.e.,* let users choose among a set of interesting configurations and not forcing them to use a unique configuration) while ensuring *safety* (*i.e.,* users should be able to select only configurations that are likely to meet performance goals).

We evaluate these aspects by answering the following research questions :

RQ1 Does our method allow to accurately classify configurations ?

RQ2 Does our method allow to maintain flexibility while being safe ?

5.4.1 Subject systems and configuration performances

We performed our experiments on a publicly-available data set used by previous works [39, 79, 87, 88]. The data set provides sets of configurations for several real-world configurable systems. The configurations have been derived, products have been executed and different performance measures measured.

Table 5.1 shows these configurable systems. They come from various domains and are implemented in different languages (C, C++ and Java). The number of options varies from as little as ten, up to about sixty ; options can be Boolean or take numerical values. The two last columns of the Table show, first, the total number of possible configurations existing for a system, and last, the number of configurations that have been generated and measured in previous works ("All" meaning all configurations have been measured).

It should be noted that most benchmarks measure a single performance value, except for x264 which was evaluated regarding 7 different performance measures.

5.4.2 Experimental setup

We want to study the impact of the size of the training set over a system regarding a specific performance measure for which a specific threshold (*i.e.,* performance goal) has been defined on our method. As said before, the threshold will define the portion of configurations that will be labeled as "interesting" (and "not interesting") which may

System	Domain	Lang.	Features	$\#[\![VM]\!]$	Meas.
Apache	Web Server	C	9/0	192	All
BerkeleyC	Database	C	18/0	2560	All
BerkeleyJ	Database	Java	26/0	400	181
LLVM	Compiler	C++	11/0	1024	All
SQLite	Database	C	39/0	10^6	4553
Dune	Solver	C++	8/3	2304	All
HIPA^{cc}	Image Proc.	C++	31/2	13485	All
HSMGP	Solver	n/a	11/3	3456	All
JavaGC	Runtime Env.	C++	12/23	10^{31}	166k
x264 (Energy)	Codec	C	8/12	10^{27}	69k
x264 (PSNR)	Codec	C	8/12	10^{27}	69k
x264 (SSIM)	Codec	C	8/12	10^{27}	69k
x264 (Speed)	Codec	C	8/12	10^{27}	69k
x264 (Size)	Codec	C	8/12	10^{27}	69k
x264 (Time)	Codec	C	8/12	10^{27}	69k
x264 (Watt)	Codec	C	8/12	10^{27}	69k

TABLE 5.1 – *Features* : number of boolean features / number of numerical features ; $\#[\![VM]\!]$: number of valid configurations ; *Meas.* : number of configurations that have been measured.

influence the difficulty of the machine learning algorithm to learn an appropriate prediction model. For each experiment : precision, recall, specificity and negative predictive value of the prediction model are measured.

Since we evaluate prediction performances, we need two data sets (*i.e.,* the training set and the test set). For each subsystem, we have randomly selected configurations to build a training set (according to the size specified before) from the ones available in the public data set. Remaining configurations constitute the test set on which prediction performance are computed. For instance, the x264 data set contains $69,000$ different configurations. If the training set size is 500, 500 randomly picked configurations will be used to learn a model and the remaining configurations (*i.e.,* $69,000 - 500 = 68,500$) will be used to evaluate prediction performances of the model.

For each configurable system, we make vary the sample size from 1 to the total number of configurations with a step of 1% of the available configurations. Further-

more, we make vary the performance goal between the lower and upper bounds that have been observed in each data set (*i.e.,* each bounds are thus specific to a particular system). Based on the observed distribution of performances, we selected 20 performance goals corresponding roughly to 5%, 10%,15%, *etc.* of interesting configurations in the data set. This method allows us to get an idea of the influence of the performance goal on learning a classification model.

To reduce the fluctuations of the dependent variables caused by random generation, we performed ten repetitions for each combination of the independent variables.

5.4.3 Presentation of results

Using 16 systems with varying parameters demand to analyze a large amount of data which is hard to summarize. For a single system, as we explained before, we need to make vary both the performance goal and the training set size independently and evaluate their impact on the performances of the system. For each system, 100 training sets are created and 20 performance goals are considered. We also repeated each measurements 10 times. In total, about $16 * 100 * 20 * 10 = 320,000$ measurements were made.

For each measure (*i.e.,* precision, recall, plasticity and sureness), we present, in the following, its average, min and max (that will be presented in separated tables). We also show the conditions of executions (*i.e.,* the number of configuration in the training set as well as the performance objective) in which such results were produced. Because of the heterogeneity of the different systems, we show the percentage of configurations used in the training set compared to the number of available configurations (*i.e.,* last column of Table 5.1) and, under brackets, the actual number of configurations. We do the same regarding the performance objective.

5.4.4 RQ1) Does our method allow to accurately classify configurations ?

Precision analysis

Regarding Figure 5.5, the average precision for all systems is above 0.80 which is high from a general point of view. They range from 0.80 to 0.98 with an average of 0.89. However, execution conditions are really different from one system to another.

Systems	Avg Prec.	% config. in train. set (# config.)	% interest. config. (Perf. goal)
Apache	0.89	34 (65)	17 (1140)
BerkeleyC	0.98	93 (2376)	54 (21)
BerkeleyJ	0.92	43 (77)	86 (14627)
LLVM	0.89	1 (11)	19 (215.47)
SQLite	0.80	38 (1711)	42 (14.39)
Dune	0.94	25 (576)	27 (10175.53)
HIPA^{cc}	0.97	30 (4021)	37 (42.576)
HSMGP	0.97	37 (1293)	10 (445.488)
JavaGC	0.98	81 (135190)	6 (830)
x264 (Energy)	0.85	74 (51061)	19 (1515.57)
x264 (PSNR)	0.84	44 (30361)	61 (47.124)
x264 (SSIM)	0.83	98 (67621)	99 (0.9967)
x264 (Speed)	0.88	87 (60031)	77 (612.38)
x264 (Size)	0.85	69 (47611)	26 (46.1)
x264 (Time)	0.85	63 (43471)	22 (11.3)
x264 (Watt)	0.81	44 (30361)	58 (154)

FIGURE 5.5 – Average Precision measures for all 16 systems along with execution conditions. Because of the heterogeneity of systems, we present the percentage of configurations used in the training set compared to the number of available configurations (see last column of Table 5.1 and the absolute number under brackets. The last column present the percentage of interesting configurations according to the distribution of performance and the performance goal (given under brackets).

Sometimes, only a few configurations are needed with a relatively difficult performance goal to reach (see LLVM system), sometimes almost all configurations are required with an easy target performance goal (see x264 (SSIM) system). Based on these averages, SQLite and x264 seem rather difficult to learn on (with measures between 0.80 and 0.85) while other systems seem rather easy (with measures equal to 0.89 or above).

Considering now the worst case, in which the precision is the worst presented in Figure 5.6, all systems can provide terrible results. All systems present a precision measure of 0, meaning they were not able to build any constraints discarding configurations. This behavior comes from extreme conditions in which only a single configuration was given in the training set. In the meantime, performance goals were not hard to achieve (with at least 25% of the configurations being able to reach the goal) except for HSMGP system which had a performance goal allowing only 14% of configurations to be interesting. In this case, training a machine learning model by only providing a single configuration does not provide enough information even without taking into ac-

Systems	Min Prec.	% config. in train. set (# config.)	% interest. config. (Perf. goal)
Apache	0	1 (1)	67 (1920)
BerkeleyC	0	<0.1 (1)	54 (21)
BerkeleyJ	0	<1 (1)	> 99 (16399)
LLVM	0	<1 (1)	93 (262)
SQLite	0	<0.1 (1)	66 (15.13)
Dune	0	<0.1 (1)	54 (13508.69)
HIPA^{cc}	0	<0.01 (1)	37(42.576)
HSMGP	0	<0.1 (1)	14 (568.792)
JavaGC	0	0.001 (1)	54 (3764.5)
x264 (Energy)	0	<0.01 (1)	56 (3740.36)
x264 (PSNR)	0	<0.01 (1)	60 (47.112)
x264 (SSIM)	0	<0.01 (1)	99 (0.9967)
x264 (Speed)	0	<0.01 (1)	65 (526.01)
x264 (Size)	0	<0.01 (1)	26 (47.12)
x264 (Time)	0	<0.01 (1)	75 (31.33)
x264 (Watt)	0	<0.01 (1)	92 (158.31)

FIGURE 5.6 – Minimum Precision measures for all 16 systems along with execution conditions. Because of the heterogeneity of systems, we present the percentage of configurations used in the training set compared to the number of available configurations (see last column of Table 5.1 and the absolute number under brackets. The last column present the percentage of interesting configurations according to the distribution of performance and the performance goal (given under brackets).

count the performance goal. No constraints can be learnt with only a single example especially when the example is interesting. Again, machine learning algorithms are based on statistics. Giving only one single example will result in an inference of every remaining configurations to belong to the same class (which is obviously not realistic).

Moving to the other extreme case now (provided by Figure 5.7), all systems provide precision higher or equal to 0.99. Most of the systems use a lot of information (the training being composed of the vast majority of available configurations), only Apache and BerkelyJ used less than 10% of available configurations (representing 17 and 8 configurations respectively). However, performance goals are rather hard to reach for all systems (except x264 (SSIM)), meaning that most of configurations should be discarded.

Sum-up over Precision : In the end, those extreme cases (worst and best cases), do not provide useful information. In substance, we learn that not providing enough information with the training set will lead to a bad model that will tend to keep all confi-

Systems	Max Prec.	% config. in train. set (# config.)	% interest. config. (Perf. goal)
Apache	1	9 (17)	35 (1410)
BerkeleyC	1	72 (1851)	4 (2)
BerkeleyJ	1	4 (8)	15 (5045)
LLVM	1	99 (1011)	4 (206)
SQLite	0.99	>99 (4546)	<0.1 (13.16)
Dune	1	>99 (2301)	<0.001 (6773.45)
HIPA^{cc}	0.99	99 (13401)	0 (21.682)
HSMGP	1	43 (1497)	23 (852.556)
JavaGC	0.99	100 (66901)	0 (470.5)
x264 (Energy)	0.99	100 (69001)	4 (634.68)
x264 (PSNR)	0.99	100 (69001)	15 (39.59)
x264 (SSIM)	0.99	100 (69001)	76 (0.9857)
x264 (Speed)	0.99	100 (69001)	0 (52.09)
x264 (Size)	0.99	100 (69001)	0 (8.07)
x264 (Time)	0.99	100 (69001)	3 (4)
x264 (Watt)	0.99	100 (69001)	<0.1 (146.95)

FIGURE 5.7 – Maximum Precision measures for all 16 systems along with execution conditions. Because of the heterogeneity of systems, we present the percentage of configurations used in the training set compared to the number of available configurations (see last column of Table 5.1 and the absolute number under brackets. The last column present the percentage of interesting configurations according to the distribution of performance and the performance goal (given under brackets).

gurations (if the example is able to reach the performance goal). On the other hand, targeting a hard performance goal (a performance goal for which a few configurations are able to reach it) is likely to create a model that will try to discard all configurations. Apart from these extreme cases, in average, learnt models provide a fairly good precision measure. But, an "average" case is really dependent on the system under study and might require a lot of experimental efforts to find a suitable training set and an adequate performance goal.

However, evaluating only precision is not enough to understand if the learnt model performs well or if it tries to put every configuration in a unique class. We have to consider recall.

Systems	Avg Recall	% config. in train. set (# config.)	% interest. config. (Perf. goal)
Apache	0.84	19 (37)	13 (1080)
BerkeleyC	0.97	1 (26)	0 (0.38)
BerkeleyJ	0.88	13 (24)	1 (3161)
LLVM	0.84	55 (561)	0.71 (248)
SQLite	0.73	87 (3961)	61 (14.96)
Dune	0.90	25 (576)	29 (10379.51)
HIPA^{cc}	0.95	64 (8577)	65 (58.173)
HSMGP	0.94	61 (2109)	8 (376.219)
JavaGC	0.97	58 (96803)	1 (557.5)
x264 (Energy)	0.73	43 (29671)	4 (634.68)
x264 (PSNR)	0.74	47 (39331)	61 (47.124)
x264 (SSIM)	0.72	55 (37951)	86 (0.9904)
x264 (Speed)	0.78	55 (37951)	25 (236.51)
x264 (Size)	0.73	59 (40711)	21 (39.61)
x264 (Time)	0.73	44 (30361)	3 (4)
x264 (Watt)	0.67	51 (35191)	61 (154.46)

FIGURE 5.8 – Average Recall measures for all 16 systems along with execution conditions. Because of the heterogeneity of systems, we present the percentage of configurations used in the training set compared to the number of available configurations (see last column of Table 5.1 and the absolute number under brackets. The last column present the percentage of interesting configurations according to the distribution of performance and the performance goal (given under brackets).

Recall analysis

Compared to precision, the range of average recall are a bit lower. Values range from 0.67 to 0.97 with an average of almost 0.83. Again, execution conditions vary a lot from one system to another.

Worst cases are similar to the ones studied with precision. All systems retrieve a recall of 0 when they use only one configuration in the training set. A recall measure equals to 0 means that the model is not able to classify correctly any of the configurations that should be discarded. No trend emerges from the performance goal, it depends on the system. Still, most systems have a performance goal fairly easy to reach (with more than half of the configurations able to reach it) except for 3 systems.

In the best case, all systems are able to reach a recall of 1 (meaning that all configurations that are not defined as interesting by the oracle are also classified as not interesting by the machine learning model). Except for x264 (SSIM), performance goal

Systems	Min Recall	% config. in train. set (# config.)	% interest. config. (Perf. goal)
Apache	0	1 (1)	67 (1920)
BerkeleyC	0	<0.1 (1)	54 (21)
BerkeleyJ	0	<1 (1)	>99 (16399)
LLVM	0	<1 (1)	93 (262)
SQLite	0	<0.1 (1)	66 (15.13)
Dune	0	<0.1 (1)	54 (13508.69)
HIPA^{cc}	0	<0.01 (1)	37 (42.576)
HSMGP	0	<0.1 (1)	14 (568.792)
JavaGC	0	<0.001 (1)	54 (3764.5)
x264 (Energy)	0	<0.01 (1)	56 (3740.36)
x264 (PSNR)	0	<0.01 (1)	60 (47.112)
x264 (SSIM)	0	<0.01 (1)	99 (0.9967)
x264 (Speed)	0	<0.01 (1)	65 (526.01)
x264 (Size)	0	<0.01 (1)	26 (47.12)
x264 (Time)	0	<0.01 (1)	75 (31.33)
x264 (Watt)	0	<0.01 (1)	92 (158.31)

FIGURE 5.9 – Minimum Recall measures for all 16 systems along with execution conditions. Because of the heterogeneity of systems, we present the percentage of configurations used in the training set compared to the number of available configurations (see last column of Table 5.1 and the absolute number under brackets. The last column present the percentage of interesting configurations according to the distribution of performance and the performance goal (given under brackets).

are hard to reach (with less than 5% of interesting configurations), and only one configuration used in the training set. It means that, almost all configurations are not interesting and providing only one configuration that is not interesting is enough to make the algorithm infer that the rest of the configurations are not interesting too.

Sum-up over Recall : Again, extreme cases are not really useful. Similar conclusions can be drawn when considering recall or precision. Not providing enough information in the training set leads to a poor model that tends to keep all configurations (if the given configuration is interesting, otherwise, it is likely that every configurations will be discarded). On the other hand, targeting a hard performance goal (a performance goal for which a few configurations are interesting) is likely to create a model that will try to discard all configurations reaching a high recall measure. Apart from these extreme cases, in average, learnt models provide a fairly good recall measure.

Systems	Max Recall	t% config. in train. set (# config.)	% interest. config. (Perf. goal)
Apache	1	1 (1)	2 (900)
BerkeleyC	1	<0.1 (1)	0 (0.38)
BerkeleyJ	1	<1 (1)	0 (3065)
LLVM	1	<1 (1)	4 (206)
SQLite	1	<0.1 (1)	<0.1 (13.16)
Dune	1	<0.1 (1)	0.001 (6773.45)
HIPA^{cc}	1	<0.01 (1)	0 (21.682)
HSMGP	1	<0.1 (1)	0 (127.142)
JavaGC	1	<0.001 (1)	0 (470.5)
x264 (Energy)	1	<0.01 (1)	<0.1 (405.04)
x264 (PSNR)	1	<0.01 (1)	0 (37.128)
x264 (SSIM)	1	<0.01 (1)	16 (0.9559)
x264 (Speed)	1	<0.01 (1)	2 (67.03)
x264 (Size)	1	<0.01 (1)	0 (8.07)
x264 (Time)	1	<0.01 (1)	1 (3)
x264 (Watt)	1	<0.01 (1)	<0.1 (146.95)

FIGURE 5.10 – Maximum Recall measures for all 16 systems along with execution conditions. Because of the heterogeneity of systems, we present the percentage of configurations used in the training set compared to the number of available configurations (see last column of Table 5.1 and the absolute number under brackets. The last column present the percentage of interesting configurations according to the distribution of performance and the performance goal (given under brackets).

Concluding remarks

In the end, our method is able to provide constraints that are able to tell interesting configurations apart from not interesting configurations rather accurately (with precision and recall measures above 0.70 in average). However, as we saw, retrieving a very high precision and recall measures (*i.e.,* above 0.9) may not be enough. From a learning point of view, presented results only evaluate the capability of the machine learning model to exclude not interesting configurations from the variability model. Precision and recall measures above 0.9 mostly occur when the performance goal is hard to reach for configurations. This may result in an empty variability model in which no valid configurations exist anymore. Precision and recall, in this case, tends to reduce choices that users can make in the configuration process. We need to evaluate also this aspect and this is the reason why flexibility and safety should be considered.

5.4.5 RQ2) Does our method allow to maintain flexibility while being safe ?

Flexibility analysis

Systems	Avg Plast.	% config. in train. set (# config.)	% interest. config. (Perf. goal)
Apache	0.90	50 (96)	15 (1110)
BerkeleyC	0.98	3 (76)	100 (38.95)
BerkeleyJ	0.90	10 (18)	3 (3532)
LLVM	0.94	32 (331)	35 (225.62)
SQLite	0.77	28 (1261)	40 (14.34)
Dune	0.93	16 (369)	27 (10175.53)
HIPA^{cc}	0.96	36 (4825)	4 (23.689)
HSMGP	0.95	42 (1463)	4 (262.707)
JavaGC	0.97	85 (141866)	0 (470.5)
x264 (Energy)	0.82	88 (60721)	2 (552.84)
x264 (PSNR)	0.90	2 (1381)	61 (47.235)
x264 (SSIM)	0.88	5 (3451)	88 (0.99)
x264 (Speed)	0.88	15 (10351)	33 (289.17)
x264 (Size)	0.83	32 (22081)	15 (29.81)
x264 (Time)	0.82	12 (8281)	15 (8.33)
x264 (Watt)	0.83	53 (36571)	66 (155)

FIGURE 5.11 – Average Plasticity measures for all 16 systems along with execution conditions. Because of the heterogeneity of systems, we present the percentage of configurations used in the training set compared to the number of available configurations (see last column of Table 5.1 and the absolute number under brackets. The last column present the percentage of interesting configurations according to the distribution of performance and the performance goal (given under brackets).

We have defined, in Def. 8, plasticity as a measure to quantify the flexibility of a system between 0 and 1. The higher it gets, the more possibilities users have to configure their systems while meeting performance requirements. Figure 5.11 shows the average plasticity retrieved for each system. For all systems, plasticity is high (above 0.75). Meaning that, the machine learning model is able to keep available most interesting configurations. Again, execution conditions are different from one system to another.

In the worst case, plasticity reaches 0 as shown in Figure 5.12. We fall back in previous cases in which a few configurations are used in the training set (*i.e.,* only one

Systems	Min Plast.	% config. in train. set (# config.)	% interest. config. (Perf. goal)
Apache	0	<1 (1)	2 (900)
BerkeleyC	0	<0.1 (1)	0 (0.38)
BerkeleyJ	0	<1 (1)	0 (3065)
LLVM	0	<0.1 (1)	4 (206.23)
SQLite	0	<0.1 (1)	0 (13.16)
Dune	0	<0.1 (1)	0 (6773.46)
HIPA^{cc}	0	<0.01 (1)	0 (21.682)
HSMGP	0	<0.1 (1)	0 (127.142)
JavaGC	0	<0.01 (1)	0 (470.5)
x264 (Energy)	0	<0.01 (1)	0 (405)
x264 (PSNR)	0	<0.01 (1)	0 (37.128)
x264 (SSIM)	0	<0.01 (1)	16 (95.586)
x264 (Speed)	0	0.01 (1)	2 (67.03)
x264 (Size)	0	<0.01 (1)	0 (8.07)
x264 (Time)	0	<0.01 (1)	1 (3)
x264 (Watt)	0	<0.01 (1)	0 (146.95)

FIGURE 5.12 – Minimum Plasticity measures for all 16 systems along with execution conditions. Because of the heterogeneity of systems, we present the percentage of configurations used in the training set compared to the number of available configurations (see last column of Table 5.1 and the absolute number under brackets. The last column present the percentage of interesting configurations according to the distribution of performance and the performance goal (given under brackets).

configuration) while defining a performance goal for which only a few configurations are defined as interesting. Once again, the machine learning algorithm will classify every configuration as not being able to meet the performance goal based on the ones given in the training set. All configurations are discarded, users have no choices left, and the systems is not usable anymore (in addition to not being configurable).

In the best case, the flexibility of the system is preserved with a plasticity of 1 as shown in Figure 5.13. In this case, again, only one configuration is used in the training set. We can make the assumption that the configuration is labeled as meeting the performance goal, contrarily to the worst case, forcing the machine learning to not build new constraints. In this case, machine learning has learnt nothing, and we are back to a non-learning approach.

Sum-up over Plasticity : Plasticity measures have shown that, in average, the machine learning model is able to keep in the valid set of configurations most of configurations that actually meet performance goals. It confirms previous results we had in

Systems	Max Plast.	% config. in train. set (# config.)	% interest. config. (Perf. goal)
Apache	1	<1 (1)	67 (1920)
BerkeleyC	1	<0.1 (1)	54 (21.25)
BerkeleyJ	1	<1 (1)	>99 (16399)
LLVM	1	<0.1 (1)	93 (262.11)
SQLite	1	<0.1 (1)	66 (15.13)
Dune	1	<0.1 (1)	54 (13.508.70)
HIPA^{cc}	1	<0.01 (1)	37 (42.576)
HSMGP	1	<0.1 (1)	14 (568.792)
JavaGC	1	<0.01 (1)	54 (3764.5)
x264 (Energy)	1	<0.01 (1)	56 (3740.36)
x264 (PSNR)	1	<0.01 (1)	60 (47.112)
x264 (SSIM)	1	<0.01 (1)	99 (99.67)
x264 (Speed)	1	<0.01 (1)	65 (526.01)
x264 (Size)	1	<0.01 (1)	26 (47.12)
x264 (Time)	1	<0.01 (1)	75 (31.33)
x264 (Watt)	1	<0.01 (1)	92 (158.31)

FIGURE 5.13 – Maximum Plasticity measures for all 16 systems along with execution conditions. Because of the heterogeneity of systems, we present the percentage of configurations used in the training set compared to the number of available configurations (see last column of Table 5.1 and the absolute number under brackets. The last column present the percentage of interesting configurations according to the distribution of performance and the performance goal (given under brackets).

Chapter 4 that machine learning can be used in the context of specializing configurable systems as they are able to keep interesting configurations that meet performance goals in the set of valid configurations of a variability model. However, as we showed, it is easy to understand conditions in which low or high values of plasticity occur and thus, we maintain that it is not the only aspect that we need to take into account.

Sureness analysis

Def. 9 defines the ability of a system to keep in the valid set of a variability model only configurations that meet performance goals. We have called this ability the safety of a system and proposed the sureness to quantify this notion (between 0 and 1). Once again, average measures are high (above 0.70) for all systems. Meaning that, at least 7 out of 10 configurations in the specialized variability model are configurations able to meet the predefined performance goal showed on the last column of the figure.

Systems	Avg Sure.	% config. in train. set (# config.)	% interest. config. (Perf. goal)
Apache	0.85	63 (121)	8 (990)
BerkeleyC	0.98	82 (2101)	<1 (0.52)
BerkeleyJ	0.85	65 (118)	2 (3299)
LLVM	0.90	25 (251)	71 (248)
SQLite	0.73	44 (1981)	40 (14.34)
Dune	0.89	87 (2002)	2 (6980)
HIPA^{cc}	0.94	10 (1341)	18 (31.9)
HSMGP	0.92	86 (3027)	0.6 (146.784)
JavaGC	0.95	34 (56747)	0.5 (501)
x264 (Energy)	0.70	62 (42781)	14 (1244.48)
x264 (PSNR)	0.82	94 (64861)	61 (47.285)
x264 (SSIM)	0.79	75 (51751)	69 (0.75)
x264 (Speed)	0.79	95 (65551)	48 (399.01)
x264 (Size)	0.71	18 (12421)	14 (28.11)
x264 (Time)	0.71	95 (65551)	15 (8.33)
x264 (Watt)	0.71	27 (18631)	67 (155.19)

FIGURE 5.14 – Average Sureness measures for all 16 systems along with execution conditions. Because of the heterogeneity of systems, we present the percentage of configurations used in the training set compared to the number of available configurations (see last column of Table 5.1 and the absolute number under brackets. The last column present the percentage of interesting configurations according to the distribution of performance and the performance goal (given under brackets).

Chances for users to select a configurations that is not interesting are thus reduced and the original try-and-error process to configure the system tends to vanish. Performance goals and number of configurations in the training set are too different from one system to another for a general trend to emerge.

In the worst case (shown by Figure 5.15), sureness is equal to zero. For all systems, only one configuration is used in the training set and we can see that performance goals are very hard to reach. Only x264 (SSIM) had over 15% of configurations able to reach this goal, for all other systems, less than 5% of configurations are interesting. Maybe more than before, in this figure, we can see how difficult it can be for a machine learning algorithm to learn anything. However, we can see also that the performance goal for 11 systems is so hard that none of the configurations are able to cope with it.

Maximum sureness for all systems is shown in Figure 5.16. Measures are above 0.98 which shows that systems can be very secure. Interesting trends show up. Except for the BerkeleyJ and BerkeleyC systems, either the performance goal is very easy to

Systems	Min Sure.	% config. in train. set (# config.)	% interest. config. (Perf. goal)
Apache	0	<1 (1)	2 (900)
BerkeleyC	0	<0.1 (1)	0 (0.38)
BerkeleyJ	0	<1 (1)	0 (3065)
LLVM	0	<0.1 (1)	4 (206.23)
SQLite	0	<0.1 (1)	0 (13.16)
Dune	0	<0.1 (1)	0 (6773.46)
HIPA^{cc}	0	<0.01 (1)	0 (21.682)
HSMGP	0	<0.1 (1)	0 (127.142)
JavaGC	0	<0.001 (1)	0 (470.5)
x264 (Energy)	0	<0.01 (1)	0 (405.04)
x264 (PSNR)	0	<0.01 (1)	0 (37.128)
x264 (SSIM)	0	<0.01 (1)	16 (0.95586)
x264 (Speed)	0	<0.01 (1)	2 (67.03)
x264 (Size)	0	<0.01 (1)	0 (8.07)
x264 (Time)	0	<0.01 (1)	1 (3)
x264 (Watt)	0	<0.01 (1)	0 (146.95)

FIGURE 5.15 – Minimum Sureness measures for all 16 systems along with execution conditions. Because of the heterogeneity of systems, we present the percentage of configurations used in the training set compared to the number of available configurations (see last column of Table 5.1 and the absolute number under brackets. The last column present the percentage of interesting configurations according to the distribution of performance and the performance goal (given under brackets).

reach either the number of configuration in the training is very large. From this Figure, we can say that, apparently BerkeleyJ and BerkeleyC are very easy to learn on as only a few configurations are needed to learn average to hard performance goals. Apache is surprising as the performance goal is very difficult to reach (only 1% of configurations are able to do so) but not all configurations are needed to reach 0.99 sureness. In other cases, experiment conditions tend to show that, for a system to be safe, either it must use all information at disposal in the training set or users must defined fairly easy performance goals which might facilitate the learning task.

Sum-up over Sureness : Sureness measures have shown that, in average, the machine learning model is able to discriminate configurations such that a majority (more than 7 out of 10 in average) of valid configurations are able to reach users defined performance goals. It confirms previous results we had in Chapter 4 that machine learning can be used in the context of specializing configurable systems as they are able to remove configurations that are not able to meet performance goals from the set of valid

Systems	Max Sure.	% config. in train. set (# config.)	% interest. config. (Perf. goal)
Apache	0.99	88.5 (170)	1 (2430)
BerkeleyC	1	2 (51)	51 (20)
BerkeleyJ	1	5 (9)	16 (5186)
LLVM	0.99	94 (961)	100 (266.32)
SQLite	0.99	73 (3331)	100 (16.13)
Dune	>0.99	89 (2048)	100 (19289.52)
HIPA^{cc}	>0.99	99 (13401)	100 (77.494)
HSMGP	>0.99	99 (3435)	56 (1902.79)
JavaGC	>0.99	100 (16901)	31 (2388)
x264 (Energy)	>0.99	55 (37951)	100 (6404)
x264 (PSNR)	0.99	22 (15181)	100 (53.646)
x264 (SSIM)	0.99	38 (26221)	100 (0.997)
x264 (Speed)	>0.99	50 (34501)	100 (779.44)
x264 (Size)	>0.99	45 (31051)	100 (155.67)
x264 (Time)	>0.99	67 (46231)	100 (41)
x264 (Watt)	0.98	48 (33121)	100 (159.25)

FIGURE 5.16 – Maximum Sureness measures for all 16 systems along with execution conditions. Because of the heterogeneity of systems, we present the percentage of configurations used in the training set compared to the number of available configurations (see last column of Table 5.1 and the absolute number under brackets. The last column present the percentage of interesting configurations according to the distribution of performance and the performance goal (given under brackets).

configurations of a variability model. However, as showed in Figure 5.16, it is easy to make a system very safe but, once again, machine learning becomes useless as performance goals might be reached by the vast majority (if not all) valid configurations of the variability model. Removing extrema cases, safeness seems to be an interesting indicator to consider as it balances the fact that good results might be retrieved with only a few configurations in the training set. Doing so, more information can be taken into account by the machine learning algorithm which may result in models that perform better at shrinking the configuration space.

Concluding remarks

In the end, our method is able to provide constraints that are able to keep interesting configurations in the specialized variability model while the number of not interesting configurations is reduced. However, as we saw, retrieving a very high plasticity and

sureness measures (*i.e.,* above 0.9) may not be enough. In particular, we see that high plasticity corresponds to not constraining anything, thus the learning process is useless. On the other hand, sureness tries to ensure that all not interesting configurations are removed from the set of valid configurations which might necessitate heavy computations (needing all configurations at disposal) and thus becomes impossible in practice.

5.5 Conclusion

This chapter followed the same approach of the one presented in Chapter 4 but significantly differ as we focus this chapter on different aspects. We have conducted new experiments on various different systems which have different characteristics (*i.e.,* number of available configurations, number and nature of options, implementation languages and application domains). Our experiments have focused on the definition of performance goals that must be reached by configurations of the variability model. Compared to our previous work, these performance goals are easier to define as they are refer to quantities that can be measured (*e.g.,* execution time or memory consumption). Users only have to define a threshold value for a performance measure that, in turn, is used in an oracle procedure in order to label configurations.

However, considering such kind of oracle procedures reveal several challenges.

First, the approach needed to be slightly adapted in order to go from performance values (which are often numerical value) to a binary classification problem.

Second, as the distribution of performances of the configurations of a system are not known a priori, classes might be unbalanced, it can be difficult to find configurations that meet (respectively do not meet) performance goals which can be a problem for the learning process.

Finally, our previous study has focused on evaluating the performance of the machine learning algorithm via precision and recall which are well-known in this domain. In our case, we have to go further and evaluate that the machine learning model that has been built does not restrict too much users possibilities in configuring their systems nor does it allow too many not interesting configurations to be picked by users. To do so, we relied on the definition of two notions : flexibility and safety of a configurable system ; and their measurable quantities : plasticity and sureness.

Based on our experiments, learning-based specialization is able to provide safer systems while maintaining some flexibility in the choice of configurations' features. However, our approach looses in flexibility compared to a non-learning based approach as classification errors are made . Meaning that, some configurations meeting performance goal might be wrongly forbidden. However, a learning-based technique can be much more efficient to discard unsafe configurations than a non-learning-based approach. In general, our specialization method has shown to achieve a good trade-off between flexibility and safety.

Future works include, among others, the exploration of other machine learning algorithms, but also to pursue efforts to properly evaluate the benefits of a machine learning approach. Even if, results were globally encouraging, we have shown that extreme cases provide either very good or very bad measures but these cases are not relevant as they excessively limit information taken into account in the training set or target a performance goal that all (respectively none) configurations are able to meet. Developing a framework that let users decide whether they want a safe system or a flexible one or a trade-off between the two is also a very interesting idea to explore.

CHAPITRE 6

MULTIMORPHIC TESTING

6.1 Introduction

The two previous chapters focused on specializing a software product line such that derived products are relevant to tackle a specific problem and meeting predefined requirements. The proposed approach aims at helping populate the first dimension of the matrix. But before the learning process can be launched, product variants still have to be executed against test cases in order to evaluate their performances. Usually, test suites contain a lot of test cases in order to cover different situations. Considering the combination of test cases with all considered program variants can be very time consuming. Furthermore, in some contexts, some test cases might be redundant with others as measured performances will be close. This can come from the fact that configurations are close to each other. Taking the example of images, it would mean that both conditions of capture as well as contents are similar. In such situation, keeping all these kinds of test cases in the test suite do not provide any additional useful information. Thus, a first problem is to find relevant test cases (*i.e.,* the ones that provide significant different measures) in order to execute them first. Doing so, we can have an idea of the performances of programs at a reduced cost (*i.e.,* without having to execute all test cases).

In the mean time, different test suites or benchmarks may exist to evaluate performances of different program variants. Since they provide different test cases, it makes their comparison difficult resulting in an other problem : how to choose a benchmark/-test suite among a collection in order to properly assess program variants at hand ?

Taking the example of computer vision and object tracking, a lot of benchmarks exist (through competitions and other evaluations) [25, 30, 38, 58, 71] with their own emphasis of specific elements that make the object tracking task difficult. For this reason choosing a specific benchmark when trying to tackle the object tracking problem as broadly as possible is complicated. Futhermore, ImageNet [25] contains millions of

images which might take time to process them all. We can wonder whether such a number is useful and necessary. Even with such large benchmarks, on May 7, 2016, a 2015 Tesla Model S collided with a tractor trailer crossing an uncontrolled intersection on a highway west of Williston, Florida, USA, resulting in fatal injuries to the Tesla driver. One might wonder why the computer vision program did not "see" this huge trailer in the middle of the road. An analysis a posteriori showed that the videos recorded during the crash by Autopilot were not under ideal lighting conditions. Background objects blended into vehicles that needed to be recognized, making it difficult for any computer to process the video stream correctly. On top of that, no wheels were visible under the trailer, which complicated its identification as a vehicle in the middle of the road. More recently, on March 18, 2018, an autonomous Uber vehicle hit a pedestrian crossing a road in Arizona, USA. The pedestrian crossed outside any near crosswalks, at night, on a road where there were no public lighting. A video of the accident was released a few weeks later showing that the vehicle did not even try to dodge or brake before it hits the person and provokes her death.

Now taking a software engineering perspective, these situations clearly lead to the usual question from the software testing community : how come that those systems were deployed without being tested under such conditions ? This is, of course, partly due to a huge input data space. Going further, since the input space (*e.g.,* videos for testing video tracking) is orders of magnitude larger than typical data, we ask the following set of questions : how much effort should we put in the testing activity ? How can we build a "good" test suite ? How do we even know that a given test suite is "better" than another one ? Depending on the software application's goals (*e.g.,* maximizing speed, computations' accuracy or even a tradeoff among several such quantitative properties), do we end up with the same "good" test set ? Structural code coverage metrics for test suites seem indeed a bit shaky for that kind of software systems, especially for handling quantitative properties related to performance aspects. To our knowledge, no method exists to assess the "coverage" of test suites with respect to their ability to reveal performance weaknesses in the software.

This general problem does not only apply to computer vision systems. For instance, generative programming techniques have become a common practice in software development to deal with the heterogeneity of platforms and technological stacks that exist in several domains such as mobile or Internet of Things. Generative programming offers a software abstraction layer that software developers can use to specify the de-

sired system's behavior and automatically generate software artifacts on different platforms. As a consequence, multiple code generators (also called compilers) are used to transform the specifications/models represented in graphical or textual languages into different general-purpose programming languages such as C, Java, C++, *etc.* or different byte code or machine code. In this case, from a testing perspective, the input data space is made of models or programs. Defective code generators with respect to performance (*e.g.,* high resource usage, low execution speed, etc.) are then hard to detect since testers need to produce and interpret numerous numerical results.

We propose a method to empirically assess the relative value of test suites. In particular, we propose a particular metric to build scores allowing us to compare test suites. By analogy with mutation testing [7, 36, 78], the core idea of multimorphic testing is to leverage the configurability of these software systems (synthetizing morphs through the variation of parameters' values and execute test cases over them) to check whether it makes any difference on the outcome of the tests : *i.e.,* are some test cases able to "kill" program's configurations ? "Killing" intuitively means exhibit high variations w.r.t. quantitative properties of the morphs (*e.g.,* some morphs are too slow). In this work, we define the relative value of a test suite as a trade-off between the size of the benchmark (*i.e.,* the number of its elements) and its discriminative power (*i.e.,* how well it is able to differentiate the execution of multiple program configurations).

This chapter is mainly copied from an article that is under submission in a journal following the first submission [99] that we have performed at the 40^{th} International Conference of Software Engineering. The main differences with the article are written in this introduction as we put back this method in the context of this thesis.

Section 6.2 presents our method including the definition of our dispersion score. To show its applicability to various domains, we validate our approach on three different applications in Section 6.3 (*i.e.,* video tracking, image recognition and code generators). Section 6.4 discusses some threats of our method and evaluation process. Section 6.5 concludes and proposes future investigation axis.

6.2 Multimorphic Testing

6.2.1 Motivation

While functional tests check the output conformance of a program (with the help of an Oracle giving a "pass"/"fail" verdict), *performance testing* aims to assess quantitative properties through the execution of software under various conditions. This assessment can be confronted to user-defined requirements (*e.g.,* "I need a system that is able to process inputs under a fixed amount of time"). In the end, the process (characterization and confrontation) can provide level of insurances about performances of a system (*e.g.,* "From the test I ran, I saw that the system was able to process inputs under 30 seconds in 7 out of 10 cases") which can be crucial, in safety contexts for instance. One important aspect of this process is that the assessment of performances heavily depends on *test cases* (or inputs) fed to software systems.

Example 1. For instance, let us consider a Computer Vision (CV) system designed to detect objects. A key performance indicator could be its *precision*, defined as the ratio of correctly identified objects with respect to ground truth. Getting a low *precision* on a given test case (*e.g.,* an image) does not necessarily mean the test case is good (and the CV program is weak), because it might simply be too difficult (*e.g.,* a scene with low contrast and poor illumination conditions resulting in objects being barely perceivable). Conversely, if the *precision* is high, does it mean that the computer vision program is efficient or that the video is simply not very challenging (*e.g.,* just one big, highly contrasted object under ideal illumination conditions) ?

Example 2. Let us consider another engineering context. The discovery of (performance) bugs in generators (*e.g.,* compilers) can be complex. In such a context, what are the most useful test cases (from a test suite) that need to be ran in order to find that a generated program using a particular programming language has performance issues (*e.g.,* unexpected high execution time) ? Once again, we can understand that very simplistic test cases are not interesting as computation times might be too short (*i.e.,* expressed in milliseconds). Conversely, executing unrealistic test cases may result in extremely long execution time or time-out. Here again, we see that what matters is the fact that we are able to discriminate the execution of different programs.

Our general observation is that we cannot assess the performance of a software system solely based on raw and absolute numbers ; *the performance should rather be put into perspective* w.r.t. *the difficulty of the task.* For doing so, we consider two

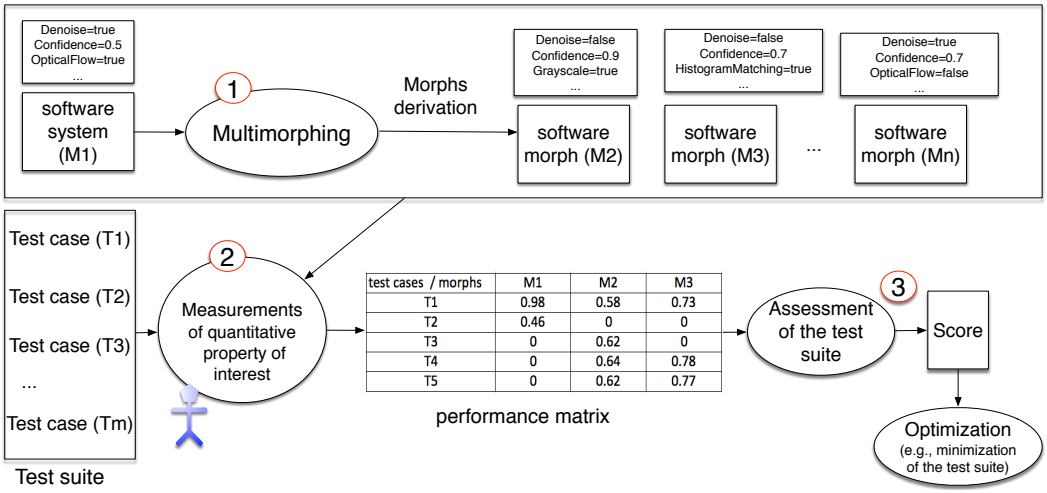


FIGURE 6.1 – Multimorphic process : morphs are automatically produced (*e.g.,* thanks to parameters) ; for each morph test suite is executed and performance measurements are gathered ; a dispersion score is finally computed to characterize the quality of a test suite

main axes. First, a software system should be confronted to other competitors (called *morphs* hereafter) in order to establish the relative difficulty of a task. Second, the quality of a test case (and by extension of test suites) is crucial. A bad test suite may not reveal the underlying difficulty in processing a certain task. We consider *a good test case as one being able to discriminate different program implementations based on their observed performances.* Overall, we aim to characterize the quality of test suites when assessing quantitative properties of a software system.

6.2.2 The principle of Multimorphic Testing

In this section, we describe how we can associate a *score* to a performance test suite [1], based on its ability to discriminate the performance behavior of variants of the same software system (called morphs hereafter).

The core idea of Multimorphic testing is *to evaluate test cases* by varying configuration settings and then comparing their outcomes. Said differently and by analogy with mutation testing : *Are some tests able to "kill" weak morphs ?* Our basic assumption is that a test is "good" when it is able to reveal significant quantitative differences in the performance behavior of software system configurations. Following the same process

1. The score of a single test case is then defined as the score of the singleton test suite made of it.

as for mutation testing, we derive and exploit morphs (instead of mutants) to reveal significant quantitative differences (instead of pass/fail verdicts) and eventually assess the quality of a test suite.

Our method, called Multimorphic testing, proactively produces *morphs* that are all tested with the same set of test cases (*i.e.,* test suite). A morph is a program variant that implements the same functionality and can possibly exhibit performance differences. Morphs typically correspond to different parameterizations of a system or to different implementation choices (choice of an algorithm for video tracking, or a strategy in a compiler) that can be selected at compile time or runtime. It can be done by leveraging software product line automatic derivation techniques [8, 9].

In the example of Figure 6.1, morphs denoted M_1, M_2, M_3, ..., M_n are derived thanks to the settings of parameters' values. For instance, in the case of computer vision systems, all morphs implement the same high-level functionality and realize the same task (for instance, tracking objects in a scene). We use different values for parameters such as Denoise, Confidence, or OpticalFlow, because these can have a significant influence on quantitative property such as execution time or precision. Once morphs are derived, they can be fed with inputs (represented by the test cases on the left part of Fig. 6.1) and their performances (*e.g.,* execution time) are measured for each pair (M_i,T_i). We represented (examples of) performances in cells of the performance matrix of Fig. 6.1 which corresponds also to the one we presented in Fig. 1.1.

Ultimately, we need a measure (or score) that reflects the ability of a test suite to exhibit different performance behaviors of a set of morphs of the same software system.

6.2.3 Properties of a measure

A measure is a function defined on a set which systematically assigns a number to each subset of the set. According to the general measure theory [96, 111], a measure must fulfill the following properties :

- *(P1)* **non-negativity** : the measure associated to a test suite should be ≥ 0
- *(P2)* **null empty set** : the measure associated with an empty test suite is 0
- *(P3)* **monotony and sub-additivity** : the measure associated with the combination of multiple test suites should not be less than the maximum measure associated to each individual test suite. In other words, adding new test cases to an existing test suite should not decrease the measure.

6.2.4 Design of dispersion measures

The three previous properties aim to restrict the design space of candidate measures and in turn eliminate some of them. For example, in this section, we demonstrate that variance, an intuitive and widely used metric, cannot be chosen in our context. We then propose a new measure that we call the *dispersion score* which does fulfill the three properties of a measure.

Variance

Variance is probably the most commonly used indicator when analyzing the spreading of measures. It computes the difference between elements of a set with the mean value of the set and average these differences to produce a value. It is usually described as : $V(X) = \mathbb{E}\left[(X - \mathbb{E}[X])^2\right]$ with X, a set of observations over a random variable and $\mathbb{E}$, the expected value.

Variance could be interesting but it does not meet the third property *(P3)* of a measure (see Section 6.2.3). We give a counterexample in Table 6.1 : adding a new test suite actually *reduces* the variance of a test suite. Specifically, Table 6.1 shows two test suites (Test suite 1 and Test suite 2) both composed of two test cases. Six Morphs (one per row) are executed on each test suite. Each execution yields a value in the range $[0.1; 0.6]$. Let us compute variances of Table 6.1 :

— The variance of Test suite 1 is 0.041 ;
— The variance of Test suite 2 is 0.049 ;
— The variance of Test suite 1 and 2 is 0.043.

The variance of the combination of the two test suites lies in between the two variances of individual test suite. This counterexample shows that *(P3)* is not always met and thus variance cannot be used in our context.

Dispersion score

Instead, we propose to use a *dispersion score* that is based on histograms, as they are one of the most popular ways of evaluating the distribution of a continuous variable [80, 90]. An example of histogram based on values given by Test suite 1 of Table 6.1 is shown in Figure 6.2.

The observed definition domain (*i.e.,* the range of observed values) is presented

Test suite 1		
	Test case 1.1	Test case 1.2
Morph 1	0.2	0.1
Morph 2	0.2	0.6
Morph 3	0.3	0.1
Morph 4	0.2	0.6
Morph 5	0.3	0.6
Morph 6	0.4	0.6

Test suite 2		
	Test case 2.1	Test case 2.2
Morph 1	0.1	0.2
Morph 2	0.6	0.3
Morph 3	0.1	0.1
Morph 4	0.6	0.2
Morph 5	0.6	0.3
Morph 6	0.6	0.1

TABLE 6.1 – An example for showing the inadequacy of variance and illustrating our measure : performance observations gathered for 2 different test suites, each composed of 2 test cases over 6 morphs.

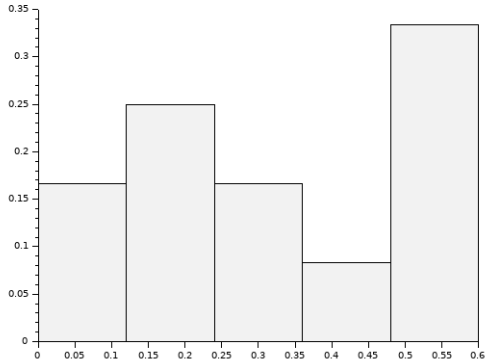


FIGURE 6.2 – An example of a histogram (based on Table 6.1).

on the X-axis, while frequency (between 0 and 1) of observations appears on the Y-axis. From left to right, it reads as follows : $16,5\%$ of observed values lie in the range $[0; 0.12]$, 25% of observed values lie in the range $[0.12; 0.24]$ ... and finally 33% lie in the range $[0.48; 0.6]$. However, we are not interested in the frequency of each value nor the absolute values but rather that at least one observation falls into each sub-range (called bin) of the X-axis. In other words, we want to build a test suite for which the retrieved histogram is as dense as possible (*i.e.,* the definition domain of observations is "covered").

Histograms are parametrized by their number of bins. Bins are defined to gather values which are close from each other (*i.e.,* with insignificant differences). Considering a small number of bins would yield a coarse histogram, gathering values with significant differences. On the other hand, too many bins would yield a very fine-grained histogram, providing more details but likely to separate observations with insignificant differences. As a trade-off, we choose to fix the number of bins as the number of morphs to execute [2]. We then define the dispersion score of a test suite as the proportion of bins activated (*i.e.,* bins with a frequency different from 0) in its histogram.

To make this definition more precise, let's consider n morphs and a test suite made of m test cases. Executing the m test cases on the n morphs yields a matrix M with m rows and n columns where each cell M_{ij} holds the measured value for some property of interest (precision, recall, execution time, etc.), as illustrated in Table 6.1).

When considering performance properties like execution time, it may however be difficult to compare different sets of executions : one test might take a few seconds while others could take minutes or more. We thus need to apply a normalization step over these observations. That is, we consider the extrema of observed performances and, for each performance, we apply the following transformation : $\frac{x-min}{max-min}$ where x is an observed performance, min and max are respectively the minimal and maximal observed performance of a test suite. After this normalization, all performance values M_{ij} are in the range $[0; 1]$.

We now have to compute the histogram for M, which is a vector V of size n (because

2. We admit this choice is rather arbitrary, but empirical assessment already shows interesting results with that choice (Section 6.3). In fact, finding the best number of bins is a well-known issue [81]. Finding the *optimum* in our context remains an open question.

we choose the number of bins to be the number of morphs) such that :

$$V_i = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m}\left(\frac{i-1}{n} \leq M_{ij} < \frac{i}{n}\right)?1:0}{n*m}$$

However we are not really interested in the histogram itself, but only on the proportion of bins that have been activated by at least one test case (*i.e.,* a column with a value different from 0). We thus apply the following transformation on V :

$$V_i = \begin{cases} 1, & \text{if } V_i \neq 0 \\ 0, & \textit{otherwise} \end{cases}$$

Now, the vector only contains 0s and 1s and the dispersion score is :

$$Disp(V) = \frac{\sum_{i=1}^{n} V_i}{n}$$

This way, the best possible test suite would activate all the bins of its histogram (*e.g.,* exactly one observation falls into each bin) ; having a dispersion score of 1, meaning it would be able to discriminate every morph. The worst test suite would have a score close to 0 (exactly 1 over the number of morphs), because all morphs would behave the same, thus filling only one bin. Dispersion score is thus defined in $[0;1]$ *satisfying the first property of a measure*.

The dispersion score of an empty set (*i.e.,* when no test suites are given) is 0 which *satisfies the second property*. To go from M to V, we only used sums which ensures that the number of activated bins cannot decrease as the number of test cases increases. Thus, the dispersion score also *fulfills the third property of a measure*.

6.3 Empirical Evaluation

We have introduced the *Dispersion Score* as a new measure to assess the relative merits of performance test suites. In the following, we want to empirically show that this *measure is right*, and that it is *a right measure* to assess the quality of test suites.

6.3.1 Research questions

Is the measure right ? We consider that the *dispersion score* is right if it is able to effectively exhibit significant differences in terms of observed performances on a set of morphs of a given software system. We also expect sufficient stability *w.r.t.* the set of morphs ; the stability of the measure is evaluated via a sensitivity analysis.

Is it a right measure ? Here, we want to assess the correlation between the actual (relative) effectiveness of performance test suites and their dispersion scores. Depending on the domain, this question will take several very different forms, from helping to select test cases able to reveal performance bugs in code generators, to reducing a test suite without loosing its ranking capabilities.

6.3.2 Evaluation settings

To answer these questions, we applied our method on several application domains and evaluate its results. For each case, we detail : *i)* what are morphs ; *ii)* what are test suites ; *iii)* how performance measurements are retrieved and used ; *iv)* how we perform the evaluation. Table 6.2 summarizes the cases ; we can notice different scales both in terms of available morphs and test cases.

Case	App. Domain	# morphs	# tests
OpenCV	Tracking in videos	252	49
COCO	Obj. rec. in images	52	40k
Haxe	Code generation	21	84

TABLE 6.2 – The three case studies

OpenCV case (object tracking in videos)

In the field of video tracking, the use of large test suites helps building confidence in the robustness of a system and its capability in performing well under various conditions. However, are all those test cases necessary ? We consider OpenCV [3], a popular library, written in C++, and implement different techniques for tracking object of interest in videos.

Morphs. By reverse-engineering a part of OpenCV and using the feature modeling formalism [2, 8, 9, 106], we have elaborated a variability model (including constraints) to formally specify the possible values of parameters and their valid combinations. From this variability model, we automatically sampled 212 configurations by assigning random values to functions' parameters. The configurations are valid *w.r.t.* the variability model and are used to derive 212 morphs.

Test suites. We use a set of 49 synthetic video sequences as a test suite. Videos have been obtained using MOTIV, the industrial video generator we have used in previous chapters [1, 4, 5, 34]. Videos are all different either in the composition of the scene (presence or not of objects we do not want to track such as tree leaves) or in the visual characteristics of the scene (different illumination conditions ; presence of heat haze and/or noise). Importantly, a ground truth is automatically generated along videos stating the position of every encrusted objects in every video images such that we can assess the ability of our programs to track objects of interest.

Measurements. Following the Multimorphic method (see Fig. 6.1), we execute all 212 programs (morphs) on the 49 videos. For each execution, we measure several quantitative properties such as : *precision*, *recall*, the *execution time* and the *CPU consumption* to cite a few. Precision and recall measures are ratios given in the range $[0; 1]$. They are measured by comparing objects' positions computed by morphs to the generated ground truth. Positions are usually defined by bounding boxes that surround objects. Then, we considered an object as being detected if the intersection between the bounding boxes retrieved by a morph and the one defined by the ground truth is not null. The execution time is given in seconds while the CPU consumption is expressed as a percentage of one CPU core usage (if the computations are distributed over multiple CPUs then this measure can be higher than 100%). Note that to stay within realistic computation boundaries, we set a time-out for every executions we launch. If

3. `https://opencv.org/`

an execution exceed this amount of time, its process is killed and its measures are reported as values showing that it has failed (high CPU and memory consumption, zero precision and recall measures, *etc.*). We have considered 13 different quantitative properties for each execution. This yields a total of $212 * 49 * 13 = 135,044$ performance measures.

Evaluation. Since our method yields a dispersion score associated to a test suite, we evaluate that a test suite created to maximize the dispersion score is actually a "better" test suite than random ones with lesser scores.

COCO case (object recognition benchmarks)

For many years, the computer vision community has been building large datasets that are used as benchmarks [25, 38, 58, 71] in competitions to rank competing image recognition techniques. Here we use the COCO dataset on which computer vision competitions are conducted every year since 2014. Results of competitions are presented on the leaderboard webpage [4]. COCO competitions address different tasks (*e.g.,* detection of objects, segmentation of images, *etc.*). Our evaluation focuses on object detection.

Morphs. Even if we do not know much about the specific details of techniques used by competitors, we know that they are all designed to recognize objects in images, which means that they can play the role of morphs when applying our Multimorphic method. It should be noted that, for this case, morphs have not been obtained by parameterization – they are simply existing competing systems realizing the same task and potentially exhibiting performance differences.

Test suites. Competitions using COCO datasets have been running for several years now and each year brings its own dataset. We focus on the 2017 challenge that ended in late November 2017. The dataset is composed of more than 160,000 images. Different object classes are specified (80 in total) and more than 886,000 object instances can be detected. Object classes are gathered into *concepts*. For instance, classes "dog", "giraffe" or "horse" are gathered under a concept called "animal". Similarly, "hot dog" or "carrot" are gathered under the "food" concept. 12 such concepts have been created in total.

To conduct the competition, organizers have decided to split the dataset into two

4. `http://cocodataset.org/#detections-leaderboard`

main subsets : first, the set given to competitors along with associated ground-truth so that they can train their models and also perform a validation step. This subset is composed of 120,000 images. The second set is also given to competitors but without associated ground-truth. It is composed of the remaining images and is used to evaluate competitors and thus to establish their ranking. We use this second subset as a test suite in the Multimorphic testing method.

Measurements. In this study, we focus on the *Average Precision* performance measure [5] available for each technique on the leaderboard. This measure is computed over the second set of images and corresponds to the overlap of bounding boxes from a computer vision technique (*i.e.,* a morph) and the ground truth (which is only known by the server).

The process is the following : (1) each morph is executed on the test suite, generating an output for each test case (*i.e.,* image) ; (2) all outputs of a morph are sent to the server following a format specified by organizers ; (3) for each test case, the server computes the overlap of the outputs of a morph and the ground truth ; (4) based on the overlap, performance measures are updated ; (5) once performance measures are all up-to-date, ranks are computed.

Even if we could have presented results for all 80 object classes, we decided to focus on the 12 concepts for the sake of compactness and exhaustive presentation. However, the method is not changed and conclusions that we present hereafter are similar considering 12 or 80 items.

Evaluation. We consider the ranking computed by the server (available online) as the ranking of reference. Using our dispersion score to rate test suites, we will try to reduce the number of test cases (*i.e.,* images) needed to evaluate competitors' techniques. We then will assess the capability of such a reduced test suite to yield a ranking similar to the original one.

Haxe case (code generator)

Today's modern generators or compilers (*e.g.,* gcc, a C compiler) become highly configurable, offering (numerous) configuration options (*e.g.,* optimization passes) to users in order to tune the produced code with respect to the target software and/or hardware platform. Haxe is an open source toolkit [6] for cross-platform development

5. Note that we could have considered other measures, it would not have affected our method
6. With about 2500 stars on GitHub in June 2018.

which compiles to a number of different programming platforms, including JavaScript, Flash, PHP, C++, C#, and Java. It involves many features : the Haxe language, multi-platform compilers, and different native libraries. Moreover, Haxe comes with a set of code generators that translate the manually-written code (in Haxe language) to different target languages and platforms. One of the main objectives of Haxe is to produce code that has better performance than a hand-written one [94] ; it shows the importance of performance aspects of the code generator.

Morphs. Based on previous works [15, 16], we selected 4 popular target languages (namely C++, C#, Java, PHP). Then, we tuned code generators according to several optimization parameters they provide. More specifically, regarding C++, we chose to apply the different optimization levels available via gcc compiler (O0, O1, O2, O3, Ofast and Os). Regarding other languages, we derived different code generators by toggling different parameters such as dead code elimination, the use (or not) of methods inlining or the use of code optimizations. For each of the generated variants in one target language, we modified one of these parameters ; others are set to default values. In total, we considered 21 *different configurations* of the Haxe code generator across the four target languages.

Test suites. We used the same 84 test suites that were used in previous studies [15, 16]. Each test suite is composed a number of test cases ranging from 5 to 50.

Measurements. We used the same testing environment that was used by Boussaa *et al.* [15, 16], running the same test suites across the 21 morphs, focusing on one property of interest, namely : *execution time*. We thus collected data relative to the execution time of each generated program. To mitigate the fact that measures could vary because of external factors (such as warm-up, or the charge of CPUs), we executed each test case several times on each morph (see [15] for details). Raw measures have been transformed and normalized as follows : (1) Finding an execution of reference for each test cases and set it to 1. The reference execution is defined as the one optimizing the considered quantitative property (*e.g.,* minimizing execution time) ; (2) expressing other observations relative to this test case as a multiplicative factor of the execution of reference. For instance, let us consider two morphs and a single test case. Assuming the property of interest is execution time, if one execution gives an observation of 35 and the other one of 70. Since the goal is probably to minimize the execution time, the first one is the execution of reference. Thus, the measure is transformed into 1 while the second execution becomes 2 as it took twice the time to be executed. Such

transformation has no impact on our proposed solution, since we are not interested in the actual values, but their dispersion. After that first step, we carried on performing a normalization in the range $[0; 1]$ to build our histograms and compute dispersion scores as explained in Section 2.

Evaluation. Here our criterion will be whether the dispersion score is helping us to select test cases able to reveal performance bugs in code generators.

Presentation of results. Hereafter, we present results for each research question using the three case studies, showing that the same method can be applied to various domains.

6.3.3 RQ1 : Is the dispersion measure right ?

Videos	Prog. 1	...	Prog. 212	Dispersion
vid 01	0.683228	...	1	0.203
	...			
vid 35	0.000396	...	0.001709	0.118
	...			
vid 49	1	...	0.177966	0.080

TABLE 6.3 – Sample of observations for precision on the OpenCV case

Table 6.3 shows a representative excerpt of *precision* measures that were observed considering the OpenCV case (similar tables for remaining examples are available in Appendix 7.2.4). In this table, rows represent test cases and columns are different morphs. As stated before, we used 49 test cases that were executed on 212 morphs. Each cell of the table reports the performance measure of the execution of the program on the video. Based on all retrieved performances for each video, we computed a dispersion score for each individual video which is presented in the last column.

Can different dispersion scores be observed ?

In the following, we want to validate the fact that we can observe variations in performances inducing different dispersion scores.

OpenCV case. Computed dispersion scores range from $\frac{17}{212} \simeq 0.08$ up to $\frac{44}{212} \simeq 0.207$. Over all 49 test cases, the mean value of dispersion scores is $\simeq 0.145$ and the standard deviation $\simeq 0.034$. While these numbers are rather low (*i.e.,* less than a quarter of the bins are activated), it seems that behaviors of all 212 programs are not equivalent as shown in Table 6.3. Meaning that not all programs give the same performance when executed on the 49 videos [7]. For instance, Prog. 1 from Table 6.3 performs very well on the last video while Prog. 2 is unable to detect anything.

COCO case. In this case, dispersion scores are larger (as shown in Appendix 1). Scores range from 0.308 to 0.423. Among all 12 concepts, the mean value of dispersion scores is $\simeq 0.359$ and the standard deviation $\simeq 0.040$. Performances of competitors over each concept are available directly on the online leaderboard. Concepts "Electronic" and "Sports" are tied with the maximum dispersion score (0.423), concepts "Indoor" and "food" are also tied but they are associated with the lower minimum score (0.308).

Haxe case. Dispersion scores from Appendix 2 range from $\frac{1}{21} \simeq 0.047$ to $\frac{3}{21} \simeq 0.143$. Over all 84 test cases, the mean value of dispersion scores is $\simeq 0.0567$ and the standard deviation $\simeq 0.0202$. These scores are low, showing consistent measures and insignificant differences in the observations. However, those numbers do not indicate the quality of test cases per se. In fact, most of dispersion scores show that only one or two bins are activated in associated histograms. In the case where two bins are activated, we assume that retrieved observations lie close to the separation between the two bins. Variations make observations fall sometimes on one side of the separation and sometimes on the other side. Only one test case (coming from the core test suite) is associated with the highest measures dispersion score. In this test case, three bins are activated due to variations. We will investigate deeper this situation in the next research question.

Taking a step backwards, from the different examples we analyzed, variations in performances can be shown and captured by the use of *dispersion score*. Every test case can give a dispersion score that is more or less high depicting how different morphs can perform differently on the same test case.

7. We obtained similar results for the other quantitative properties we have measured (such as recall, performance, etc.).

Is dispersion score stable and sensitive to the set of morphs ?

Our hypothesis is that the dispersion score associated with test cases only slightly changes depending on morphs considered in the set that is used. In other words, we want to perform a sensitivity analysis about the dispersion score.

To conduct this experiment, we randomly remove some morphs out of the original pool. That is, we only build dispersion score taking into account the measures coming from remaining morphs. Taking back the OpenCV case : at each iteration $it \in 0..100$, we remove it morphs from the original set of morphs and observe the effect on dispersion scores for each 49 videos. The removal of 100 programs boils down to the removal of about half of our observations. The whole process of selecting up to 100 programs and computing dispersion score is repeated 50 times in order to flatten the impact of random choices in the removal of the morphs. Algorithm1 describes how measures have been retrieved.

Algorithm 1 Procedure to assess stability of the method

(1) $current_iter = 1;$
(2) $max_iter = 50;$
(3) $\#_morph_remove = 0;$
(4) $\#_max_morph_remove = 100$
while $\#_morph_remove = 0 <= \#_max_morph_remove$ **do**
  for all $videos\ in\ the\ set\ of\ videos$ **do**
    while $current_iter <= max_iter$ **do**
      (5) select randomly $current_iter$ morphs to remove ;
      (6) compute dispersion score w.r.t. remaining morphs ;
      (7) store and average dispersion score ;
      (8) store best dispersion score ;
      (9) store worst dispersion score ;
      (10) $current_iter++$;
    end while
    (11) $current_iter = 1$;
  end for
  (12) $\#_morph_remove++$
end while

OpenCV case. Fig. 6.3 shows the evolution of the dispersion score of two videos (*i.e.,* with the best and worst score) depending on the number of morphs that have been removed. On this figure, the X-axis represents the number of morphs that have been removed (from 0 to 100) and the Y-axis represents the associated dispersion scores.

Six curves are plotted :

— the three top curves represent results for the video providing the best dispersion score ;

— the three bottom curves represent results for the video with the worst dispersion scores.

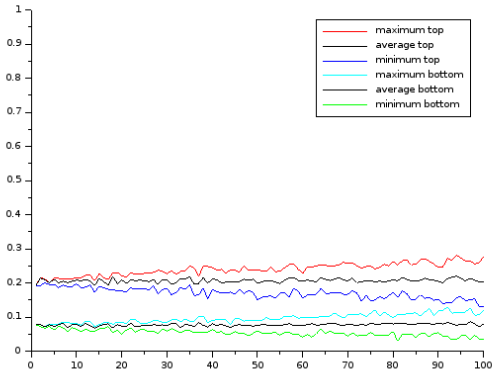


FIGURE 6.3 – Stability results for the property precision ; X-axis : number of morphs removed ; Y-axis : dispersion score

Specifically, considering the best video : (1) The curve in the middle, called average top, reports the average of the dispersion scores (it is obtained through line 7 of Algorithm 1) ; (2) the top curve, called maximum top, reports the maximum dispersion score (see line 8 of Algorithm 1) ; (3) The so-called minimum top reports the minimum dispersion score (line 9 of Algorithm 1). We depict the same curves for the worst video.

For each curve, we can notice that the results are stable. The dispersion score variations are less than 10%. The average curves (average top and average bottom) are very stable. The four others (maximum top, minimum top, maximum bottom and minimum bottom) tend to be noisy once fifty morphs or more have been removed. However, despite those variations, none of the curves between the top plots and the bottom ones cross each other. Overall, the results show a stability and consistency in their positions [8].

COCO case We apply the same process on the COCO dataset but because there are only 52 morphs available, we remove up to 26 morphs instead of 100 which corres-

8. Again, similar observations can be made for other quantitative properties (*e.g.,* recall, performance or even a composition of different properties).

ponds to remove half our observations. Conclusions are similar to the previous case and dispersion score remains stable and consistent in their positions.

Haxe case We run again the same sensitivity analysis over Haxe. For this case, as we only have a small number of morphs, we remove only up to 10 morphs over the 21 available. Retrieved curves are stable as plateaus appeared. Similarly to previous cases (*e.g.,* Fig. 6.3), top curves and bottom curves never interchange.

6.3.4 RQ2 : Is the dispersion score a right measure ?

We showed that our dispersion score was able to "rate" test suites with different scores. In the following, we would like to evaluate whether those scores have the desired intuitive meaning : is a test suite with a higher score really better than one with a lesser score ?

For addressing this qualitative question, we first used an exhaustive search to create an optimal test suite (according to their dispersion score) of exactly 5 test cases for each of our 3 cases studies. We then relied on domain specific ground truth to assess whether these "optimal" test suites were indeed any good.

As the newly built test sets (of 5 test cases) maximize the dispersion score, we can compare histograms from the original test set to this one and observe how far they are one to the other. For instance, taking the Haxe case, the histogram of the original test set (using 84 test cases) is presented in Fig. 6.4. Blue light bars represent activated bins. Fig. 6.5 shows the histogram of the second test set. Red bars represent differences in activated bins.

Only 4 bins are different (*i.e.,* bins 3, 6, 18 and 21) between the two histograms. Meaning that, at most, 4 test cases are needed, in the reduced test set, to retrieve the same histogram as the original one. In the end, with those hypothetical 9 test suites, we ensure to retrieve the same diversity in the observed quantitative properties but drastically diminishing testing efforts.

(OpenCV case) Can we create a "good" test suite that is able to differentiate morphs that perform well from others ?

The selected "optimal" test suite of 5 test cases was associated with a score of $\simeq 0.6$ activating 127 bins over 212 in total. From this perspective, this "optimal" test suite is at least 3 times better than any individual test case. But is it really any good ?

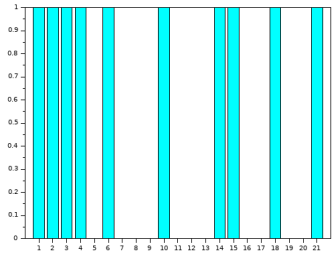


FIGURE 6.4 – The number of bins activated with the original test suite (composed of 84 test cases)

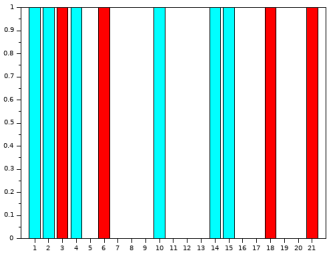


FIGURE 6.5 – The number of bins activated with the smaller test suite composed of 5 test cases that maximizes the dispersion score.

FIGURE 6.6 – On the X-axis are the index of the bins. On the left, the original histogram when all test cases are taken into account. On the right is the histogram associated with our smaller test suite. Bars in blue represent activated bins of histograms. Bars in red are bins that are activated with the original test suite but that we fail to activate with our smaller test suite.

To answer that question, we have asked a computer vision expert to cherry-pick twelve new morphs in such a way that six of them are expected to perform well on average and six others would be likely to perform poorly/moderately well. Note that we did not ask the expert to choose very bad configurations (*i.e.,* that would not recognize anything) : since any test case would be able to tell that they are bad ; that would tell us nothing about the relative merit of our test selection process.

Then we ran these 12 morphs on the test suite of 5 videos. For each of these morphs, we plot in Fig. 6.7 the obtained precision averaged over the 5 executions. The supposedly 6 moderately poor morphs correspond to index 1 to 6 on the X-axis while indexes 7 to 12 correspond to the 6 others (supposed to perform well).

From Fig. 6.7, two classes of programs can be identified : programs which have an average score below 0.4 and programs which reach an average score above 0.5. This separation corroborates the expert's classification since programs which reach an average *precision* above 0.5 (respectively below 0.4) are exactly the ones expected to perform well (respectively moderately poorly).

Even if our selected test suite is probably not the best possible one (it is just the best set of exactly 5 test cases), it is already quite able to tell good from poor configurations apart.

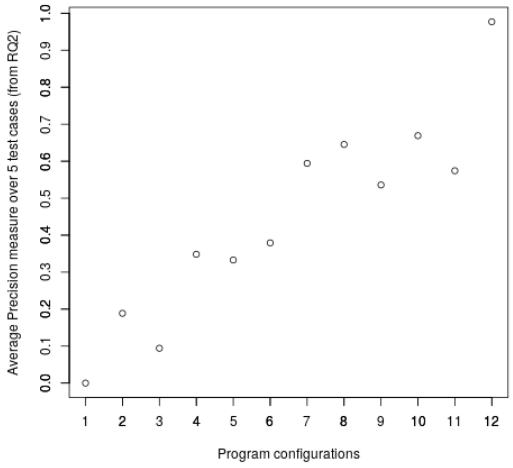


FIGURE 6.7 – Average precision measures over the 5 videos from RQ2. On X-axis are the CV morphs : first, the 6 first morphs that are supposed to perform moderately badly ; the 6 last morphs are supposed to be good. Y-axis reports the averaged *precision* measure over the test suite.

But is this test suite really better at that than any random test suite of size 5 ? To answer this question, we created a test suite composed of 5 videos randomly picked among our 49 videos. We ran the 12 selected programs on this randomly picked test suite. For each program, we computed the *precision* measure averaged over the test suite (similarly as presented in Fig. 6.7). Then, we confront the classification of morphs (depending on whether the average of their performance is above 0.5 or below 0.4) with the intuition of the expert and count how many times they disagree. To mitigate the potential bias induced by random picking, we run this process 10 times. The result is that a minimum of 2 programs out of 12 have been misclassified, with a worst case of 5 misclassified programs.

In average, over the 10 runs, almost 4 programs over 12 are misclassified (with a standard deviation of almost 1). As a conclusion, we got some evidence that our optimal test suite of 5 videos performs significantly better that a random one of the same size (and a lower score) to tell good from poor configurations apart.

Concepts	Dispersion score
accessory	0.673
animal	
appliance	
electronic	
food	

TABLE 6.4 – The 5 concepts that maximize the dispersion score

(COCO case) Can a smaller test suite built such that it maximizes its dispersion score provide a similar ranking of morphs as the original test suite ?

Again, we created a reduced test suite containing 5 concepts following the same process as before. The 5 concepts are presented in Table 6.4 and yield a score of 0.673 (which is very close to the dispersion score of the entire COCO benchmark). Note that, once again, the choice of these 5 concepts are not the same as picking the 5 top rows of Appendix 1.

With this new smaller benchmark, we rank again the competitors. We check that the two rankings are similar in order to assess that not all categories are needed when performing continuous evaluations of morphs' performances. The similarity between the two rankings is established using the Spearman correlation coefficient. This coefficient indicates whether two ranked lists are strongly correlated when the value is close to 1 or -1, showing whether the evolution follows the same tendency or opposite directions. On the other hand, if the coefficient is close to 0, then no correlation can be deduced.

Appendix 3 provides a comparison of performances of a sample of competitors' techniques *w.r.t.* the two benchmarks. The first column presents competitors' names. We selected 13 techniques out of the 52 competitors that are shown on the leaderboard. That is, we show approximately one out of five techniques. The second column shows the performances provided by the leaderboard. We assume such performances to be an average over all 12 concepts.

The third column shows the averaged performances over the set of 5 concepts (see Table 6.4) that we retrieved from our method. Finally, the last column shows the absolute difference between the two performances. The differences are rather low which shows how close we are from the original measures but considering fewer data.

We also computed some statistics over all 52 techniques which compare the differences between the two set of performances. The average difference in performances

over all 52 techniques is ≈ 0.013 with a standard deviation of ≈ 0.004. This shows that, most of the time, differences in retrieved performances are in the range $[0.01; 0.02]$ approximately. We also searched for the maximum and minimum differences, they are respectively ≈ 0.025 and 0.008.

Regarding ranks directly : Appendix 4 shows two rankings on the second and third columns. First, the one retrieved from the COCO leaderboard. Second, the one we computed considering our reduced benchmark.The two rankings are similar with only some ranks that are permuted with the one above or below. The result Spearman correlation coefficient of 0.998 shows a strong correlation between the two rankings.

This strong correlation shows that a small subset of test cases selected via the use of dispersion score is nearly as powerful as the full test suite to rank competitors. The concrete consequence of that is out of all concepts in COCO, only a smaller number is needed to assess the global behavior of competitors. In the end, their continuous evaluations could be reduced to assess performances on a smaller number of concepts such that competitors can get an idea of their rank in the competition more quickly, with fewer resources which in turn could help them improve their solutions more quickly and more frequently.

(Haxe case) Can we discover bugs thanks to the dispersion score ? Can we build a smaller test suite that is able to select interesting test cases ?

For this case, we want to check that a higher dispersion score might be correlated with the detection of performance bugs. In particular, we focus on the test case with a maximal dispersion score (*i.e.,* with 3 bins activated).

Because most dispersion scores activated one or two bins, we assumed that performances observed lied close to a boundary between those two bins and little variations caused observations to lie sometimes in one bin and sometimes in another. However, when three bins are activated, it is unlikely that little variations could cause this behavior. There must be something else : we analyzed observations of the test case associated with the highest retrieved dispersion score. Retrieved performances were rather consistent except for code generator variants targeting the PHP language. Performances regarding those morphs drastically increased. In fact, execution times were at least 40 times longer than for morphs targeting other languages. Checking results with authors from previous work [15, 16], they also have noticed this anomaly and reported it to Haxe community in a bug report. Developers responded that they knew

about it, it was fixed already but the patch was not live when Boussaa *et al.* conducted their study.

6.3.5 Concluding remarks over the method

In light of our results, we can answer to the research questions :

Performance diversity (RQ1). *Is our dispersion score able to capture different performance values and thus assign different scores to different test cases ?* Results showed that we are able to build a dispersion score that assigns a higher value to test cases able to capture significant differences in performance values. Giving an overview of results we retrieved : the OpenCV case showed dispersion scores ranging from 0.08 to 0.207. The minimal score of 0.08 is assigned to a test case on which most of OpenCV morphs performed similarly. On the other hand, with a score of 0.207, about a fifth of morphs performed significantly different on this test case. Regarding COCO and Haxe, dispersion scores ranged respectively from 0.308 to 0.423 and from 0.047 to 0.143. For every case studies, dispersion scores were able to differentiate test cases by assigning them different dispersion score whether they are able to capture more or less different performance values.

Stability (RQ1). *Are dispersion scores very sensitive to the use of particular morphs ?* Our sensitivity analysis showed that dispersion scores were rather stable regardless of used morphs. Hence our method is able to deal with morphs having similar performances ; we can certainly avoid the costly use of some equivalent morphs. It also suggests that our method is robust even when the selection of morphs is realized in an agnostic way (*e.g.,* randomly). However, we cannot claim that domain knowledge will not be beneficial to our method (*e.g.,* for selecting optimal configurations and morphs, see discussions in Section 6.4).

Applicability (RQ2). *Can dispersion scores be correlated with an external evaluation (for each case) of what would be "good" test cases ?* We showed that, dispersion scores can be used to rank test cases in order of importance *w.r.t.* the different performance values they are able to capture. They can also be used to build smaller test suites that will maximize this score. With smaller test suites, we were able to :

— execute a set of **OpenCV** morphs (selected by a computer vision expert) and match the intuition of the experts regarding morphs' behaviors. That is, the 6 morphs supposed to perform better according to the expert were indeed the

ones performing the best on our smaller test suite composed of 5 test cases ;
— retrieve a similar ranking of **COCO** competitors as the original one (available online) with a strong Spearman correlation coefficient (*i.e.,* 0.998) between the two rankings. Our test suite took into account 5 of the 12 concepts originally present in the COCO dataset ;
— exhibit a bug in generated PHP code with only 5 test cases out of the 84 composing the original test suite.

6.3.6 Reproducibility of experiments

Data, code and results are available in a public repository on Github at the following link : `https://github.com/templep/TSE_MM_test.git`.

For practitioners interested in reproducing the analysis of our data, we provide all configurations, test suites and observations through CSV files. Statistical results presented this article are available through text files or plots. The code for producing such results is written in Scilab, an open-source software (close to Matlab) and is also included. For practitioners interested in reproducing the computation of performance data (OpenCV and Haxe cases), we provide specific instructions as well as the code to instrument the whole process.

6.4 Discussions and Threats to Validity

6.4.1 Internal threats

To compute our dispersion score, we used histograms that provide information in a 2D graphical representation. Despite not being interested in the actual frequency value of each bin, we need to know which one are activated to compute our dispersion score. While the frequency representation is questionable, histograms are a common representation which provides information that we need even if not fully exploited.

On the X-axis, histograms define bins and their number is crucial in our method. Defining the right number of bins remains an open question since it depends on the application : trying to analyze a color distribution of an image, comparing two different data distribution requires a very fine-grained analysis and thus a larger number of bins ; while, in our case, requirements are different. Using a small number of bins would provide a coarse analysis of the variability in the results while more bins might isolate every execution into its own single bin and thus would show differences that are not significant. We fixed the number of bins to the number of used morphs as a reasonable trade-off, but this is only true if the number of morphs is large enough.

In RQ2, we validate the usefulness of our dispersion score by building test suites composed of 5 test cases that maximized the dispersion scores. 5 is an arbitrary value chosen to limit the amount of time taken by the exhaustive search. We are aware that, depending on the application domain, the number of test cases to consider may vary and 5 is certainly not the optimal number to use in every occasion. However, test suites that we created were already quite effective despite their small number of test cases.

In the end, our dispersion score, the histogram representation and exhaustive search consist in only one instantiation of the Multimorphic testing approach. Even if it seems to work surprisingly well (at least for the different application we have considered), other options might perform better and need to be explored.

6.4.2 External threats

Applicability. Can Multimorphic testing and our proposed dispersion score be used in different domains ? Our experiments took three different application domains (*i.e.,* tra-

cking of objects in videos, object recognition in images and program generation). The method was able to detect test cases emphasizing interesting behaviors of morphs. While two application domains are rather close, associated tasks were different. Results presented in Section 6.3 mitigate this first threat as it shows that, at least in presented situations, Multimorphic testing can be applied.

Performance dimensions and metrics. About generalization, we have presented results regarding only one performance measure (*i.e.,* precision or execution time) at a time. Further experiments have been running taking into account other measures (such as recall or memory consumption or even a combination of precision and recall or else), similar conclusions were drawn from those extra experiments but we do not show them in this document as it would not provide more insight about the method. They are available on the companion GitHub repository though.

On test suites (OpenCV). Regarding the OpenCV case specifically, test sets were composed of synthetic videos only. The merit of synthetic videos is that (1) the associated ground truths are of high quality (by construction since they are synthesized) ; (2) we can better control the properties of the videos and thus increase the diversity of situations. Synthetic videos are getting more and more used in the industry or in research as a substitute or complement of real assets [34, 47, 72, 86]. However, natural videos may not provide as diverse behaviors and performance measurements as we observed. Note that other experiments (*e.g.,* the COCO case) used "natural" images and still gave fairly good results which mitigates this threat.

On test suites (COCO). Focusing on the COCO data set, we did not have access to raw images ; we only considered concepts and classes of objects. It seems realistic to assume that object classes are not equally represented over all the images of the dataset. For instance, there might be fewer objects labeled "hair dryer" than objects labeled "cat". A hypothesis is that classes that are more represented might have more chances to provide "extra diversity" and thus to show differences among competing approaches. Results we present in Section 6.3 might be biased as they could simply reveal that the reduced set of 5 concepts is composed of the 5 most represented ones. However, it is only a hypothesis that does not necessarily hold in general, *i.e.,* there is not necessarily a correlation between the size of the data set and performance diversity. In the context of COCO, we can hardly validate this hypothesis since concepts are

coarse abstractions averaging performances of large subsets of images. In any case, we have shown that our method is able to retain the key elements of the dataset that exhibit diversity within the performances of competing systems.

Data preprocessing. In the Haxe case, our evaluation is based on results from prior measures (on the same Haxe code generator). Measures that we retrieved were preprocessed (as explained in 6.3.2) which might exacerbate or alleviate differences in observed performances. However, the fact that we were able to retrieve a test case that highlighted the presence of a bug in PHP generated code is encouraging. Furthermore, the fact that measures were consistent, with only one or two bins activated, shows that the dispersion score is somehow invariant to this preprocessing. Further investigations should be conducted in order to understand which kind of preprocessings do not affect drastically the dispersion score.

Morphs' selection The ability of Multimorphic Testing to observe different performance behavior is dependent of the nature of morphs. Using only very similar morphs (with only a small delta in the value of one parameter) might not be enough to observe differences while the morphs will be different in their configurations. Used morphs should be somehow representative of the whole population of morphs of a system. This is an assumption that we used for all our experiments : we have generated morphs of OpenCV for a specific task with the goal of exploring the configuration space as much as possible ; we considered competitors to the COCO competitions to be representative of the state-of-the-art techniques in terms of object recognition ; and similarly for the Haxe case, we considered that target languages and optimization options to be representative of what users might look for.

The underlying problem is how to sample those morphs efficiently ? This remains an open problem in the Software Product Lines community that is addressed by several works, mainly for finding functional bugs [24, 62, 68, 79].

6.5 Conclusion

We applied multimorphic testing to assess the effectiveness of a test suite in revealing performance weaknesses of different systems. We showed that our method can be applied for quantitative properties such as precision, recall or execution time. The core idea was to generate system variants (called morphs) by varying their parameters' values and to check whether it makes any difference on the outcome of test cases in terms of such quantitative properties. Intuitively a "good" test has a good discriminating power over the set of morphs. Conversely, a "bad" (or useless) test returns more or less the same results whatever the morphs. We proposed to use the *dispersion score* to embody this intuition, and have empirically shown over 3 different applications its applicability. Also, we have shown its usefulness regarding different goals that are detailed for every application. Above all, thanks to our method, we can envision to remove unnecessary, redundant test cases from test suites, or improve existing test data sets.

Future work first includes investigating other dispersion metrics, since we do not claim it is the best possible one. Also, we would like to pursue the idea of using this method in different contexts. We could use different code generators and compilers (*e.g.,* ThingML, Num, TypeScript) but also try to investigate new domains like databases. Other candidate domains are highly configurable systems featuring some form of recognition (*e.g.,* video or speech recognition) or more generally any software applications where quantitative properties are of prior importance. Another obvious research direction would be to combine multimorphic testing to well-known test selection techniques (*e.g.,* search-based techniques) in order to concretely build optimal test suites, thereby providing cheaper and better test suites than current hand-crafted benchmarks.

CHAPITRE 7

CONCLUSION AND FUTURE WORK

This chapter concludes this thesis by summarizing contributions and highlighting some points that could be developed in the future.

7.1 Conclusion

Modern software are configurable in the sense that their behavior can be tuned (via the use of parameters and options) to meet users' defined requirements. However, the number of options and parameters is getting out of hand fostering the generation of tons of variants that are specifically tuned to meet particular goals. Considering a specific user and associated goals, a number of variants are unlikely to meet these goals. Thus, it can be hard to find a configuration leading to a variant that actually meet requirements.

On the other hand, variants' performances are assessed by executing configured systems in various settings. Such assessment demands time and energy yelling to a need to minimize test sets. But, how to choose a priori (*i.e.,* before any executions) which test cases to discard ?

We conceptualized these two aspects (*i.e.,* regarding the selection of program variants and test cases) as a matrix in which a cell corresponds to the execution of a program variant on a test case and contains observed performances. Each of our main contributions aims at shrinking the size of the matrix along either dimension.

We proposed to use a machine learning based specialization approach in order to automatically reduce the number of configurations that a user has to consider when choosing a variant. The approach relies on the prediction capabilities of machine learning algorithms that will exploit configurations' similarity and their observed performances to build a model that will decide whether a configuration should be kept or not. Based on this model, constraints are created and added to the variability model

underlying the configurable system, resulting in automatically discarding configurations having a high probability not to meet users' requirements. It succeeded in producing understandable constraints (helping practitioners to understand interactions between features of the variability model) and introduced few classification errors while considering a relatively low number of examples to train the model. We extended this approach by focusing on user-defined requirements which make the use of machine learning possible. We have further analyzed how the definition of such requirements may impact the performances of machine learning techniques in turn leading to more or less helpful constrained software product lines. With both works, this approach has shown to work well in various settings. We applied it to different configurable systems, considering different performance objectives and different performance measures.

The second contribution proposes a new method to reduce the size of a test suite based on observed performances on different program variants. This method, inspired from metamorphic testing, is based on the assumption that a good test case is a test case that is able to highlight differences in the behavior of product variants coming from a unique software product line. Thus, the more significant differences are shown by a test case, the higher the chance it has to be kept in the final, reduced test suite. This method is also interesting, despite its initial cost (*i.e.,* tests have to be made to be able to observe performance differences), when more variants will be derived in the future as the test suite will be reduced, saving time in the end. In its implementation, the method gives a score to each test case of a test suite which creates an order. Users can define a threshold or select a number of test cases according to budget allocated to test. We proposed to use the dispersion score which measures how different observed performances are. It relies on the use of histograms and the definition of bins, if two observations fall into different bins, they are considered as significantly different.

In the following, we discuss different points and several thoughts that could help improve the work we have presented in this thesis.

7.2 Perspectives

7.2.1 Machine Learning, Variability and Software Product Lines

In this thesis, we applied machine learning to configurable systems and variability of systems. Still, we mentioned in Chapter 2 that machine learning algorithms contain

numerous parameters. They have to be tuned for every specific applications in order to maximize classification performances. Therefore, we can consider machine learning algorithms as being part of a software product line. However, a few works have also studied variability of machine learning processes. Camillieri *et al.* [18] proposed a model-based approach to help users define their machine learning framework. It results in a general framework that guides users to create a machine learning process suited to their specific needs. This shows that machine learning can be a playground for variability and software product lines methods which can be very interesting. Machine learning is a domain that evolves fast and encompasses a lot of different techniques that have their own particularities (in the task they tackle, in the way that they tackle problems, in the kind of data they take as inputs and their representations, *etc.*). All these aspects can bring challenges in modeling these techniques using variability models.

In the end, taking a user perspective, modeling machine learning techniques as a software product line can be interesting to produce reusable tools, easing the creation of new machine learning based systems as they are likely to be based on already developed functional bricks. Furthermore, it may help users to select a configuration that is likely to meet their performance requirements more quickly instead of using brute force methods in order to fine-tuned parameters of the algorithms.

Nonetheless, brute force methods are a first step. For instance, in 2015, ChaLearn [1] organized a challenge called AutoML [45] asking to create automatic machine learning systems that are able to automatically self-organize their pipeline and adapt it to different tasks. From this challenge, other work [32, 52, 65, 103] pursued in the automatic selection of a model and its hyper-parametrization. While most of these works have a try-and-error approach, they have to deal with the variability of machine learning models that can be created. Thorton *et al.* [103] have focused on applying this strategy on the Weka software : a very popular software that gathers a large range of machine learning techniques. Kotthoff *et al.* [52] proposed an improved version of autoWeka while Feurer *et al.* [32] preferred to focus on scikit-learn, a Python library gathering multiple machine learning algorithms, similarly to Weka, and that is probably more popular nowadays. Finally, Mendoza *et al.* [65] propose to adapt the concept and applies it on neural networks that are growing in interest with recent advances in deep learning.

From an academic point of view, it can be interesting to produce a Machine Learning

1. an organization running challenges, every year, around different aspects of machine learning tasks and its framework.

Product Line that would gather existing machine learning techniques and that could be easily extended to encompass new techniques to be created.

7.2.2 Developing an appropriate sampling method

Work presented in this thesis rely on sampling the variability space of a system. We have proposed to use a simple random procedure to select configurations but it is not really efficient in exploring the configuration space, especially when Boolean and numerical values are mixed in the variability model or when constraints are expressed (diminishing chances to explore some nested configurations).

A better sampling technique that is able to explore different facets of the configuration space would help having a better overview of systems' performances, in turn, improving prediction performances of machine learning techniques since the entire configuration space would be covered.

Novelty search based algorithms [57] or bacteriological algorithms [11] could be used since they try to produce diverse set of data that may try to cover as much space as possible. However, other problems remain : since the number of possible configurations is very large, the number of configurations needed to build an accurate view of performances is likely to require lots of configurations. In addition, novelty search and bacteriological techniques may be resources demanding, taking a long time and a large amount of power to establish a set of configurations.

The benefits of building such a diverse set that covers as many parts of the original space as possible need to be evaluated and compared to other approaches that may be less accurate but much faster to compute.

We have the certainty it is important to explore the configuration space for various reasons. In particular : from the users' perspective, to be aware, as much as possible, of the range of possible performances that can be reached ; from the engineering point of view, avoid unwilling behaviors from the machine learning (*e.g.,* too much extrapolation between configurations or connecting two sub-spaces that are very different *w.r.t.* configurations that they contain).

7.2.3 Adversarial Machine Learning and Software Product Lines

Adversarial machine learning is a trend in the world of machine learning that is getting evermore popular. The initial context of this field was to study how and why machine learning algorithms could be fooled in such a way that they produce a lot of prediction errors. By doing so, machine learning based systems can become unusable and, with recent advances in autonomous vehicles, this behavior can have tragic consequences.

We do not place ourselves in such a context. However, the idea seems interesting to use adversarial machine learning techniques in order to better understand the distribution of configurations of a software product line.

With the use of adversarial machine learning, we foresee that we could select new configurations that are valid *w.r.t.* the variability model but that would not match users' point of view (*i.e.,* being not interesting). Getting back to the video generator used in Chapter 4, we saw that the notion of interesting videos (for an object tracking algorithm) was not easy to define. Using adversarial machine learning on the model given after training might create new video configurations that are supposed to be valid but once derived into a video sequence will be blurry or contain too many distractors, *etc.* After being derived, we could use these new configurations to create new constraints that could be added to the variability model and enter an iterative process in which adversarial machine learning selects new configurations that have to be discarded, generating new constraints *etc.* until configurations would always fit users requirements.

A major problem in this approach is that current adversarial machine learning techniques are applied on derivable machine learning models which is not the case of decision trees (and other machine learning techniques). A simple solution would be to move from decision trees to other derivable machine learning algorithms. However, derivable machine learning techniques (such as support vector machines) might not produce constraints which are easily understandable by users. This might be a problem when trying to reason about interactions between features. An other solution might be to use two different machine learning algorithms : a decision tree and a derivable machine learning technique and make them co-evolve.

7.2.4 Taking into account the surrounding context

In this thesis, we have emphasized the fact that everything had to be done *w.r.t.* a task, for a specific reason, taking into account what users wanted, *etc.* However, it is really hard to take all of that into account despite potentially having crucial importance on the outcomes of studies.

For instance, taking the example of Haxe and Multimorphic testing (see Chapter 6), we have emphasized that test cases had to be executed under the same environment (*i.e.,* same machines, *etc.*). Another example is performances that can be observed when testing a configuration of the x264 software to encode videos. Usually, performances are measured in a specific context using a specific input, however, both aspects have their importance. Considering x264, inputs are videos and their content might activate (or require to activate) specific options which will impact performances. On the other hand, encoding a video on the fly for streaming or not necessitating a real-time encoding, using a laptop machine to encode or a distributed architecture, using GPU processing to parallelize computations or using only CPU are all aspects that need to be evaluated as well and that are used in different contexts. As we said, configurations are tuned to answer specific needs, these needs can come from users' requirements but they are also driven by the context in which they are executed.

What would be interesting is to develop an approach allowing to map software configurations to specific contexts. Doing so, knowing the context of execution could improve the selection process by proposing users partial configurations of a system such that they can focus on options and features that matter.

We have discussed about such approach in [100]. We foresee a similar approach as we proposed in Chapter 4, however, a major challenge will be to take into account even more heterogeneous pieces of information, requirements, *etc.* and make a link between them and features of the variability model representing the system. The particularity was that we proposed to consider the context and the system (that can be modeled via two different and separated variability model) as one such that bridges can be created when trying to learn how features from one side impact the other.

Appendices

Concepts	Dispersion Score
electronic	0.423
sports	0.423
animal	0.404
appliance	0.365
kitchen	0.365
person	0.365
accessory	0.346
vehicle	0.346
furniture	0.327
outdoor	0.327
food	0.308
indoor	0.308

TABLE 1 – Dispersion scores for all concepts in the COCO dataset

Test Cases	Prog. 1	Prog. 2	...	Prog. 21	Disp.
tc 01	1.1425	93.4543	...	1	0.143
tc 02	15.3550	1	...	1.4269	0.095
	...				
tc 35	1	1.0634	...	1.5155	0.047
	...				
tc 84	1.0255	18.4676	...	1.0117	0.095

TABLE 2 – Sample of Execution time observations regarding generated Haxe programs. (similar table regarding other examples are available in appendixes).

Concepts	Init. Perf.	Comp. Perf.	Diff. Perf.
Megvii (Face++)	0.526	0.539	0.013
bharat_umd	0.482	0.494	0.012
IL	0.420	0.430	0.010
umd_det	0.408	0.416	0.008
Deformable R-FCN	0.375	0.391	0.016
HRI	0.367	0.392	0.025
Imagine Lab	0.357	0.372	0.015
CMU_A2_VGG16	0.324	0.338	0.014
Ttester	0.294	0.312	0.018
CMU_A2	0.256	0.276	0.020
drl	0.235	0.247	0.012
1026	0.178	0.196	0.018
IRONYUN	0.153	0.154	0.001

TABLE 3 – Excerpt of competitors' performances differences between the original benchmark (Initial Perf) and our reduced set of 5 concepts (Computed Perf)

Techniques' names	Init. Rank	New Rank
Megvii (Face++)	1	1
UCenter	2	2
MSRA	3	3
FAIR Mask R-CNN	4	4
Trimps-Soushen+QINIU	5	6
bharat_umd	6	5
DANet	7	7
BUPT-Priv	8	8
DL61	9	10
DeNet	10	9
IL	11	12
G-RMI	12	11
VCA	13	13
LDL	14	14
PingAn AI Lab	15	15
umd_det	16	16
MSRA_2016	17	17

DeepInsight	18	19
RetinaNet (1 model)	19	18
DGIST-FATRC	20	21
Deformable R-FCN	21	23
MSRA_2015	22	20
HRI	23	22
FPN (single model)	24	25
Trimps-Soushen	25	24
Imagine Lab	26	26
mcc_lab	27	28
Wall	28	27
ANLV	29	30
FAIRCNN	30	29
CMU_A2_VGG16	31	31
DeNet	32	32
ION	33	33
CMU Cylab	34	34
COCO VGG16 Baseline	35	36
Ttester	36	35
ToConcoctPellucid	37	38
MCLAB	38	37
hust-mclab	39	39
MCPRL	40	40
CMU_A2	41	41
Darknet	42	43
UofA	43	42
FRCNN CNET	44	45
COCO Baseline	45	44
drl	46	46
Decode	47	47
Wall_2015	48	48
SinicaChen	49	49

UCSD	50	51
1026	51	50
iRONYUN	52	52

TABLE 4 – Excerpt of differences in the ranks for the 52 competitors : Initial Rank is computed over the entire dataset while New Rank is computed over the reduced dataset

TABLE DES FIGURES

1 Un exemple de matrice de performance exploitée dans cette thèse. Chaque cellule est le résultat d'une exécution d'un programme (colonne) sur un cas de test (ligne). Dans cet exemple le temps d'exécution a été mesuré et reporté dans les cellules de la matrice exprimé en secondes. 5

1.1 An example of the performance matrix we exploit in this thesis. Each cell is the result of the execution of a Program Variant (columns) with a Test case (rows). Let us consider that execution time is measured and expressed in seconds. 22
1.2 Our two contributions in perspective of the performance matrix 23

2.1 Common view of the software product line development process. 27
2.2 A feature model representing how to create a Video Sequence 30
2.3 3 categories of irises with a representative of each category 35
2.4 Irises are described by the length and width of their petals and sepals. . 35
2.5 the process which is used to learn how to separate configurations . . . 36

3.1 The studied state of the art in relation with the performance matrix . . . 43

4.1 Sampling, testing, learning : process for inferring constraints of product lines 60
4.2 Constraining the configuration space 63
4.3 Variability model excerpt of the generator 69
4.4 Learning method on the video generator 70
4.5 An excerpt of the decision tree built from a sample of 500 configurations/videos 70
4.6 three constraint extracted from our case study 72

5.1 Configuration of a specialized x264. Given a performance objective, the specialization method infers and fixes some options values (no_mbtree and crfRatio) ; users still have some flexibility to configure other options. 86

5.2 Number of x264 configurations running under a certain time : X-axis represents a number of configurations ; Y-axis represents the execution speed (in seconds) to encode a video benchmark ; *e.g.,* about 25994 configurations can encode the video in less than 145.01 seconds. . . . 86

5.3 Sampling, measuring, learning : given a performance objective, there is an automated method for specializing a configurable system 88

5.4 Confusion matrix and classification metrics : with machine learning *vs* without learning (example) 91

5.5 Average Precision measures for all 16 systems along with execution conditions. Because of the heterogeneity of systems, we present the percentage of configurations used in the training set compared to the number of available configurations (see last column of Table 5.1 and the absolute number under brackets. The last column present the percentage of interesting configurations according to the distribution of performance and the performance goal (given under brackets). 97

5.6 Minimum Precision measures for all 16 systems along with execution conditions. Because of the heterogeneity of systems, we present the percentage of configurations used in the training set compared to the number of available configurations (see last column of Table 5.1 and the absolute number under brackets. The last column present the percentage of interesting configurations according to the distribution of performance and the performance goal (given under brackets). 98

5.7 Maximum Precision measures for all 16 systems along with execution conditions. Because of the heterogeneity of systems, we present the percentage of configurations used in the training set compared to the number of available configurations (see last column of Table 5.1 and the absolute number under brackets. The last column present the percentage of interesting configurations according to the distribution of performance and the performance goal (given under brackets). 99

5.8 Average Recall measures for all 16 systems along with execution conditions. Because of the heterogeneity of systems, we present the percentage of configurations used in the training set compared to the number of available configurations (see last column of Table 5.1 and the absolute number under brackets. The last column present the percentage of interesting configurations according to the distribution of performance and the performance goal (given under brackets). 100

5.9 Minimum Recall measures for all 16 systems along with execution conditions. Because of the heterogeneity of systems, we present the percentage of configurations used in the training set compared to the number of available configurations (see last column of Table 5.1 and the absolute number under brackets. The last column present the percentage of interesting configurations according to the distribution of performance and the performance goal (given under brackets). 101

5.10 Maximum Recall measures for all 16 systems along with execution conditions. Because of the heterogeneity of systems, we present the percentage of configurations used in the training set compared to the number of available configurations (see last column of Table 5.1 and the absolute number under brackets. The last column present the percentage of interesting configurations according to the distribution of performance and the performance goal (given under brackets). 102

5.11 Average Plasticity measures for all 16 systems along with execution conditions. Because of the heterogeneity of systems, we present the percentage of configurations used in the training set compared to the number of available configurations (see last column of Table 5.1 and the absolute number under brackets. The last column present the percentage of interesting configurations according to the distribution of performance and the performance goal (given under brackets). 103

5.12 Minimum Plasticity measures for all 16 systems along with execution conditions. Because of the heterogeneity of systems, we present the percentage of configurations used in the training set compared to the number of available configurations (see last column of Table 5.1 and the absolute number under brackets. The last column present the percentage of interesting configurations according to the distribution of performance and the performance goal (given under brackets). 104

5.13 Maximum Plasticity measures for all 16 systems along with execution conditions. Because of the heterogeneity of systems, we present the percentage of configurations used in the training set compared to the number of available configurations (see last column of Table 5.1 and the absolute number under brackets. The last column present the percentage of interesting configurations according to the distribution of performance and the performance goal (given under brackets). 105

5.14 Average Sureness measures for all 16 systems along with execution conditions. Because of the heterogeneity of systems, we present the percentage of configurations used in the training set compared to the number of available configurations (see last column of Table 5.1 and the absolute number under brackets. The last column present the percentage of interesting configurations according to the distribution of performance and the performance goal (given under brackets). 106

5.15 Minimum Sureness measures for all 16 systems along with execution conditions. Because of the heterogeneity of systems, we present the percentage of configurations used in the training set compared to the number of available configurations (see last column of Table 5.1 and the absolute number under brackets. The last column present the percentage of interesting configurations according to the distribution of performance and the performance goal (given under brackets). 107

5.16 Maximum Sureness measures for all 16 systems along with execution conditions. Because of the heterogeneity of systems, we present the percentage of configurations used in the training set compared to the number of available configurations (see last column of Table 5.1 and the absolute number under brackets. The last column present the percentage of interesting configurations according to the distribution of performance and the performance goal (given under brackets). 108

6.1 Multimorphic process : morphs are automatically produced (*e.g.,* thanks to parameters) ; for each morph test suite is executed and performance measurements are gathered ; a dispersion score is finally computed to characterize the quality of a test suite 117
6.2 An example of a histogram (based on Table 6.1). 120
6.3 Stability results for the property precision ; X-axis : number of morphs removed ; Y-axis : dispersion score 131
6.4 The number of bins activated with the original test suite (composed of 84 test cases) 133
6.5 The number of bins activated with the smaller test suite composed of 5 test cases that maximizes the dispersion score. 133
6.6 On the X-axis are the index of the bins. On the left, the original histogram when all test cases are taken into account. On the right is the histogram associated with our smaller test suite. Bars in blue represent activated bins of histograms. Bars in red are bins that are activated with the original test suite but that we fail to activate with our smaller test suite. . . . 133
6.7 Average precision measures over the 5 videos from RQ2. On X-axis are the CV morphs : first, the 6 first morphs that are supposed to perform moderately badly ; the 6 last morphs are supposed to be good. Y-axis reports the averaged *precision* measure over the test suite. 134

LISTE DES TABLEAUX

2.1 An example of a confusion matrix for a 2-class problem (*i.e.,* class +1 and -1). 39

4.1 Confusion matrix of our experiment 74

5.1 *Features* : number of boolean features / number of numerical features ; $\#[\![VM]\!]$: number of valid configurations ; *Meas.* : number of configurations that have been measured. 95

6.1 An example for showing the inadequacy of variance and illustrating our measure : performance observations gathered for 2 different test suites, each composed of 2 test cases over 6 morphs. 120
6.2 The three case studies 123
6.3 Sample of observations for precision on the OpenCV case 128
6.4 The 5 concepts that maximize the dispersion score 135

1 Dispersion scores for all concepts in the COCO dataset 149
2 Sample of Execution time observations regarding generated Haxe programs. (similar table regarding other examples are available in appendixes). 149
3 Excerpt of competitors' performances differences between the original benchmark (Initial Perf) and our reduced set of 5 concepts (Computed Perf) 150
4 Excerpt of differences in the ranks for the 52 competitors : Initial Rank is computed over the entire dataset while New Rank is computed over the reduced dataset 152

BIBLIOGRAPHIE

[1] Mathieu Acher, Mauricio Alférez, José A Galindo, Pierre Romenteau, and Benoit Baudry. Vivid : A variability-based tool for synthesizing video sequences. In *Proceedings of the 18th International Software Product Line Conference : Companion Volume for Workshops, Demonstrations and Tools-Volume 2*, pages 143–147. ACM, 2014.

[2] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. Familiar : A domain-specific language for large scale management of feature models. *Science of Computer Programming (SCP)*, 78(6) :657–681, 2013.

[3] Mustafa Al-Hajjaji, Thomas Thüm, Jens Meinicke, Malte Lochau, and Gunter Saake. Similarity-based prioritization in software product-line testing. In *Proceedings of the 18th International Software Product Line Conference-Volume 1*, pages 197–206. ACM, 2014.

[4] Mauricio Alférez, Mathieu Acher, José A Galindo, Benoit Baudry, and David Benavides. Modeling variability in the video domain : Language and experience report. *Software Quality Journal*, pages 1–41, 2014.

[5] Mauricio Alférez, Mathieu Acher, José A Galindo, Benoit Baudry, and David Benavides. Modeling Variability in the Video Domain : Language and Experience Report. *Software Quality Journal*, pages 1–28, January 2018.

[6] James H Andrews, Lionel C Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments ? In *Proceedings of the 27th international conference on Software engineering*, pages 402–411. ACM, 2005.

[7] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8) :608–624, 2006.

[8] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-oriented software product lines*. Springer, 2016.

[9] Kacper Bąk, Krzysztof Czarnecki, and Andrzej Wąsowski. Feature and meta-models in clafer : mixed, specialized, and coupled. In *International Conference on Software Language Engineering*, pages 102–122. Springer, 2010.

[10] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing : A survey. *IEEE Transactions on Software Engineering*, 41(5) :507–525, May 2015.

[11] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. From genetic to bacteriological algorithms for mutation-based testing. *Software Testing, Verification and Reliability*, 15(2) :73–96, 2005.

[12] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. In *International Conference on Advanced Information Systems Engineering*, pages 491–503. Springer, 2005.

[13] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later : a literature review. *Information Systems*, 35(6) :615–636, 2010.

[14] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. A study of variability models and languages in the systems software domain. *IEEE Transactions on Software Engineering*, 39(12) :1611–1640, 2013.

[15] Mohamed Boussaa. *Automatic Non-functional Testing and Tuning of Configurable Generators*. PhD thesis, Inria Rennes - Bretagne Atlantique ; University of Rennes 1, 2017.

[16] Mohamed Boussaa, Olivier Barais, Benoit Baudry, and Gerson Sunyé. Automatic non-functional testing of code generators families. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming : Concepts and Experiences*, volume 52, pages 202–212. ACM, 2016.

[17] Michael Buhrmester, Tracy Kwang, and Samuel D Gosling. Amazon's mechanical turk : A new source of inexpensive, yet high-quality, data ? *Perspectives on psychological science*, 6(1) :3–5, 2011.

[18] Cécile Camillieri, Luca Parisi, Mireille Blay-Fornarino, Frédéric Precioso, Michel Riveill, and Joël Cancela-Vaz. Towards a Software Product Line for Machine Learning Workflows : Focus on Supporting Evolution. In *10th Workshop on Models and Evolution co-located with ACM/IEEE 19th International Conference*

on Model Driven Engineering Languages and Systems (MODELS 2016), Saint Malo, France, October 2016.

[19] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. On parallel scalable uniform sat witness generation. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035*, pages 304–319, Berlin, Heidelberg, 2015. Springer-Verlag.

[20] Shiping Chen, Yan Liu, Ian Gorton, and Anna Liu. Performance prediction of component-based applications. *Journal of Systems and Software*, 74(1) :35–43, 2005.

[21] Paul Clements and Linda M. Northrop. *Software Product Lines : Practices and Patterns*, volume 3. Addison-Wesley Professional, 2002.

[22] Myra B Cohen, Matthew B Dwyer, and Ieee Computer Society. Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints : A Greedy Approach. *IEEE Transactions on Software Engineering*, 34(5) :633–650, 2008.

[23] M. Cordy, P.-Y. Schobbens, P. Heymans, and A Legay. Beyond boolean product-line model checking : Dealing with feature attributes and multi-features. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 472–481, May 2013.

[24] Krzysztof Czarnecki, Steven She, and Andrzej Wasowski. Sample spaces and feature models : There and back again. In *12th International Software Product Line Conference*, pages 22–31. IEEE, 2008.

[25] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet : A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[26] Ivan do Carmo Machado, John D. McGregor, and Eduardo Santana de Almeida. Strategies for testing products in software product lines. *ACM SIGSOFT Software Engineering Notes*, 2012.

[27] Alastair F. Donaldson and Andrei Lascu. Metamorphic testing for (graphics) compilers. In *Proceedings of the 1st International Workshop on Metamorphic Testing, MET@ICSE 2016, Austin, Texas, USA, May 16, 2016*, pages 44–47, 2016.

[28] Richard W. Dosselman and Xue Dong Yang. No-reference noise and blur detection via the fourier transform. Technical report, University of Regina, CANADA, 2012.

[29] Jose Angel Galindo Duarte. *Evolution, testing and configuration of variability intensive systems*. PhD thesis, Université de Rennes 1 & Université de Séville, 2015.

[30] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes challenge : A retrospective. *International Journal of Computer Vision*, 111(1) :98–136, January 2015.

[31] Tom Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8) :861–874, 2006.

[32] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015.

[33] Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2) :179–188, 1936.

[34] José A. Galindo, Mauricio Alférez, Mathieu Acher, Benoit Baudry, and David Benavides. A variability-based testing approach for synthesizing video sequences. In *International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 293–303. ACM, 2014.

[35] Gonzalo Génova, María Cruz Valiente, and Mónica Marrero. On the difference between analysis and design, and why it is relevant for the interpretation of models in model driven engineering. *Journal of Object Technology*, 8(1) :107–127, 2009.

[36] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 302–313. ACM, 2013.

[37] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[38] Greg Griffin, Alex Holub, and Pietro Perona. Caltech-256 image dataset. 2007.

[39] Jianmei Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski. Variability-aware performance prediction : A statistical learning approach. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 301–311, 2013.

[40] Jianmei Guo, Jules White, Guangxin Wang, Jian Li, and Yinglin Wang. A genetic algorithm for optimized feature selection with resource constraints in software product lines. *Journal of Systems and Software*, 84(12) :2208 – 2221, 2011.

[41] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. Test them all, is it worth it ? assessing configuration sampling on the jhipster web development stack. *Empirical Software Engineering*, Jul 2018.

[42] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. Bypassing the combinatorial explosion : Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Trans. Software Eng.*, 2014.

[43] JC Huang. An approach to program testing. *ACM Computing Surveys (CSUR)*, 7(3) :113–128, 1975.

[44] Arnaud Hubaux, Thein Than Tun, and Patrick Heymans. Separation of concerns in feature diagram languages : A systematic survey. *ACM Computing Surveys (CSUR)*, 45(4) :51, 2013.

[45] F Hutter, Balázs Kégl, R Caruana, I Guyon, H Larochelle, and E Viegas. Automatic machine learning (automl). In *ICML 2015 Workshop on Resource-Efficient Machine Learning, 32nd International Conference on Machine Learning*, 2015.

[46] Martin Fagereng Johansen, Ø ystein Haugen, and Franck Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the 16th International Software Product Line Conference*, volume 1, pages 46–55. ACM, 2012.

[47] Matthew Johnson-Roberson, Charles Barto, Rounak Mehta, Sharath Nittur Sridhar, Karl Rosaen, and Ram Vasudevan. Driving in the matrix : Can virtual worlds replace human-generated annotations for real world tasks ? In *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, pages 746–753. IEEE, 2017.

[48] Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. Featureide : A tool framework for feature-oriented software development. In *ICSE*, pages 611–614. IEEE Computer Society, 2009.

[49] Chang Hwan Peter Kim, Don S. Batory, and Sarfraz Khurshid. Reducing combinatorics in testing product lines. In *Proceedings of the tenth international conference on Aspect-oriented software development*, pages 57–68. ACM, 2011.

[50] Chang Hwan Peter Kim, Sarfraz Khurshid, and Don Batory. Shared execution for efficiently testing product lines. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 221–230. IEEE, 2012.

[51] Chang Hwan Peter Kim, Darko Marinov, Sarfraz Khurshid, Don Batory, Sabrina Souto, Paulo Barros, and Marcelo D.Amorim. Splat : Lightweight dynamic analysis for reducing combinatorics in testing configurable systems. In *ESEC/FSE 2013*, 2013.

[52] Lars Kotthoff, Chris Thornton, Holger H Hoos, Frank Hutter, and Kevin Leyton-Brown. Auto-weka 2.0 : Automatic model selection and hyperparameter optimization in weka. *The Journal of Machine Learning Research*, 18(1) :826–830, 2017.

[53] Charles W Krueger. Software reuse. *ACM Computing Surveys (CSUR)*, 24(2) :131–183, 1992.

[54] Charles W Krueger. New methods in software product line development. In *Software Product Line Conference, 2006 10th International*, pages 95–99. IEEE, 2006.

[55] CharlesW Krueger. Easing the transition to software mass customization. In *International Workshop on Software Product-Family Engineering*, pages 282–293. Springer, 2001.

[56] Beatriz Pérez Lamancha and Macario Polo Usaola. Testing product generation in software product lines using pairwise for features coverage. In *IFIP International Conference on Testing Software and Systems*, pages 111–125. Springer, 2010.

[57] Joel Lehman and Kenneth O Stanley. Exploiting open-endedness to solve problems through the search for novelty. In *ALIFE*, pages 329–336, 2008.

[58] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco : Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[59] Huai Liu, Fei-Ching Kuo, Dave Towey, and Tsong Yueh Chen. How effectively does metamorphic testing alleviate the oracle problem ? *IEEE Transactions on Software Engineering*, 40(1) :4–22, 2014.

[60] Rafael Lotufo. On the complexity of maintaining the linux kernel configuration. *Technical Report, Electrical and Computer Engineering*, 2009.

[61] Alessandro Maccari and Anders Heie. Managing infinite variability in mobile terminal software : Research articles. *Softw. Pract. Exper.*, 35(6) :513–537, 2005.

[62] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 643–654, 2016.

[63] Marcilio Mendonca, Moises Branco, and Donald Cowan. S.p.l.o.t. : Software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 761–762, New York, NY, USA, 2009. ACM.

[64] Marcilio Mendonca, Moises Branco, and Donald Cowan. S.p.l.o.t. : software product lines online tools. In *OOPSLA'09 (companion)*. ACM, 2009.

[65] Hector Mendoza, Aaron Klein, Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. Towards automatically-tuned neural networks. In *Workshop on Automatic Machine Learning*, pages 58–65, 2016.

[66] Hung Viet Nguyen, Christian Kästner, and Tien N Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 907–918. ACM, 2014.

[67] Simeon C. Ntafos. A comparison of some structural testing strategies. *IEEE transactions on software engineering*, 14(6) :868–874, 1988.

[68] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. Finding near-optimal configurations in product lines by random sampling. In *Proceedings of*

the 2017 11th Joint Meeting on Foundations of Software Engineering, pages 61–71. ACM, 2017.

[69] Mike Papadakis and Nicos Malevris. Automatic mutation test case generation via dynamic symbolic execution. In *Software reliability engineering (ISSRE), 2010 IEEE 21st international symposium on*, pages 121–130. IEEE, 2010.

[70] Sachin Patel, Priya Gupta, and Vipul Shah. Feature interaction testing of variability intensive systems. In *Product Line Approaches in Software Engineering (PLEASE), 2013 4th International Workshop on*, pages 53–56. IEEE, 2013.

[71] Luis Patino, Tom Cane, Alain Vallee, and James Ferryman. Pets 2016 : Dataset and challenge. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–8, 2016.

[72] Matthew Patrick, Matthew D Castle, Richard OJH Stutt, and Christopher A Gilligan. Automatic test image generation using procedural noise. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 654–659. IEEE, 2016.

[73] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore : Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 1–18, New York, NY, USA, 2017. ACM.

[74] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. *Software product line engineering : foundations, principles and techniques*. Springer Science & Business Media, 2005.

[75] Adam Porter, Cemal Yilmaz, Atif M Memon, Douglas C Schmidt, and Bala Natarajan. Skoll : A process and infrastructure for distributed continuous quality assurance. *IEEE Transactions on Software Engineering*, 33(8) :510–525, 2007.

[76] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco3 Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2014.

[77] Cyrus Rashtchian, Peter Young, Micah Hodosh, and Julia Hockenmaier. Collecting image annotations using amazon's mechanical turk. In *Proceedings of the NAACL HLT 2010 Workshop on Creating Speech and Language Data with Amazon's Mechanical Turk*, pages 139–147, 2010.

[78] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering*, 22(2) :852–893, 2017.

[79] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. Cost-efficient sampling for performance prediction of configurable systems (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 342–352. IEEE, 2015.

[80] David W Scott. On optimal and data-based histograms. *Biometrika*, 66(3) :605–610, 1979.

[81] David W Scott. *Multivariate density estimation : theory, practice, and visualization*. John Wiley & Sons, 2015.

[82] Sergio Segura, Gordon Fraser, Ana B. Sánchez, and Antonio Ruiz Cortés. A survey on metamorphic testing. *IEEE Trans. Software Eng.*, 42(9) :805–824, 2016.

[83] Sergio Segura, Javier Troya, Amador Durán Toro, and Antonio Ruiz Cortés. Performance metamorphic testing : Motivation and challenges. In *39th IEEE/ACM International Conference on Software Engineering : New Ideas and Emerging Technologies Results Track, ICSE-NIER 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 7–10, 2017.

[84] Samuel Sepúlveda, Ania Cravero, and Cristina Cachero. Requirements modeling languages for software product lines : A systematic literature review. *Information and Software Technology*, 69 :16 – 36, 2016.

[85] Hazim Shatnawi and H. Conrad Cunningham. Mapping spl feature models to a relational database. In *ACM Southeast Regional Conference*, 2017.

[86] Ashish Shrivastava, Tomas Pfister, Oncel Tuzel, Joshua Susskind, Wenda Wang, and Russell Webb. Learning from simulated and unsupervised images through adversarial training. In *CVPR*, volume 2, page 5, 2017.

[87] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 284–294, 2015.

[88] Norbert Siegmund, Sergiy S Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. Predicting performance via automated feature-interaction detection. In *Proceedings of the 34th International Conference on Software Engineering*, pages 167–177. IEEE Press, 2012.

[89] Norbert Siegmund, Marko RosenmüLler, Christian KäStner, Paolo G. Giarrusso, Sven Apel, and Sergiy S. Kolesnikov. Scalable prediction of non-functional properties in software product lines : Footprint and memory consumption. *Inf. Softw. Technol.*, 2013.

[90] Bernard W Silverman. *Density estimation for statistics and data analysis*. Routledge, 2018.

[91] Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Is the linux kernel a software product line ? In *Proc. SPLC Workshop on Open Source Software and Product Lines*, 2007.

[92] Julio Sincero, Wolfgang Schroder-Preikschat, and Olaf Spinczyk. Approaching non-functional properties of software product lines : Learning from products. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 147–155. IEEE, 2010.

[93] Sabrina Souto, Divya Gopinath, Marcelo d'Amorim, Darko Marinov, Sarfraz Khurshid, and Don Batory. Faster bug detection for software product lines with incomplete feature models. In *Proceedings of the 19th International Conference on Software Product Line*, pages 151–160. ACM, 2015.

[94] Domagoj Štrekelj, Hrvoje Leventić, and Irena Galić. Performance overhead of haxe programming language for cross-platform game development. *International Journal of Electrical and Computer Engineering Systems*, 6(1) :9–13, 2015.

[95] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques : Research articles. *Softw. Pract. Exper.*, 35(8) :705–754, 2005.

[96] Terence Tao. *An introduction to measure theory*, volume 126. American Mathematical Soc., 2011.

[97] Rasha Tawhid and Dorina C Petriu. Automatic derivation of a product performance model from a software product line model. In *Software Product Line Conference (SPLC), 2011 15th International*, pages 80–89. IEEE, 2011.

[98] Richard N. Taylor, David L. Levine, and Cheryl D. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, 18(3) :206–215, 1992.

[99] Paul Temple, Mathieu Acher, and Jean-Marc Jézéquel. Multimorphic testing. In *Proceedings of the 40th International Conference on Software Engineering : Companion Proceeedings*, pages 432–433. ACM, 2018.

[100] Paul Temple, Mathieu Acher, Jean-Marc Jezequel, and Olivier Barais. Learning contextual-variability models. *IEEE Software*, 34(6) :64–70, 2017.

[101] Paul Temple, Mathieu Acher, Jean-Marc Jézéquel, Léo Noel-Baron, and José Galindo. Learning-based performance specialization of configurable systems. Technical report, Univ Rennes, Inria, CNRS, IRISA, 2017.

[102] Paul Temple, José A Galindo, Mathieu Acher, and Jean-Marc Jézéquel. Using machine learning to infer constraints for product lines. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 209–218. ACM, 2016.

[103] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Auto-weka : Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 847–855. ACM, 2013.

[104] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys*, 2014.

[105] Thomas Thum, Don Batory, and Christian Kastner. Reasoning about edits to feature models. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 254–264. IEEE, 2009.

[106] Thomas Thüm, Christian Kstner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. Featureide : An extensible framework for feature-oriented software development. *Science of Computer Programming*, 2012.

[107] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest : Automated testing of deep-neural-network-driven autonomous cars. *CoRR*, abs/1708.08559, 2017.

[108] Pavel Valov, Jianmei Guo, and Krzysztof Czarnecki. Empirical comparison of regression methods for variability-aware performance prediction. In *Proceedings of the 19th International Conference on Software Product Line*, pages 186–190. ACM, 2015.

[109] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. A classification of product sampling for software product lines. In *Proceeedings of the 22nd International Conference on Systems and Software Product Line-Volume 1*, pages 1–13. ACM, 2018.

[110] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. Presence-condition simplification in highly configurable systems. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 178–188. IEEE Press, 2015.

[111] Nik Weaver. *Measure theory and functional analysis*. World Scientific Publishing Company, 2013.

[112] Jules White, Brian Dougherty, Doulas C Schmidt, and David Benavides. Automated reasoning for multi-step feature model configuration problems. In *Proceedings of the 13th International Software Product Line Conference*, pages 11–20. Carnegie Mellon University, 2009.

[113] Cemal Yilmaz, Myra B Cohen, and Adam A Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, 32(1) :20–34, 2006.

[114] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. Deeproad : Gan-based metamorphic autonomous driving system testing. *arXiv preprint arXiv :1802.02295*, 2018.

[115] Yi Zhang, Jianmei Guo, Eric Blais, and Krzysztof Czarnecki. Performance prediction of configurable software systems by fourier learning (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 365–373. IEEE, 2015.