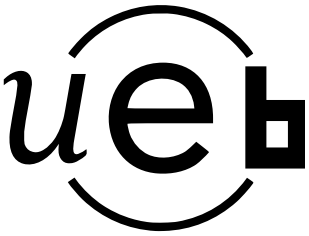
Approche à contraintes pour la sélection de Covering Array

Aymeric Hervieu

ANNÉE 2014



THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Aymeric HERVIEU

préparée à l'unité de recherche INRIA
Rennes Bretagne Atlantique

Approche à contraintes pour la sélection de *Covering Array*.

Thèse soutenue à Rennes
le 9 Décembre 2013

devant le jury composé de :

Thierry JERON
Directeur de recherche à l'INRIA / *Président*

Jordi CABOT
Maître de conférences, École des mines de Nantes / *Rapporteur*

Camille SALINESI
Professeur, Université Paris 1 / *Rapporteur*

Mireille DUCASSE
Professeur, INSA Rennes / *Examinatrice*

Benoit BAUDRY
Chercheur, INRIA Rennes - Bretagne Atlantique / *Directeur de thèse*

Arnaud GOTLIEB
Chercheur, Simula research lab / *Co-directeur de thèse*

Alain RIBAULT
Directeur technique, Kereval / *Co-directeur de thèse*

Table des matières

Chapitre 1

Introduction

Comment un développeur peut-il s'assurer que l'application Android qu'il a développé s'exécute correctement sur tous les téléphones mobiles existants, ou comment peut-il être sûr qu'une application web fonctionne correctement pour tous les utilisateurs ? Comment s'assurer, devant la multitude de plugins Eclipse existants, que n'importe quel arrangement de plugin ne déclenche pas une erreur ? Avec 199 paramètres différents comment s'assurer que le compilateur GCC fonctionne dans toutes ses configurations possibles ?

Toutes ces questions ont une même origine : la variabilité. Dans un produit logiciel, cette variabilité se manifeste de différentes façons :

- elle peut être issue du produit logiciel développé comme une application Eclipse, qui résulte de l'assemblage de différents composants logiciels (plugins).
- Elle peut être relative aux contextes d'exécutions : de multiples téléphones mobiles ou bien un ensemble hétérogène de machines qui consultent un site web.
- Elle peut être relative aux paramètres logiciels comme le cas de l'application GCC.

Quel que soit le domaine où elle s'applique, la variabilité pose des problèmes aux équipes de test. Comment peut-on quantifier cette variabilité ? Comment peut-on tester le logiciel ? Quelles sont les configurations qui doivent être testées ? Toutes ces questions font partie du quotidien des testeurs de logiciel.

Dans cette thèse, nous adressons ces problématiques de test afin de fournir des moyens pratiques aux testeurs et développeurs pour résoudre ces problèmes de variabilité.

1.1 Motivations

Aujourd'hui, les éditeurs logiciels ne conçoivent, développent et ne maintiennent plus leurs offres logicielles en ne ciblant qu'un client unique. Au contraire, les offres logicielles sont conçues pour cibler plusieurs entités. Par conséquent, ces applications doivent s'in-

tégrer dans des environnements différents et s'adapter aux besoins des clients. Ainsi, les produits développés ne sont plus des programmes uniques, mais des familles de produits [Par76]. Tous ces produits présentent donc des similarités, mais également des différences en certains points, dues aux contraintes métiers du client.

Dans ces familles de produits, les produits possèdent un ensemble de fonctionnalités communes, et un certain nombre de fonctionnalités qui leurs sont propres. Ces fonctionnalités spécifiques permettent de différencier les produits en les spécialisant. Ainsi, on définit la variabilité logicielle comme la capacité qu'a un système logiciel ou un élément de ce système à être changé, personnalisé, ou configuré pour un contexte particulier [VGBS01].

Les logiciels, conçus comme des familles de produits, sont appelés *systèmes configurables*. Ces systèmes configurables permettent de réaliser un produit logiciel particulier suite à un processus de configuration. Ce processus consiste à sélectionner les fonctionnalités présentes dans le produit, puis à sélectionner les artefacts logiciels réalisant ces fonctionnalités pour les assembler.

La validation de ces systèmes configurables est une tâche complexe. Un système configurable peut générer plusieurs millions de configurations possibles. Il ne s'agit donc plus de valider un seul et unique produit, mais un ensemble de produits. Le processus de validation consiste en l'exécution d'un ou plusieurs cas de test sur les différentes configurations du système. Les projets de test industriels sont souvent contraints par le temps, par conséquent le nombre de configurations pouvant être réellement testées est lui aussi limité. Les testeurs doivent alors choisir les configurations les plus pertinentes à tester. Une partie de l'activité du testeur consiste alors à sélectionner des configurations de test pertinentes qui tiennent compte des points communs et des différences des produits issus du système configurable.

1.2 Problématique

La gestion de la variabilité dans l'activité de test logiciel lève différentes problématiques pour les ingénieurs chargés de la validation.

Expliciter la variabilité :

Dans les projets industriels rencontrés au cours de cette thèse, la variabilité n'était pas gérée. Les éléments composant le système étaient stockés dans des tableurs ou des fichiers textes et les développeurs ou testeurs n'avaient qu'une vision partielle du système. Sans représentation explicite de la variabilité, les ingénieurs ne sont pas capables de la quantifier et donc d'estimer l'effort de test. De plus, cette variabilité peut évoluer lors du déroulement du projet de test : des incompatibilités ou des dépendances entre les éléments composant les configurations peuvent être découvertes. Si ces informations

demeurent informelles, il y a un risque de perte des connaissances acquises lors du projet.

Sélection de configurations à tester :

Face à des millions de configurations possibles, les ingénieurs chargés de la validation n'ont pas de critères objectifs pour sélectionner les configurations à tester. Le choix des configurations de test peut se faire de façon empirique en utilisant les informations relatives aux configurations clients, où de façon totalement arbitraire par les ingénieurs en charge de la validation. Ces approches ne sont pas entièrement satisfaisantes. Un processus de sélection manuel des configurations de test peut en effet introduire un biais. Certaines configurations de test peuvent être très semblables, tandis que d'autres éléments de configurations ne sont pas du tout testés. Il faut donc fournir aux testeurs des moyens d'identifier ces configurations de test.
De plus, dans un contexte où les professionnels du test visent des standards qualité de plus en plus élevés, certaines organisations proposent d'apposer à leurs prestations l'étiquette ISO 17025 [ISO05]. Cette norme établit les exigences générales pour effectuer des activités de test et des essais en laboratoire. Une des exigences de cette norme consiste en la reproductibilité d'un processus de test par un tiers. Pour respecter ce cadre normatif, il faut fournir des méthodes systématiques de sélection des configurations de test. Alors que les projets industriels de test sont tous bornés dans le temps, le nombre de configurations de test peut être un facteur de réussite ou d'échec pour le projet de test. Si le nombre de configurations sélectionnées est trop important, elles ne pourront pas toutes être testées dans le temps imparti. Deux choix s'offrent alors à la personne responsable des tests : soit cesser les activités de test en sachant pertinemment que toutes les configurations choisies n'ont pas été testées, soit poursuivre et dépasser le budget de test. Afin de prévenir ces problèmes, il faut pouvoir fournir aux testeurs un moyen de sélectionner un nombre minimal de configurations de test.

Évaluation objective des techniques de test proposées :

Alors que les projets de tests logiciel sont soumis à d'importantes contraintes de coûts et de temps, les industriels ne sont pas toujours enclins à faire évoluer leurs processus de test. Il est risqué de passer d'une méthode de test à une autre : il faut faire évoluer les processus, les outils et former les parties prenantes. Si une nouvelle méthodologie de test est proposée, il faut ajouter à cette méthode des arguments chiffrés qui justifient son efficacité. Toute nouvelle méthode doit être évaluée sur un ou plusieurs projets de façon à détecter les faiblesses et mesurer les gains : que ce soit un gain de temps ou de qualité du logiciel.

1.3 Contributions

Conscient de ces difficultés, le monde académique propose déjà des solutions pour répondre en partie à ces questions. Les modèles de *features* permettent de représenter

et de raisonner sur un produit logiciel en terme de points communs et de différences. Ces outils sont particulièrement efficaces pour représenter l'ensemble des configurations d'un système (c'est à dire tous les produits que l'on peut créer), ou pour représenter les multiples contextes d'exécutions. La sélection de configurations de test a déjà été étudiée par la recherche : les techniques de test combinatoire ont montré leurs efficacités à sélectionner un sous-ensemble de configurations de test. L'utilisation conjointe des modèles de *features*, qui capturent les configurations du système à tester, et du test combinatoire a aussi été étudiée par le monde académique. Cependant, ces travaux ne sont pas ou peu connus du monde industriel et sont décorrélés des pratiques de test en industrie.

Dans cette thèse, nous présentons trois contributions qui visent à traiter les problèmes de test de systèmes variables.

La première contribution est une **analyse détaillée de deux projets industriels de test**, faisant face à des problématiques de variabilité. Il s'agit du projet de test du logiciel *Business and Internet EveryWhere* (BIEW) développé par Orange et du projet de test du logiciel TéléPrésence développé par Cisco. Nous présentons les techniques de test retenues par les testeurs et les défis rencontrés.

La deuxième contribution est une nouvelle approche permettant de résoudre, à l'aide de la programmation par contraintes, le problème de la sélection de configurations à partir d'un modèle de *features*. Cette méthodologie repose sur une approche à contraintes. Nous proposons une **implémentation à l'aide des contraintes des opérateurs utilisés dans un modèle de *features***. Nous introduisons une nouvelle contrainte appelée *Pairwise*. Nous montrons comment l'utilisation conjointe des contraintes du modèle de *features* et l'utilisation de la contrainte *Pairwise* nous permettent de **résoudre le problème de la sélection des configurations de test respectant le critère *Pairwise***. L'utilisation de la programmation par contrainte nous permet de proposer une approche reproductible avec le même solveur qui peut minimiser le nombre de configurations. L'approche proposée est comparée à trois outils de l'état de l'art, sur 224 modèles de *features* différents. Nous montrons que notre approche est plus performante en terme de taille de solution retournée que deux des outils existant, et a des performances similaires à la troisième approche. Cette nouvelle technique est embarquée dans un produit logiciel : Pacogen. Ce produit logiciel fait 7 000 lignes de code et est disponible en téléchargement. [1]

La troisième contribution consiste en **une étude de l'impact qu'aurait l'utilisation de la technique de test *pairwise* sur les deux projets de test industriels de Orange et Cisco**. Cette étude montre que l'utilisation des modèles de *features* et de Pacogen permettrait un gain de temps sur les projets de test tout en préservant la qualité logicielle. Elle fournit aussi un ensemble de données chiffrées qui permettent de juger

1. http ://www.hervieu.info

de l'efficacité de l'utilisation d'une telle méthode dans le cadre de projets industriels.

1.4 Plan

La première partie de cette thèse est dédiée au contexte industriel et à l'état de l'art.

Le **chapitre 2** présente les deux applications logicielles et leurs projets de test associés qui seront utilisés tout au long du document. Ces projets de test logiciels sont issus de deux entreprises. La validation du logiciel BIEW (*Buisness EveryWhere*), développé par Orange, dispose d'un cahier des charges composé de 1 493 exigences qui doivent être validées sur plus de 1 900 000 environnements de test. La validation du logiciel TéléPrésence, développé par Cisco, concerne un produit qui possède plus d'un milliard de configurations.

Dans le **chapitre 3** sont présentés les principaux modèles de variabilité qui ont été proposés par la recherche. Nous détaillons le modèle utilisé pour les activités de recherche de cette thèse : les modèles de *features*. Et nous présentons les principaux travaux réalisés sur ces modèles de variabilité : les opérations existantes et les méthodes proposées pour raisonner automatiquement sur ces modèles.

Le **chapitre 4** présente l'approche de test appelée test combinatoire. Nous présentons les notions mathématiques associées à cette technique de test, puis nous détaillons une technique particulière issue du test combinatoire : le *pairwise*. Nous présentons les approches existantes qui traitent de la sélection de configurations de test selon le critère *Pairwise*.

Le **chapitre 5** présente la programmation par contraintes, c'est sur cette technique de programmation que les travaux présentés reposent. Nous présentons les grands principes de cette technique et les mécanismes de résolution.

La deuxième partie de cette thèse est dédiée aux contributions scientifiques.

Le **chapitre 6** montre comment les techniques de programmation par contraintes peuvent résoudre les problèmes de sélection de configuration *pairwise* sur les modèles de *features*. Nous présentons le moteur à contraintes développé pour raisonner sur les modèles de *features*, la contrainte *pairwise*, et le processus de résolution. Nous montrons comment les mécanismes développés nous permettent de fournir un système de résolution déterministe et qui permet d'extraire un ensemble de configurations de taille minimale. Nous présentons également Pacogen, l'application créée qui intègre le moteur à contraintes que nous avons conçu.

Le **chapitre 7** est dédié à l'évaluation de notre approche. Pacogen est évalué dans ces différentes configurations, et est comparé aux autres approches proposées par le

monde académique.

L'applicabilité de l'approche de test dans le cas des deux projets industriels du chapitre 2 est évaluée dans le **chapitre 8**. Nous revenons sur les principaux besoins des ingénieurs et nous étudions comment l'approche *Pairwise* y répond. Puis nous utilisons des données chiffrées pour quantifier l'impact de l'application d'une telle méthodologie.

La dernière partie est dédiée à la conclusion et aux perspectives ouvertes par ces travaux de recherches.

1.5 Publications liées à cette thèse

Cette section présente les publications et qui ont été présentées dans des conférences internationales avec comités de relecture :

- L'article *Pacogen : Automatic generation of pairwise test configurations from feature model* [HBG11] écrit par Hervieu, Gotlieb et Baudry présente Pacogen un outil permettant la sélection de configurations de test à partir de modèle de *features*. Cet article a été présenté à la conférence ISSRE en 2011.

- L'article *Managing variability during software testing* [HBG12] écrit par Hervieu, Gotlieb et Baudry présente le projet de test BIEW de Orange. Nous expliquons la méthode de test utilisée et les défis rencontrés. Cet article a été présenté à la conférence ICTSS en 2012.

- L'article *Practical pairwise testing for software product lines* [MGSH13] écrit par Marijan, Sen, Gotlieb et Hervieu présente l'application les techniques de test combinatoire sur le projet de test du logiciel TéléPrésence de Cisco. Cet article a été présenté à la conférence SPLC en 2013.

Ces travaux ont fait l'objet d'une présentation en workshop :

– L'article *Minimum pairwise coverage using constraint programming techniques* écrit par Gotlieb, Hervieu et Baudry [GHB12] présente comment les techniques de programmation par contraintes permettent de minimiser le nombre de configurations de test sélectionnées. Cet article a été présenté au workshop CSTVA de la conférence ICST en 2012.

Le journal TOSEM *Optimal Minimization of Pairwise-covering Test Configurations Using Constraint Programming* est actuellement en révision. C'est une extension des travaux initialement présentés dans *Pacogen : Automatic generation of pairwise test configurations from feature model*

Première partie

Contexte Industriel et État de l'art

Chapitre 2

Cas d'étude issus de l'industrie

Dans ce chapitre, nous présentons deux projets de test issus de l'industrie. Dans ces deux projets, les équipes chargées de de la validation font face à des problématiques de variabilité. Nous étudions le déroulement de ces deux projets : pour chacun d'eux nous présentons le produit logiciel testé, le public visé et décrivons les approches de test mises en place. Par la suite, nous identifions les points difficilement traités par les équipes de test.

Ces deux projets industriels sont issus de deux grandes entreprises. Le premier porte sur le logiciel *Business and Internet EveryWhere TM* (BIEW) développé par l'entreprise Orange. C'est un logiciel destiné aux professionnels en mobilité qui permet de se connecter de différentes façons à Internet (WiFi, Ethernet, 3G ...). Cette application est destinée à être exécutée sur des ordinateurs professionnels. Pour s'assurer que cette application fonctionne sur la majorité des ordinateurs, Orange a défini un cahier des charges qui explicite les environnements à tester représentant la diversité des configurations clients (sur quels systèmes d'exploitations, avec quelles cartes WiFi...). Selon ce cahier des charges, l'application doit être validée sur près de deux millions d'environnements différents. Les équipes de tests font face à une explosion du nombre d'environnements de test et elles doivent trouver un moyen de valider ce logiciel sur tous ces environnements. Comme ce la se révèle difficile, les testeurs ont mis en place des techniques pour sélectionner des environnements de test.

Le second projet industriel traite de la validation d'une solution de vidéo-conférence développée par Cisco, TéléPrésence. Ce logiciel est un système configurable, les ingénieurs sont capables de développer très rapidement de nouvelles versions de l'application en ajoutant ou supprimant des briques logicielles. Cette modularité permet à l'entreprise de produire très rapidement des logiciels répondant au mieux aux besoins utilisateurs. Cette production logicielle simplifiée pose un problème aux équipes de validation pour lesquelles le test d'un produit demande beaucoup de temps.

La revue de ces deux projets industriels nous permet d'identifier les verrous techno-

logiques qui motivent les travaux de recherche présentés dans cette thèse. Ces deux cas industriels serviront de support expérimental tout au long de cette thèse.

2.1 Validation de l'application Business Everywhere

2.1.1 Le logiciel Business Everywhere

Business and Internet EveryWhere TM (BIEW) est un logiciel de connexion à Internet développé par Orange. L'application BIEW a été créée pour répondre aux besoins des professionnels en mobilité. Son objectif est de fournir à l'utilisateur la capacité de se connecter à Internet en utilisant différents moyens. Le logiciel BIEW est capable de gérer des connexions en 3G, WiFi ou Ethernet. De plus, il intègre une base de données des réseaux WiFi partenaires (en France et à l'étranger) ce qui permet aux professionnels de se connecter automatiquement et gratuitement. Aujourd'hui, l'application est utilisée par plus de 1.5 million d'utilisateurs à travers le monde.

Pour l'utilisateur final, l'application est un gestionnaire de connexion comme celui de Microsoft Windows, qui intègre quelques fonctionnalités supplémentaires. L'application rassemble dans une seule et même fenêtre les différents moyens de se connecter à Internet. La figure 2.1 est une capture d'écran de l'application. La zone grise représente l'état de la connexion : sa nature (WiFi, 3G...), le temps écoulé, la qualité de la connexion, et le volume de données transférées. La figure 2.1 est une capture d'écran de l'application lorsque celle-ci est connectée en 3G, depuis 13 secondes, et 694 octets ont été échangés. En bas, un ensemble d'icônes permet à l'utilisateur d'accéder aux différentes fonctionnalités de l'application.

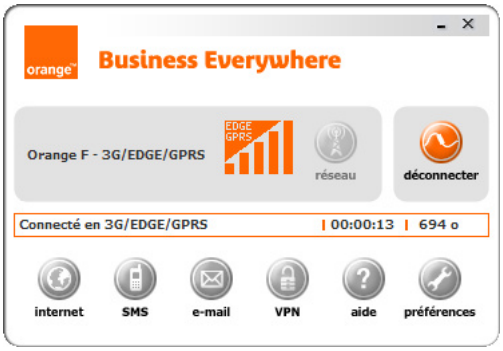


FIGURE 2.1 – Capture d'écran du logiciel BIEW

Comme BIEW doit être capable de fournir à l'utilisateur la capacité de se connecter à Internet en utilisant de multiples protocoles, le logiciel doit pouvoir gérer un nombre important de périphériques physiques. Le cahier des charges de l'application spécifie que celle-ci doit correctement s'exécuter sur près de deux millions d'environnements différents. Ces environnements sont composés d'un système d'exploitation, de périphériques 3G, de clés WiFi, de navigateurs et de clients courriers électroniques. Cette complexité

est accrue par les 1493 exigences qui définissent le produit logiciel et qui doivent être vérifiées.

BIEW est une application critique pour le groupe Orange : c'est un système destiné aux professionnels, les impactant directement dans leur travail. Un bug peut causer d'importantes pertes financières et en terme de réputation. Par conséquent, la phase de recette doit être particulièrement rigoureuse pour prévenir ce genre d' événements. Afin d'assurer une qualité logicielle optimale, Kéréval, une PME spécialisée dans le test de logiciel a été chargée de cette phase de validation.

2.1.2 Une approche de test systématique

Dans cette section, nous décrivons la méthodologie déployée par Kéréval pour la validation de ce logiciel. Pour valider ce logiciel, les ingénieurs ont fait face à deux défis :

1. **La gestion du nombre important d'environnements à tester.**
   Les utilisateurs du logiciel BIEW ont potentiellement tous des machines différentes. Pour capturer cet ensemble hétérogène d'environnements d'exécutions, les ingénieurs d'Orange ont retenu un ensemble d'éléments logiciels (systèmes d'exploitation, client courriers électroniques) et de périphériques matériels (clés WiFi, 3G..) représentant la diversité des configurations clients. Ces exigences spécifient que le logiciel doit s'exécuter correctement sur 1 920 000 environnements. Ce nombre élevé rend difficile l'utilisation d'une approche exhaustive. Il est donc nécessaire de trouver une solution afin de sélectionner les configurations à tester.
2. **Une traçabilité entre les exigences, les environnements de tests et les cas de tests.**
   Les évolutions dans les exigences peuvent impacter la stratégie de test. Ces changements doivent être tracés pour identifier facilement le nombre de cas de test impactés par une évolution. Cette traçabilité facilite l'identification des parties de l'environnement et du logiciel à tester lors d'évolutions des exigences de l'application, ou de l'environnement de test (par exemple l'ajout d'un nouveau système d'exploitation).

Ce projet de test a duré cinq ans. De nombreuses versions de l'application ont été testées. L'effort de test, pour la validation d'une version du logiciel est estimé entre 100 et 400 jours/homme, avec en moyenne une charge 300 jours/homme.

2.1.2.1 Un processus de test

Pour répondre au mieux à ces deux problématiques, les ingénieurs ont conçu une méthodologie de test, celle-ci est décrite ci-dessous. Cette approche se base sur deux documents : les spécifications environnementales et les exigences du logiciel. Les spécifications environnementales contiennent les informations relatives à l'environnement d'exécution du logiciel. C'est dans ce document que l'on retrouve les systèmes d'exploitation, les logiciels ou les matériels supportés. Le document décrivant les exigences

du logiciel contient toutes les informations détaillant le comportement de l'application. C'est à partir de ce document que sont créés les cas de tests à exécuter.

La méthodologie conçue permet de produire un ensemble de cas de tests concrets (qui sont exécutables sur le produit logiciel) et de sélectionner un ensemble d'environnements de tests. La figure 2.2 présente la méthodologie mise en place par Kéréval pour tester cette application.

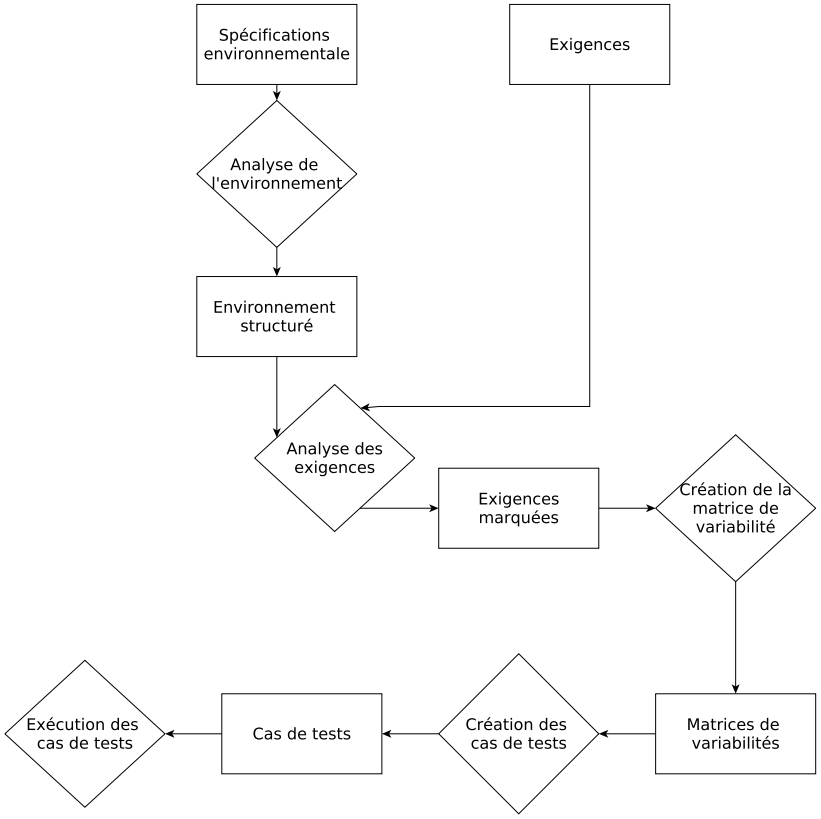


FIGURE 2.2 – Processus de test mis en place pour la validation de l'application BIEW

2.1.2.2 Analyse de l'environnement

Durant cette étape, l'ingénieur chargé de la validation formalise les spécifications environnementales. Dans le projet BIEW, les éléments composant l'environnement d'exécution sont classés selon huit catégories :

- **OS** (5) : Win. 2000, Win. XP 32 bits, Win. XP 64 bits, Win. Vista 32 bits, Win. Vista 64 bits
- **Mobile** (25) : Novatel Xu870, GT Max GX0301, Lucent Merlin U530, Huawei E870...
- **Wifi Interne** (5) : intel centrino 2100, 2200, 2915, 3945,

- **Wifi Externe** (8) : Sagem 706 A, Sagem 703...
- **Modem** (8) : Sagem F@st 800 USB, Thomson ST330, Siemens A100, ZTE ZXDSL 852...
- **VPN** (4) : Safenet, Cisco, Avasy, empty
- **Client Mail** (4) : Outlook, Outlook Express, Windows Live Mail, empty
- **Navigateur** (3) : Firefox 2.0, Firefox 1.5, Internet Explorer 5.5,

La catégorie mobile répertorie l'ensemble des périphériques permettant de fournir un accès à internet en utilisant une connexion mobile (3G...). La distinction entre les catégories WiFi interne et WiFi Externe s'explique par deux raisons : alors qu'une clé WiFi externe peut être connectée ou déconnectée à la discrétion de l'utilisateur, les systèmes permettant de se connecter en WiFi en interne ne le peuvent pas. Cette capacité à être branchée ou débranchée à la volée va induire toute une batterie de tests dédiés. De plus, les cartes WiFi internes sont déjà reconnues par le système, car les pilotes sont embarqués dans le système d'exploitation. Alors que pour les clés WiFi externes un pilote est parfois nécessaire. Selon les exigences d'Orange, ce pilote doit être intégré dans le logiciel BIEW. La valeur empty dans chaque catégorie signifie qu'aucun élément n'a été choisi. Ce mot clé permet par exemple la mise en place d'environnements ne possédant pas de VPN.

Une fois cette classification réalisée, les ingénieurs ont défini la notion de configuration de test, qui est une sélection d'un composant de chaque catégorie. Selon cette définition, ce sont 1 920 000 configurations qui peuvent être mises en place.

2.1.2.3 Analyse des exigences

A l'origine, les exigences fournies par Orange sont contenues dans des fichiers Word, et arrangées par domaines fonctionnels.

Cette structure est conservée tout au long du projet, car elle permet une répartition efficace des tâches. Il y a au total 73 domaines fonctionnels dans le projet BIEW.

Pour chaque domaine fonctionnel, les exigences sont séparées en deux catégories : celles dépendant de l'environnement et celles n'en dépendant pas. Sur les 73 domaines fonctionnels, 33 contiennent des exigences dépendant de l'environnement, soit 841 exigences. Les exigences qui dépendent de l'environnement sont étiquetées avec une ou deux des huit catégories environnementales (**O.S.**, **Mobile**, **WiFi Interne**...) identifiées précédemment. Par exemple si une exigence se réfère à une connexion à un réseau réalisée grâce à une clé $3G$, alors celle-ci est marquée "Mobile". Cette marque est utile lors de la sélection des configurations à tester. Cette activité de tri des exigences n'est pas laissée à la seule responsabilité de l'équipe de validation, elle est le résultat d'une activité collective entre les différentes parties prenantes du projet.

La figure 2.3 présente une exigence qui dépend des périphériques WiFi. Dans ce cas précis, la distinction entre WiFi interne et WiFi externe n'est pas faite, ce qui signifie que l'exigence dépend de façon générale des deux types de périphériques WiFi.

```
[RQ 01980 _V8.0.4_ Other WLAN Descriptor security key]
When the end-user chooses, in the access point list, one of the sniffed
 'Other WLAN Descriptor', he is prompted to enter
 the security key, when one is required.
```

FIGURE 2.3 – Une exigence qui dépend du périphérique WiFi

2.1.2.4 La création des cas de tests

C'est à cette étape que sont créés les cas de tests qui vont permettre de valider les exigences du système. Ce sont ces cas de tests qui seront exécutés sur le produit logiciel à valider. Un exemple de cas de test est présenté figure 2.4, il est extrait du jeu de cas de tests créé à partir de l'exigence présentée figure 2.3.

[353-RQ01980][WiFi]
Objective: Check the prompt display for one descriptor and one security key WPA2.
Pre Requise:
Business EveryWhere Kit installed.
Acces Point AP1 selected
Wireless lan seted up with WPA 2 security
Test Procedure:
- Launch the BIEW application
- Click on the button connect, on the main screen
- Select the access Point AP1

FIGURE 2.4 – Un exemple de cas de test

2.1.2.5 La création de la matrice de variabilité

Une matrice de variabilité est créée pour chacun des 33 domaines fonctionnels qui contiennent des exigences dépendant de l'environnement. La *matrice de variabilité* met en relation les cas de tests et les environnements à tester. La figure 2.5 est une capture d'écran d'une matrice utilisée dans le projet BIEW. C'est dans cette matrice que l'on retrouve quels cas de tests ont été exécutés et dans quels environnements. Les premières colonnes contiennent le nom du cas de test, sa priorité ainsi que la ou les catégories environnementales dont dépendent le cas de test et son verdict. Les colonnes qui suivent précisent dans quels environnements exécuter le cas de test. Sur cette figure, les tests sont déjà partiellement exécutés, mais juste après la création de la matrice tous les tests ont le statut `No Run`.

Une fois cette matrice créée, trois règles sont appliquées pour diminuer le nombre d'environnements à tester :

– Si un cas de test n'a aucune marque, alors il n'est exécuté que dans un seul des environnements présents dans la matrice.

Projet ouvert : BESS_V8_0				Vista Business												
			120 test(s) >	Centrino 2100	Centrino 2200	Centrino 2915	Centrino 3945	Dongle Inventel InexQ	Inventel Cohiba	Sagem 703	Sagem 760A	Sagem 760N	Cameo	Broadcom Devices	Centrino 2100	Centrino 2200
Plan: Name	Plan: Dependancies	Plan: Priority	Status	20	44	20	20	20	80	20	20	20	20	20	15	15
353 - RQ 01500	Aucun	P1 - Important	Passed						OK							
353 - RQ 02900 - 002	OS + Wifi	P0 - Indispensable	No Run									NA	NA	NA		
353 - RQ 03000	OS + Wifi	P0 - Indispensable	Failed									fail	fail	fail		
353 - RQ 03050	OS + Wifi	P0 - Indispensable	Failed									OK	OK	fail		
353 - RQ 03200	Aucun	P0 - Indispensable	Passed						OK							

FIGURE 2.5 – Matrice de variabilité

- Si un cas de test dépend d'une caractéristique environnementale, alors celui-ci devra être exécuté sur un ensemble d'environnements qui couvrent tous les éléments composants cette caractéristique environnementale. Par exemple, un cas de test marqué `VPN` sera exécuté sur 4 environnements de test différents afin de tester tous les VPN contenus dans "caractéristiques environnementales" VPN.
- Si le cas de test est marqué avec deux catégories environnementales, alors il sera exécuté sur toutes les combinaisons possibles des éléments qui composent les deux catégories environnementales. Par exemple, un cas de test marqué VPN et OS est exécuté sur $4 * 5 = 20$ configurations différentes.

Dans la figure 2.5, les cases grise représentent les configurations dans lesquelles les cas de tests ne devront pas être exécutés. Les cases en blanc identifient les configurations sur lesquelles les testeurs vont devoir exécuter les cas test. En bleu figurent les combinaisons d'environnements et de cas de test non applicables (à cause, par exemple d'une incompatibilité). Finalement, en rouge et vert sont identifiées les combinaisons de cas de test et environnements aboutissant à l'échec ou à un succès du cas de test.

2.1.2.6 L'exécution des cas de tests

L' **exécution des cas de tests** est une étape où l'ingénieur chargé de la validation distribue les tâches du projet de test aux membres de l'équipe : de la mise en place de l'environnement de test à l'exécution des cas de tests. Dans ce projet, l'ingénieur attribue à chaque testeur une matrice de variabilité, puis les testeurs mettent en place les environnements choisis et exécutent les cas de tests.

2.1.3 Défis rencontrés

Lors de ce projet de test, les testeurs font face à un certain nombre de défis :

Une absence de représentation explicite de la variabilité.

La classification en huit catégories des éléments qui composent l'environnement d'exécution choisie par les ingénieurs a permis de répondre aux besoins levés par les activités de test de ce projet. Cependant, une analyse précise des choix des testeurs met en avant un certain nombre de faiblesses.

1. Le choix de représenter une configuration comme la sélection d'un composant par catégorie ne reflète pas parfaitement la réalité. Par exemple, rien n'interdit d'avoir plusieurs navigateurs sur une même machine.
2. L'utilisation du mot empty pour caractériser l'absence d'un composant dans une configuration relève plus d'un traitement ad'hoc que d'une solution pérenne.
3. Le processus actuel ne prend pas en compte les relations inter-catégories. Il est par exemple impossible de spécifier une incompatibilité entre un navigateur et un système d'exploitation. Toutes ces informations restent informelles, ce qui augmente la probabilité de mettre en place un environnement invalide.

Cette absence de représentation explicite de la variabilité peut être source de retard pour le projet de test. La mise en place d'un environnement de test prend du temps : il faut récupérer le matériel nécessaire, installer le système d'exploitation, les pilotes... Si l'environnement choisi s'avère invalide, par exemple à cause d'un cas de test non applicable dans une configuration, tout le temps passé à préparer l'environnement est perdu.

Une difficulté à sélectionner les configurations de test.

L'étude du projet de test met en avant une tâche consommatrice de temps : la mise en place des environnements de tests. Il apparaît que de nombreux environnements de tests, parfois très similaires sont mis en place de façon concurrente et qu'il n'y a que très peu de réutilisation. Sur les 390 configurations utilisées, 149 sont redondantes. Ce phénomène s'explique par la structure du projet de test : les configurations de tests sont cloisonnées dans les différentes matrices de variabilité, ce qui empêche toute capitalisation des environnements de test.

L'important nombre de matrices, les différentes interactions entre les éléments composant les environnements et le nombre important d'environnements à mettre en place rendent très difficile la prise de recul nécessaire pour pouvoir mettre en place une méthode capitalisant ces environnements de test. Les testeurs privilégient une méthode plus coûteuse qui consiste à traiter les matrices de variabilité de façon indépendante et à mettre à chaque fois en place les environnements de test associés. Afin de réduire le temps passé à mettre en place les environnements de tests, les testeurs ont utilisé un système permettant d'automatiser partiellement la mise en place des configurations de tests.

2.2 Le système de vidéo conférence de Cisco

2.2.1 L'application

La solution de vidéo-conférence, TéléPrésence, est une application proposée par Cisco. Cette solution est destinée à un public de professionnels. Elle fournit aux organisations qui possèdent plusieurs sites ou aux professionnels en mobilité une solution pratique et efficace pour communiquer et tenir des réunions. Cette solution vise un

marché très large : des PME aux grandes entreprises. Pour répondre aux besoins de ces entreprises, Cisco propose plusieurs versions de sa solution. Chaque version diffère par son prix, ses caractéristiques techniques et le matériel utilisé. C90 [1] est un des produits de la solution TéléPrésence. La figure 2.6 présente ses principales caractéristiques techniques. Chaque produit possède de multiples options de configurations : dans le cas de *C*90 on peut choisir le pare-feu, la bande passante, la qualité de la vidéo... Dans cette solution de TéléPrésence *C*90, il y a 169 éléments configurables, ce qui représente plus d'un milliard de configurations.

FIGURE 2.6 – Extrait des caractéristiques techniques de la solution logicielle TéléPrésence Cisco C90

2.2.2 Une méthodologie de test empirique

La validation du produit Téléprésence est réalisée par une équipe dédiée dans les locaux de Cisco. Les équipes de tests reçoivent un produit (par exemple *C*90) et exécutent une batterie de tests (250 pour *C*90) pour le valider. Toute la difficulté pour les testeurs est de sélectionner les configurations logicielles (par exemple une résolution, une entrée audio...) du produit pour exécuter les cas de tests.

La sélection des configurations de test est une activité manuelle fastidieuse. Les testeurs doivent interpréter les spécifications fournies pour identifier les éléments présents dans les configurations. Cette phase d'interprétation est source de nombreuses erreurs, car la validité de la configuration sélectionnée dépend du degré de compréhension qu'a le testeur du système. De plus, les dépendances ne sont pas explicites, ce qui augmente

1. http ://www.cisco.com/en/US/products/ps11330/index.html

le risque d'oubli lors de la phase de configuration. Une fois ces configurations établies, les 250 tests définis sont exécutés.

2.2.3 Défis levés

L'approche manuelle de test choisie par les ingénieurs-testeurs de Cisco fait face à plusieurs problèmes :

Des activités de tests variables et bornées dans le temps.

Un des problèmes rencontrés par les testeurs est l'ajustement du rythme des activités de test avec le rythme d'évolution et de développement d'un produit. Alors que le marché des solutions de visioconférences croît, Cisco doit, en tant que leader du domaine, suivre sans cesse ce marché pour rester compétitif. Cela implique des innovations régulières qui offrent à l'utilisateur final de nouvelles fonctionnalités. Les innovations étant bien souvent demandées par le marché, les activités de tests sont souvent réalisées en temps très contraint et le temps alloué ne tient pas souvent compte de la complexité ou de la taille du système.

Des configurations de test invalides.

Le choix des configurations dans le cadre de la validation de la solution C90 de Cisco résulte d'une procédure manuelle. Les testeurs choisissent les éléments qui constituent la configuration de test. Le processus de configuration est complexe, de nombreuses dépendances cachées entre les éléments existent. Le risque de sélectionner une configuration dont toutes les dépendances ne sont pas satisfaites est élevé. Si une configuration invalide est mise en place (c'est à dire qui ne respecte pas les contraintes entre les différents éléments la composant) sa validation mènera à un échec. Le problème est qu'il est impossible pour le testeur de savoir rapidement si la faute est due à une mauvaise configuration, ou à une erreur dans le logiciel. Le temps perdu à déterminer l'origine de la faute peut alors être très important.

Une couverture de test insuffisante.

La sélection des configurations de test étant une activité manuelle, le choix des éléments à tester est laissé à la discrétion du testeur. Cette absence de critère systématique peut provoquer un déséquilibre quant aux éléments testés : une partie de l'application peut être testée excessivement aux dépends d'une autre partie. C'est ce qui a été constaté lors d'une analyse manuelle de ces configurations, il a été rapporté que certaines configurations étaient extrêmement similaires, tandis que d'autres parties de l'application étaient délaissées. Il est donc nécessaire d'apporter aux testeurs des critères objectifs permettant de les aider dans la sélection des configurations.

2.3 Des verrous à traiter

Bien que ces deux projets logiciels aient été développés pour des objectifs différents, par des compagnies différentes, les équipes de testeurs ont fait face à des problèmes similaires. L'important nombre de périphériques dans le cas de BIEW, ou l'important nombre de paramètres de configurations dans le cas du logiciel TéléPrésence empêchent l'utilisation d'approches exhaustives. Ces problèmes ne sont pas nouveaux, et se retrouvent dans d'autres projets de test :

- la validation d'applications mobiles : les systèmes d'exploitation sont de plus en plus nombreux sur le marché et tous les smartphones diffèrent de par leurs configurations matérielles et logicielles.
- la validation des logiciels embarqués sur les automobiles où de nombreux contrôleurs électroniques sont présents et peuvent être configurés de multiples façons.
- les tests de sites Internet devant maintenant, en plus de s'afficher correctement sur tous les navigateurs existants, être compatibles avec la navigation à partir d'un appareil mobile.

Nos expériences et nos discussions avec les différents acteurs des milieux industriels et académiques nous ont permis d'identifier trois problèmes récurrents rencontrés par les testeurs lors de leurs activités.

1. **Expliciter la variabilité** : les deux projets ont rencontré des problèmes de configurations invalides dues à des incompatibilités entre le matériel ou les paramètres. Les équipes n'ont aucun moyen de formaliser ces incompatibilités. Cette absence de formalisation peut engendrer une perte de temps, car il est possible d'instancier des configurations non valides.
2. **Des moyens d'échantillonner les configurations** : quand les deux projets de tests ont été mis en place, les testeurs se sont retrouvés démunis face à l'explosion combinatoire causée par les nombreuses configurations possibles du système. Ils ne connaissaient aucune approche ou méthode permettant de réduire ce nombre de configurations. Les testeurs ont donc choisi des approches basées sur leurs connaissances du système.
3. **La mise en place d'un mécanisme de traçabilité** : permettant la mise en relation des exigences et des éléments de l'environnement pour pouvoir effectuer des analyses d'impact lors de l'évolution des fractionnements ou des exigences.

Certains de ces défis ont déjà été résolus par le milieu académique, mais d'autres restent à traiter. Ce sont une partie de ces problèmes que nous proposerons de résoudre tout au long de cette thèse.

Chapitre 3

Modèles de variabilité

Alors que les deux projets de tests présentés dans le chapitre 2 ont tous les deux souffert de problèmes avec des configurations erronées et de gestion de la diversité, le monde académique, conscient de ces problèmes, propose déjà des solutions qui permettent de résoudre partiellement ces difficultés. Dans ce chapitre, nous présentons ces éléments. Il est organisé en deux parties : dans un premier temps, nous présentons les modèles de variabilité existants. Nous détaillons leurs grands principes et leurs particularités. Puis, dans un deuxième temps, nous nous focalisons sur un type de modèle : les modèles de *features*. Ces modèles serviront de support aux activités de recherche présentées dans ce document.

3.1 Les modèles de variabilité

3.1.1 Panorama des différents modèles de variabilité

Dans cette section, nous présentons les principaux modèles de variabilité qui existent. Cette présentation n'est pas exhaustive, car de nombreux formalismes ont été proposés dans la littérature. Ces formalismes sont présentés par Schobbens & Al. [SHTB07]. Nous nous concentrons sur les principaux.

3.1.1.1 Les modèles de *features*

Les modèles de *features* sont un langage de modélisation graphique. Ils permettent de capturer l'ensemble des configurations possibles d'un système par une représentation graphique. Les modèles de *features* ont été présentés pour la première fois en 1990 par Kang [KCH+90]. Depuis, de nombreux formalismes ad-hoc ont été créés. Tous ces formalismes ont été répertoriés dans la littérature [SHTB07]. Dans cette section, nous en présentons deux : les modèles de *features* et les modèles de *features* attribués.

Les modèles de *features*

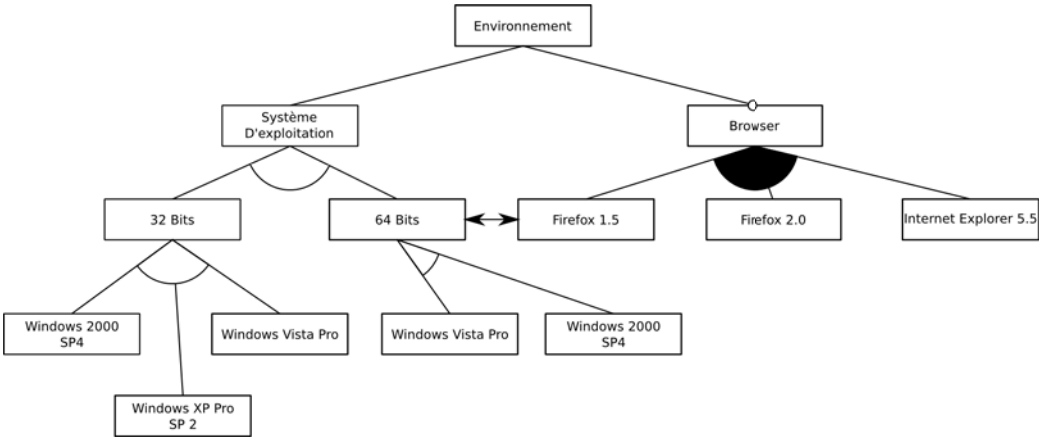


FIGURE 3.1 – Modèle de *features* issu du cas industriel BIEW représentant une partie des environnements de test

Un modèle de *features* est une représentation graphique d'un ensemble de configurations. Il est composé de plusieurs *features*. Chaque *feature* représente une fonctionnalité, un plug-in, un morceau de code, ou encore un concept. Un extrait d'un modèle de *features* est présenté figure 3.1. C'est un extrait du modèle de *features* que nous avons construit dans le cadre du projet de test BIEW. Ce modèle représente l'ensemble des environnements de tests possibles du logiciel BIEW d'Orange.

L'élément racine, la *feature* Environnement représente le concept le plus général du modèle. Il nous informe sur la nature de l'objet modélisé. En dessous de la *feature* Environnement se trouvent deux *features* : Système d'exploitation et Browser. La présence de ces deux *features* signifie qu'un Environnement est composé d'un Système d'exploitation et d'un Browser. La *feature* Système d'exploitation possède deux *features* filles qui sont 64 *bits* et 32 *bits*. Ces deux *features* filles sont reliées entre elles par un opérateur xor, signifiant une exclusivité : on ne peut pas avoir à la fois un système d'exploitation 32 bits et 64 bits. Le même raisonnement est appliqué ensuite pour les *features* filles des deux architectures.

La *feature* Browser est reliée à la *feature* Environnement par l'opérateur optionnel, ce qui signifie que dans une configuration il peut y avoir, ou non, de navigateurs. Les navigateurs sont reliés à la *feature* Browser par l'opérateur Or, ce qui permet de sélectionner un, deux ou trois navigateurs dans la configuration de test. La présence de la double flèche entre les *features* $Firefox$ 1.5 et 64$bits$ symbolise une exclusion, dans ce cas, elle signifie que $Firefox$ 1.5 et 64$bits$ ne peuvent pas être ensemble dans une même configuration.

Il existe quatre types d'opérateurs hiérarchiques :

- **And :** quand le parent est sélectionné dans un produit, la *feature* fille reliée à ce parent est obligatoirement sélectionnée.
- **Or :** quand le parent est sélectionné, une ou plusieurs *features* filles peuvent être sélectionnées.

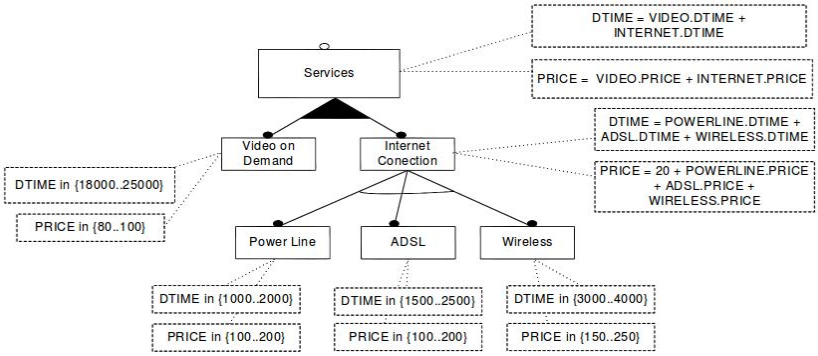


FIGURE 3.2 – Un *feature* modèle attribué extrait de [BTRC05]

- **Xor :** quand le parent est sélectionné, seule une des *features* filles peut être sélectionnée.
- **Optionnel :** quand le parent est sélectionné, la *feature* fille peut ou non être sélectionnée dans un produit.

En plus des quatre opérateurs hiérarchiques, il existe deux opérateurs transverses :

- **Require :** A require B signifie que si A est sélectionné alors B doit l'être aussi.
- **Mutex :** A mutex B signifie que si A est sélectionné alors B ne peut pas l'être et vice-versa.

La sémantique de ces opérateurs, ainsi que les éléments graphiques associés sont détaillés section 3.2.

Les modèles de *features* attribués

Benavides et al. [BTRC05], a proposé plus récemment un autre type de modèle de *features*. Ce modèle a été créé pour pouvoir ajouter des attributs qualité aux *features*. Grâce à ces attributs, les utilisateurs peuvent ajouter des informations comme le coût, un temps, ou encore un nombre de ligne de code à chaque feature, et ainsi caractériser les configurations sélectionnées.

La figure 3.2 présente un modèle de *feature* attribué où les *features* possèdent chacune deux attributs. Une particularité de ce modèle est que l'auteur associe aux *features* la façon de raisonner avec les attributs. Par exemple le coût d'une connexion est égal à la somme des coûts des *features* filles plus 20. Cette façon de raisonner peut varier en fonction de la nature de l'attribut. On ne calcule pas de la même façon le coût d'une configuration et la probabilité d'erreur.

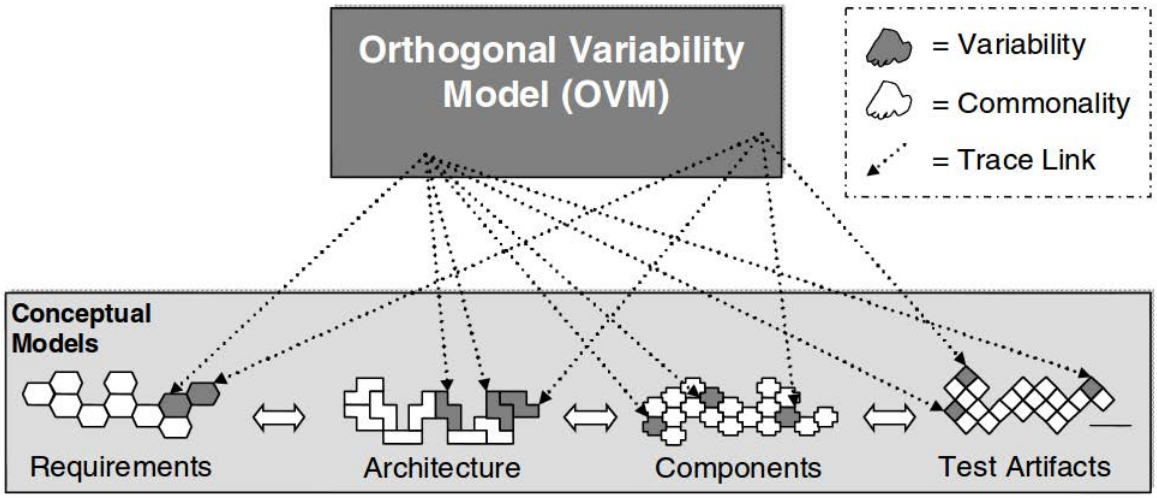


FIGURE 3.3 – OVM et les relations entre les différents artefacts logiciels. Extrait de [MP07]

3.1.1.2 Orthogonal Variability Model (OVM)

OVM est un langage de modélisation proposé par Pohl et al.[PBvdL05]. Ce langage permet de relier les différents artefacts logiciels (Exigences, architecture...) entre eux. La figure 3.3 illustre les relations entre OVM et les artefacts logiciels. Ces artefacts logiciels sont identifiés dans la figure comme *Conceptuals Models*.

Un modèle OVM est composé de deux éléments centraux : les points de variation et les variants.

Point de variation :
Il documente ce qui peut varier dans la plateforme logicielle. Si on se réfère au cas d'étude BIEW, chacune des caractéristiques environnementales est un point de variation : Mobile, OS, WiFi...

Variant :
Un variant est associé à un point de variation et représente une des possibilités existantes permettant de réaliser le point de variation. Par exemple, dans le cas de la caractéristique environnementale O.S. il y a 5 variants : Windows 2000, Windows XP 32 bits, Windows XP 64 bits, Windows Vista 32 bits, Windows Vista 64 bits.

Le langage OVM permet d'établir des relations entre ces points de variations et ces variants. On distingue dans le langage OVM deux types de relations :

Dépendances de variabilité :

Elles définissent les règles régissant le comportement des variants les uns par rapport aux autres dans un point de variation. Il existe trois types de dépendances de variabilité : optionnel, obligatoire, et alternatif. Alternatif est composé de deux attributs min et max qui spécifient le nombre de variants pouvant être sélectionnés en même temps. Dans le cas de la caractéristique environnementale O.S., le point de variation possède une dépendance de variabilité de type obligatoire, car un environnement de test possède obligatoirement un système d'exploitation.

Les contraintes :
Elles établissent des relations entre les différents variants. Ces contraintes permettent de modéliser les contraintes d'exclusions ou de besoins entre différents variants.

Un méta modèle pour ce langage a été défini dans [PBvdL05].

3.1.1.3 CVL

Le langage CVL [1] (*Common variability language*) est un langage destiné à spécifier la variabilité. Ce langage permet de faciliter la spécification, la configuration et la réalisation (l'instanciation du produit) de la variabilité pour n'importe quel langage décrit sous la forme d'un DSL respectant MOF [2]. L'architecture CVL est composée de 3 couches : le modèle d'abstraction de la variabilité (*Variability abstraction layer*, VAM), qui est le pendant du modèle de *features*. C'est dans ce modèle qu'on spécifie les points de variations. En dessous du VAM se trouve la couche réalisation du modèle de variabilité (*Variability Realization Model*), elle va faire le lien entre les éléments du *base model* (les éléments concrets) et le VAM. C'est cette couche qui va propager les conséquences des choix réalisés dans le VAM dans le *base model*. En-dessous se trouve le *base model*, qui contient les éléments du DSL (*domain specific langage*). Ce sont ces éléments qui seront instanciés lors du processus de sélection d'une configuration.

3.1.2 Des langages textuels pour représenter la variabilité TVL et Familiar

Plusieurs chercheurs se sont intéressés à l'utilisation de langages textuels pour représenter la variabilité. La principale motivation justifiant l'utilisation de texte pour représenter la variabilité est que les outils de représentation graphique sont bien souvent des prototypes de recherches et sont moins performants que les éditeurs textuels. De plus, les langages graphiques ne sont pas toujours adaptés aux standards industriels.

Deux langages textuels gérant la variabilité ont été proposés ces dernières années : le langage TVL par Classen et al. [CBH11] et Familiar *(FeAture Model scrIpt Language for manIpulation and Automatic Reasoning)* par Acher et al. [ACLF13].

1. http ://www.omgwiki.org/variability
2. http ://www.omg.org/mof

Le langage TVL (*text-based variability language*) est un langage permettant de représenter sous une forme textuelle des modèles de *features*. Il est possible de représenter dans ce langage tous les modèles de *features* déjà existants. Ce langage intègre une gestion des modèles attribués. TVL est fourni avec une grammaire et une implémentation java.

Familiar [ACLF13] est un outil qui permet de raisonner sur les modèles de variabilité. Il est basé sur le framework xtext et permet de représenter et d'effectuer de multiples opérations sur les modèles de *features*. Il fournit une solution opérationnelle à la gestion de multiples modèles de *features* et permet de cacher à l'utilisateur les détails de l'implémentation comme les appels à des solveurs permettant d'effectuer des raisonnements.

3.2 Les modèles de *features* : formalisme et définition

3.2.1 Définitions

Dans cette section nous présentons une formalisation des modèles de *features*. Il est inspiré de la définition de modèle de *features* proposé par Schobbens [SHTB07]

Definition 1 *Un modèle de features est un tuple* $FD = \langle G, r, \mathcal{F}, \mathcal{CCT} \rangle$ *où :*
- $G = \langle \mathcal{N}, \mathcal{E} \rangle$ *est un arbre où* $\mathcal{N}$ *est un ensemble de features* n *et* $E \subseteq \mathcal{F} \times \mathcal{F}$ *est un ensemble fini d'arêtes.*
- $r \in \mathcal{N}$ *est la feature racine*
- $\mathcal{F} = \{ F_{xor}, F_{or}, F_{and}, F_{optionnal}, F_{card} \}$ *un ensemble de fonctions opérateurs* $F_i \subseteq \mathcal{P}(\mathcal{N}) \times \mathcal{N}$ *ou* $F_{card} \subseteq \mathcal{P}(\mathcal{N}) \times \mathcal{N} \times \mathbb{N}$*(valeur min.)*$\times \mathbb{N}$*(valeur max.) associant un groupe de features filles à un père commun* $\mathcal{P}$*.*
- $\mathcal{CCT}$ *: un ensemble de contraintes, contenant les contraintes suivantes* $\{$*mutex, require*$\}$ *ou* $mutex \subseteq \mathcal{N} \times \mathcal{N}$ *et* $require \subseteq \mathcal{N} \times \mathcal{N}$
- *Un parent peut avoir plusieurs groupes de features filles et une feature fille n'a qu'un seul père.*

On peut définir la sémantique des opérateurs en utilisant la logique propositionnelle $(\vee, \wedge, \Leftrightarrow, \Rightarrow, \neg)$. Cette sémantique a été originellement proposée par [Man02, MC04]. La figure 3.4 établit la correspondance entre les opérateurs, les éléments graphiques et la logique propositionnelle. Notons que bien que l'opérateur Card (cardinalité) subsume tous les opérateurs pour une question de lisibilité, l'utilisation des opérateurs Or, And et Xor est privilégiée.

Un modèle de *features* représente un ensemble de configurations, une configuration est définie comme ceci :

Definition 2 ***Configuration*** *: une configuration est la sélection d'un ensemble de features, qui mène à l'assemblage d'un ensemble d'artefacts logiciel. Pour qu'une configuration soit correcte, il faut que les artefacts sélectionnés respectent les opérateurs du modèle de features, sinon on parle de configuration incorrecte ou invalide.*

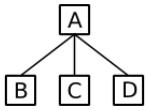
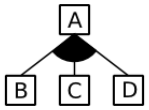
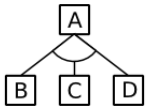
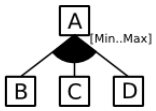
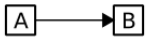
Opérateur	Représentation Graphique	Logique propositionnelle
And		$A \Leftrightarrow (B \wedge C \wedge D)$
Or		$A \Leftrightarrow (B \vee C \vee D)$
Xor		$A \Leftrightarrow (B \wedge \neg C \wedge \neg D) \vee$ $A \Leftrightarrow (\neg B \wedge C \wedge \neg D) \vee$ $A \Leftrightarrow (\neg B \wedge \neg C \wedge D)$
Optionnel		$B \Rightarrow A$
Card		Si $Min = 1$ et $Max = 2$: $A \Leftrightarrow (B \wedge \neg C \wedge \neg D) \vee$ $A \Leftrightarrow (\neg B \wedge C \wedge \neg D) \vee$ $A \Leftrightarrow (\neg B \wedge \neg C \wedge D) \vee$ $A \Leftrightarrow (B \wedge C \wedge \neg D) \vee$ $A \Leftrightarrow (B \wedge \neg C \wedge D) \vee$ $A \Leftrightarrow (\neg B \wedge C \wedge D)$
Mutex		$\neg(A \wedge B)$
Require		$A \Rightarrow B$

FIGURE 3.4 – Tableau de correspondance des opérateurs, des éléments graphiques et de la logique propositionnelle

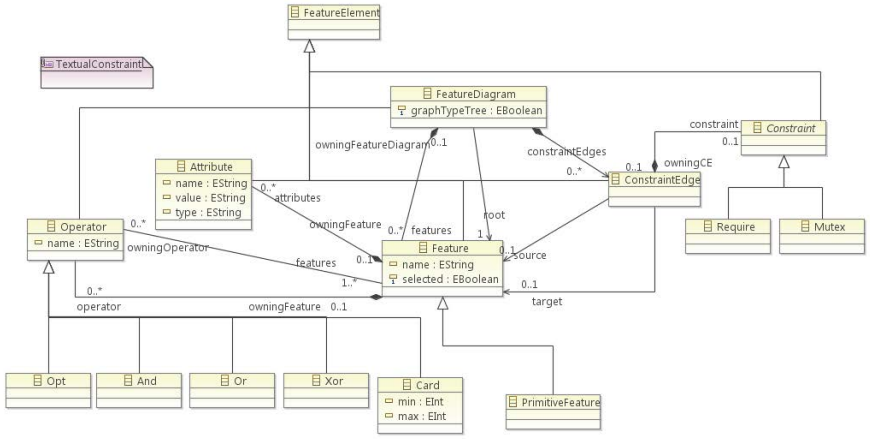


FIGURE 3.5 – Métamodèle de la syntaxe abstraite du langage FFD

On distingue deux types de *features* [TKESug] dans un modèle de *features* . Les *features* dites "concrètes", associées aux éléments concrets d'une configuration (plug-in, morceau de code, librairies) et les *features* "abstraites" permettant de structurer le modèle. Ces types de *features* sont définis ainsi :

Definition 3 ***Feature concrète***[3] *: Une feature est dite concrète si celle-ci, ou l'artefact logiciel qu'elle représente, fait partie d'une configuration.*

Definition 4 ***Feature Abstraite***[4] *: Une feature est dite abstraite si celle-ci n'est liée à aucun artefact logiciel.*

Le modèle de *features* issu du projet BIEW présenté figure 3.1 possède des *features* de ces deux types. Les *features* abstraites sont les suivantes : Environnement, Système d'exploitation System... Ces *features* n'impactent pas directement la configuration finale. Les *features* concrètes participent directement à la configuration, comme Windows 2000 Sp4, ou encore Firefox 2.0.

3.2.2 Un métamodèle pour les modèles de *features*

Un des bénéfices direct de la formalisation proposée par [SHTB07] a été la définition d'un métamodèle de la syntaxe abstraite du langage FFD. Ce métamodèle a été proposé par Perrouin et al. dans [PKGJ08]. Ce métamodèle, décrit figure 3.5. *FeatureElement*, est une métaclasse abstraite dont le rôle est de fournir un super type commun à tous les éléments du modèle pour faciliter leurs manipulations. *FeatureDiagram* est la racine du métamodèle. Cette classe a un attribut *graphTypeTree* permettant de considérer le modèle de *features* comme un arbre ou comme un graphe (ce qui est le cas en présence de contraintes entre les *features*). La présence de l'attribut *graphTypeTree*

3. Parfois nommée Primitive *Feature* dans la littérature
4. Parfois nommée Compound *Feature* dans la littérature

permet de représenter les modèles de *features* basés sur des formalismes ne contenant pas de contraintes entre les *features*. Le métamodèle contient également la liste des *features* (class $Feature$) contenues dans le modèle. Parmi elles, une est de type racine. Elle est l'élément le plus haut de la hiérarchie. Elle est identifiée par la référence *root* liant $FeatureDiagram$ à $Feature$.
Les opérateurs permettent de décrire la variabilité. On retrouve dans le métamodèle les opérateurs présentés précédemment.

3.3 Raisonner sur les modèles de *features*

3.3.1 Opérations sur les modèles de *features*.

La formalisation de la sémantique des opérateurs des modèles de *features* en logique propositionnelle a permis aux chercheurs de proposer toute une gamme d'analyses sur ces modèles. Benavides et al. dans [BSRC10] propose une revue réalisée sur 53 articles de toutes les analyses existantes sur ces modèles. Dans cette section, nous présentons les principales :

3.3.1.1 Modèle de *features* vide (*void feature model*) [KCH+90]

Cette opération prend en entrée un modèle de *features* et retourne une valeur booléenne. Un modèle de *features* est dit vide (*void*) si celui-ci ne représente aucune configuration. Un modèle de *features* vide a pour origine une mauvaise utilisation des contraintes inter-*features* (mutex, require, et les contraintes complexes sous la forme CNF).

3.3.1.2 Configuration valide [KCH+90]

Cette opération prend en entrée un modèle de *features* et une configuration puis vérifie que cette configuration est correcte par rapport au modèle ; que les *features* sélectionnées et les *features* non sélectionnées dans la configuration ne violent aucune des contraintes du modèle.

3.3.1.3 Configuration partielle valide [KCH+90]

Cette opération prend en entrée un modèle de *features* et une configuration partielle : une configuration contenant des *features* pouvant être sélectionnées ou non sélectionnées. Elle vérifie que cette configuration partielle est correcte par rapport au modèle : que les *features* sélectionnées et non sélectionnées dans la configuration partielle ne violent aucune contrainte du modèle.

3.3.1.4 Toutes les configurations [Man02]

Cette opération prend en entrée un modèle de *features* et retourne l'ensemble des configurations valides.

3.3.1.5 Nombre de configurations [Man02]

Cette opération prend en entrée un modèle de *features* et retourne le nombre de configurations valides.

3.3.1.6 Filtre [BCT04]

L'opération de filtre prend en entrée un modèle de *features* et une configuration partielle puis retourne la liste des configurations valides possibles contenant la configuration partielle.

3.3.1.7 Détection d'anomalies

Une partie des analyses sur les modèles de *features* traite de la détection d'anomalies sur les modèles de *features*.

Feature morte : [KCH+90] Une *feature* est dite morte si celle-ci n'appartient à aucune des configurations du modèle de *features*.

Feature faussement optionnelle : [ZZM04] Une *feature* est dite faussement optionnelle à partir du moment où celle-ci est présente dans tous les produits où son père est présent.

3.3.1.8 Optimisation [BCT04]

Cette opération prend en entrée un modèle de *features* attribué et une fonction de coût. Elle rend une configuration valide maximisant ou minimisant cette fonction de coût.

3.3.1.9 *features* centrales (*core features*) [TRC09]

Cette opération prend en entrée un modèle de *features* et retourne les *features* qui sont toujours sélectionnées, celles qui sont présentes dans tous les produits.

3.3.1.10 *features* variantes [TRC09]

Cette opération prend en entrée un modèle de *features* et retourne les *features* pouvant être sélectionnées ou non sélectionnées.

3.3.1.11 Analyse de dépendances [KCH+90]

Cette opération prend en entrée un modèle de *features* et une configuration partielle et complète la configuration partielle en ajoutant ou en supprimant des *features* grâce à l'utilisation de la propagation des contraintes.

3.3.1.12 Métriques

En plus des opérations retournant une ou des configurations sur les modèles de *features*, les chercheurs ont proposé plusieurs métriques sur les modèles de *features*.

Facteur de variabilité [BRCT05] : Cette métrique prend en entrée un modèle de *features* et retourne un facteur de variabilité. Ce facteur est égal au nombre de configurations divisé par 2^N où N est le nombre de *features*. Une valeur proche de 1 signifie que le modèle est peu contraignant, tandis que si le facteur se rapproche de 0 le modèle est plus contraignant.

***Commonality* [BRCT05] :** Prend en entrée une configuration partielle et retourne le pourcentage de configurations du modèle qui contiennent cette configuration partielle.

***Extra Constraint Representativeness* [KCH$^+$90] :** Cette opération prend en entrée un modèle de *features* et réalise le rapport du nombre de *features* impliquées dans des contraintes inter-*features* par le nombre de *features*.

Plus petit ancêtre commun [MWC09] : Cette opération prend en entrée un ensemble de *features* et un modèle de *features* puis retourne le plus petit ancêtre commun de l'ensemble de *features*.

3.3.1.13 Composition de modèles de *features* [ACLF10, Ach11]

Acher et al. proposent des opérations permettant de raisonner sur de multiples modèles. Constatant qu'un seul modèle de *features* n'est pas toujours suffisant pour les utilisateurs, les auteurs proposent des opérations de raisonnement sur plusieurs modèles. Les auteurs proposent trois opérations :

- le *merge*, qui va combiner deux ou plusieurs modèles pour n'en faire qu'un.
- l'insertion, permettant d'insérer un modèle dans un autre
- l'agrégation, qui prend en entrée plusieurs modèles et les agrège pour n'en faire qu'un seul.

La figure 3.6 présente les trois opérations existantes. On y voit plusieurs modèles de *features* (FM sur la figure) sur lesquels de multiples opérations sont effectuées. Sur la gauche, deux opérations d'insertions sont présentées. Deux modèles de *features* FM_{mdata} et FM_{MRI} sont ajoutés au modèle $FM_{MIsupport}$. Sur la droite, une opération de *merge* des trois modèles est présentée. Cette opération est réalisée sur des modèles possédant des *features* communes et permet la création d'un seul et unique modèle. L'opération d'agrégation va simplement proposer un modèle résultant de l'addition des deux modèles.

3.3.1.14 *Slicing* de modèle de *features* [ACLF11]

Plus récemment, Acher et al. ont proposé le *slicing* de modèle *features*. Cette opération prend en entrée un modèle de *features* et retourne une projection de ce modèle

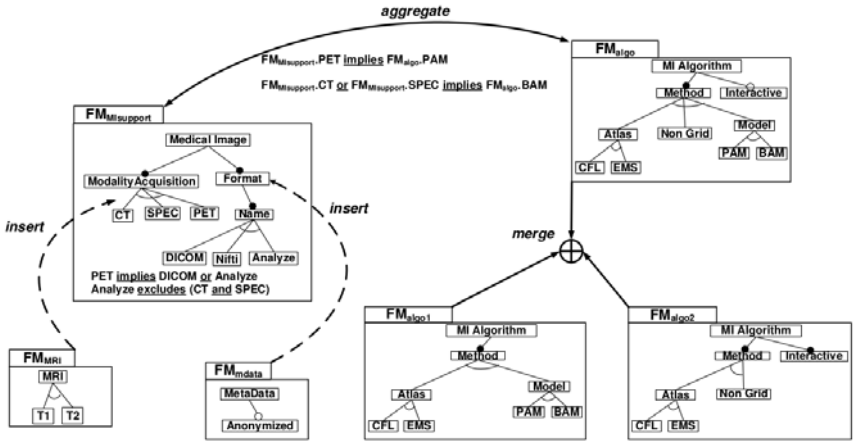


FIGURE 3.6 – Composition de modèles de *features* extraite de [Ach11]

de *features*. Le *slicing* permet d'obtenir un modèle de *features* , qui ne contient que les *features* choisies et qui respecte le modèle original.

3.3.1.15 Un besoin de support automatisé

Toutes les opérations présentées précédemment peuvent se réaliser manuellement. Cependant, ces opérations deviennent extrêmement complexes dès que le nombre de *features* croît. Les risques d'erreurs deviennent très importants pour un opérateur humain. De plus, toutes ces opérations sont répétitives. Elles demandent du temps et de la concentration. C'est pourquoi il est intéressant d'automatiser ces opérations.

3.3.2 Support à l'analyse automatique de modèles de *features*

L'automatisation des opérations sur les modèles de *features* repose sur l'utilisation d'un moteur de raisonnement. Dans la littérature, deux approches dominent : celles basées sur la logique propositionnelle et celles basées sur la programmation par contraintes.

3.3.2.1 Approches basées sur la logique propositionnelle

Les approches qui utilisent la logique proportionnelle pour raisonner sur les modèles de *features* se basent toutes sur les mêmes principes. Les *features* sont modélisées comme des variables booléennes et les relations entre les variables se font via l'utilisation des opérateurs de la logique propositionnelle. Une fois le modèle de *features* transformé en logique propositionnelle, un solveur SAT ou un solveur qui utilise les diagrammes de décision binaire (BDD pour *Binary Decision Diagram*) est appelé. Les règles qui permettent de transformer un modèle de *features* en formule de la logique propositionnelle sont présentées figure 3.4

Les solveurs SAT sont utilisés pour vérifier la satisfiabilité d'une formule propositionnelle, c'est à dire s'il est possible de trouver une valeur pour toutes les variables de la formule vérifiant la formule. Certains de ces solveurs ne prennent qu'en entrée des formules de forme normale conjonctive (CNF). Ce sont des formules booléennes exprimées avec seulement trois connecteurs : $\neg, \vee, \wedge$.

Les diagrammes de décision binaire construisent en mémoire une représentation sous la forme d'un graphe de la formule propositionnelle. Les approches basées sur les BDD sont efficaces pour compter le nombre de solutions et vérifier la satisfiabilité [Bry86].

Ci-dessous nous présentons les principaux travaux qui permettent de raisonner sur les modèles de *features* et qui utilisent la logique propositionnelle.

Approche proposée par Don Batory

Batory et el. [Bat05] proposent de lier modèle de *features* et logique propositionnelle. Les auteurs montrent que l'utilisation de la logique propositionnelle permet d'aider les utilisateurs durant les activités de conception. L'approche est implémentée dans un outil, AHEAD [5] et utilise un solveur SAT.

Approche proposée par Benavides et al.

Benvides et al.[TBRC+08] proposent un environnement écrit en java nommé Fama, qui permet de raisonner sur les modèles de *features*. Fama propose de raisonner sur les modèles de *features* en utilisant des solveurs SAT et BDD. Fama implémente plusieurs des opérations présentées section 3.3 et permet d'utiliser soit un solveur BDD, soit un solveur SAT pour les accomplir. Dans leurs travaux, les auteurs suggèrent que les BDD sont plus efficaces pour compter le nombre de configurations.

Approche proposée par Mendonca

Dans leurs travaux Mendonca et al.[MWC09] utilisent des solveurs SAT pour raisonner sur les modèles de *features*. Les auteurs utilisent des règles de correspondances similaires à celles présentées figure 3.4 et évaluent les performances des solveurs SAT sur des modèles de *features* de différentes tailles générés aléatoirement. Ils montrent que les solveurs SAT sont capables de vérifier qu'un modèle de 10000 *features* n'est pas vide en 0.4 seconde.

Les auteurs s'intéressent aussi aux raisonnements utilisant les BDD [MWCC08]. Ils montrent qu'avec un BDD et un bon paramétrage (un bon choix dans l'ordre des variables du BDD), on peut calculer efficacement le nombre de configurations de modèles de *features* contenant juqu'à 2000 *features*. Les auteurs proposent aussi un éditeur en ligne de modèles de *features* appelé SPLOT [MBC09]. Ce site permet à l'utilisateur de réaliser en ligne son propre modèle de *features* et d'y appliquer différentes analyses (nombre de configurations, nombre de *features* centrales...).

5. http ://www.cs.utexas.edu/users/schwartz/ATS.html

Approche proposée par Acher

Plus récemment Acher et al. ont proposé un outil qui permet de manipuler les modèles de *features* : Familiar [ACLF13]. Cet outil se base sur la logique propositionnelle pour raisonner sur les modèles de variabilité. Sa particularité est de raisonner sur plusieurs modèles de *features*. En plus d'implémenter les opérations de base comme entre autre le calcul du nombre de configurations, Familiar implémente les opérations d'agrégation, *merge* et d'insertion.

3.3.2.2 Approches à contraintes

Les techniques de programmation par contraintes ont aussi été utilisées pour raisonner sur les modèles de *features*. Dans cette partie, nous passons en revue les trois approches principales qui permettent de raisonner sur les modèles de *features*.

Approche proposée par Benavides

Benavides et al. sont les premiers à utiliser les techniques de programmation par contraintes pour raisonner sur les modèles de *features* [BTRC05]. Les auteurs proposent un ensemble de règles qui traduisent un modèle de *features* sous la forme d'un problème à contraintes. La figure 3.9 présente les règles de transformation que les auteurs ont établies. Le modèle à contraintes proposé permet de gérer les modèles attribués. L'implémentation est faite avec Choco [JRL08], un solveur de contraintes écrit en langage java. Cette modélisation à contraintes a été intégrée dans le framework FaMa [TBRC$^+$08].

La figure 3.7 présente un extrait du code source de Fama [6]. Cet extrait montre la méthode appelée lors de la création d'une contrainte optionnelle entre une *feature* Père et son fils. La déclaration de la contrainte se fait lignes 558 et 559. Les auteurs s'appuient sur l'implémentation native de l'implication dans Choco. L'utilisation de cette implémentation se fait par appel à la méthode *implies*. La méthode *eq* est une contrainte traduisant l'égalité : si parentVar est égal à 0, alors la contrainte *eq* est satisfaite, sinon elle ne l'est pas. Une traduction littérale de la contrainte serait la suivante : si $parentVar$ est égal à 0 alors $childVar$ est égal à 0.

Pour évaluer l'efficacité de ce modèle, les auteurs calculent le temps [BTRC05, BSTRC06] nécessaire pour compter le nombre de configurations de plusieurs modèles de *features*. Les expérimentations se font sur des modèles de taille inférieure à 20 *features*. Les expériences montrent que le temps nécessaire pour calculer toutes les configurations d'un modèle de 19 *features* est de 4000 ms.

Approches proposées par Salinesi et Maso

6. https ://code.google.com/p/famats/source/browse/branches/multistep/src/es/us/isa /ChocoReasoner/attributed/ChocoReasoner.java

```
553         protected Constraint createOptional(GenericRelation rel,
554                         GenericFeature child, GenericFeature parent) {
555
556                 IntegerVariable childVar = variables.get(child.getName());
557                 IntegerVariable parentVar = variables.get(parent.getName());
558                 Constraint optionalConstraint = implies(eq(parentVar, 0), eq(childVar,
559                                 0));
560                 getDependencies().put(rel.getName(), optionalConstraint);
561                 return optionalConstraint;
562
563         }
```

FIGURE 3.7 – Extrait du code source du modèle à contraintes Choco de Fama

$$children \in \{0,1\}, father \in \{0,1\}, children \leqslant father \tag{3.1}$$

FIGURE 3.8 – Utilisation des contraintes arithmétiques pour représenter la contrainte optionnelle

Salinesi et Maso utilisent aussi les techniques de programmation par contraintes pour raisonner sur les modèles [SMDD10] de *features* et les modèles attribués. L'implémentation proposée repose sur l'utilisation de la programmation logique via l'utilisation de GNU Prolog. Les activités de recherche ont été formalisées dans un outil appelé VariaMos [MSD+12b]. Les auteurs s'appuient sur un modèle à contraintes [SMDD10] utilisant des contraintes arithmétiques et logiques entre les variables. Dans ce langage, les *features* ont deux valeurs 0 (non sélectionnées), ou 1 (sélectionnées). Ce langage permet aussi d'associer aux *features* des variables énumérées, par exemple une *feature widht resolution* peut avoir 4 valeurs possibles : {0,800,1024,1366}. Les contraintes arithmétiques permettent de contraindre les variables en utilisant des opérateurs mathématiques : $+, -, >$ ... L'utilisation des contraintes arithmétiques pour représenter la contrainte optionnelle est présentée figure 3.8. Plus récemment, les auteurs ont établi différents modèles à contraintes pour les différents langages de variabilité proposés par le monde académique (FODA, OVM, TVL) ainsi que pour des langages de variabilité créés par des industriels. [MSD+12a].

Approche proposée par Karatas

Karatas et al. [KOD10a, KOD10b] proposent une modélisation à contraintes basée sur de la programmation logique. Les auteurs utilisent le logiciel Sicstus prolog et la libraire CLP(FD) pour modéliser le comportement des contraintes. La modélisation à contraintes gère les modèles de *features* et les modèles de *features* attribués. Pour capturer les comportements des opérateurs, les auteurs utilisent les contraintes globales (expliquées chapitre 5) fournies par le solveur à contraintes. Le modèle à contraintes de

l'auteur permet d'exprimer des contraintes de la forme suivante : le coût total du prix du produit ne doit pas excéder le budget autorisé.

Cependant, l'implémentation est peu détaillée. L'auteur évalue son approche avec un modèle de 52 *features*. Il effectue différentes opérations (modèle de *features* vide...) et rapporte seulement les temps d'exécution.

Les méthodes de raisonnement sur les modèles de *features* à l'aide des contraintes reposent toutes sur des approches similaires : les *features* sont modélisées comme des entiers à deux valeurs possibles 0 ou 1. Les relations entres ces *features* sont établies soit en utilisant des contraintes globales ([KOD10a]), des contraintes arithmétiques ([SMDD10]), ou des contraintes fournies par la librairie utilisée [BTRC05].

3.3.2.3 Logique propositionnelle ou contrainte

Les deux approches qui permettent de raisonner sur les modèles de *features* ont chacune des avantages et des inconvénients. Dans cette section, nous détaillons les points forts et les points faibles de ces deux paradigmes.

Benavides et al. sont les seuls auteurs à avoir exploré les différentes façons d'analyser un modèle de *features*. Malheureusement, ils ne proposent pas de comparaison des différences en terme de performances des deux paradigmes.

Les approches basées sur la logique propositionnelle montrent deux principales forces : une capacité à traiter de très grands modèles sans problèmes de passage à l'échelle, et une facilité pour compter le nombre de configurations d'un modèle.

Les approches basées sur la logique propositionnelle souffrent néanmoins d'un inconvénient : la difficulté à gérer les modèles de *features* attribués. En effet, les formules propositionnelles reposent sur l'utilisation de variables booléennes, ayant pour valeur 0 ou 1. Dans le cas d'un raisonnement sur un modèle à attributs, il est nécessaire d'avoir des variables qui ont un domaine de valeur plus large pour modéliser un prix, une résolution ou un poids.

Les approches à contraintes quant à elles n'ont pas été formellement évaluées sur leurs performances sur des grands modèles à contraintes.

3.3.3 Les modèles de *features* pour la recherche expérimentale

Alors que la recherche autour des modèles de *features* dure depuis plus de 20 ans, les activités de recherches ont créé plusieurs modèles de *features*. Dans cette section, nous présentons les principaux modèles de *features* utilisés dans la littérature.

La plus grande source de modèle de *features* a été à l'initiative de Mendonca [MBC09] avec la création du dépôt de modèle de *features* SPLOT. (*Software Product Line Tools*) [7]. Ce dépôt est actuellement riche de 325 modèles de *features*, allant de neuf *features* à 290. La richesse de ce dépôt s'explique par deux raisons :

– un éditeur en ligne de modèle de *features* qui permet de créer facilement un modèle de *features* et permet plusieurs analyses (nombre de configurations, *feature* morte...) automatiques.

7. http ://www.splot-research.org/

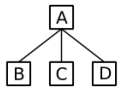
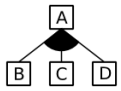
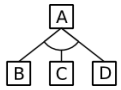
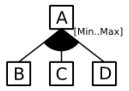
Opérateur	Représentation Graphique	Comportement
And		$A = B = C = D$
Or		**if** $A > 0$ **then** $B + C + D > 0$ **else** $B = C = D = 0$ **end if**
Xor		**if** $A > 0$ **then** $B + C + D = 1$ **else** $B = C = D = 0$ **end if**
Optionnel		**if** $B = 1$ **then** $A = 1$ **end if**
Card		**if** $A > 0$ **then** $Min < B+C+D < Max$ **else** $B = C = D = 0$ **end if**
Mutex		$A + B \leqslant 1$
Require		**if** $A > 0$ **then** $B > 0$ **end if**

FIGURE 3.9 – Tableau de correspondance des opérateurs, des éléments graphiques et du comportement de la contrainte

– une API java permettant la création et la transformation de modèles issus de SPLOT.

En plus des modèles fournis par SPLOT, il existe trois autres modèles proposés par [SLB+11]. Ces trois modèles résultent d'une activité de *reverse engineering* et ont la particularité d'être les plus gros qui existent à l'heure actuelle. Ces trois modèles sont issus du système d'exploitation linux et font respectivement 6888 *features*, 1396 *features* et 1244 *features*.

3.4 Résumé

Le raisonnement automatique sur les modèles de *features* a été largement traité ces dernières années. De multiples opérations ont été proposées et implémentées sur différents outils. Pour automatiser ces opérations, les chercheurs ont proposé des règles qui permettent de transformer un modèle de *features* vers un moteur de raisonnement. Les système utilisés pour raisonner sur les modèles de *features* peuvent se classer en deux familles : les approches utilisant des solveurs SAT/BDD et les approches se basant sur la programmation par contraintes.

Chapitre 4

Variabilité et Test

Dans les projets de test présentés dans le chapitre 2, les équipes de testeurs font face à une explosion combinatoire du nombre de configurations de test possibles : près de 2 millions pour le projet BIEW et plus de 1 milliard pour le projet de test du logiciel Téléprésence. Dans ce chapitre nous présentons les techniques de test existantes qui permettent de traiter ces problèmes d'explosion combinatoire.

Dans une première partie nous présentons le principe du test combinatoire, sa définition et comment cette technique de test a été traitée par le monde académique. Puis nous présentons comment ces techniques ont été combinées aux modèles de *features* pour sélectionner des configurations de test.

4.1 Test combinatoire

4.1.1 Motivations

Dans cette thèse, nous nous intéressons à l'utilisation d'objets mathématiques appelés *Covering Array* pour les activités de test combinatoire (*Combinatorial Interaction Testing*(CIT)). L'utilisation des techniques de test combinatoire pour le test logiciel a été proposée pour la première fois par Cohen et al. [CDG97].

La principale motivation quant à l'utilisation des techniques de test combinatoire est que de nombreux défauts logiciels sont dus à l'interaction de quelques paramètres logiciels [KW04], [BV05], [GOA05]. Les cas de tests générés à l'aide des techniques de test combinatoire visent à couvrir toutes les combinaisons possibles de t paramètres d'entrées. Ces suites de tests sont alors capables de détecter des fautes dues à l'interaction d'au plus t composants . Les techniques de test combinatoire ont été appliquées avec succès dans de multiples domaines et sur des problèmes différents [CE00], systèmes distribués [KW04], tests d'interfaces hommes-machines [AM03], localisation de fautes [YC06].

Dans cette section, nous présentons les principales définitions des objets mathématiques appelés *Covering Arrays* qui sont utilisés au cours de cette thèse, puis nous présentons les différentes approches qui permettent de créer ces objets. Ces *Covering arrays* sont des tableaux dont chaque ligne est une configuration de test.

4.1.2 Modèle combinatoire

Le test combinatoire repose sur l'utilisation d'objets mathématiques. Dans cette section, nous présentons deux de ces objets : les *covering arrays* et les *mixed covering arrays*. Finalement, nous présentons un cas particulier de ces modèles combinatoires appelé *pairwise testing*. Ce cas particulier est support à de nombreuses expérimentations dans le monde académique.

4.1.2.1 *Mixed Level covering Arrays* et *covering Array*

Formellement, les techniques de test combinatoire reposent sur des structures mathématiques appelées *covering Arrays* (CA) et *Mixed Level covering Arrays* (MCA) qui permettent d'extraire des configurations de test. Ces deux structures sont proches, les *Mixed Level covering Array* sont une version générique des *covering Arrays*.

Définition 4.1 (Un Mixed Level covering Arrays) *Un mixed level covering array* $MCA_\lambda(N; t, k, (v_1, v_2, ..., v_k))$*, de force* t *est un tableau de taille* $N \times k$ *à* v *symboles, ou* $v = \sum_{i=1}^{k} v_i$ *ayant les propriétés suivantes :*

- *Chaque colonne* $i (1 \leq i \leq k)$ *contient des éléments qui appartiennent au domaine* D_i *de taille* v_i*.*
- *Les lignes de chaque* $N \times t$ *sous-tableau couvrent tous les t-uplets de valeurs issus des t-combinaisons de colonnes au moins* λ *fois.*

Dans la pratique, dans des activités de test logiciel une valeur de 1 est choisie pour λ. En pratique si $\lambda = 1$ alors cette valeur est omise.

La figure 4.1 présente un MCA de force $t = 2$ composé de 4 variables dont les domaines de définitions sont définis ci-dessous :

- $A \in \{0, 1, 2, 3\}$
- $B \in \{a, b, c\}$
- $C \in \{4, 5, 6\}$
- $D \in \{d, e\}$

Lorsque la force d'un MCA est de deux on parle de *pairwise testing*. Pour chaque paire de variables, toutes les combinaisons possibles sont testées. Ainsi pour le couple (A, B) les paires suivantes sont testées :

$\{0, a\}$,$\{0, b\}$,$\{0, c\}$,$\{1, a\}$,$\{1, b\}$,$\{1, c\}$,$\{2, a\}$, $\{2, b\}$,$\{2, c\}$,$\{3, a\}$,$\{3, b\}$, $\{3, c\}$

toutes ces paires vont donc être présentes dans le MCA créé. Le même principe est appliqué à toutes les paires possibles de tous les couples suivants :

(A, C),(A, D),(B, C),(B, D),(C, D)

.

La figure 4.1 présente le MCA de taille minimum : elle est de 12 lignes et il y a 12 paires possibles entre A et B. La matrice de couverture a une taille minimale.

Définition 4.2 (Valeur minimale d'un MCA) *La valeur de matrice de couverture* $MCA(N; t, k, (v_1, v_2, ..., v_k))$ *est le plus petit* N *tel que le* $MCA(b; t, k, (v_1, v_2, ..., v_k))$ *de taille* b *existe.*

0	a	4	d
2	b	6	e
3	c	5	e
2	c	4	d
0	b	5	d
1	a	6	e
1	b	4	d
3	a	6	d
0	c	6	e
2	a	5	e
3	b	4	e
1	c	5	d

FIGURE 4.1 –
$MCA(12; 2, 4, (4, 3, 3, 2)$ extrait de [CGMC03]

Trouver une valeur optimale de N est un problème NP-complet. Il est difficile de trouver la plus petite valeur possible pour N. Un des défis rencontrés lors de la création de MCA est de déterminer le nombre de lignes nécessaires permettant de couvrir l'ensemble des t-combinaisons. Il n'existe pas de formule exacte permettant de déterminer pour toute valeur de t,k et v_i la valeur optimale de N.

Les *covering Array* sont une spécialisation des MCA, ils sont définis de cette façon :

Définition 4.3 (covering Array) *Un covering array $CA_\lambda(N; t, k, v)$ est un tableau de taille $N \times k$ sur v variables tel que les lignes de chaque $N \times t$ sous tableaux couvrent au moins une fois tous les t-tuples de valeurs issues des t-combinaisons de colonnes au moins λ fois.*

Quand la valeur *lambda* est égale à 1, celle-ci n'est pas affichée dans la définition.

Définition 4.4 (La valeur minimale du covering Array) *La valeur de matrice de couverture CAN(t,k,v) (covering Array Number) est le plus petit N tel que le $CA(t, k, v)$ detaille N existe.*

La majeure différence entre un *covering Array* et un *Mixed Level covering Array* réside dans les valeurs que peuvent prendre les variables v. Pour un *Mixed Level covering Array* pour chaque v_i, D_i peut être différent tandis que pour un *covering Array* tous les D_i sont identiques : $D_1 = D_2 \cdots = D_k$.

4.1.2.2 Le *pairwise* testing

Le pairwise testing est une activité de test qui utilise un cas particulier de *covering array*, dans lequel on cherche à couvrir des paires et non plus des tuples. Lors d'une activité de test *pairwise*, les données de test sont sélectionnées de telle façon que chaque

Var 1	Var 2	Var 3	Var 4	Var 5	Var 6	Var 7	Var 8	Var 9	Var 10
0	0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	0	0	0
1	0	1	1	1	0	0	1	1	0
1	1	0	1	0	0	1	0	1	1
1	1	1	0	0	1	0	1	0	1
0	0	0	0	1	1	1	1	1	1

FIGURE 4.2 –
$CA(6; 2, 6, 2)$– *covering array* couvrant Pairiwse pour 6 variables

combinaison de chaque paire de valeurs soit couverte. L'utilisation du *pairwise testing* dans le domaine du test logiciel a été présentée pour la première fois par Cohen et al. [CDG97]. L'intérêt du *pairwise testing* est de détecter des défauts déclenchés par l'interaction de deux paramètres. Il est évidemment possible d'augmenter le degré de couverture (3,4,..,n) mais la conséquence directe est une augmentation du nombre de cas de test, et donc du temps et de l'effort de test.

Mathématiquement, le *pairwise testing* se traduit de cette façon :
- $MCA(N; 2, k, (v_1, v_2, ..., v_k))$
- $CA(N; 2, k, v)$

Dans le cadre du *pairwise testing* sur des variables à deux valeurs ($k = v = 2$), il est possible de déterminer pour différentes valeurs de N (nombre de lignes) le nombre d'interactions de variables possibles que l'on peut couvrir (i.e. la valeur de k (nombre de variables)). Cette formule a été mise en évidence indépendamment par Kleitman et Spencer [KS73] et Katona [Kat73].

$$\binom{n-1}{\lceil \frac{n}{2} \rceil} \geq k \tag{4.1}$$

Pour une valeur de k importante, la taille de N croît de façon logarithmique. Pour une matrice de 6 lignes, il est possible de couvrir jusqu'à $k = 10$ variables. Un exemple de *covering Array* couvrant le critère *Pairwise* est présenté figure 4.2. Pour $N = 7$, il est possible d'appliquer le critère *Pairwise* à 140 variables.

4.1.2.3 Approches existantes

Plusieurs approches existent pour générer des configurations de test couvrant y interactions pour un ensemble de variables. Cohen et al. [CGMC03] présentent la plupart des méthodes de génération de *covering array* existantes. Dans cette section, nous présentons les principales approches.

Approches utilisant des constructions mathématiques :

Plusieurs méthodes de création de *covering array* utilisent des techniques mathématiques [SM99]. Ces méthodes sont déterministes et reproductibles. L'un des inconvénients de ces approches est qu'elles ne sont pas aussi génériques que les méthodes basées sur des heuristiques et ne fonctionnent que pour un sous ensemble de valeurs de k, t et v.

Approches utilisant des algorithmes gloutons

De nombreux travaux explorent la génération de *covering array* avec des algorithmes gloutons. Il existe deux façons d'utiliser les algorithmes gloutons pour créer des covering array.

La première est utilisée par les approches suivantes : AETG [CDG97], Deterministic Density Algorithm [CCT04] et PICT [Cze06]. L'algorithme de génération construit le *covering Array* ligne par ligne. Chaque nouvelle ligne ajoutée permet de couvrir de nouvelles interactions. L'algorithme glouton maximise le nombre d'interactions couvertes dans chaque ligne en y mettant un maximum de paires. Les approches proposées dans l'académie se distinguent entre elles dans la façon de choisir les paires couvertes dans les configurations.

La seconde méthode qui permet de créer des *covering Array* à l'aide d'algorithmes gloutons est celle utilisée par In Parameter Order Algorithm (IPO) [TL02, LKK$^+$08]. Cet algorithme sélectionne d'abord un petit nombre de variables ($k' < k$), et résout le problème *pairwise* sur ce sous problème. Puis l'algorithme ajoute de façon incrémentale les variables et fait grossir la solution de façon horizontale (résultant de l'ajout d'une variable au *covering Array*) et de façon verticale (dû au besoin d'ajouter de nouvelles lignes pour atteindre le critère *pairwise*).

Méta-heuristiques :

Pour la génération de *covering array* à l'aide de méta heuristique quatre techniques sont utilisées : le recuit simulé [CCL03, GCD09] et la méthode de descente (*hill climbing*), les algorithmes génétiques [Sta01], algorithme de recherche tabou [Nur04]. Ces algorithmes fonctionnent tous selon le même principe : une solution de taille N est trouvée. Et, à l'aide d'une fonction de coût, plusieurs solutions sont essayées, et évaluées, jusqu'à ce qu'une solution de taille $N' < N$ soit trouvée ou que le temps imparti soit écoulé.

Approche basée sur la programmation par contraintes :

Des chercheurs [HPSS06] ont exploré comment utiliser les techniques de programmation par contrainte pour la génération de *covering array*. Dans l'approche proposée, les auteurs associent à chaque tuple un entier. Par exemple dans le cas d'une paire de variables A et B, ayant pour valeur 0 ou 1, la paire $(0, 0)$ est associé à la valeur 0, la paire $(0, 1)$ est associé à la valeur 1, et ainsi de suite. Dans la solution proposée les colonnes ne représentent plus des *features* mais des tuples. Par exemple la colonne 1 peut représenter la paire (A, B). Les lignes elles, correspondent aux configurations. Si

la première cellule de la première colonne est égale à la valeur 0, ça signifie que dans la configuration numéro 1 la paire $(0, 0)$ est présente, et ainsi de suite. L'approche présentée ne gère pas les contraintes entre les *features*, mais permet la minimisation.

Approches permettant une gestion des contraintes :

Certaines de ces approches ont été étendues pour considérer les contraintes entre les variables. Ces approches reposent sur l'utilisation conjointe d'un algorithme et d'un solveur SAT. Cohen et al. ont étendu leurs travaux originaux (algorithmes gloutons et méta heuristiques) pour gérer des contraintes. Des approches plus récentes se sont aussi intéressées à ce problème, comme celle proposée par Segall reposant sur l'utilisation de diagramme binaire de décisions [STBF11]. Ces approches utilisent des contraintes exprimées grâce à l'utilisation de la logique booléenne.

Le tableau 4.4 présente une comparaison des différents outils. Ces outils sont étudiés selon les trois critères détaillés ci-dessous.

Gestion des contraintes :

L'un des inconvénients majeurs est que ces approches ne peuvent pas générer des configurations *pairwise* sur des modèles de features. En effet, soit les outils génèrent des configurations de test sans contraintes, ce qui rend ces configurations inutilisables telles quelles. Soit les outils prennent en entrée un langage à contraintes différent du formalisme des modèles de features. Ce langage est bien souvent trop large, et même s'il est possible d'utiliser ce langage pour exprimer le formalisme des modèles de *features*, un travail d'adaptation est nécessaire. La figure 4.3 présente un extrait du langage utilisé par l'outil CASA pour exprimer des contraintes entre les variables. Cet exemple est extrait de la documentation de CASA [1]. Les deux premières lignes indiquent que l'on fait du *pairwise testing* (deux), et que 5 variables sont impliquées (5). La troisième ligne décrit le nombre de valeurs que peuvent prendre ces 5 variables : 2 2 2 2 3. Ici il y 4 variables binaires et une ternaire. Les lignes suivantes permettent d'exprimer les contraintes entre les variables. La quatrième ligne indique qu'il y aura deux contraintes exprimées sous la forme de disjonction. La cinquième ligne signifie que pour la prochaine contrainte, 2 variables seront concernées. La ligne $-0-8$ est la contrainte elle-même et signifie qui si la variable ternaire prend la plus petite valeur, alors la première variable binaire ne peut pas prendre la valeur la plus petite.

1. http ://cse.unl.edu/ citportal/tools/casa/documentation2.php

```
[1] 2
[2] 5
[3] 2 2 2 2 3
[4] 2
[5] 2
[6] - 0 - 8
```

FIGURE 4.3 – Langage utilisé par l'outil CASA pour exprimer des contraintes entre les variables.

Déterministe :

Tous les algorithmes ne proposent pas des approches déterministes. Ce non-déterminisme peut poser problème lors de l'utilisation en industrie. En effet la norme ISO 17025 [ISO05] (destinée à l'origine aux laboratoires d'analyse, mais qui peut se porter pour les activités de test logiciel) impose qu'un processus de test soit reproductible dans le temps, or l'utilisation d'algorithmes non déterministes ne permet pas de reproduire les résultats d'une exécution : pour les mêmes paramètres fournis en entrée, les résultats diffèrent.

Optimum Global :

Tous les algorithmes ne permettent pas d'atteindre la valeur optimale de configuration, c'est-à-dire un nombre minimum de configurations de test. Dépendant du projet de test, le nombre de configurations à tester peut être très important. Si le projet de test a des contraintes de temps fortes, il peut être stratégique de vouloir obtenir le plus petit nombre possible de configurations à tester.

Tous les outils présentés précédemment couvrent partiellement ces critères. Cependant, aucun ne permet de générer des configurations *covering Array* couvrant le critère *pairwise*, de façon déterministe, gérant les contraintes et permettant d'atteindre un nombre optimal de configurations.

4.2 Test Combinatoire et variabilité

Dans cette section nous présentons les travaux qui permettent de justifier la pertinence de l'utilisation des techniques de test combinatoire pour valider des systèmes configurables. Puis nous présentons les approches existantes pour appliquer le test combinatoire sur les modèles de features. Cette problématique est une problématique de recherche actuelle, une part importante des travaux présentés ont été menés au cours des trois dernières années, en parallèle des travaux de cette thèse.

Approche	Déterministe	Minimum Global	Gestions des contraintes
AETG [CDG97]	✗	✗	✓
CASA [GCD09]	✗	✓	✓
IPOG [TL02, LKK+08]	✗	✗	✓
Mathématique [SM99]	✓	✓	✗
Deterministic Density Algorithm [CCT04]	✓	✗	✗
Test Case Generator [TA00]	✓	✗	✓
PICT [Cze06]	✓	✗	✓
Algorithme Tabou [Nur04]	✗	✗	✗
Algorithme Génétique [Sta01]	✗	✗	✗
Approche à contraintes [HPSS06]	?	✓	✓

FIGURE 4.4 –
Tableau comparatif des différentes approches existantes

4.2.1 Test combinatoire et systèmes configurables

Yilman et al. [YC06] montrent l'efficacité du test combinatoire. En utilisant des *covering array* les auteurs diminuent de 50 % à 99 % le nombre de configurations à tester tout en obtenant des résultats quasi-similaires, en terme de défauts détectés, que les approches exhaustives.

Plus récemment, Johanssen et al. [JHF+12b] montrent la pertinence de l'utilisation des *covering Array* couvrant *pairwise* dans deux produits logiciels : un module de sûreté développé par ABB ainsi que l'environnement de développement Eclipse. Pour réaliser cette évaluation, les auteurs réutilisent les cas de tests unitaires. Ils supposent que si les cas de tests unitaires d'un module sont tous réussis lorsque celui-ci est testé unitairement, alors les cas de tests unitaires doivent être de nouveau réussis lors de l'ajout de ce module dans une configuration. Les auteurs appliquent cette théorie sur les deux produits logiciels testés. Ils sélectionnent un ensemble de configurations qui couvrent le critère *pairwise* et exécutent les cas de tests unitaires des différents composants. Pour le module de sûreté, onze configurations sont testées. Cinq configurations sont identifiées comme erronées : pour quatre de ces configurations, les erreurs ont pour origine une mauvaise modélisation. La dernière erreur est due quant à elle à une faute logicielle causée par une mauvaise interaction de composants. Pour la ligne de produit Eclipse, 13 configurations couvrant *pairwise* sont créées et 8 contiennent des modules défaillants, i.e. dont au moins un des tests unitaires a échoué.

Plusieurs auteurs ont travaillé sur la génération de configurations de test *pairwise* à partir de modèles de *features*. Les sections suivantes décrivent ces différents travaux.

4.2.2 Les approches proposées par Perrouin et al.

Perrouin et Al proposent deux approches pour la génération de configuration de test couvrant le critère *pairwise* sur les modèles de *features*.

4.2.2.1 *Pairwise* et Alloy

La première approche proposée par Perrouin et al. permet de générer des *covering Array* de différents degrés (jusqu'à trois) sur des modèles de *features*. Les auteurs utilisent Alloy qui permet de raisonner sur les modèle de *features*.

Alloy est un outil d'analyse formelle. Il permet de spécifier des contraintes entre des variables. Le solveur vérifie que les valeurs de ces variables respectent les contraintes du modèle.

Perrouin et al. utilisent une méthodologie en quatre étapes pour couvrir le critère *pairwise* sur les modèles de *features*.

Le modèle de *features* est transformé en un modèle Alloy, ce qui permet de raisonner sur ce modèle. Ensuite, l'ensemble des paires possibles est généré. Ce qui inclut les paires invalides, c'est-à-dire celles qui ne seront jamais présentes dans une configuration car étant interdites par les contraintes du modèle de *features*. Ces paires sont ensuite analysées et les paires invalides sont supprimées en vérifiant que chaque paire est présente dans au moins une configuration valide. Puis les auteurs utilisent un algorithme qui permet d'arranger les paires.

Deux stratégies sont proposées par les auteurs pour arranger les paires valides : *Binary Split* et *Incremental Growth*.

Binary Split consiste à réduire la taille du problème en divisant les pairs valides identifiées en deux sous-problèmes. Si le problème de plus petite taille ne peut être résolu, il est redivisé et ainsi de suite.
Incremental Growth est un algorithme où les paires sont ajoutées de façon incrémentale. L'algorithme sélectionne une paire, l'ajoute à la configuration courante, si la configuration est correcte c'est à dire que la configuration est valide au regard du modèle de *features*), une autre paire est sélectionnée et le processus se répète. Sinon la paire est rejetée et une autre est sélectionnée.

Cette approche souffre d'un problème de passage à l'échelle, il a été impossible pour les auteurs de couvrir *pairwise* sur des modèles de tailles supérieures ou égales à 88 features.
Basé sur ce constat, et sur le fait que les modèles sont de plus en plus gros (le plus gros modèle de *features* existant contient 5323 *features* [SLB+10]), les auteurs explorent une autre approche dans laquelle le but n'est plus de couvrir toutes les paires possibles, mais d'en couvrir un maximum en un temps contraint.

4.2.2.2 Maximisation du nombre de paires couvertes

Dans cette approche, les auteurs [PH12] changent de méthode de génération de configurations en relâchant la contrainte de couverture de 100% de paires.
L'approche proposée diffère de l'approche précédente : il ne s'agit plus de couvrir 100 % des paires valides, mais de couvrir le maximum de paires, en un temps imparti pour un nombre de configurations donné. Pour réaliser cet objectif, un solveur SAT est utilisé. L'algorithme prend en entrée un nombre de configurations, un temps de génération, et a pour objectif de maximiser le nombre de paires couvertes en respectant le nombre de configurations donné dans un temps imparti.

L'approche est évaluée sur trois modèles proposés par She et al. [SLB+11]. Ces modèles portent sur la variabilité de Linux. L'évaluation expérimentale montre qu'en 30 minutes il est possible de couvrir plus de 90 % des paires valides avec 50 configurations et plus de 99 % des paires valides en 100 configurations.

Bien que cette approche offre des performances intéressantes, elle est difficile à appliquer dans un processus de test industriel rigoureux. En effet, en ayant couvert 90 % des paires, il reste toujours 10 % des paires à tester. Il faut donc identifier ces paires, et sélectionner les configurations nécessaires permettant de couvrir les paires restantes.

4.2.3 L'approche proposée par Oster : MosoPolite

Dans cette approche les auteurs représentent les modèles de *features* sous leurs formes propositionnelles pour ensuite utiliser un algorithme glouton.

Grâce à cette représentation sous la forme de la logique propositionnelle, l'outil extrait l'ensemble de paires valides. Puis l'algorithme appliqué est le suivant. Une fois la première paire sélectionnée, l'algorithme utilise le mécanisme de propagation par contrainte pour enlever les paires incompatibles avec la configuration courante. Puis l'algorithme sélectionne parmi les paires compatibles restantes une paire à ajouter et ainsi de suite.

Les approches proposées par [OMR10] et [PSK+10] ont été comparées dans ce journal [POS+12]. Il en ressort que l'approche basée sur les solveurs à contraintes et un algorithme glouton est plus efficace aussi bien en temps de résolution qu'en nombre de configurations obtenues que l'approche proposée par Perrouin et al.

4.2.4 L'approche proposée par Johanssen : SPLCATool

L'approche proposée par Johanssen [JHF11, JHF12a] est à la fois la plus récente des trois et une des plus complètes. L'auteur utilise une méthode similaire : génération des paires avant l'arrangement de ces paires pour couvrir *pairwise* sur les modèles de features.

Approche	Déterministe	Minimum Global	Disponible en Téléchargement
SPLCATool [JHF11, JHF12a]	✗	✗	✓
MosoPolite [OMR10]	✗	✗	✓(intégré dans un produit logiciel payant)
Perrouin et al [PSK+10]	✗	✗	✗

FIGURE 4.5 –
Tableau comparatif des outils de générations de configurations *pairwise* à l'aide de modèles de *features*

Dans ce travail, l'auteur propose une solution 100% java utilisant un solveur SAT, Sat4j [LBP10]. Le solveur SAT permet de vérifier la satisfiabilité des configurations générées par rapport au modèle de features.

L'auteur propose deux algorithmes pour couvrir *pairwise* sur les modèles de *features* : une adaptation de l'algorithme de Chvatal [Chv79] qui permet de gérer les contraintes entre les variables. Le deuxième algorithme proposé est nommé ICPL. Il repose sur les mêmes principes que le premier mais les actions redondantes sont supprimées.

L'approche de Johanssen est la première à réussir à passer le problème de passage à l'échelle. L'outil est capable de générer des *covering Array* sur des modèles de grandes tailles, dont un possédant plus de 7 000 features, en 33 000 secondes (9 heures et 10 minutes). Pour les modèles de *features* de taille plus petite, l'outil proposé permet d'atteindre le critère *pairwise* en 240 secondes pour un modèle de 1 396 features, et cinq secondes pour un modèle de 224 features.

Le logiciel est open-source et est librement téléchargeable [2] ce qui permet une reproduction des expériences.

4.3 Discussion

Toutes ces approches montrent que les techniques de test combinatoire sur les modèles de *features* donnent de bons résultats : le processus d'échantillonnage permet de réduire drastiquement la combinatoire, et il a été montré dans plusieurs projets [JHF+12b, YC06] que ces techniques permettent de détecter efficacement les fautes. Toutes les techniques existantes ne sont pas égales en termes de performances : certaines sont très rapides et fournissent de bon résultats, tandis que les premières approches sont plus lentes et moins bonnes en terme de configurations échantillonnées.

Cependant, les techniques de générations de configurations *pairwise* sur les modèles de *features* actuelles souffrent de deux faiblesses. Le tableau figure 4.5 en fait le résumé.

2. http ://heim.ifi.uio.no/martifag/

Aucune des approches ne propose un algorithme déterministe, ce qui empêche toute reproductibilité du processus de génération des configurations. Cette incapacité à reproduire les configurations de test peut être un frein quant à l'applicabilité en industrie. Les processus de tests de certaines organisations imposent la reproductibilité. De plus, aucune des approches proposées ne permet de minimiser ce nombre de configurations. Les outils fournissent un résultat définitif et il est impossible pour le testeur de réduire ce nombre. Cependant, il peut parfois être utile pour les industriels de réduire ce nombre de configurations, même de une ou deux si le coût de test est très élevé. Ces trois outils ne sont pas tous disponibles en téléchargement libre. L'outil SPLCATool est open source sous une licence *Eclipse Public License*. L'outil MosoPolite a été intégré à une solution commerciale de gestion de la variabilité pure variants [3]. Tandis que l'outil utilisé par Perrouin et al. n'est pas disponible en ligne.

3. http ://www.pure-systems.com

Chapitre 5

Programmation par contraintes

De nombreux travaux de recherche utilisent les techniques SAT pour raisonner sur les modèles de features ou pour aider à la résolution pairwise ([JHF12a, CCL03, ACLF13, Bat05, KTS+09, MWC09, MBC09]. Au contraire, les travaux autour de l'utilisation de la programmation par contraintes pour raisonner sur les modèles de features et la résolution pairwise sont minoritaires [BTRC05, KOD10a, OMR10]. Dans cette thèse, nous nous intéressons à comment cette technique de programmation peut aider à tester des systèmes configurables, comme le logiciel BIEW ou l'application Cisco Téléprésence. Dans ce chapitre nous présentons les principaux mécanismes associés à la programmation par contraintes. Nous présentons d'abord les principes de ce paradigme puis nous définissons formellement un problème à contraintes. Finalement, nous expliquons les mécanismes élémentaires utilisés pour la résolution d'un système à contraintes.

5.1 Principes de la programmation par contraintes

La programmation par contraintes est un paradigme de programmation où un problème est décrit sous la forme de variables et de relations entre ces variables. Ces relations sont appelées contraintes. La résolution d'un problème à contraintes se déroule en deux étapes distinctes : l'expression du problème à résoudre sous la forme d'un problème de satisfaction de contraintes, appelé aussi CSP (*Constraint Satisfaction Problem*), et une phase de résolution.

Les techniques de programmation par contraintes sont très utilisées pour la résolution des problèmes d'ordonnancements [MJPL92] ou des problèmes avec une combinatoire élevée, tels que les problèmes de planification, placement, emploi du temps ou gestion des ressources.

5.2 Définition d'un problème de satisfaction de contraintes

On définit un problème de satisfaction de contraintes de cette façon :

Définition 5.1 (Problème de satisfaction de contraintes) ***(Constraint Satisfaction Problem, CSP)***

Un CSP est un tuple $< X, D, C >$ *ou :*

- X *est un n-uplet de variables :* $X =< x_1, x_2, ..., x_n >$.
- D *est un n-uplet de domaines, où chaque domaine est associé à une variable :* $D =< D_1, D_2, ..., D_n >$ *où* $x_i \in D_i$
- C *est un ensemble de contraintes* $C =< C_1, C2, ..., C_t >$

En programmation par contraintes, une contrainte sur un ensemble de variables est une restriction sur les valeurs que ces variables peuvent prendre simultanément. Les solveurs à contraintes proposent deux types de contraintes : les contraintes primitives et les contraintes utilisateurs.

Les contraintes primitives sont des contraintes gérées nativement par le solveur. La librairie CLP(FD) [COC97], du solveur à contraintes Sicstus Prolog [COC97], fournit des contraintes primitives de plusieurs types.

- Les contraintes arithmétiques, qui établissent des relations arithmétiques entre les variables par exemple : $A > B$, qui spécifie que A doit être strictement supérieur à B.
- Les contraintes combinatoires, qui permettent de mettre en relation un ensemble de variables comme la contrainte $all\_different$ imposant que les variables soient de valeurs différentes.

En plus de ces contraintes préexistantes, les solveurs à contraintes fournissent un mécanisme d'extension : les contraintes utilisateurs. Les contraintes utilisateurs permettent à l'utilisateur de créer ses propres contraintes. Pour créer une contrainte utilisateurs, il faut définir plusieurs éléments :

- Son identifiant
- Un ensemble de variables
- Un algorithme de filtrage
- Des conditions de réveil

Afin d'illustrer le rôle de ces différents éléments nous présentons le processus de création de la contrainte utilisateur $exactly(X, L, N)$. Cette contrainte est satisfaite si l'entier X est présent exactement N fois dans la liste L. Elle a pour identifiant $eaxctly$. Nous supposons que les valeurs de X et de N sont données. Les variables de la contrainte sont les éléments L_i de la liste L. Les conditions de réveil portent sur chacun des L_i de la liste L. Dès que le domaine de définition d'un L_i est modifié, une condition de réveil est satisfaite, ce qui déclenche l'algorithme de filtrage. L'algorithme de filtrage permet d'effectuer des choix sur les domaines des variables, ici les L_i. Par exemple l'algorithme de filtrage peut supprimer une ou plusieurs valeurs du domaine d'une variable. L'algorithme de filtrage de cette contrainte est présenté dans la figure 5.1. Il se comporte de quatre façons différentes. Ces comportements sont les suivants :

1. Dans le premier cas, si la valeur N est égale à 0, c'est-à-dire si la valeur X ne doit pas être présente dans la liste L, alors l'algorithme de filtrage impose que tous les L_i soient différents de la valeur X.
2. Dans le deuxième cas, si le nombre total d'occurrences de la valeur X, N, est égale au nombre d'éléments m de la liste L alors l'algorithme de filtrage impose

que tous les L_i contenus dans L soient égaux à X.

3. Le troisième cas correspond à la situation où un élément L_i de la liste L prend la valeur X. Dans ce-cas là il faut que parmi tous les éléments de L restant l'élément X apparaisse $N-1$ fois.
4. Le quatrième cas correspond à la situation où un élément L_i de la liste L prend une valeur différente de X. Dans ce-cas là il faut que parmi tous les éléments de L restant l'élément X apparaisse N fois.

[1] $N = 0 \Rightarrow L_1 \neq X \dots L_m \neq X$
[2] $N = m \Rightarrow L_1 = X \dots L_m = X$
[3] $L_i = X \Rightarrow exactly(X, L_1 \dots L_{i-1}, L_{i+1} \dots L_m, N-1)$
[4] $L_i \neq X \Rightarrow exactly(X, L_1 \dots L_{i-1}, L_{i+1} \dots L_m, N)$

FIGURE 5.1 – Algorithme de filtrage de la contrainte exactly

La résolution d'un CSP aboutit à l'obtention de une ou plusieurs solutions.

Définition 5.2 (Solution) *Une solution est une assignation de toutes les variables du problème à un ensemble de valeurs qui ne viole aucune contrainte.*

5.3 Mécanismes de résolution d'un problème à contraintes.

Une fois l'ensemble des variables du problème définies et que l'ensemble les contraintes ont été spécifiées, la phase de résolution peut être déclenchée. Le but de la phase de résolution est de trouver pour chacune des variables une valeurs appartenant à son domaine de définition afin que l'ensemble des contraintes du problème soit satisfait. Pour trouver cet ensemble de valeur, un arbre de recherche est construit. C'est cet arbre de recherche qui est parcouru pour obtenir une solution au problème.

Les techniques de programmation par contraintes utilisent deux mécanismes imbriqués :

- un algorithme de recherche avec des capacités de *backtracking* (capacité à revenir sur une décision antérieure pour explorer de nouvelles pistes) qui parcourt l'arbre de recherche
- la propagation des contraintes qui permet de réduire l'ensemble des valeurs possibles des variables

Ces deux techniques, utilisées de façon complémentaire, permettent au solveur de contraintes d'aboutir plus rapidement à une solution.

5.3.1 La propagation des contraintes

La propagation des contraintes est un mécanisme qui consiste à réduire le domaine de variables impliquées dans une contrainte, en supprimant les valeurs incompatibles avec les contraintes. Soit le CSP suivant :

- $X =< A, B >$

- $A \in [0, 5]$ et $B \in [3, 7]$
- $C = A > B$

La propagation des contraintes permet de réduire le domaine des variables du problème. Dans ce CSP, les deux variables sont reliées par la contrainte $A > B$. À partir de cette contrainte il est possible d'opérer la réduction de domaine suivante : $A' \in [4, 5]$ et $B' \in [3, 4]$, où A' et B' sont respectivement les nouveaux domaines de A et B après application de la contrainte. Cette réduction du domaine des variables permet de réduire la taille de l'arbre de recherche (de 42 paires de valeurs possibles avec les domaines de A et B originaux, nous passons à quatre paires de valeurs possibles $(4, 3), (5, 3), (5, 4)$).

5.3.2 Résolution des contraintes

La résolution d'un problème à contraintes repose sur l'utilisation d'algorithmes d'énumération ayant des capacités de *backtracking*. Ces algorithmes vont parcourir l'arbre de recherche du problème afin d'obtenir un ensemble de valeurs pour chacune des variables respectant les contraintes du problème.

On appelle arbre de recherche une représentation sous la forme d'un arbre de l'ensemble des valeurs possibles de chaque variable du problème. La figure 5.2 présente un extrait de l'arbre de recherche créé pour le CSP suivant :

- $X =< A, B >$
- $A \in [0, 5]$ et $B \in [3, 7]$
- $C = \{\varnothing\}$

Pour ce problème il est possible de créer deux arbres de recherche : celui qui est présenté figure 5.2, où la première variable énumérée est A, et un deuxième ou la première variable énumérée est B. L'ordre d'énumération des variables est l'un des paramètres qu'il est possible de modifier pour optimiser la résolution du problème. Ces paramètres seront décrits dans la section suivante.

Les contraintes entre les variables permettent de réduire la taille de l'arbre de recherche. Soit le CSP suivant :

- $X =< A, B >$
- $A \in [0, 5]$ et $B \in [3, 7]$
- $C = \{A > B\}$

L'introduction de la contrainte $A > B$ permet de réduire les domaines de valeurs de A et B, ce qui impacte directement la taille de l'arbre de recherche. Sans avoir fait aucune hypothèse sur les valeurs de A ou de B il est possible de réduire leurs domaines de variations : A passe de $[0, 5]$ à $[4, 5]$ et B passe de $[3, 7]$ à $[3, 4]$. L'arbre de recherche résultant est présenté figure 5.3. Cette propagation est dynamique, si un choix de valeur est fait sur une des variables du problème, une réduction des domaines a lieu. Par exemple si A a la valeur 4 alors B doit avoir la valeur 3.

Le parcours de ces arbres se fait grâce à des algorithmes de *backtracking*, dont le principe est présenté figure 5.4. Pour chaque variable du problème, l' algorithme prend une valeur du domaine, instancie cette variable à cette valeur, puis prend une autre variable du domaine et ainsi de suite.

L'algorithme de *backtracking* est appelé lors de l'étape de résolution du problème.

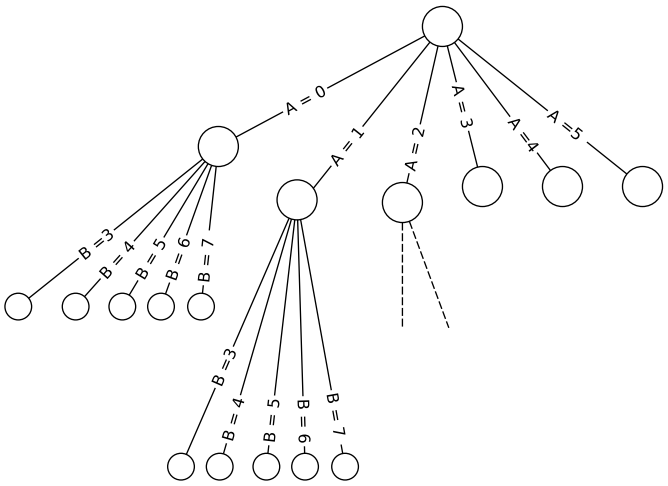


FIGURE 5.2 – Extrait de l'arbre de recherche d'un CSP simple

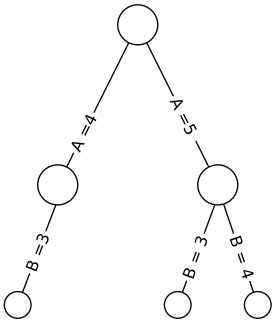


FIGURE 5.3 – Arbre de recherche d'un CSP, avec la contrainte $A > B$

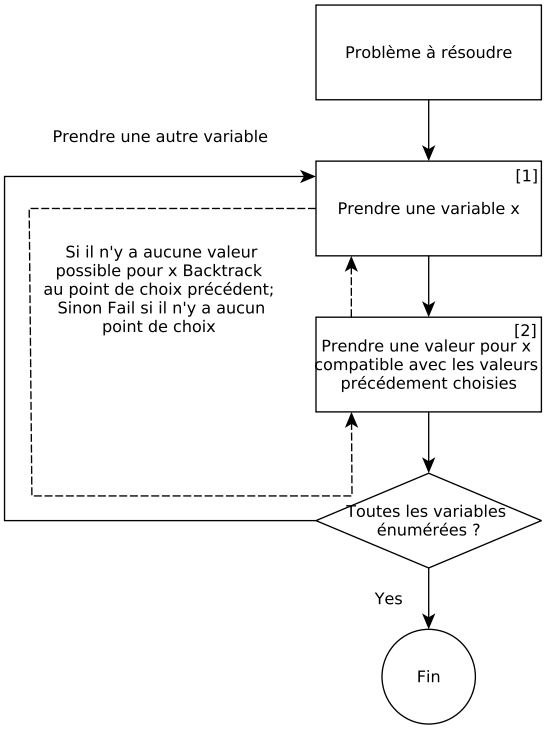


FIGURE 5.4 – Représentation d'un algorithme d'énumération extrait de [Tsa93]

Il peut être paramétré de plusieurs manières. Les points paramétrables sont identifiés figure 5.4 avec des étiquettes numérotées. Ces paramètres sont les suivants :

- l'ordre des variables à énumérer ([1]).
- l'ordre de parcours du domaine ([2])

5.3.2.1 L'ordre d'énumération des variables

Un problème à contraintes est composé d'un ensemble de variables. L'efficacité de la résolution dépend parfois de l'ordre des variables sélectionnées. Sur le CSP présenté précédemment, il y a deux ordres possibles pour les variables : soit la variable A est énumérée en première puis B ou vice versa. C'est une énumération naïve où l'algorithme prend la première variable, puis la seconde et ainsi de suite.

Il existe plusieurs stratégies d'énumération. La première, la plus simple est appelée *left-most*. Cette stratégie d'énumération prend en entrée la liste des variables à énumérer et va sélectionner les variables dans l'ordre de la liste : de la gauche, vers la droite. Pour la résolution de problèmes plus complexes, il existe des stratégies de choix de variables plus fines comme le *first-fail constraint (FFC)*, qui se base sur le principe suivant : « Élaguer l'arbre de recherche au plus tôt ». Cette stratégie utilise deux critères pour effectuer le choix de la variable à énumérer : la taille du domaine de la variable et le nombre de contraintes où la variable est impliquée. L'algorithme *first-fail constraint* choisit d'abord les variables ayant le plus petit domaine, c'est-à-dire celles ayant un nombre de valeurs limitées. Si plusieurs variables ont la même taille de domaine, *first-fail constraint* choisit celle qui est impliquée dans le plus de contraintes, et sinon la variable la plus à gauche est choisie. *First-fail constraint* est une fonction de choix dynamique, c'est-à-dire que dès que l'algorithme d'énumération fait un choix, *first-fail constraint* va parcourir une nouvelle fois la liste des variables restantes à instancier, réévaluer les différents critères et effectuer un nouveau choix.

5.3.2.2 L'ordre du parcours du domaine

L'ordre du parcours du domaine permet de préciser comment les valeurs du domaine de la variable vont être énumérées. Le domaine d'une variable peut être parcouru de plusieurs façons : de la valeur la plus grande à la plus petite, de la plus petite à la plus grande ou encore de façon aléatoire. Dans l'arbre de recherche présenté figure 5.2 le parcours du domaine de définitions des variables se fait de la plus petite valeur à la plus grande : A est d'abord assigné à la valeur 0, puis 1, et ainsi de suite...

5.3.3 Résolution de problèmes d'optimisation

Il est possible d'associer à un problème de résolution de contraintes une fonction de coût à optimiser. Si un objectif de coût est donné, le solveur à contraintes peut parcourir l'arbre de recherche de façon à trouver une solution au problème maximisant ou minimisant cette fonction.
Un des mécanismes qui rentre en jeu dans le cadre de la résolution d'un problème d'optimisation est le *branch and bound*. À chaque nœud de l'arbre de recherche, le solveur

à contraintes est capable de déterminer l'intervalle de valeurs dans lequel se trouve la valeur à optimiser. Grâce à cette estimation, il est possible de couper des branches de l'arbre de recherche pour atteindre plus rapidement la valeur minimum ou maximum recherchée. Dans le cas d'une minimisation, si la valeur minimum du coût courant est supériure à la valeur du plus petit coût trouvé, alors le solveur va arrêter l'exploration de la branche et passer à la suivante.
Il arrive cependant que l'arbre de recherche soit très important, à cause d'une combinatoire très élevée. Il est alors possible de joindre au mécanisme d'optimisation un *time out* qui a pour effet, à son arrivée à zéro, de retourner la meilleure solution trouvée dans le temps imparti.
Supposons que l'on souhaite minimiser la fonction de coût $f = A + B$ sur l'arbre de recherche présenté figure 5.2. L'algorithme de recherche va parcourir la première branche : $A = 0$ et $B = 3$ et va obtenir pour première valeur de f la valeur 3. Ensuite, l'algorithme va automatiquement couper les autres branches de l'arbre, car si $B = 4$, la nouvelle valeur de f va être supérieure à la plus petite valeur trouvée. De cette façon le mécanisme de *branch and bound* va couper l'ensemble des branches de l'arbre de recherche.

5.4 Résumé

Dans ce chapitre, nous avons présenté les principes élémentaires de la programmation par contraintes ainsi que les principaux mécanismes associés à ce paradigme. Dans cette thèse, nous proposons d'utiliser cette technique de programmation pour résoudre le problème de la génération de configurations respectant le critère pairwise sur les modèles de features. Les contraintes sont utilisées de deux façons : pour raisonner sur les modèles de features, et pour résoudre le problème de l'extraction de configurations permettant de couvrir le critère *pairwise*.

Deuxième partie

Contributions

Chapitre 6

Génération de configurations de test à l'aide de la programmation par contraintes

6.1 Introduction

Un des défis rencontrés par les équipes de test des projets BIEW et Téléprésence réside dans la sélection des configurations de test. Nous proposons d'utiliser le critère *pairwise* pour la sélection de ces configurations. Plusieurs approches ont été proposées par le monde académique pour la sélection de configurations de test *pairwise* à partir d'un modèle de *features* : SPLCATool [JHF11, JHF12a], MosoPolite [OMR10] et Perrouin et al. [PSK+10]. Cependant, toutes ces méthodes ne donnent pas les mêmes résultats en terme de nombre de configurations. Alors que les projets de test sont bien souvent soumis à des contraintes de temps et d'argent importantes, il est intéressant de sélectionner un nombre de configurations de la plus petite taille possible. Pour pouvoir être utilisable dans des contextes industriels répondants à la norme 17025 [ISO05], le processus de résolution doit être reproductible.

Dans ce chapitre, nous présentons les mécanismes que nous avons développés afin de sélectionner des configurations de test qui couvrent le critère *pairwise*. Nous présentons une vue d'ensemble de notre solution technique de sélection de configurations à l'aide des contraintes. Puis nous expliquons pourquoi la sélection de configurations de test *pairwise* est un problème complexe en présence d'un modèle de feature. Nous décrivons le processus de résolution déployé pour la sélection des configurations et le modèle à contraintes créé. Nous présentons comment nous pouvons paramétrer la résolution afin d'obtenir des configurations de tailles plus petites et comment nous avons pu embarquer un processus de minimisation. Ces différents paramètres sont évalués dans le chapitre 7. Finalement, nous décrivons le produit logiciel qui embarque tous les développements présentés : Pacogen.

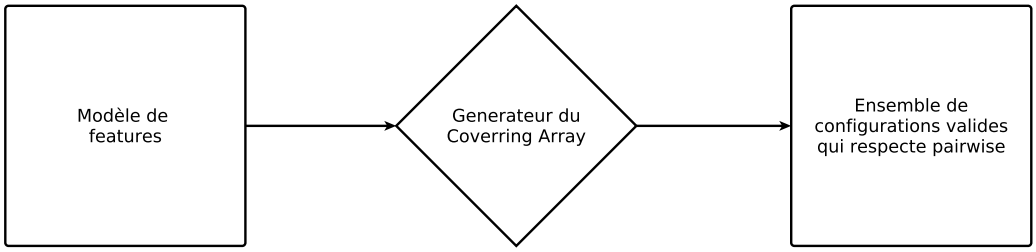


FIGURE 6.1 – Vue globale de la solution proposée

6.2 Sélection de configurations de test qui couvrent *pairwise* à partir d'un modèle de *features*

Pour résoudre le problème de la sélection de configurations de test qui couvrent le critère *pairwise* sur des modèles de features, nous proposons une solution qui prend en entrée un modèle de *features* et qui sélectionne un ensemble de configurations de test respectant le critère *pairwise*. Ces configurations sont conservées dans une structure de donnée appelée Matrice Solution. La figure 6.1 présente une vue globale de la solution que nous proposons. Dans cette section, nous détaillons les éléments présentés dans la figure 6.1.

6.2.1 Les modèles de *features*

L'approche proposée prend en entrée un modèle de features. Ce sont des configurations issues de ce modèle qui sont sélectionnées pour atteindre le critère *pairwise*. Les modèles pris en entrée sont des modèles issus du dépôt de modèle de *features* SPLOT [MBC09] ou des instances du métamodèle proposé par Perrouin et al. [PKGJ08] présenté section 3.2.2. Un exemple de modèle de *features* est présenté figure 6.2. Ce modèle de *features* a été créé à l'aide des spécifications environnementales du projet BIEW présenté dans le chapitre 2 ainsi que grâce à la connaissance des testeurs du projet.L'extrait présenté est composé de 13 *features* et représente 64 configurations. Cet exemple sera utilisé tout au long de cette section pour illustrer les mécanismes développés.

6.2.2 Le générateur de *Covering Array*

Le générateur de *Covering Array* est chargé de la sélection des configurations de test qui vont couvrir le critère *pairwise*. La génération repose sur la création d'un modèle à contraintes du problème de la sélection des configurations de test qui couvrent Pairwise. Ce modèle est composé de deux types de contraintes : celles qui permettent de capturer la sémantique du modèle de *features* et celles qui permettent d'imposer la présence de paires. C'est cette modélisation à contraintes qui est détaillée dans ce chapitre.

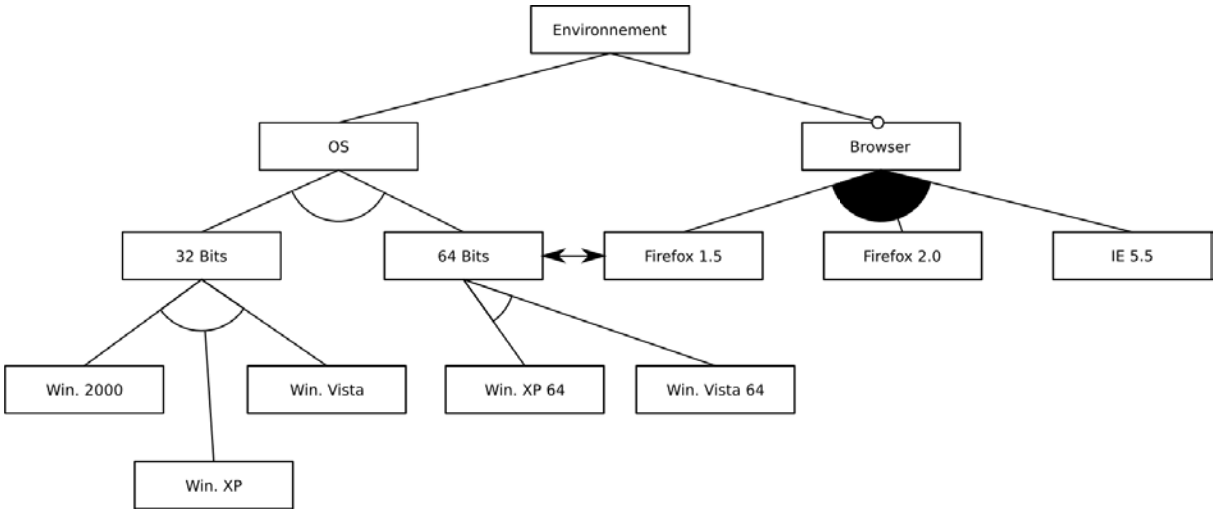


FIGURE 6.2 – Extrait du modèle de *features* représentant les environnements de test du logiciel BIEW

6.2.3 Ensemble de configurations valides qui respecte *pairwise*

L'ensemble de configurations valides qui respecte *pairwise* est produit par le générateur de *Covering Array*. Cet ensemble est contenu dans une structure de données appelée Matrice Solution, c'est un tableau. Dans ce tableau, chaque *feature* du modèle de *features* est représentée par une colonne. Chaque ligne de la matrice est une configuration. Une cellule peut prendre deux valeurs : 0 qui signifie que la *feature* n'est pas sélectionnée et 1 qui signifie que la *feature* est sélectionnée. La figure 6.3 présente la Matrice Solution produite sur le modèle de *features* du projet BIEW de la figure 6.2. Cette matrice possède 13 colonnes : une par *feature*, et fait 11 lignes : une par configuration. Dans la suite de ce document, les *features* ENVIRONNEMENT, FIREFOX 1.5, FIREFOX 2 et INTERNET EXPLORER 5.5 seront respectivement nommées ENV, FF 1.5, FF 2 et IE 5.5

6.3 Défis liés à la sélection de configurations de test *pairwise* sur les modèles de features

La sélection des configurations de test qui respectent le critère *pairwise* sur les modèles de *features* soulève trois problèmes :

1. La détection et la suppression de paires invalides
2. L'assemblage des paires valides de façon à aboutir à un ensemble de configurations respectant le modèle de features
3. L'incapacité à prédire le nombre minimum de configurations nécessaires pour couvrir pairwise.

Env	OS	32 Bits	64 Bits	Win. 2000	Win. XP	Win. Vista	Win. Xp 64	Win. Vista 64	Browser	FF 1.5	FF 2	IE 5.5
1	1	0	1	0	0	0	0	1	1	0	1	1
1	1	1	0	0	0	1	0	0	1	1	1	0
1	1	1	0	0	0	1	0	0	1	1	0	1
1	1	0	1	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	0	1	0	1	0	1	1
1	1	0	1	0	0	0	1	0	0	0	0	0
1	1	1	0	0	0	1	0	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0	0
1	1	1	0	1	0	0	0	0	0	0	0	0
1	1	1	0	0	1	0	0	0	1	1	1	1
1	1	1	0	1	0	0	0	0	1	1	1	1

Figure 6.3 – Matrice solution des 11 configurations couvrant *pairwise*, créée à partir du modèle de *features* BIEW présenté figure 6.2

Dans cette section, nous présentons et illustrons ces défis.

6.3.1 La détection des paires invalides

Si l'on fait abstraction de l'ensemble des opérateurs qui composent un modèle de features, il faut que les configurations sélectionnées, pour couvrir *pairwise*, pour ces trois *features* 64 Bits, Win. Xp 64 et Win. Vista 64 à deux valeurs, contiennent les paires suivantes :

(64 Bits,Win. Xp 64) : $(0,0)$,$(0,1)$,$(1,1)$,$(1,0)$
(64 Bits,Win. Vista 64) : $(0,0)$,$(0,1)$,$(1,1)$,$(1,0)$
(Win. Xp 64,Win. Vista 64) : $(0,0)$,$(0,1)$,$(1,1)$,$(1,0)$

Cependant, l'application des techniques de test combinatoire, pour extraire des configurations qui couvrent le critère *pairwise* sur les modèles de *features*, est plus complexe.

La présence de contraintes entre les *features* invalide certaines paires : certaines interactions entre *features* sont interdites par les opérateurs liant ces features.

Par exemple dans le modèle de *features* présenté figure 6.2 les *features* 64 Bits, Win. Xp 64 et Win. Vista 64 sont liées par l'opérateur xor. 64 Bits est la *feature* père des deux *features* filles Win. Xp 64 et Win. Vista 64. Les *features* Win. Xp 64 et Win. Vista 64 sont reliées entres-elles par une relation d'exclusivité : xor(64Bits,[Win. Xp 64, Win. Vista 64]). Cet opérateur permet de préciser qu'un environnement de test est composé d'un seul et unique système d'exploitation 64 Bits : soit Win. Xp 64 soit

Features	64 Bits	Win. Xp 64	Win. Vista 64
	1	0	0

Figure 6.4 – Un assemblage de paires valides aboutissant à une configuration invalide

Win. Vista 64. La présence de l'opérateur xor invalide certaines paires comme la paire (1,1) pour le couple de *features* (Win. Xp 64, Win. Vista 64) car il n'est pas possible d'avoir deux systèmes d'exploitation en même temps. Après application de l'opérateur xor sur ces trois *features* 64 Bits, Win. Xp 64 et Win. Vista 64, il n'y a plus que neuf paires valides :

(64 Bits, Win. Xp 64) : $(0,0)$,$(1,0)$,$(1,1)$
(64 Bits, Win. Vista 64) : $(0,0)$,$(1,0)$,$(1,1)$
(Win. Xp 64, Win. Vista 64) : $(0,0)$,$(0,1)$,$(1,0)$

Les trois paires invalides sont les suivantes : la paire $(0,1)$ pour les couples (64 Bits, Win. Vista 64) et (64 Bits, Win. Xp 64), car il n'est pas possible qu'un système d'exploitation soit sélectionné si la *feature* 64 Bits ne l'est pas. Et la paire $(1,1)$ pour le couple (Win. Xp 64, Win. Vista 64) car il n'est pas possible d'avoir dans une même configuration les *features* Win. Xp 64 et Win. Vista 64. Pour sélectionner des configurations de test qui respectent le critère *pairwise* à partir d'un modèle de features, il faut développer des mécanismes pour identifier et supprimer les paires invalides.

6.3.2 La sélection de configurations qui respectent le modèle de features

La suppression des paires invalides n'est pas suffisante dans le cadre de la résolution du *pairwise* sur les modèles de features. Les opérateurs du modèle n'autorisent pas n'importe quels arrangements de paires valides. Ces arrangements doivent respecter les opérateurs du modèle de variabilité. Dans l'exemple précédent, nous identifions neuf paires valides :

(64 Bits, Win. Xp 64) : $(0,0)$,$(1,0)$,$(1,1)$
(64 Bits, Win. Vista 64) : $(0,0)$,$(1,0)$,$(1,1)$
(Win. Xp 64, Win. Vista 64) : $(0,0)$,$(0,1)$,$(1,0)$

Pour atteindre le critère *pairwise*, il faut combiner ces différentes paires dans plusieurs configurations. La présence de l'opérateur xor ne permet pas de combiner librement ces paires. Par exemple, la configuration présentée figure 6.4 est composée de trois paires valides : (64 Bits,Win. Xp 64) : $(1,0)$, (64 Bits,Win. Vista 64) : $(1,0)$ et (Win. Xp 64,Win. Vista 64) : $(0,0)$, mais elle est fausse. En effet la *feature* père est sélectionnée mais aucune des *features* fille ne l'est.

Au contraire l'assemblage des paires valides suivantes (64 Bits,Win. Xp 64) : $(1,1)$, (64 Bits,Win. Vista 64) : $(1,0)$ et (Win. Xp 64,Win. Vista 64) : $(1,0)$, présenté dans la figure 6.5 est valide.

Features	64 Bits	Win. Xp 64	Win. Vista 64
	1	1	0

Figure 6.5 – Un assemblage de paires valides aboutissant à une configuration valide

Un simple filtrage des paires invalides n'est pas suffisant, il est nécessaire de prendre en compte les opérateurs du modèle lors de l'assemblage des paires valides pour aboutir à la sélection de configurations valides.

6.3.3 Incapacité à prévoir à priori le nombre de configurations nécessaires pour couvrir *pairwise*

Dans cette thèse, nous proposons d'utiliser la programmation par contraintes pour la sélection de configurations de tests qui respectent le critère *pairwise*. Pour réaliser cet objectif, nous posons un ensemble de contraintes (qui seront détaillées section 6.4.1) sur la Matrice Solution. Un des prérequis avant la résolution du problème est donc de définir cette Matrice Solution : en spécifier le nombre de colonnes et le nombre de lignes. Ces informations doivent être définies avant la résolution, car c'est sur cette Matrice Solution que sont posées les contraintes. Une fois les contraintes posées, il n'est plus possible de faire évoluer la Matrice Solution pour ajouter une ligne. La structure de donnée n'est pas capable de grandir dynamiquement. Il faut donc définir arbitrairement la taille de la matrice.
Le nombre de colonnes de la Matrice Solution est égal au nombre de *features* du modèle. La détermination du nombre de lignes est une tâche plus complexe. La présence des opérateurs entre les *features* ne permet pas l'utilisation de la formule proposée par Kleitman, Spencer et Katone présentée dans l'équation 6.1 pour estimer le nombre de lignes nécessaire. Cette formule permet dans le cas du critère pairwise de déterminer le nombre k de variables que l'on peut arranger dans n lignes.

$$\left(\binom{n-1}{\lceil \frac{n}{2} \rceil}\right) \geq k \tag{6.1}$$

On ne peut pas savoir s'il faudra plus de configurations ou moins de configurations que la valeur proposée par la formule. Pour illustrer ce problème, prenons un cas d'application du critère *pairwise* pour trois *features* ($k = 3$) à deux valeurs, 0 ou 1. Le nombre minimal de configurations nécessaires selon la formule est de quatre ($n = 4$). Les configurations associées dans ce cas-là (trois *features* et quatre lignes) sont présentées figure 6.6.

Si la contrainte $C = \{(\neg A \wedge \neg B \wedge \neg C) \vee (A \wedge B \wedge C)\}$ est posée, alors seulement deux configurations sont nécessaires pour couvrir pairwise : $(A = B = C = 1)$ et $(A = B = C = 0)$. Cependant, si la contrainte suivante est posée : $C_3 = \{\neg(\neg A \wedge \neg B \wedge \neg C) \vee (A \wedge B \wedge C)\}$ alors il faut six configurations pour couvrir pairwise. Ces

ligne	A	B	C
[1]	1	1	1
[2]	1	0	0
[3]	0	1	0
[4]	0	0	1

FIGURE 6.6 – $CA(4; 2, 3, 2)$ de taille minimale

configurations sont présentées figure 6.7

ligne	A	B	C
[1]	0	1	1
[2]	0	1	0
[3]	1	0	1
[4]	1	0	0
[5]	0	0	1
[6]	1	1	0

FIGURE 6.7 – $CA(4; 2, 3, 2)$ de taille minimale avec pour contrainte $C = \{(\neg A \vee \neg B \vee \neg C) \wedge (A \vee B \vee C)\}$

Une Matrice Solution de taille inadaptée peut avoir deux conséquences : soit la matrice est de taille trop petite, c'est-à-dire plus petite que la plus petite des solutions existantes, dans ce cas il sera impossible de trouver une solution. Soit la Matrice Solution est de taille trop grande, ce qui augmentera la taille du problème à contraintes et dégradera les performances.

6.4 Processus de résolution

Dans cette section, nous présentons en détail la procédure que nous avons créée pour répondre aux deux premiers défis présentés. Le problème du choix de la taille de la matrice sera abordé dans le chapitre 7. Ce processus de résolution est présenté figure 6.8. Les étapes de ce processus sont détaillées dans cette section. La partie grisée du processus présente le cœur de la contribution. La section suivante (section 6.5) est dédiée à son explication.

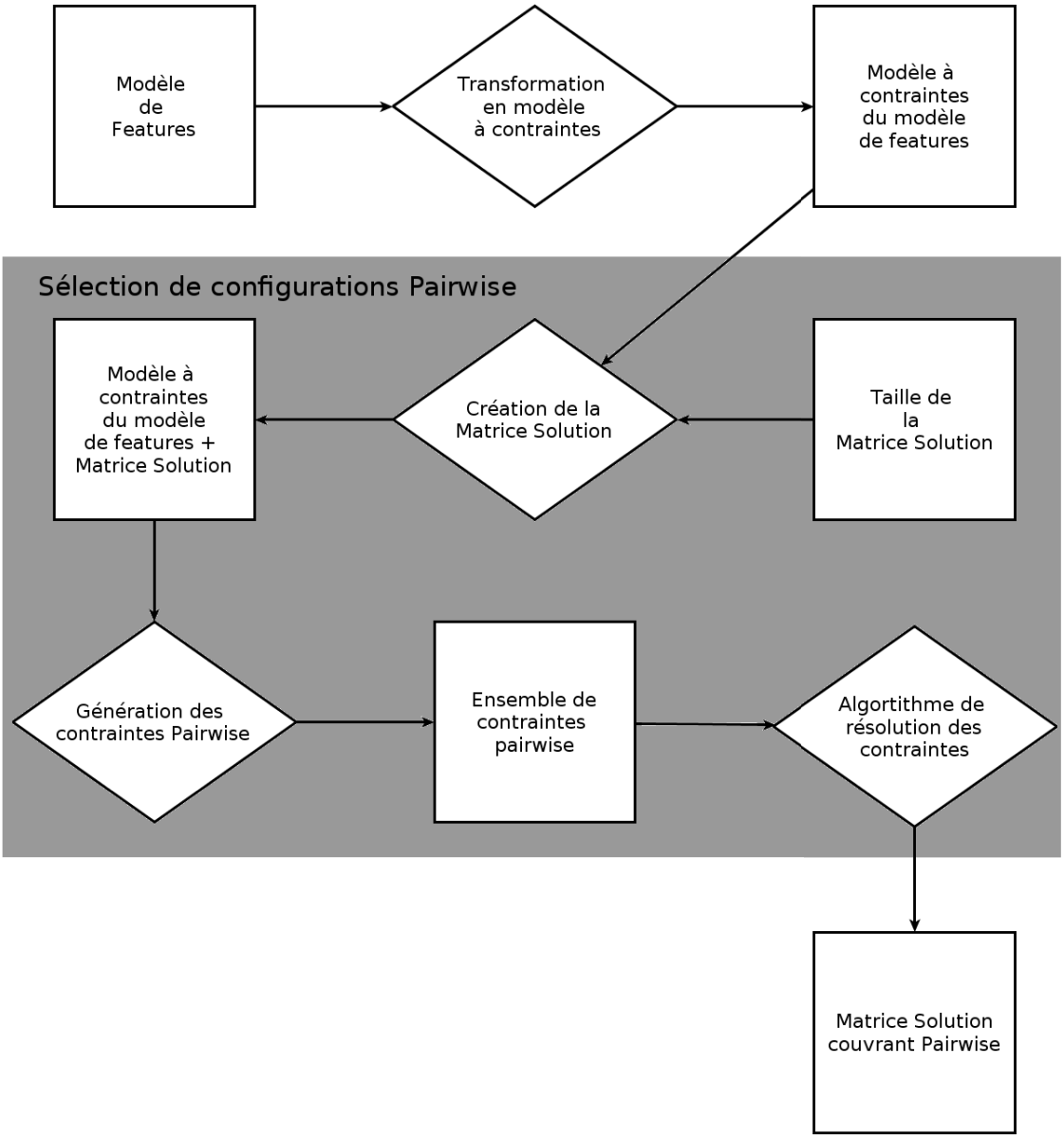


FIGURE 6.8 – Processus de sélection de configurations qui respectent le critère *pairwise*

6.4.1 Transformation du modèle de *features* en modèle à contraintes

Pendant cette étape, le modèle de *features* pris en entrée est transformé en un modèle à contraintes. C'est ce modèle à contraintes qui va permettre de raisonner sur le modèle de features. Le modèle à contraintes du modèle de *features* sera utilisé pour s'assurer de la validité des paires sélectionnées (problème numéro 1 présenté 6.3) et pour s'assurer que les configurations sélectionnées sont bien valides (problème numéro 2 présenté 6.3).

Dans le modèle à contraintes créé, une *feature* du modèle de *features* est représentée par une variable qui peut avoir pour valeur 0 (feature non sélectionnée) ou 1 (feature sélectionnée). Ces variables sont ensuite reliées entres-elles avec des contraintes pour pouvoir raisonner, c'est-à-dire être capable de déduire l'état d'une variable en fonction de l'état d'une autre variable.

Les contraintes utilisateurs permettent d'encapsuler le comportement d'un ensemble de variables sous une interface claire. Nous avons créé sept contraintes utilisateurs qui permettent de modéliser les comportements des opérateurs d'un modèle de *features* :

- $\mathsf{and}(A, [B, C, ...])$
- $\mathsf{or}(A, [B, C, ...])$
- $\mathsf{xor}(A, [B, C, ...])$
- $\mathsf{opt}(A, [B, C, ...])$
- $\mathsf{card}(N, M, A, [B, C, ...])$
- $\mathsf{mutex}(A, B)$
- $\mathsf{require}(A, B)$

La correspondance entre les opérateurs, les représentations graphiques, les comportements et les contraintes utilisateurs est présentée dans la figure 6.9.

Ici A, B, C, sont les variables associées aux *features* du modèle et $and, or, opt, ..$ sont les opérateurs du modèle de features. On distingue deux types de contraintes : les contraintes dites hiérarchiques, reliant une *feature* à ses *features* filles : $and, or, opt, xor, card$ et les contraintes inter-features : $mutex$ et $require$. La représentation à contraintes du modèle de *features* du projet BIEW présenté figure 6.2 est présentée figure 6.10.

Pour chacune des contraintes utilisateurs, nous avons conçu un algorithme de filtrage dédié qui capture le comportement attendu de l'opérateur. L'algorithme de filtrage associé à la contrainte utilisateur *and* est présenté figure 6.11. Le prédicat $and\_solver(P, LF, Actions)$ est composé de trois paramètres : P est la variable qui représente la *feature* Père, LF est une liste qui contient l'ensemble des *features* filles, et $Actions$ est une liste qui contiendra l'ensemble des actions à réaliser après l'exécution de cet algorithme de filtrage.

Le premier prédicat appelé par l'algorithme est $filter(LF, M, N)$ (ligne 3). Ce prédicat parcourt la liste des *features* filles. La variable M contiendra le nombre de *features* non sélectionnées (dont les valeurs sont égales à 0) et la variable N contiendra le nombre de *features* sélectionnées (dont leurs valeurs sont égales à 1). En fonction des valeurs de P, M et N, certaines actions sont réalisées. Dans le premier cas, ligne 1 de la figure 6.11 : si M est supérieur à 0, c'est-à-dire si au moins une des *features* n'est pas sélection-

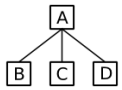
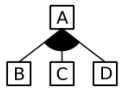
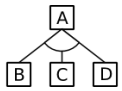
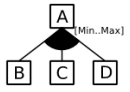
Opérateur	Représentation Graphique	Comportement	Contrainte Utilisateur
And		$A = B = C = D$	and(A,[B,C,D])
Or		**if** $A > 0$ **then** $B + C + D > 0$ **else** $B = C = D = 0$ **end if**	or(A,[B,C,D])
Xor		**if** $A > 0$ **then** $B + C + D = 1$ **else** $B = C = D = 0$ **end if**	xor(A,[B,C,D])
Optionnel		**if** $B = 1$ **then** $A = 1$ **end if**	opt(A,[B])
Card		**if** $A > 0$ **then** $Min \leqslant B+C+D \leqslant Max$ **else** $B = C = D = 0$ **end if**	card(A,[B,C,D],Min,Max)
Mutex		$A + B \leqslant 1$	mutex(A,B)
Require		**if** $A > 0$ **then** $B > 0$ **end if**	require(A,B)

FIGURE 6.9 – Tableau de correspondance des opérateurs, des éléments graphiques, comportements et contraintes utilisateurs

```
and(Env,[OS]),
opt(Env,[Browser]),
xor(OS,[32Bits,64Bits]),
xor(32Bits,[Win2000,WinXP,WinVista]),
xor(64Bits,[WinXP64,WinVista64]),
or(Browser,[FF15,FF,IE5.5]),
mutex(64Bit,FF15)
```

FIGURE 6.10 – Modèle à contraintes du modèle de *features* présenté figure 6.2

née, alors on impose que toutes les autres *features* soient égales à 0 grâce à l'appel du prédicat $ex\_eq([P|LF], 0, Ps)$. Pour imposer que toutes les autres *features* (le père et les autres *features* filles) soient égales à 0, une liste d'action est stockée dans la variable Ps du prédicat ex_eq. Dans ce cas, les actions contenues dans la liste stipulent que toutes les variables doivent être égales à zéro.

L'ensemble des actions à réaliser à la fin de l'exécution de l'algorithme de filtrage sont stockées dans une liste nommée Actions grâce à cette instruction : $Actions = [exit|Ps]$. La liste Action est composée de la liste d'action Ps à laquelle on a ajouté l'action $exit$. L'action $exit$ permet de signifier au solveur de contraintes que la contrainte est satisfaite et qu'elle n'a plus besoin d'exister.

Dans un deuxième cas : ligne 4 de la figure 6.11, si le P est égal à 0, c'est-à-dire si le père n'est pas sélectionné, on impose que toutes les *features* filles soient elles aussi non sélectionnées.

Dans le troisième cas : ligne 5 de la figure 6.11, si le père est sélectionné, on impose que toutes les *features* filles soient égales à 1. Dans le quatrième cas : ligne 5 de la figure 6.11, si une des *features* filles est sélectionnée, on impose que le reste des *features* le soient aussi. Finalement si aucun de ces cas présentés n'est rencontré, rien ne se passe : $Actions = []$, ligne 7 de la figure 6.11.

Le système à contraintes permet aussi de gérer des contraintes sous la forme normale conjonctive (CNF). Il est possible d'exprimer des contraintes complexes en utilisant la logique propositionnelle : $\neg A_1 \vee \neg A_2 \vee \ldots \vee B_1 \vee B_2 \vee \ldots$.
La gestion de ce type de contraintes est nécessaire pour que le modèle à contraintes puisse prendre en entrée des modèles de variabilité intégrant ce type de contraintes, comme ceux proposés dans SPLOT [MBC09].

6.4.2 Sélection de configurations Pairwise

L'étape de sélection de configurations *Pairwise* du processus contient tous les mécanismes qui permettent d'extraire d'un modèle de *features* un ensemble de configurations

```
1 and_solver(P,LF,Actions) :-
2 filter(LF,M,N),
3 ( M > 0  -> Actions = [exit|Ps], ex_eq([P|LF],0,Ps)
4 ; P == 0 -> Actions = [exit|Ps], ex_eq(LF,0,Ps)
5 ; P == 1 -> Actions = [exit|Ps], ex_eq(LF,1,Ps)
6 ; N > 0  -> Actions = [exit|Ps], ex_eq([P|LF],1,Ps)
7 ; Actions = []
8 ).
```

FIGURE 6.11 – Code source de l'algorithme de filtrage de la contrainte utilisateur and

à tester qui respectent le critère pairwise. Une fois cette étape réalisée, une Matrice Solution qui contient un ensemble de configuration qui respectent *pairwise* est retournée.

6.5 Des contraintes pour résoudre le problème pairwise

Dans cette section nous décrivons le modèle à contraintes créé et le processus de résolution utilisé afin de remplir la Matrice Solution.

6.5.1 Création de la Matrice Solution

L'étape de création de la Matrice Solution permet de définir la structure de données qui sera utilisée lors de la résolution. Cette étape consiste en la création d'un tableau avec un nombre de colonnes égal au nombre de *features* et un nombre de lignes égal à un paramètre fourni par l'utilisateur. Cette matrice contient un ensemble de variables. À la création de la matrice, toutes les variables contenues dans la matrice ne peuvent prendre que deux valeurs, soit 0 soit 1. Avant le début de la résolution, cette Matrice Solution contient un ensemble de variables non instanciées : toutes les variables contenues dans cette matrice peuvent prendre la valeur 0 ou la valeur 1. Cette Matrice Solution est présentée figure 6.12. Pour chaque feature, une colonne de variables est créée. Dans la première colonne de la matrice figure 6.12, chaque A_i représente l'état de la *feature* ENV dans la i-ème configuration.

C'est dans cette matrice que sont réalisés les assemblages de paires. Pour garantir que chaque arrangement de paires respecte bien les opérateurs du modèle de *features*, cette matrice est contrainte. Sur chacune de ses lignes sont appliquées les contraintes du modèle de *features*. Pour ce faire, les contraintes du modèle de *features* sont dupliquées et les variables du modèle original sont remplacées par les variables de ligne correspondante. Les contraintes présentées figure 6.10 sont reproduites 11 fois. Les contraintes associées aux deux premières lignes de la Matrice Solution présentée figure 6.12 sont présentées figure 6.13.

Env	OS	32 Bits	64 Bits	Win. 2000	Win. XP	Win. Vista	Win. Xp 64	Win. Vista 64	Browser	FF 1.5	FF 2	IE 5.5
A_1	B_1	C_1	D_1	E_1	F_1	G_1	H_1	I_1	J_1	K_1	L_1	M_1
A_2	B_2	C_2	D_2	E_2	F_2	G_2	H_2	I_2	J_2	K_2	L_2	M_2
A_3	B_3	C_3	D_3	E_3	F_3	G_3	H_3	I_3	J_3	K_3	L_3	M_3
A_4	B_4	C_4	D_4	E_4	F_4	G_4	H_4	I_4	J_4	K_4	L_4	M_4
A_5	B_5	C_5	D_5	E_5	F_5	G_5	H_5	I_5	J_5	K_5	L_5	M_5
A_6	B_6	C_6	D_6	E_6	F_6	G_6	H_6	I_6	J_6	K_6	L_6	M_6
A_7	B_7	C_7	D_7	E_7	F_7	G_7	H_7	I_7	J_7	K_7	L_7	M_7
A_8	B_8	C_8	D_8	E_8	F_8	G_8	H_8	I_8	J_8	K_8	L_8	M_8
A_9	B_9	C_9	D_9	E_9	F_9	G_9	H_9	I_9	J_9	K_9	L_9	M_9
A_{10}	B_{10}	C_{10}	D_{10}	E_{10}	F_{10}	G_{10}	H_{10}	I_{10}	J_{10}	K_{10}	L_{10}	M_{10}
A_{11}	B_{11}	C_{11}	D_{11}	E_{11}	F_{11}	G_{11}	H_{11}	I_{11}	J_{11}	K_{11}	L_{11}	M_{11}

Figure 6.12 – Matrice solution vide pouvant accueillir 11 configurations. Matrice Solution créée à partir du modèle de *features* BIEW présenté figure 6.2 et un paramètre de taille de matrice de taille 11

$and(A_1, [B_1]), opt(A_1, [J_1]), xor(B_1, [C_1, D_1]), xor(C_1, [E_1, F_1, G_1]),$
$xor(D_1, [H_1, I_1]), or(J_1, [K_1, L_1, M_1]), mutex(D_1, K_1)$
$and(A_2, [B_2]), opt(A_2, [J_2]), xor(B_2, [C_2, D_2]), xor(C_2, [E_2, F_2, G_2]),$
$xor(D_2, [H_2, I_2]), or(J_2, [K_2, L_2, M_2]), mutex(D_2, K_2)...$

Figure 6.13 – Extrait des contraintes créées qui permettent de vérifier que les configurations de la Matrice Solution respectent bien le modèle de feature

6.5.2 Génération des contraintes *pairwise*

L'étape de génération des contraintes *pairwise* permet, à partir d'un modèle de *features*, d'extraire un ensemble de contraintes *pairwise* qui portent sur la Matrice Solution et qui imposent que toutes les paires valides soient présentes.

Lors de cette étape, nous identifions les paires valides et nous les transformons en contraintes *pairwise*.

Dans cette section, nous présentons d'abord le fonctionnement des contraintes *pairwise* puis nous décrivons le mécanisme d'identification des paires valides et de création de ces contraintes.

6.5.2.1 La contrainte *pairwise*

Pour respecter le critère *pairwise* sur un modèle de *features*, il faut que pour toutes les paires de features, toutes les combinaisons de valeurs des paires autorisées soient présentes dans l'ensemble de configurations sélectionnées. C'est l'implémentation de la contrainte utilisateur *pairwise* qui permet d'atteindre cet objectif.

Cette contrainte met en relation une variable R représentant une ligne (c'est-à-dire une configuration) de la Matrice Solution, deux vecteurs associés aux colonnes de la matrice et un couple de valeurs. Chaque vecteur représente les valeurs possibles d'une *feature* dans un ensemble de configurations. La première valeur du vecteur contient l'état de la *feature* dans la première configuration, la deuxième valeur, l'état de la *feature* dans la deuxième configuration et ainsi de suite...

Pour illustrer le comportement de la contrainte pairwise nous utilisons un exemple simple où l'on souhaite couvrir la paire $(0, 0)$ pour le couple de *features* (X, Y). La contrainte suivante est créée :

$\mathsf{pairwise}(R, ([X_1, X_2, X_3], [Y_1, Y_2, Y_3]), (0, 0))$.

Elle impose qu'à la ligne R de la matrice se trouve la paire $(0, 0)$. Les vecteurs $[X_1, X_2, X_3]$ et $[Y_1, Y_2, Y_3]$ représentent les deux colonnes de la Matrice Solution associées aux variables X et Y. Ici R peut avoir trois valeurs, car les vecteurs ($[X_1, X_2, X_3]$ et $[Y_1, Y_2, Y_3]$) sont de taille 3. À l'état initial, le domaine des variables est le suivant :

$Requete$:
$\mathsf{pairwise}(R, ([X_1, X_2, X_3], [Y_1, Y_2, Y_3]), (0, 0))$.
$Resultats$:
$R \in [1, 3]$
$X_1, X_2, X_3, Y_1, Y_2, Y_3 \in [0, 1]$

R peut prendre une valeur entre 1 et 3, et tous les X_i, Y_j peuvent tous être égaux à 0 ou à 1. Une fois cette contrainte chargée en mémoire, celle-ci va avoir différents comportements en fonction des choix réalisés sur les variables qui composent cette contrainte.

Comportement 1 : Une valeur est imposée à R ($R = 2$)

En d'autres termes, on impose qu'à la ligne 2 la paire $(0, 0)$ soit présente. Les variables (X_2, Y_2) prendront pour valeurs $(0, 0)$.

> *Requete* :
> $R = 2$
> $\mathsf{pairwise}(R, ([X_1, X_2, X_3], [Y_1, Y_2, Y_3]), (0, 0))$.
> *Resultats* :
> $X_1, X_3, Y_1, Y_3 \in [0, 1]$
> $X_2 = 0$
> $Y_2 = 0$

Comportement 2 : Une valeur est donnée à une des variables des vecteurs colonnes

Une des variables X_i ou Y_j est instanciée. Par exemple, la variable X_3 est instanciée à la valeur 1. La valeur 3 sera supprimée du domaine des valeurs possibles de R, car la valeur $X_3 = 1$ n'est pas compatible avec la paire $(0, 0)$. À la place la contrainte sera mise en attente, jusqu'à l'obtention de nouvelles informations.

> *Requete* :
> $X_3 = 1$
> $\mathsf{pairwise}(R, ([X_1, X_2, X_3], [Y_1, Y_2, Y_3]), (0, 0))$.
> *Resultats* :
> $R \in [1, 2]$
> $X_1, X_2, Y_1, Y_2, Y_3 \in [0, 1]$
> $X_3 = 1$

Ces deux comportements sont réalisés grâce à l'utilisation de l'algorithme de filtrage présenté algorithme 1. Il est déclenché dès qu'une valeur du domaine des variables de la contrainte est modifiée. L'algorithme explore les valeurs possibles de R et détermine celles qui sont consistantes par rapport aux informations disponibles sur les autres variables. La complexité de l'algorithme est linéaire.

Ce sont ces contraintes *pairwise* qui sont créées pour chaque paire valide. Si toutes les contraintes *pairwise* sont satisfaites, cela signifie que toutes les paires valides sont présentes dans la Matrice Solution.

6.5.2.2 Identification des paires valides

Comme montré section 6.3.1, en présence d'un modèle de *features* toutes les paires de valeur des *features* ne sont pas valides. Une procédure permettant de distinguer les paires valides des paires invalides doit être mise en place.

L'identification et l'extraction de l'ensemble des paires valides d'un modèle de *features* reposent sur l'utilisation du mécanisme de propagation des contraintes. Ce mécanisme permet de répercuter l'effet d'un choix fait sur une variable sur les autres variables du problème. Pour les modèles de *features*, il est possible d'examiner l'effet qu'a la sélection d'une *feature* (c'est-à-dire lui donner la valeur 1) ou la non-sélection

Input: R une variable à domaine fini, L_1, L_2, deux listes de variables à domaine fini de même taille (v_1, v_2) un couple d'entier
Output: $Fail$ une valeur pour (R, L_1, L_2)

function pairwise$(R, (L_1, L_2), (v_1, v_2))$
$R' \leftarrow I,\ T_1 \leftarrow \emptyset,\ T_2 \leftarrow \emptyset;$
foreach $r \in R$ **do**
  if $(v_1 \notin L_1[r])$ ***or*** $(v_2 \notin L_2[r])$ **then**
    $R' = R' \backslash \{r\}$
  end
  else
    $T_1 \leftarrow T_1 \cup L_1[r],\quad L'_1[r] \leftarrow L_1[r];$
    $T_2 \leftarrow T_2 \cup L_2[r],\quad L'_2[r] \leftarrow L_2[r];$
  end
end
if $R' = \{a\}$ **then**
  $L'_1[a] = v_1;\ L'_2[a] = v_2;$ **return** $(\{a\}, L'_1, L'_2);$
end
else if $(R' = \emptyset$ ***or*** $v_1 \notin T_1$ ***or*** $v_2 \notin T_2)$ **then**
  return *Fail*
end
else
  return (R', L'_1, L'_2)
end

Algorithm 1: L'algorithme de filtrage de la contrainte pairwise

Requete :
$domain([A, B, C], 0, 1), or(A, [B, C]), B = 1$
Resultats :
$A = 1$
$B \in [0, 1]$

FIGURE 6.14 – Mécanisme de propagation des contraintes

Input: N, un ensemble de feature, M, la Matrice Solution, Ctr, les contraintes du modèle de *features* présentées figure 6.16
Output: S un ensemble de contraintes pairwise sur N, permettant de couvrir pairwise dans M

function Valid Pairs Generator(N, M)
$S = \emptyset$;
$Call(Ctr)$;
foreach $n \in N$ **do**
  $N' = N \backslash \{n\}$;
  $S_1 = GeneratePairwiseConstraint(n, 0, N, M)$;
  $S_2 = GeneratePairwiseConstraint(n, 1, N, M)$;
  $S \leftarrow S \cup S_1 \cup S_2$;
  $N = N'$;
end
return S

Algorithm 2: Génération des contraintes *pairwise* et filtrage des paires invalides.

(c'est-à-dire lui donner la valeur 0) d'une *feature* sur les autres features. Le mécanisme de propagation des contraintes est illustré figure 6.14. Dans cette figure, le modèle utilisé est composé de trois features, A, B et C, qui sont reliées par l'opérateur *or*. Le prédicat $domain([A, B, C], 0, 1)$ stipule que les *features* ont pour domaine de définition deux valeurs : soit 0 soit 1. Dans cet exemple, la valeur 1 est choisie pour la variable B : la *feature* B est sélectionnée dans le modèle de features. Au moment d'appliquer la valeur 1 à B, l'algorithme de filtrage de la contrainte *or* est déclenché et permet de déduire des informations sur le domaine des autres variables. Le résultat est visible à la ligne résultat. Comme la *feature* fille B est sélectionnée, il est nécessaire que la *feature* père le soit aussi, donc A doit prendre la valeur 1. La *feature* C, à cause de la sémantique de l'opérateur *or*, peut prendre les deux valeurs possibles de son domaine.

Pour extraire toutes les paires valides, l'algorithme $Valid\ Pairs\ Generator$ présenté Algorithme 2 est utilisé. L'algorithme prend en entrée la liste des *features* N, la Matrice Solution M et le modèle à contraintes du modèle de *features* Ctr. À la fin est produit un ensemble de contraintes *pairwise*, ce sont les contraintes associées aux paires valides identifiées.

Input: n, la *feature* courante, v la valeur à affecter à la *feature* courante, N l'ensemble des autres features, M la Matrice Solution
Output: S un ensemble de contraintes *pairwise* qui permettent de couvrir des paires dans M

function GeneratePairwiseConstraint(n, v, N, M)
$S = \emptyset$;
$n = v$;
foreach $n' \in N$ **do**
  $D = getDomainList(n')$;
  foreach $d \in D$ **do**
    $S = S \cup pairwise(I, (v, d), (M[n], M[n']))$
  end
end
return S

Algorithm 3: Génération des contraintes *pairwise* pour une valeur d'une *feature* et un ensemble de features.

L'algorithme charge en mémoire l'ensemble des contraintes du modèle de *features* grâce à l'appel de la méthode $Call(Ctr)$. Il est ainsi possible d'utiliser le mécanisme de propagation des contraintes présenté figure 6.14. A chaque affectation d'une valeur à une feature, le mécanisme de propagation des contraintes peut réduire le domaine des autres features, c'est de cette façon que les paires invalides sont supprimées.
L'algorithme prend une *feature* n de l'ensemble de *features*, puis génère l'ensemble des contraintes *pairwise* quand cette *feature* est égale à 0, grâce à l'appel de la méthode $GeneratePairwiseConstraint(n, 0, N, M)$.

Ensuite, l'algorithme génère l'ensemble des paires valides quand cette *feature* est égale à 1 grâce à l'appel $GeneratePairwiseConstraint(n, 1, N, M)$. L'instruction $N' = N \backslash \{n\}$ permet de supprimer la *feature* n de l'ensemble des *features* pour éviter de considérer deux fois la même paire de *feature* ((n, n'), puis (n', n)).

L'algorithme 3 présente l'algorithme de $GeneratePairwiseConstraint$. Cet algorithme prend en entrée quatre paramètres : la *feature* courante n, une valeur v, l'ensemble des autres *features* N et M la Matrice Solution. À son déclenchement, la *feature* n est affectée à la valeur v. En affectant la *feature* n à la valeur v, le mécanisme de propagation des contraintes est activé. Le même phénomène que celui présenté dans la figure 6.14 se produit. C'est de cette façon que les paires invalides sont supprimées. Puis les contraintes *pairwise* sont créées pour toutes les paires qui concernent n et les autres *features* $n' \in N$. L'appel à la méthode $getDomainList$ permet de transformer le domaine de variation d'une *feature* n' en une liste. Ainsi si n' ne peut prendre que deux valeurs : 0 ou 1, D sera une liste de deux éléments : $[0, 1]$, au contraire si n' n'a qu'une seule valeur possible D sera une liste d'un unique élément.

Pour toutes les valeurs de D, une contrainte *pairwise* est construite. Cette contrainte *pairwise* impose la présence de la paire (v,d), où $d \in D$ est issu de la liste des valeurs

possible que peut prendre n', dans les colonnes $M[n]$ et $M[n']$. $M[n]$ et $M[n']$ qui représentent respectivement les colonnes de la Matrice Solution associées aux *features* n et n'. Ainsi si n représente la *feature* 64 Bits de la figure 6.12 et n' la *feature* Win. Xp 64, alors $M[n] = [D_1,D_2,D_3,D_4,....,D_{10},D_{11}]$ et $M[n'] = [H_1,H_2,H_3,H_4,....,H_{10},H_{11}]$.

A la fin de cette procédure, pour toutes les paires valides, un ensemble de contraintes pairwise est créé. La figure 6.15 présente un extrait de ces contraintes *pairwise*, pour les paires valides des *features* (64 Bits, Win. Xp 64).

$\mathsf{pairwise}(R_1, ([D_1,D_2,D_3,D_4,\cdots,D_{11}],[H_1,H_2,H_3,H_4,\cdots,H_{11}]), (1,0))$
$\mathsf{pairwise}(R_2, ([D_1,D_2,D_3,D_4,\cdots,D_{11}],[H_1,H_2,H_3,H_4,\cdots,H_{11}]), (1,1))$
$\mathsf{pairwise}(R_3, ([D_1,D_2,D_3,D_4,\cdots,D_{11}],[H_1,H_2,H_3,H_4,\cdots,H_{11}]), (0,0))$

Figure 6.15 – Contraintes *pairwise* créées pour la paire de *feature* (64 Bits, Win. Xp 64)

6.5.3 Résolution du problème à contraintes

Pour obtenir un ensemble de configurations qui couvrent le critère pairwise, il faut résoudre le problème à contraintes défini précédemment. Le principe est le suivant : pour chaque paire valide, il faut trouver une valeur de rang R telle que cette paire soit dans la Matrice Solution et respecte les contraintes du modèle. Ce processus est déterministe : si les paramètres fournis en entrée ne varient pas, le résultat sera identique pour deux exécutions. Dans cette section nous présentons une partie du processus de résolution dans le cas du modèle de *features* présenté figure 6.2. Le modèle à contraintes associé à ce modèle de *features* est rappelé figure 6.16.

Nous présentons six étapes de la résolution. Pour ce modèle de *features*, 190 contraintes *pairwise* ont été générées.

La Matrice Solution initiale est présentée figure 6.17. Pour des raisons de lisibilité, nous masquons les colonnes et lignes qui n'interviennent pas dans la résolution. Cette matrice, à la différence de la Matrice Solution présentée figure 6.3, est déjà partiellement remplie : les deux premières colonnes sont remplies par la valeur 1. Ce remplissage s'explique par le fait que la *feature* Env est toujours sélectionnée. La configuration valide aucune des *features* sélectionnées (c'est à dire composée seulement de 0) n'est

```
and(Env,[OS]),
opt(Env,[Browser]),
xor(OS,[32Bits,64Bits]),
xor(32Bits,[Win2000,WinXP,WinVista]),
xor(64Bits,[WinXP64,WinVista64]),
or(Browser,[FF15,FF,IE5.5]),
mutex(64Bit,FF15)
```

Figure 6.16 – Modèle à contraintes du modèle de *features* présenté figure 6.2

Env	OS	32 Bits	64 Bits	...	Win. Xp 64	Win. Vista 64	Browser	FF 1.5	FF 2	IE 5.5
1	1	C_1	D_1	$\cdots$	H_1	I_1	J_1	K_1	L_1	M_1
1	1	C_2	D_2	$\cdots$	H_2	I_2	J_2	K_2	L_2	M_2
1	1	C_3	D_3	$\cdots$	H_3	I_3	J_3	K_3	L_3	M_3
1	1	C_4	D_4	$\cdots$	H_4	I_4	J_4	K_4	L_4	M_4
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	1	C_{11}	D_{11}	F_{11}	H_{11}	I_{11}	J_{11}	K_{11}	L_{11}	M_{11}

$\mathsf{pairwise}(R_1, ([L_1, L_2, L_3, L_4, \cdots, L_{11}], [M_1, M_2, M_3, M_4, \cdots, M_{11}]), (1, 1))$
$\mathsf{pairwise}(R_2, ([L_1, L_2, L_3, L_4, \cdots, L_{11}], [M_1, M_2, M_3, M_4, \cdots, M_{11}]), (1, 0))$
$\mathsf{pairwise}(R_3, ([L_1, L_2, L_3, L_4, \cdots, L_{11}], [M_1, M_2, M_3, M_4, \cdots, M_{11}]), 0, 1))$
$\mathsf{pairwise}(R_4, ([L_1, L_2, L_3, L_4, \cdots, L_{11}], [M_1, M_2, M_3, M_4, \cdots, M_{11}]), (0, 0))$
$\mathsf{pairwise}(R_5, ([J_1, J_2, J_3, J_4, \cdots, J_{11}], [M_1, M_2, M_3, M_4, \cdots, M_{11}]), (0, 0))$
$\mathsf{pairwise}(R_6, ([K_1, K_2, K_3, K_4, \cdots, K_{11}], [M_1, M_2, M_3, M_4, \cdots, M_{11}]), (1, 0))$

Figure 6.17 – Matrice Solution et contraintes *pairwise* générées à l'état initial.

pas considérée, car elle correspond à un système vide. Par propagation des contraintes, la *feature* OS est elle aussi sélectionnée car les deux *features* sont reliées par l'opérateur and.

Pour illustrer le processus de résolution nous avons sélectionné 6 contraintes *pairwise* parmi les 190 contraintes générées. Ces contraintes sont présentées en-dessous de la Matrice Solution de la figure 6.17. Elles concernent les paires de *features* (FF 1.5 , FF 2) et (Browser , IE 5.5). Les rangs R_i peuvent prendre une valeur entre 1, la première configuration (c'est à dire première ligne de la matrice) et 11 la dernière configuration (c'est à dire dernière ligne de la matrice). Le processus de résolution va chercher à donner une valeur à tous les rangs R_i. Comme nous souhaitons avoir un nombre de configurations le plus petit possible, l'algorithme va toujours choisir la première valeur du domaine de définition (qui est la plus petite), de cette façon la paire sera arrangée au plus tôt dans la Matrice Solution.

Première itération :

Lors de la première itération, l'algorithme d'énumération va donner une valeur à R_1. Dans le cas présent $R1 \in [1, 11]$. La valeur 1 est donnée à R_1. Ainsi, la paire $(1, 1)$ est

ajoutée à la ligne de la Matrice Solution, dans les colonnes FF 2 et IE 5.5.

Env	OS	32 Bits	64 Bits	...	Win. Xp 64	Win. Vista 64	Browser	FF 1.5	FF 2	IE 5.5
1	1	C_1	D_1	$\cdots$	H_1	I_1	**1**	K_1	**1**	**1**
1	1	C_2	D_2	$\cdots$	H_2	I_2	J_2	K_2	L_2	M_2
1	1	C_3	D_3	$\cdots$	H_3	I_3	J_3	K_3	L_3	M_3
1	1	C_4	D_4	$\cdots$	H_4	I_4	J_4	K_4	L_4	M_4
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	1	C_{11}	D_{11}	F_{11}	H_{11}	I_{11}	J_{11}	K_{11}	L_{11}	M_{11}

$\mathsf{pairwise}(1, ([\mathbf{1}, L_2, L_3, L_4, \cdots, L_{11}], [\mathbf{1}, M_2, M_3, M_4, \cdots, M_{11}]), (1,1))$
$\mathsf{pairwise}(R_2([\mathbf{1}, L_2, L_3, L_4, \cdots, L_{11}], [\mathbf{1}, M_2, M_3, M_4, \cdots, M_{11}]), (1,0))$
$\mathsf{pairwise}(R_3, ([\mathbf{1}, L_2, L_3, L_4, \cdots, L_{11}], [\mathbf{1}, M_2, M_3, M_4, \cdots, M_{11}]), 0, 1))$
$\mathsf{pairwise}(R_4([\mathbf{1}, L_2, L_3, L_4, \cdots, L_{11}], [\mathbf{1}, M_2, M_3, M_4, \cdots, M_{11}]), (0,0))$
$\mathsf{pairwise}(R_5, ([\mathbf{1}, J_2, J_3, J_4, \cdots, J_{11}], [\mathbf{1}, M_2, M_3, M_4, \cdots, M_{11}]), (0,0))$
$\mathsf{pairwise}(R_6, ([K_1, K_2, K_3, K_4, \cdots, K_{11}], [\mathbf{1}, M_2, M_3, M_4, \cdots, M_{11}]), (1,0))$

Figure 6.18 – Matrice Solution et contraintes pairwise générées après la première itération

Le résultat de l'ajout de cette paire est visible sur la matrice présentée figure 6.18. Dans cette Matrice Solution la *feature* Browser a pris la valeur 1. Cet événement est dû aux contraintes du modèle de features. Si une des *features* FF 2 ou IE 5.5 ou FF 1.5 est sélectionnée, alors la *feature* Browser doit elle aussi être sélectionnée (du à la relation du filiation entre les *features* IE 5.5, FF 1.5 et FF 2 et leur père Browser).

Le fait d'ajouter la paire $(1,1)$ dans les colonnes FF 2 et IE 5.5 impacte aussi les autres contraintes *pairwise*. Par exemple la paire associée au rang R_2 : $(1,0)$ qui concerne les *features* FF 2 et IE 5.5 ne peut plus être arrangée à la ligne 1, car dans cette ligne la *feature* IE 5.5 a pris la valeur 1 et la paire nécessite que la *feature* IE 5.5 soit égale à la valeur 0. Par conséquent, après la pose de la paire R_1 le domaine des valeurs possibles de R_2 est réduit et $R_2 \in [2, 11]$. Le même phénomène s'applique aux autres rangs.

Lors de l'ajout de la paire (1,1) à la ligne 1 de la matrice, les variables M_1et L_1 sont toutes les deux affectées à la valeur 1. Cette affectation se répercute sur l'ensemble des contraintes *pairwise* et leurs vecteurs colonnes sont modifiés.

Env	OS	32 Bits	64 Bits	...	Win. Xp 64	Win. Vista 64	Browser	FF 1.5	FF 2	IE 5.5
1	1	C_1	D_1	$\cdots$	H_1	I_1	1	K_1	1	1
1	1	C_2	D_2	$\cdots$	H_2	I_2	**1**	K_2	**1**	**0**
1	1	C_3	D_3	$\cdots$	H_3	I_3	J_3	K_3	L_3	M_3
1	1	C_4	D_4	$\cdots$	H_4	I_4	J_4	K_4	L_4	M_4
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	1	C_{11}	D_{11}	F_{11}	H_{11}	I_{11}	J_{11}	K_{11}	L_{11}	M_{11}

$\mathsf{pairwise}(1, ([1,\mathbf{1},L_3,L_4,\cdots,L_{11}],[1,\mathbf{0},M_3,M_4,\cdots,M_{11}]),(1,1))$
$\mathsf{pairwise}(2, ([1,\mathbf{1},L_3,L_4,\cdots,L_{11}],[1,\mathbf{0},M_3,M_4,\cdots,M_{11}]),(1,0))$
$\mathsf{pairwise}(R_3, ([1,\mathbf{1},L_3,L_4,\cdots,L_{11}],[1,\mathbf{0},M_3,M_4,\cdots,M_{11}]),0,1))$
$\mathsf{pairwise}(R_4([1,\mathbf{1},L_3,L_4,\cdots,L_{11}],[1,\mathbf{0},M_3,M_4,\cdots,M_{11}]),(0,0))$
$\mathsf{pairwise}(R_5, (1,\mathbf{1},J_3,J_4,\cdots,J_{11}],[1,\mathbf{0},M_3,M_4,\cdots,M_{11}]),(0,0))$
$\mathsf{pairwise}(R_6, ([K_1,K_2,K_3,K_4,\cdots,K_{11}],[1,\mathbf{0},M_3,M_4,\cdots,M_{11}]),(1,0))$

Figure 6.19 – Matrice Solution et contraintes *pairwise* générées après la deuxième itération

Deuxième itération :

Lors de cette étape, la paire associée au rang R_2 est arrangée à la ligne deux de la matrice. Les résultats sur la Matrice Solution et les contraintes pairwise sont visibles figure 6.19.

Troisième itération :

Le comportement associé à l'arrangement de la paire du rang R_3 est similaire aux étapes précédentes. Les résultats sur la Matrice Solution et les contraintes *pairwise* sont visibles figure 6.20.

Quatrième itération :

Lors de la quatrième itération, la paire $(0,0)$ associée à la paire de *features* FF 2 et IE 5.5 est ajoutée ligne 4. Le résultat est présenté figure 6.21. À la différence des

Env	OS	32 Bits	64 Bits	...	Win. Xp 64	Win. Vista 64	Browser	FF 1.5	FF 2	IE 5.5
1	1	C_1	D_1	$\cdots$	H_1	I_1	1	K_1	1	1
1	1	C_2	D_2	$\cdots$	H_2	I_2	1	K_2	1	0
1	1	C_3	D_3	$\cdots$	H_3	I_3	**1**	K_3	**0**	**1**
1	1	C_4	D_4	$\cdots$	H_4	I_4	J_4	K_4	L_4	M_4
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	1	C_{11}	D_{11}	F_{11}	H_{11}	I_{11}	J_{11}	K_{11}	L_{11}	M_{11}

pairwise$(1, ([1, 1,$**0** $, L_4, \cdots, L_{11}], [1, 0,$**1**$, M_4, \cdots, M_{11}]), (1, 1))$
pairwise$(2, ([1, 1,$**0** $, L_4, \cdots, L_{11}], [1, 0,$**1**$, M_4, \cdots, M_{11}]), (1, 0))$
pairwise$(3, ([1, 1,$**0** $, L_4, \cdots, L_{11}], [1, 0,$**1**$, M_4, \cdots, M_{11}]), 0, 1))$
pairwise$(R_4([1, 1,$**0** $, L_4, \cdots, L_{11}], [1, 0,$**1**$, M_4, \cdots, M_{11}]), (0, 0))$
pairwise$(R_5, ([1, 1,$**1**$, J_4, \cdots, J_{11}], [1, 0,$**1**$, M_4, \cdots, M_{11}]), (0, 0))$
pairwise$(R_6, ([K_1, K_2, K_3, K_4, \cdots, K_{11}], [1, 0,$**1**$, M_4, \cdots, M_{11}]), (1, 0))$

Figure 6.20 – Matrice Solution et contraintes *pairwise* générées après la troisième itération

itérations précédentes la *feature* Browser dans cette configuration n'est pas mise à la valeur 1 car il est possible que le navigateur ne soit pas présent dans une configuration. C'est le résultat de l'opérateur optionnel du modèle à contraintes présenté 6.16.

Cinquième itération :

Au cours de cette cinquième itération, la paire $(0, 0)$ qui concerne le couple de *features* Browser et IE 5.5 doit être arrangée dans la matrice. Le rang R_5 a pour domaine de définition : $R_5 \in [4, 11]$. L'algorithme de résolution impose que cette paire soit à la ligne 4. La *feature* Browser étant non sélectionnée, les contraintes du modèle imposent que la *feature* IE 5.5 ne soit pas sélectionnée. La matrice résultante ainsi que l'impact sur les contraintes est présenté figure 6.22.

Sixième itération :

La sixième paire à ajouter concerne la paire de *features* (FF 1.5, IE 5.5), et a pour valeur $(0, 1)$. Le domaine de R_6 est le suivant $R_6 \in 2 \cup [5, 11]$. L'algorithme d'énu-

Env	OS	32 Bits	64 Bits	...	Win. Xp 64	Win. Vista 64	Browser	FF 1.5	FF 2	IE 5.5
1	1	C_1	D_1	$\cdots$	H_1	I_1	1	K_1	1	1
1	1	C_2	D_2	$\cdots$	H_2	I_2	1	K_2	1	0
1	1	C_3	D_3	$\cdots$	H_3	I_3	1	K_3	0	1
1	1	C_4	D_4	$\cdots$	H_4	I_4	J_4	K_4	**0**	**0**
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	1	C_{11}	D_{11}	F_{11}	H_{11}	I_{11}	J_{11}	K_{11}	L_{11}	M_{11}

pairwise$(1, ([1,1,0,\mathbf{0},\cdots,L_{11}], [1,0,1,\mathbf{0},\cdots,M_{11}]), (1,1))$
pairwise$(2, ([1,1,0,\mathbf{0},\cdots,L_{11}], [1,0,1,\mathbf{0},\cdots,M_{11}]), (1,0))$
pairwise$(3, ([1,1,0,\mathbf{0},\cdots,L_{11}], [1,0,1,\mathbf{0},\cdots,M_{11}]), 0,1))$
pairwise$(4, ([1,1,0,\mathbf{0},\cdots,L_{11}], [1,0,1,\mathbf{0},\cdots,M_{11}]), (0,0))$
pairwise$(R_5, ([1,1,1,J_3,J_4,\cdots,J_{11}], [1,0,1,\mathbf{0},\cdots,M_{11}]), (0,0))$
pairwise$(R_6, ([K_1,K_2,K_3,K_4,\cdots,K_{11}], [1,0,1,\mathbf{0},\cdots,M_{11}]), (1,0))$

Figure 6.21 – Matrice Solution et contraintes *pairwise* générées après la quatrième itération

mération donne à R_6 la valeur 2. Le résultat est visible sur la figure 6.23. Le fait de sélectionner la *feature* FF 1.5 impose que la *feature* 64 bit soit mise à la valeur 0 : c'est la conséquence de la contrainte Mutex. La *feature* 64 Bits dans la ligne deux de la Matrice Solution est donc mise à 0. La contrainte xor qui lie la *feature* 64 Bits aux *features* Win XP. 64 et Win. Vista 64 se réveille et impose aux deux systèmes d'exploitation de ne pas être présents dans la configuration : les *features* Win. Vista 64 et Win XP. 64 sont donc mises à la valeur 0 à la ligne deux de la Matrice Solution. De même, comme les *features* 32 Bits et 64 Bits sont liées par un xor, la *feature* 32 Bits est automatiquement sélectionnée, car la *feature* 64 Bits ne l'est pas.

L'utilisation des contraintes du modèle de *features* permet de réduire les combinaisons possibles de paires. De plus, dès que ces contraintes sont activées, de nouvelles paires sont automatiquement placées dans la matrice. Ainsi dans la matrice présentée figure 6.23, le fait de passer les *features* 64 Bits, Win. Xp 64 et Win. Vista 64 à 0 et de sélectionner la *feature* 32 Bits permet de couvrir plusieurs paires comme la paire $(1,0)$ de la paire de *feature* (32 Bits ,64 Bits).

Env	OS	32 Bits	64 Bits	...	Win. Xp 64	Win. Vista 64	Browser	FF 1.5	FF 2	IE 5.5
1	1	C_1	D_1	$\cdots$	H_1	I_1	1	K_1	1	1
1	1	C_2	D_2	$\cdots$	H_2	I_2	1	K_2	1	0
1	1	C_3	D_3	$\cdots$	H_3	I_3	1	K_3	0	1
1	1	C_4	D_4	$\cdots$	H_4	I_4	**0**	**0**	0	**0**
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	1	C_{11}	D_{11}	F_{11}	H_{11}	I_{11}	J_{11}	K_{11}	L_{11}	M_{11}

$\mathsf{pairwise}(1, ([1,1,0,0\cdots,L_{11}], [1,0,1,0,\cdots,M_{11}]), (1,1))$
$\mathsf{pairwise}(2, ([1,1,0,0\cdots,L_{11}], [1,0,1,0,\cdots,M_{11}]), (1,0))$
$\mathsf{pairwise}(3, ([1,1,0,0\cdots,L_{11}], [1,0,1,0,\cdots,M_{11}]), 0,1))$
$\mathsf{pairwise}(4, ([1,1,0,0\cdots,L_{11}], [1,0,1,0,\cdots,M_{11}]), (0,0))$
$\mathsf{pairwise}(4, ([1,1,1,\mathbf{0},\cdots,J_{11}], [1,0,1,0,\cdots,M_{11}]), (0,0))$
$\mathsf{pairwise}(R_6, ([K_1,K_2,K_3,\mathbf{0},\cdots,K_{11}], [1,0,1,0,\cdots,M_{11}]), (1,0))$

Figure 6.22 – Matrice Solution et contraintes pairwise générées après la cinquième itération

Dans cet exemple, nous avons considéré un sous-ensemble de contraintes pairwise. Lors de la résolution sur des grands modèles de *features* ce processus est reproduit avec plusieurs dizaines de milliers de paires.

La façon dont sont énumérées les paires a un impact sur le nombre de configurations de la Matrice Solution. Dans la prochaine section, nous montrons comment il est possible d'optimiser la taille de la solution obtenue grâce aux techniques d'énumérations existantes. Dans le chapitre 7 nous évaluons différentes techniques d'énumération et nous montrons que le choix de la technique influence le nombre de configurations contenu dans la Matrice Solution.

6.6 Optimisation

Dans cette section nous montrons comment utiliser les informations structurelles du modèle de *features*, les différentes techniques d'énumération et les capacités de minimisation fournies par les techniques de programmation par contraintes pour réduire le nombre de configurations sélectionnées. L'évaluation de ces heuristiques se fera dans le

Env	OS	32 Bits	64 Bits	...	Win. Xp 64	Win. Vista 64	Browser	FF 1.5	FF 2	IE 5.5
1	1	C_1	D_1	$\cdots$	H_1	I_1	1	K_1	1	1
1	1	C_2	**0**	$\cdots$	**0**	**0**	1	**1**	1	**0**
1	1	C_3	D_3	$\cdots$	H_3	I_3	1	K_3	0	1
1	1	C_4	D_4	$\cdots$	H_4	I_4	0	0	0	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	1	C_{11}	D_{11}	F_{11}	H_{11}	I_{11}	J_{11}	K_{11}	L_{11}	M_{11}

$\mathsf{pairwise}(1, ([1,1,0,0,\cdots,L_{11}],[1,0,1,0,\cdots,M_{11}]),(1,1))$
$\mathsf{pairwise}(2, ([1,1,0,0,\cdots,L_{11}],[1,0,1,0,\cdots,M_{11}]),(1,0))$
$\mathsf{pairwise}(3, ([1,1,0,0,\cdots,L_{11}],[1,0,1,0,\cdots,M_{11}]),0,1))$
$\mathsf{pairwise}(4, ([1,1,0,0,\cdots,L_{11}],[1,0,1,0,\cdots,M_{11}]),(0,0))$
$\mathsf{pairwise}(4, ([1,1,1,0,\cdots,J_{11}],[1,0,1,0,\cdots,M_{11}]),(0,0))$
$\mathsf{pairwise}(2, ([K_1,\mathbf{1},K_3,K_4,\cdots,K_{11}],[1,0,1,0,\cdots,M_{11}]),(1,0))$

Figure 6.23 – Matrice Solution et contraintes pairwise générées après la sixième itération

chapitre 8.

6.6.1 Les heuristiques de recherche

Nous avons identifié plusieurs heuristiques de recherche. Dans cette section nous présentons ces différentes heuristiques.

6.6.1.1 L'ordre de sélection des variables : un facteur important de la résolution

L'ordre de sélection des variables a une influence sur la taille de la Matrice Solution. Dans l'exemple présenté section 6.5.3, l'algorithme d'énumération prend en entrée une liste de rang : $[R_1, R_2, R_3, R_4, R_5, R_6]$ et affecte à chaque rang une valeur qui correspond à une ligne de la Matrice Solution. Dans cet exemple, les rangs sont pris dans l'ordre de la liste. En modifiant cet ordre, il est possible de trouver des solutions de tailles différentes. La figure 6.1 présente deux Matrices Solutions, qui contiennent toutes les deux un ensemble de configurations qui couvrent *pairwise*. Ces deux matrices contiennent des configurations qui couvrent *pairwise* pour le même modèle de *features* : un opérateur

or composé de 4 fils : B,C,D,E, et un père A. Pour ces deux matrices, nous avons fait varier l'algorithme de sélection des variables. La première matrice à gauche a été obtenue en utilisant l'algorithme *left-most* (même algorithme que celui utilisé pour le cas d'étude précédent). La seconde matrice à droite a été obtenue en utilisant l'algorithme de sélection des variables *first-fail*. Ces deux matrices ne présentent pas le même nombre de configurations. La matrice à gauche présente 6 configurations tandis que celle de droite 5.

A	B	C	D	E
1	1	1	1	1
1	1	1	1	0
1	0	0	0	1
1	1	0	0	0
1	0	1	0	0
1	0	0	1	0

A	B	C	D	E
1	1	1	1	1
1	1	0	1	0
1	0	0	0	1
1	1	1	0	0
1	0	1	1	0

TABLE 6.1 – Deux Matrices Solutions pour un même modèle de *features* : or(A,[B,C,D,E]), mais avec deux heuristiques de recherche différentes.

Cet exemple montre que pour un même modèle de features, deux ordres de sélection des variables différents vont mener à deux Matrices Solutions de tailles différentes.

6.6.1.2 Sélection des variables

En utilisant l'algorithme de sélection des variables *left-most* décrit dans la section 5.3.2.1., le solveur à contraintes va énumérer les variables dans l'ordre d'apparition dans la liste. C'est cet algorithme qui a été utilisé dans l'exemple présenté dans la section 6.5.3. Une liste de rangs $[R_1, R_2, R_3, R_4, R_5, R_6...]$ est fournie au solveur, et le solveur commence par R_1, qui est le premier rang, puis le deuxième et ainsi de suite.

L'algorithme de sélection *first-fail* est plus complexe. À chaque fois qu'une valeur est attribuée à un rang, l'algorithme parcourt la liste des rangs restant et étudie leurs domaines. Il regarde pour chaque rang quelle valeur le rang peut prendre, en d'autre terme, dans combien de configurations la paire peut être placée. Au fur et à mesure de la progression de la résolution, certains rangs ne pourront plus être arrangés dans certaines configurations, car les paires à arranger sont incompatibles. Par exemple si l'on prend ces deux contraintes :

$$\mathsf{pairwise}(R_1, ([A_1, A_2, A_3], [B_1, B_2, B_3]), (0, 0))$$
$$\mathsf{pairwise}(R_2, ([A_1, A_2, A_3], [B_1, B_2, B_3]), (1, 1))$$

Si l'on place la paire R_1 à la ligne 1, alors la valeur 1 sera exclue du domaine de R_2. L'algorithme va sélectionner les rangs qui ont le petit domaine et va les placer en priorité. S'il y a plusieurs candidats, c'est le premier qui est placé.

Dans la section 7.1.4, nous évaluons ces différentes heuristiques pour déterminer laquelle est la plus efficace.

6.6.1.3 Le Tri des paires

Il est possible d'utiliser la structure du modèle de *features* pour améliorer la résolution. Pour utiliser ces informations structurelles, nous calculons différentes métriques, pour toutes les paires valides du modèle. Nous définissons trois métriques, qui peuvent être utilisées pour sélectionner les paires à placer en priorité lors de la résolution. Ce sont ces métriques qui sont utilisées pour trier la liste des rangs $[R_1, R_2, R_3, R_4, R_5, R_6...]$ juste avant l'appel de l'algorithme d'énumération. Ces trois métriques sont les suivantes et sont illustrées dans la table 6.25.

1. *Profondeur_features* : qui est la somme des profondeurs des *features* de la paire. Cette métrique permet de capturer le phénomène de filiation dans un modèle de features. Si une *feature* fille est sélectionnée, son père l'est automatiquement. Le fait de placer en priorité les *features* filles va accélérer la propagation des contraintes. Plus une *feature* est profonde, plus elle est contrainte. Le couple (Win. 2000, Win. Xp 32) est composé de *features* qui ont toutes les deux une profondeur de trois : trois arêtes relient les *features* à la *feature* racine Env. La valeur de la métrique pour le couple est de six. Pour le couple (Win. Vista 64, IE 5.5), cette métrique est de cinq, car la *feature* Win. Vista 64 a une profondeur de trois et IE 5.5 une profondeur de deux. Avec ce tri, les premières paires énumérées sont celles qui ont la valeur calculée la plus importante et ainsi de suite.
2. *Profondeur_père* : la profondeur du plus proche ancêtre commun d'une paire de features. Cette métrique représente la distance entre deux features. Plus la valeur de cette métrique est importante, plus les *features* de la paire sont profondes dans l'arbre. Le père du couple de *features* : (Win. 2000, Win. Xp 32) est la *feature* 32 Bits. Celle-ci a une profondeur de deux car deux arêtes la séparent de la *feature* racine Env. Le couple de *features* (Win. Vista 64, IE 5.5) a pour père la *feature* Env. La profondeur du père est donc de 0. Avec ce tri, les premières paires énumérées sont celles qui ont la valeur calculée la plus importante et ainsi de suite.
3. *Opérateur_commun* : cette métrique se base sur le type du plus proche ancêtre commun d'une paire de feature. Comme tous les opérateurs d'un modèle de *features* n'ont pas la même capacité à contraindre, nous ordonnons ces opérateurs du plus contraignant au moins contraignant.

   $$and > xor > card > or > opt > none$$

   Lors de l'étape d'énumération, les premières paires énumérées seront celles reliées par l'opérateur *and* tandis que celles reliées avec l'opérateur *opt* seront énumérées plus tard. L'opérateur *and* a un pouvoir déductif plus important. Si le père est sélectionné, alors ses fils le sont aussi, tandis que pour l'opérateur *opt* si le père est sélectionné on ne peut rien déduire de l'état de ses fils. L'opérateur *none* permet de considérer les *features* qui n'ont pas d'opérateur commun par exemple les *features* OS et Browser : la *feature* OS est liée grâce à l'opérateur *and* tandis que Browser est lié à la *feature* Env grâce à l'opérateur Optionnel. Le couple de

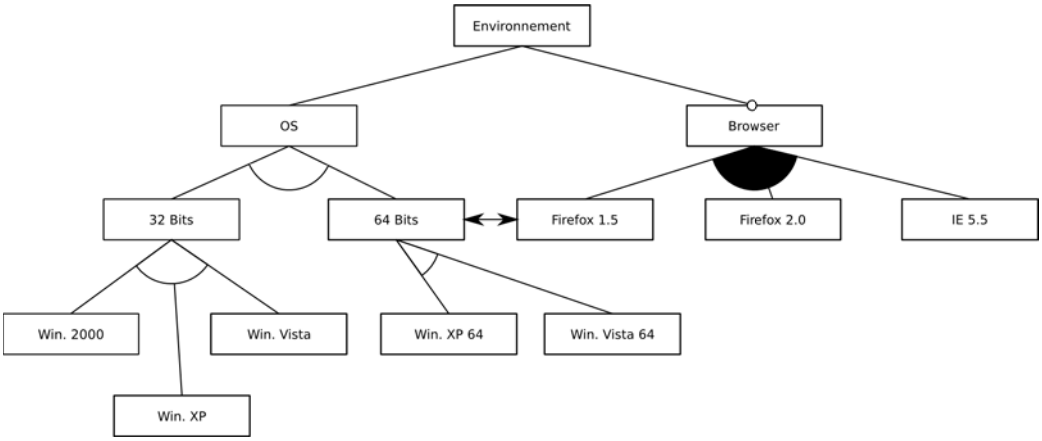


FIGURE 6.24 – Extrait du modèle de features du projet BIEW

Paires	♯1 : WIN. 2000 ; WIN. XP	♯2 : WIN VISTA 64 ; IE 5.5
profondeur_père	2	0
profondeur_features	6	5
opérateur_commun	*xor*	*none*

FIGURE 6.25 – Métriques utilisées pour le tri des paires

(WIN. 2000,WIN XP. 32) lui à pour opérateur commun *xor* car ces deux *features* sont liées.

La table 6.25 présente les métriques calculées pour le modèle du projet de test BIEW. Ce modèle est rappelé figure 6.2.

Ces différentes heuristiques de recherches sont évaluées dans le chapitre 7 de ce document.

6.6.2 Processus de minimisation

Les techniques de programmation par contraintes permettent de minimiser une fonction de coût. Dans cette section, nous montrons comment ces techniques de minimisation peuvent être appliquées au problème de la sélection de configurations de test *pairwise* sur des modèles de features. Puis nous montrons comment nous pouvons mettre un contrat de temps pour résoudre le problème en un temps précis.

Trouver un nombre de configurations de taille minimale permettant de couvrir le critère pairwise revient à résoudre ce problème :

$$\textbf{Trouver un } R_1, ..., R_{4n^2} \textbf{ tel que } Min(f) \textbf{ et} \forall i, j \textbf{ dans} 1..n,$$
$$\mathsf{pairwise}(R_k, C_i, C_j, (0,0)), \mathsf{pairwise}(R_{k+1}, C_i, C_j, (0,1)),$$
$$\mathsf{pairwise}(R_{k+2}, C_i, C_j, (1,0)), \mathsf{pairwise}(R_{k+3}, C_i, C_j, (1,1))$$

où n est le nombre de features, C_i, C_j les colonnes de la matrice. Le problème présenté ci-dessus considère que toutes les paires sont valides. C'est une version générique du problème de minimisation où tous les couples possibles sont présents. Dans le cas d'un modèle de *features* la valeur maximale du rang R serait $R'_n < R_{4n^2}$ où n' serait égal au nombre de paires valides, et tout les paires possibles ne seraient pas présentes.

Nous étudions deux fonctions de coûts :

$$f_1 = \sum_{k \in 1..4n^2} R_k$$
$$f_2 = \mathsf{Max}_{k \in 1..4n^2} R_k$$

Ces deux fonctions peuvent être utilisées pour trouver la valeur minimum des R_k, tel que le critère *pairwise* soit satisfait dans la matrice. Le mécanisme de minimisation utilisé repose sur l'utilisation du *branch & bound*. Cette méthode consiste à explorer toutes les solutions possibles tout en maintenant la fonction de coût le plus bas possible. A chaque nœud de l'arbre de recherche, le *branch & bound* évalue la fonction de coût et supprime les branches pour lesquelles le coût sera supérieur au coût minimum courant.

La taille de l'arbre de recherche est potentiellement très grande : le nombre de façons d'arranger les paires est très important. Pour s'assurer de trouver la valeur minimale, il est nécessaire de parcourir entièrement cet arbre de recherche. Ce parcours peut prendre beaucoup de temps. Une solution possible à ce problème est d'utiliser la minimisation en temps contraint. Cette minimisation repose sur l'allocation d'un temps pour trouver une solution [1]. C'est cette technique qui sera utilisée dans les expérimentations présentées dans le chapitre 7.

Grâce à cette minimisation en temps contraint, l'algorithme est capable de rendre, lorsque le temps alloué est écoulé, la meilleure solution trouvée. Il est cependant possible qu'en allouant un temps de minimisation plus long l'algorithme puisse trouver une meilleure solution, mais cette approche reste un bon compromis dans le cas où il est important de contrôler le temps dédié à la création de configurations de test et à l'activité de test elle-même.

1. Lorsque la minimisation est réalisée en temps contraint, il n'est pas possible de garantir pour deux exécutions avec un même temps de minimisation que les configurations retournées soient identiques. Ce non déterminisme est dû aux conditions d'exécution du programme permettant de couvrir le critère *pairwise* : en fonction de l'occupation de la machine, de la température extérieure et des processus en cours, pour deux exécutions identiques l'algorithme ne pourra pas parcourir exactement la même portion de l'arbre. Le processus de résolution aura la même méthode parcours, cependant il ne s'arrêtera pas au même endroit dans l'arbre de recherche : plus la machine est occupée, plus l'algorithme de parcours s'arrêtera tôt.

Dans le cas d'une minimisation sans contrat de temps, deux exécutions rendront le même résultat, la seule différence résidera dans le temps d'exécution : plus la machine est occupée plus le temps nécessaire pour trouver le minimum sera important.

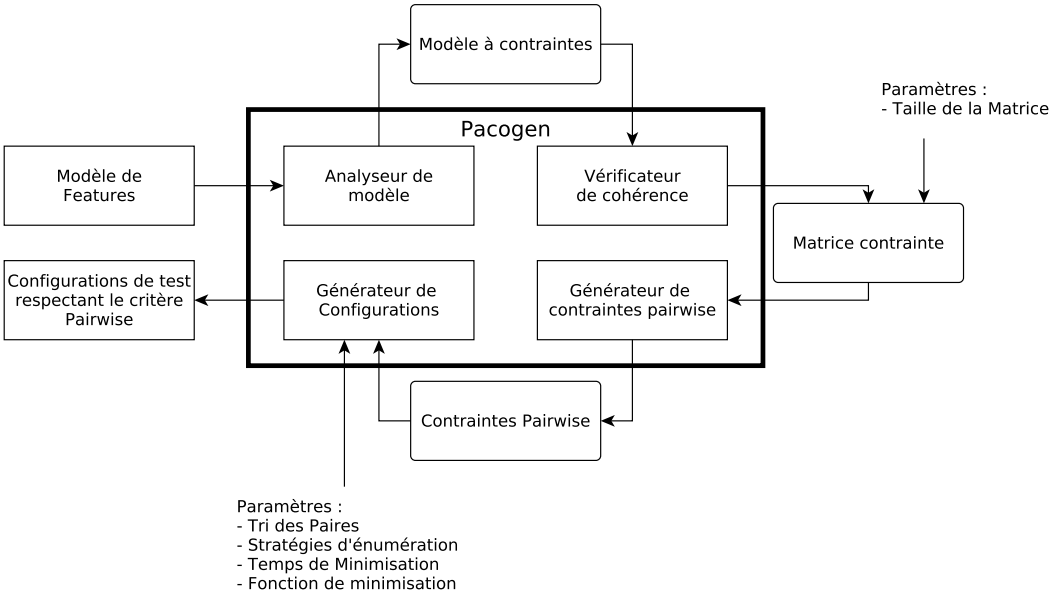


FIGURE 6.26 – Architecture de Pacogen

6.7 Une contribution outillée : Pacogen

Dans cette section, nous présentons le logiciel réalisé. Nous détaillons l'architecture et les différents modules composant l'application. Nous présentons la méthode de validation que nous avons employée et qui nous a permis de valider le modèle à contraintes et les configurations *pairwise* obtenues.

6.7.1 Implémentation

L'implémentation de l'approche à contraintes proposée est faite dans un logiciel appelé Pacogen (PAirwaise COnfiguration GENeration). L'application est développée en Prolog et Java et fait environ 7 000 lignes de code. Ce logiciel utilise la librairie SICStus Prolog `clpfd` qui permet de résoudre des problèmes d'optimisation à l'aide des techniques de programmation par contraintes. Pacogen est disponible en téléchargement libre [2]. Pacogen est fait de quatre composants, présentés figure 6.26. Cette figure présente l'architecture globale de Pacogen, les relations entre les composants et les paramètres utilisés. Chaque composant est détaillé ci-dessous :

1. **Analyseur de modèle** transforme un *modèle de features* en un *modèle à contraintes*. Ce composant transforme toutes les relations du modèle de *features* en un ensemble de contraintes Prolog. Il réalise l'étape Transformation en modèle à contraintes du processus figure 6.1.

2. http ://hervieu.info/pacogen/

Ce module prend en entrée des modèles exprimés dans deux formats :
- le format **.sxfm** : format utilisé par le dépôt de modèle de *features* SPLOT [MBC09]. La lecture du format sxfm permet à Pacogen de lire tous les modèles qui ont été créés via le site web de Splot, soit plus de 300 modèles.
- le format **.xmi** : format dans lequel sont enregistrés les instances du métamodèle proposé par Perrouin et al. [PKGJ08] présenté section 3.2.2. La capacité à pouvoir lire des instances du métamodèle proposé par Perrouin et al. permet à Pacogen une compatibilité avec de nombreux formats. C'est un métamodèle du langage proposé par Schobbens et al. [SHTB07]. Ce langage est une généralisation de plusieurs langages existants. Ainsi, il est possible d'exprimer dans ce langage des modèles de *features* issus de formalismes différents.

La figure 6.27 est un exemple de modèle à contraintes généré, ce modèle à contraintes est issu du modèle de *features* présentés dans la figure 6.2.

Le paramètre `FListe` représente la liste des features, `CListe` représente les relations entre les features. Ces relations sont exprimées à l'aide des contraintes utilisateurs que nous avons présentées dans la figure 6.9 ($and(A, [B, C, ...])...$). La méthode `solver` est l'appel au solveur de contraintes, elle déclenche le processus de résolution. Sa signature est la suivante :

$$solver(FListe, CListe, Taille, Tri, Enum, Temps, Minimization).$$

Cette méthode a sept paramètres : `FListe` - la liste des features, `CListe` - la liste des contraintes, `Taille` -la taille de la matrice. Les valeurs à utiliser pour ces différents paramètres seront étudiées section 7.1.4. Les paramètres `Tri` et `Enum` permettent à l'utilisateur de choisir la fonction de tri et la technique d'énumération. Le paramètre `Temps` permet à l'utilisateur de spécifier le temps dédié à la minimisation. En pratique, la valeur de ce paramètre est choisie en fonction du temps disponible pour les activités de test et du temps nécessaire pour tester chaque configuration. Il est par exemple inutile d'allouer plusieurs heures pour la minimisation alors que le temps nécessaire pour tester une configuration n'est que de quelques secondes. Au contraire si le temps nécessaire pour tester une configuration est de l'ordre de la journée ou de la semaine, il devient pertinent d'augmenter le temps de minimisation de façon à réduire le nombre de configurations à tester. Le paramètre `Minimization` permet à l'utilisateur de sélectionner la fonction de minimisation : f_1 ou f_2. Si les paramètres ne sont pas spécifiés par l'utilisateur, des valeurs par défauts sont utilisées. La figure 6.27 présente un appel à la méthode solveur, pour une matrice de taille 100, un tri des paires : profondeur_pere, une technique d'énumération *first-fail* ([ff]), un temps de minimisation de 50000 ms et une fonction de coût f1.

2. **Vérificateur de cohérence** va produire la Matrice Solution contrainte destinée à accueillir les paires valides. La matrice est de taille $K \times N$, où K est le nombre de colonnes, égal au nombre de *features* et N le nombre de lignes, défini par l'utilisateur, qui représente le nombre de configurations maximal. La Matrice Solution créée est présentée figure 6.12. Ce composant réalise l'étape *Création de la Matrice Solution* du processus figure 6.1.

```
FListe =        [Env, OS, Browser, 32Bits,64Bits, Win2000,WinXP,WinVista, WinXP64,WinVista64,
FF15,FF2,IE55]
CListe =        and(Env,[OS]),
                opt(Env,[Browser]),
                xor(OS,[32Bits,64Bits]),
                xor(32Bits,[Win2000,WinXP,WinVista]),,
                xor(64Bits,[WinXP64,WinVista64]),
                or(Browser,[FF15,FF2,IE5.5]),
                mutex(64Bit,Firefox15),
                solver (FListe, CListe, 100, profondeur_pere,[ff],50000,f1).
```

FIGURE 6.27 – Modèle à contraintes généré dans le cas C3S

3. Le **Générateur de contraintes *pairwise*** prend en entrée la Matrice Solution et crée un ensemble de contraintes *pairwise* permettant de couvrir toutes les paires du modèle de features. Ce composant réalise l'étape *Génération des contraintes Pairwise* du processus figure 6.1.
4. Le **générateur de configurations en temps contraint** déclenche la résolution du problème d'optimisation. Il réalise la partie *Algorithme de résolution des contraintes* du processus 6.1. Il permet d'obtenir un ensemble de configurations couvrant toutes les paires existantes. Le processus de résolution remplit la Matrice Solution en assignant à chaque rang des paires générées une valeur. A la fin de la résolution, la matrice ne contient que des configurations valides couvrant *pairwise*. Dans ce processus, le temps de minimisation est optionnel. Si un temps est fourni, Pacogen minimisera durant ce temps imparti et retournera la meilleure solution qu'il ait pu trouver. Si aucun temps n'est fourni en entrée, Pacogen minimisera jusqu'à ce que le minimum de configuration ai été trouvé. Cependant comme le nombre de façons d'arranger les paires dans la matrice est très élevé, il est possible que le temps de minimisation nécessaire pour trouver le minimum global soit très important (de l'ordre de la semaine ou du mois). Les configurations générées sont fournies au format CSV.

6.7.2 Validation de l'implémentation de Pacogen

Le processus de validation de l'outil s'est fait en deux temps :
- vérification de la correction du modèle à contraintes du modèle de features
- vérification que toutes les paires valides étaient bien couvertes

Pour valider l'outil, nous nous sommes appuyé sur un outil déjà existant : SPLCATool, proposé Johanssen [JHF12a].

6.7.2.1 Validation du modèle à contraintes du modèle de features

La validation des contraintes utilisateurs qui représentent la sémantique des modèles de *features* s'est déroulée en deux parties : une validation unitaire puis une validation

plus générale à l'aide des modèles de *features* entiers.

Lors de la validation unitaire, nous avons manuellement testé les différentes contraintes utilisateurs que nous avons créées (and, or, xor...) et nous avons vérifié que le comportement de ces opérateurs était conforme à la sémantique des opérateurs du modèle de *features* présenté chapitre 3.

Ensuite, nous avons comparé le nombre de configurations obtenues grâce à notre moteur à contraintes et celui obtenu par SPLCAtool. Cette comparaison a été réalisée sur 10 modèles de *features* de différentes tailles issus du référentiel de modèle SPLOT. La comparaison des résultats obtenus avec ceux fournis par SPLCATool nous a permis de valider la modélisation. Les modèles choisis sont de tailles modestes (inférieure à 50 features) car l'énumération des configurations est un processus coûteux avec les technique de programmation par contraintes.

6.7.2.2 Validation des configurations *pairwise*

Une fois le modèle à contraintes validé, nous avons vérifié que la solution générée couvre bien *pairwise*. c'est-à-dire que toutes les paires valides du modèle étaient bien couvertes. Pour couvrir cette dimension du problème nous avons choisi de comparer le nombre de paires valides présentes dans la solution fournie par Pacogen et le nombre de paires valides présentes dans la solution fournie par SPLCATool. Cette méthode permet de vérifier que toutes les paires sont bien présentes dans la solution retournée par Pacogen.

L'utilisation conjointe de ces deux validations : la comparaison des deux moteurs de raisonnement sur les modèles de et le dénombrement du nombre de paires obtenues dans les deux jeux de configurations fournis par les deux outils nous permet de nous assurer de la correction de Pacogen.

6.8 Discussion

Dans ce chapitre nous présentons le cœur de la contribution de cette thèse. Une approche basée sur la programmation par contraintes pour la sélection de configurations de test qui respectent le critère *pairwise*. L'utilisation des techniques de programmation par contraintes permet de proposer une solution déterministe. Pour un même ensemble de paramètres d'entrées, les configurations générées resteront identique. De plus nous montrons comment utiliser les mécanismes de minimisations offerts par les solveurs à contraintes pour obtenir une solution de la plus petite taille possible.
L'approche proposée permet de minimiser la taille de la solution selon un contrat de temps. Ce processus de minimisation permet d'obtenir la plus petite des solutions existantes. Cependant, il est complexe de prouver que la solution trouvée est la plus petite. En effet, la sélection de configurations de test qui couvrent le critère *pairwise* est un problème à la combinatoire très importante : il y a souvent plusieurs millions de façons d'arranger les paires valides. Ces millions de façons d'arranger les paires mènent à la

création d'un arbre de recherche très grand. Pour prouver qu'une solution est la plus petite existante il faut parcourir tout l'arbre de recherche, ce qui peut être très long pour de grands modèles de *features*.

Chapitre 7

Configuration et évalutation expérimentale de Pacogen

Dans le chapitre précédent, nous avons présenté une solution basée sur les techniques de programmation par contraintes pour la sélection de configurations de test respectant le critère *Pairwise*. Cette approche fournit un processus de résolution reproductible et intègre une procédure permettant de minimiser le résultat obtenu. L'utilisation de cette approche nécessite la définition par l'utilisateur de différents paramètres présentés chapitre 6. Dans ce chapitre, nous classons ces paramètres en deux catégories. Nous proposons des méthodes pour déterminer la taille de la Matrice Solution et le temps de minimisation. Puis nous déterminons expérimentalement quelle combinaison de paramètres utiliser (tri des paires, fonction de minimisation et technique d'énumération) afin que Pacogen produise des configurations de test de taille minimale. Nous comparons Pacogen aux autres outils de l'état de l'art : SPLCATool, MosoPolite et CASA présentés sections 4.2.3 et 4.2.4. Ces expériences montrent que Pacogen retourne un nombre de configurations plus petit que SPLCATool sur 79% des modèles évalués et dans 18% des cas un résultat de taille identique. La comparaison entre Pacogen et MosoPolite montre que Pacogen donne un résultat de plus petite taille dans 100% des cas. La comparaison avec Casa montre que les deux outils ont des résultats similaires. Sur 17 % des modèles (37 modèles), CASA retourne un nombre de configurations plus petit que Pacogen. Sur 15 % des modèles (34 modèles) Pacogen retourne un nombre de configurations plus petit que CASA. Pour les autres modèles les deux outils retournent le même nombre de configurations.

7.1 Configuration optimale de Pacogen

On distingue deux types de paramètres : les paramètres de classe 1, où le choix d'une mauvaise valeur peut empêcher Pacogen de trouver une solution, et les paramètres de classe 2 qui améliorent le processus de résolution en permettant de trouver plus rapidement des Matrices Solutions de plus petites tailles.

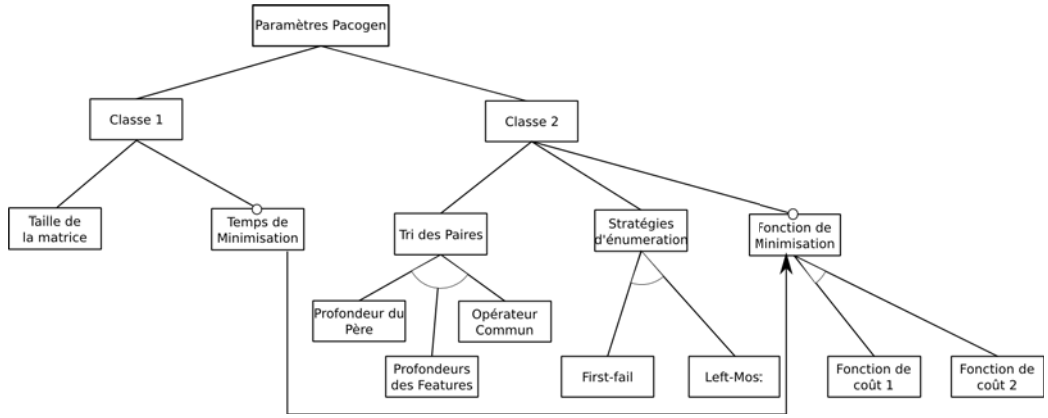


FIGURE 7.1 – Modèle de *features* des paramètres de Pacogen

7.1.1 Deux classes de paramètres

Dans cette section, nous présentons les deux classes de paramètres. L'ensemble des paramètres ainsi que leur classification est faite dans le modèle de *features* figure 7.1. On retrouve dans ce modèle les paramètres présentés dans le chapitre 6. Le temps de minimisation et la fonction de coût sont optionnels, car il est possible d'utiliser Pacogen dans une version sans minimisation. Dans ce cas, le programme retourne la première solution trouvée. Pour utiliser Pacogen dans sa version permettant la minimisation, il faut préciser quelle fonction de coût utiliser pour minimiser la taille de la Matrice Solution. Il n'est pas obligatoire de préciser un temps de minimisation : dans ce cas là, Pacogen explorera toutes les matrices solutions existantes et rendra la plus petite de toutes. Si un temps de minimisation est ajouté, alors Pacogen minimisera jusqu'à l'écoulement du temps imparti et rendra la plus petite matrice solution trouvée.

7.1.1.1 Les paramètres de classe 1

La classe 1 contient les paramètres dont les valeurs peuvent impacter la terminaison de Pacogen. Cette classe contient deux paramètres :

- la taille de la matrice solution
- le temps de minimisation.

Parmi ces deux paramètres, la taille de la matrice est obligatoire, mais le temps de minimisation est optionnel, car il est possible de configurer Pacogen pour ne fournir que la première solution sans enclencher de processus de minimisation.

Le choix de la taille de la matrice solution est important, car une taille inadaptée peut avoir deux conséquences : soit un échec du processus de résolution, soit des performances dégradées. Un échec du processus de résolution a pour origine une taille de matrice solution trop petite. Si la taille choisie est plus petite que la plus petite des solutions possible, cela revient à demander à Pacogen de trouver une solution plus petite que la plus petite des solutions existantes, ce qui n'est pas possible. Cette situation est

Paramètres	Valeurs
Tri des paires	profondeur_père profondeur_*features* opérateur_commun
Stratégie d'énumération	$left\_most$ $first\_fail$
Fonctions de coûts	$f_1 = \mathsf{Max}_{k \in 1..4n^2} R_k$ $f_2 = \sum_{k \in 1..4n^2} R_k$

TABLE 7.1 – Paramètres internes de Pacogen

due au fait qu'en présence de contraintes sur les modèles de *features*, il n'existe pas à l'heure actuelle de formule mathématique qui permet de calculer la taille de la plus petite solution couvrant le critère *pairwise*. D'autre part, une taille de matrice trop grande a pour conséquence une augmentation de la taille du problème à contraintes. Les contraintes utilisateurs *pairwise* vont contenir plus de variables, par conséquent les performances du processus de résolution vont être dégradées.

Le choix du temps de minimisation pose aussi un problème : si le temps de minimisation choisi est inférieur au temps nécessaire à Pacogen pour trouver une première solution, alors Pacogen sera incapable de trouver une solution.

Pour prévenir ces risques, il faut fournir à l'utilisateur des moyens pratiques pour sélectionner ces deux valeurs.

7.1.1.2 Les paramètres de classe 2

Les paramètres de classe 2 sont les paramètres qui ne peuvent pas remettre en cause la terminaison de Pacogen. Ces paramètres sont présentés section 6.6.2 et sont destinés à améliorer le processus de résolution. Pacogen en possède trois :

- le tri des paires
- la stratégie d'énumération
- les fonctions de minimisation.

Sont présentés, tableau 7.1 les différentes valeurs possibles des paramètres de classe 2. Il est possible de configurer Pacogen de 18 façons différentes ($3 \times 3 \times 2$), car le choix de la fonction de minimisation est optionnel dans le modèle de *features* présenté figure 7.1. Le choix de la bonne combinaison des paramètres peut améliorer la taille de la solution retournée.

7.1.2 Méthode de sélection des paramètres de classe 1

La sélection de valeurs optimales pour la taille de la Matrice Solution et le temps de minimisation sont primordiaux. Un mauvais choix peut entraîner un échec dans la résolution. Il est difficile pour un utilisateur de choisir à priori ces valeurs. Sans une expérience avec Pacogen et la sélection de configurations de test *pairwise*, un utilisateur

est incapable, à priori de donner une taille de matrice solution et un temps de minimisation minimum. Dans cette section nous présentons différentes méthodes qui permettent de définir des valeurs optimales pour ces deux paramètres.

7.1.2.1 Identification de la taille de matrice solution optimale

Pour estimer la taille de la matrice solution, nous proposons deux méthodes.

La première se base sur l'utilisation d'autres outils générant des configurations de test respectant le critère *Pairwise* comme SPLCATool, MosoPolite ou CASA, ou sur l'utilisation des résultats de l'état de l'art connus. Ces outils ou ces résultats permettent d'obtenir une première estimation du nombre de configurations nécessaires. Les expériences menées avec cet outil seront présentées dans la section 7.2.2, elles montrent que dans de nombreux cas, la solution fournie par l'outil est un très bon candidat comme taille de matrice solution. En moyenne, les tailles fournies par SPLCATool sont 15 % plus grandes (soit 1.9 configuration de plus), que celles retournées par Pacogen. Cette estimation est proche de la solution retournée par Pacogen, et l'utiliser permet d'une part de s'assurer que la Matrice Solution est assez grande, car une solution a été trouvée et d'autre part de ne pas avoir une taille trop grande.

La seconde méthode consiste à utiliser Pacogen dans une version légèrement modifiée qui intègre une double résolution. Cette méthodologie vise à permettre de traiter des modèles inconnus, où il n'y a pas de résultats déjà existant. Le principe est le suivant : on fournit en entrée à Pacogen un modèle de *features* et une matrice de grande taille (plus grande que nécessaire, généralement une taille de 200 est suffisante). Les paires valides sont générées et une première résolution est effectuée : Pacogen rend alors un nombre de configurations N. Nous utilisons ce nombre pour créer une nouvelle contrainte qui va contraindre que tous les rangs des paires valides soient strictement inférieurs à cette valeur N. Avec cette nouvelle contrainte un nouveau processus de résolution, intégrant les mécanismes de minimisations, est déclenché.

L'estimation de la taille de la Matrice Solution peut aussi se faire empiriquement en utilisant l'expérience de l'utilisateur avec les modèles de *features*.

7.1.2.2 Sélection du temps de minimisation optimal

Lors de l'utilisation de Pacogen dans la recherche d'une solution de la plus petite taille possible, il est possible de préciser un temps de minimisation. Cette minimisation est optionnelle, car il est possible de laisser Pacogen minimiser sans limites de temps et dans ce cas-là, la solution donnée sera le minimum global. Néanmoins, ce temps de minimisation peut être très important car les problèmes traités ont une combinatoire élevée. Pour limiter cette durée de minimisation, il est possible de fournir un temps de minimisation. Il est nécessaire de fournir un moyen de trouver le temps minimum de le moins élevé requis. Pour réaliser cet objectif, nous proposons deux solutions. La première

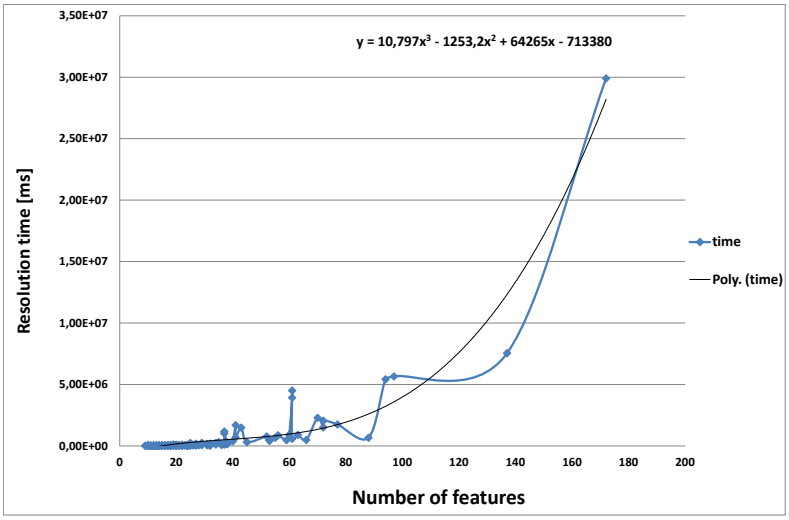


FIGURE 7.2 – Courbe d'estimation du temps de résolution en fonction du nombre de *features*

solution réutilise la version de Pacogen modifiée présentée dans la section précédente. Une première exécution de Pacogen permet, en plus de donner une première valeur pour la taille de la Matrice Solution, de donner le temps nécessaire pour obtenir une première solution. Le temps de minimisation doit alors être supérieur au temps nécessaire pour trouver la première solution. De cette façon, le risque d'avoir un temps de minimisation trop court est écarté.

La seconde méthode pour estimer le temps de minimisation se base sur l'utilisation de la fonction estimant le temps de résolution en fonction de la taille du modèle. Cette courbe a été élaborée sur la base des résultats expérimentaux qui seront présentés section 7.1.3. Pour créer cette courbe, nous avons utilisé 224 modèles de *features* issus du dépôt de modèle SPLOT [MBC09] présenté section 3.3.3. Pour chaque modèle est représenté le temps nécessaire pour trouver une première solution en fonction du nombre de *features*. Une courbe polynomiale de degrés 3 est tracée. La courbe résultante est présentée figure 7.2. À l'aide cette courbe il est possible d'estimer un temps de minimisation minimum en fonction d'un nombre de *features*. L'un des inconvénients est que les valeurs fournies par cette estimation ne sont valables que pour les modèles expérimentaux. On ne peut pas savoir, à priori, si cette technique pour estimer le temps de minimisation donnera des résultats. Néanmoins, cette estimation peut être utilisée pour donner une indication quant au temps à allouer à Pacogen pour la résolution de modèles inconnus.

Technique d'énumération	*first-fail*	*left-most*
Nombre moyen de conf.	12.40	16.13

TABLE 7.2 – Évaluation de la technique d'énumération Paramètres : nombre de modèles = 224 ; tri = profondeur_père ; sans minimisation ; taille de matrice = 200.

7.1.3 Configuration optimale des paramètres de classe 2

Pacogen propose plusieurs paramètres internes pour optimiser le processus de résolution des contraintes. Ces paramètres sont les suivants : profondeur_père, opérateur_commun et profondeur_*features* qui permettent de réaliser un tri des paires juste avant la début de la résolution, $first\_fail$ et $left\_most$ les deux stratégies de résolution et les deux fonctions de coûts :

$$f_1 = \mathsf{Max}_{k \in 1..4n^2} R_k$$
$$f_2 = \sum_{k \in 1..4n^2} R_k.$$

Ces paramètres sont listés table 7.1. Avec ces paramètres, il est possible de paramétrer Pacogen de 18 façons différentes. Dans cette section, nous souhaitons déterminer la configuration de Pacogen donnant, en moyenne, le plus petit nombre de configurations à tester. Nous ne testons pas Pacogen dans toutes ses combinaisons de paramètres possibles. Nous appliquons le processus suivant : nous prenons un paramètre et nous le faisons varier tout en fixant les autres paramètres. Cette expérience permet d'obtenir la valeur optimale (celle qui donne le plus petit nombre de configurations) de ce paramètre. Une fois ce paramètre déterminé, nous répétons le même processus pour les autres paramètres, en utilisant la valeur optimale du paramètre précédent. Nous déterminons d'abord quelle stratégie d'énumération retourne le meilleur résultat, puis quel tri des paires et finalement les deux fonctions de coûts. Pour chaque paramètre évalué, les résolutions sont réalisées sur 224 modèles de *features* issus de SPLOT [MBC09].

Les expériences ont été réalisées sur machine linux 64-bit avec deux CPU Intel Xeon $E5520$ et 16 GB de mémoire vive. Dans cette section est présenté une synthèse des expériences réalisées. L'ensemble des résultats est disponible en ligne [1].

7.1.3.1 Identification de la stratégie d'énumération optimale

La comparaison du nombre de configurations obtenues avec les deux techniques d'énumération a été réalisée sur 224 modèles. La fonction de tri des paires choisie est celle de profondeur_père et la résolution se fait sans minimisation. Seule la technique d'énumération varie. Le tableau 7.2 présente le nombre moyen de configurations obtenu pour chaque technique d'énumération.

Les résultats montrent que *first-fail* réalise de meilleures performances que *left-most*. La moyenne de la taille des matrices solutions *first-fail* est de 12.40 contre 16.13 pour *left-most*. Pour 83% des modèles *first-fail* donne des solutions jusqu'à 73% plus petites.

1. http ://hervieu.info/pacogen/

Modèle	*first-fail*	*left-most*
Aircraft PL	9	9
Arcade game	13	35
Car PL	6	7
Coche Ecologico	90	93
Connector PL	14	14
Doc generation	12	18
Fame DBMS	10	10
Inventory	12	14
Model Trans.	23	49
Movie App PL	6	6
Search Engine	11	13
Sienna	20	22
Smart Home	11	18
Stack PL	9	14
Web Portal	15	20

TABLE 7.3 – Évaluation des techniques d'énumération. Paramètres : nombre de modèles = 15 ; tri = profondeur_père ; sans minimisation ; taille de matrice = 200.

Pour 17% des modèles, les deux approches fournissent les mêmes résultats en nombre de configurations. Un extrait des résultats est présenté table 7.3. Ce tableau montre que les performances, en terme de nombre de configurations obtenues sont meilleures lorsque la technique d'énumération utilisée est *first-fail*. Cette différence de performance s'explique par la nature de *first-fail* qui est une technique de sélection dynamique. Avant de placer une paire dans la matrice, *first-fail* inspecte tous les domaines des paires non placées et sélectionne celle qui a le plus petit domaine, et la place dans la matrice.

7.1.3.2 Identification de la fonction tri des paires optimale

L'évaluation des trois fonctions de tri des paires a été réalisée sur les 224 modèles issus de SPLOT. Nous souhaitons déterminer lequel de ces tris fournit le plus petit nombre de configurations. Pour chaque modèle, la technique d'énumération utilisée est *first-fail* et les résolutions sont faites sans processus minimisation, seulement l'algorithme de tri des paires a été changé. La taille moyenne des configurations obtenue pour chaque tri est présentée dans le tableau 7.4. Les résultats montrent que l'application d'un tri des paires permet une amélioration du nombre de configurations trouvé par rapport à une absence de tri. La comparaison des trois fonctions de tri montre que le tri le plus performant en terme de nombre de configurations est profondeur_père, suivi de opérateur_commun puis profondeur_*features*.

Tri	opérateur commun	profondeur *features*	profondeur père	Aucun tri
Nombre moyen de configurations trouvé	12.57	12.79	12.43	13.42

TABLE 7.4 – Évaluation du tri des paires. Paramètres : nombre de modèles : 224 ; énumération : *first-fail* ; sans minimisation ; taille de matrice : 200.

Fonction de minimisation	Fonction 1	Fonction 2
Nombre moyen de conf.	11.86	12.40

TABLE 7.5 – Évaluation des deux fonctions de minimisation. Paramètres : nombre de modèles : 224 ; tri : profondeur père ; Énumération : *first-fail* ; taille de matrice : 200.

7.1.3.3 Identification de la fonction de coût optimale

Dans cette section sont évaluées les deux fonctions de coûts. Ces fonctions sont évaluées sur 224 modèles, avec le tri profondeur_père et la technique d'énumération *first-fail*. La taille moyenne des configurations pour les deux fonctions de coût est présentée dans le tableau 7.5. Ce tableau montre que la moyenne des tailles des configurations est plus petite pour la fonction $f1$. L'analyse des résultats expérimentaux montre que sur 100 % des modèles, la fonction $f1$ donne un nombre de configurations inférieur ou égal à la fonction $f2$. Sur 40 % des modèles, la fonction $f1$ fournit un nombre de configurations strictement inférieur à la fonction $f2$. Sur certains modèles nous constatons que la fonction $f1$ permet d'obtenir un nombre de configurations jusqu'à 46% plus petit que le nombre de configurations retourné par la fonction de coût $f2$.

Pour illustrer comment les deux fonctions de minimisation diffèrent, nous avons réalisé une contrainte utilisateur nous permettant de surveiller le processus minimisation. Dès qu'une paire est placée dans la matrice, cette contrainte utilisateur se réveille et indique le nombre de paires restantes à arranger. La figure 7.3 présente les résultats des observations réalisées sur les deux fonctions de minimisation f_1 et f_2 sur le modèle de *features Application* FM de SPLOT. Les nombres de la forme $\#Val$ représentent la taille de la configuration trouvée. L'axe des abscisses représente le temps en seconde, l'axe des ordonnées le nombre de paires restantes à mettre dans la matrice. Pour ce modèle de *features* il y a l'origine 1131 paires à arranger.

Quand le nombre de paires restantes atteint 0, une solution a été trouvée. Dans la figure 7.3 Pacogen trouve neuf solutions grâce à la fonction de coût f_1, la solution la plus petite est de huit configurations. La fonction de coût f_2, Pacogen trouve deux solutions, dont la plus petite est de taille 16. L'allure de la courbe illustre le phénomène de backtracking présenté dans la section 5.3.2 utilisé durant la résolution. Le backtracking permet au solveur à contraintes de revenir dans un état antérieur du processus de résolution.

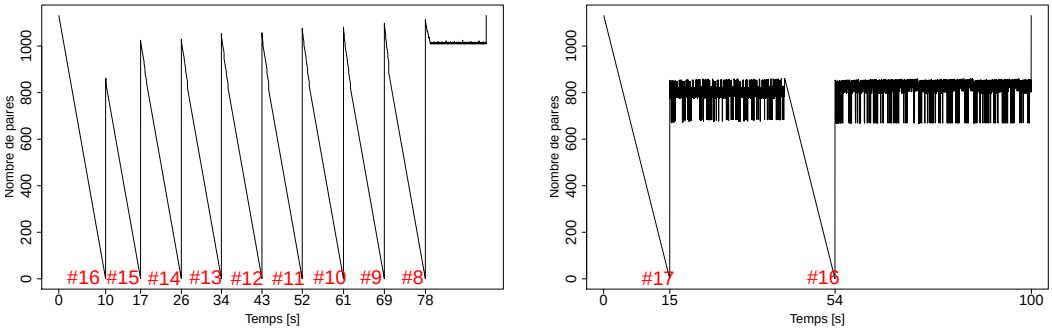


FIGURE 7.3 – Comparaison des deux processus de minimisation pour le modèle de *features Applications* avec (a) la fonction de coût $f1$ et (b) la fonction de coût $f2$.

Cette expérience montre que la fonction de coût $f1$ donne de meilleurs résultats : elle fournit une solution plus petite et teste plus de configurations (neuf configurations sont testées avec les fonction $f1$ contre deux seulement avec $f2$).

7.1.4 Configuration optimale

Les expériences précédentes permettent de définir une combinaison optimale de paramètres : profondeur_père pour le tri des paires, $first\_fail$ pour la sélection des variables et la fonction de coût $f1$ comme configuration optimale de Pacogen.

7.2 Comparaison de Pacogen à l'état de l'art de la génération *pairwise*

Dans cette section, Pacogen est comparé à deux approches existantes qui traitent du problème de la génération de configuration *pairwise* sur les modèles de *features* : celle proposée par Johanssen avec SPLCATool, [MFJF11, JHF12a] et celle proposée par Oster, [OMR10], avec l'outil MosoPolite. Nous comparons Pacogen à une approche plus générale proposée par Garvin & Al. [GCD11] traite de la sélection de configurations de test *pairwise* en présence de contraintes en général. Les expériences sont réalisées sur une machine linux 64-bit avec deux CPU Intel Xeon $E5520$ et 16GB de mémoire vive. Les résultats présentés dans cette section sont une synthèse des expériences réalisées. Les résultats sont disponibles en ligne [2].

7.2.1 Données expérimentales

La comparaison de Pacogen, SPLCATool et CASA se fait sur 224 modèles de *features* issus de SPLOT [MBC09]. Ces modèles ont une taille comprise entre 9 et 290

2. http ://hervieu.info/pacogen/

features. Comme SPLCATool et CASA sont disponibles en téléchargement, nous avons pu évaluer les outils avec les 224 modèles. L'outil Mosopolite n'étant pas disponible en téléchargement, nous n'avons pas pu reproduire les expériences pour le comparer exhaustivement à Pacogen. La comparaison entre Pacogen et Mosopolite [OMR10] est réalisée sur la base des résultats publiés par l'auteur sur sept modèles de variabilités.

7.2.2 Résultats et analyses

Le tableau 7.2.2 présente les résultats en terme de nombre de configurations et temps de résolution pour 17 modèles de *features*. Ces modèles ont été choisis, car ils sont de tailles différentes et ont été utilisés par d'autres chercheurs pour la présentation de leurs résultats [OMR10] [JHF11].

Pour déterminer la taille de la matrice nous avons utilisé la première des deux méthodes qui sont présentées section 7.1.2.1. Nous nous avons utilisés les résultats retournés par l'outil CASA.

Les deux premières colonnes du tableau présentent les propriétés des modèles de *features* : le nombre de *features* et le nombre de configurations valides. Les trois colonnes suivantes présentent le nombre de configurations de test valides trouvées par les outils SPLCATool, Mosopolite et CASA. Les deux suivantes présentent le nombre de configurations obtenues par Pacogen. La première présente les résultats sans processus de minimisation (#1), tandis que la seconde présente les résultats obtenus après minimisation (#2). Les trois dernières colonnes du tableau 7.2.2 montrent la différence en nombre de configurations trouvées par Pacogen, SPLCATool, MosoPolite et CASA, en pourcentage. Une valeur inférieure à zéro signifie que Pacogen trouve un plus petit nombre de configurations.
Ce tableau permet d'illustrer l'efficacité de l'utilisation du critère *pairwise* pour la sélection de configurations de test à base de modèle de *features*. Par exemple, dans le cas du modèle *Model Transformation*, l'application du critère *Pairwise* permet de passer de plus 10^{13} configurations à tester en test exhaustif à 25 avec Pairwise. Sur les 17 modèles présentés, Pacogen des résultats meilleurs ou égaux en terme de nombre de configurations que les outils MosoPolite, SPLCATool et Casa. Pacogen est capable de proposer des solutions jusqu'à 33 % plus petites que les autres outils. Les outils Pacogen et CASA ont des performances proches très proches.

Modèle	Métriques		SPLCAT	MosoPolite	CASA	Pacogen	Amélioration		
	♯F	♯Conf					vsSPLCAT	vsMosoPolite	vsCASA
Car PL	9	315	7	–	6	6	−14%	–	0%
Aircraft PL	13	315	9	–	8	8	−11%	–	0%
Movie APP PL	13	24	7	–	6	6	−14%	–	0%
Search Engine	14	126	13	–	11	11	−15%	–	0%
Stack PL	17	24	12	–	10	9	−25%	–	−10%
Connector PL	20	30	15	–	14	14	−7%	–	0%
Fame DBMS	21	320	9	–	8	8	−11%	–	0%
Smart Home	35	1048576	10	11	10	8	−20%	−27%	−20%
Inventory	37	2028096	13	12	10	10	−23%	17%	0%
Sienna	38	2520	22	24	20	20	−9%	−17%	0%
Doc generation	44	55700000	12	18	12	12	0%	−33%	0%
Web Portal	43	2120800	19	26	15	15	−21%	−42%	0%
Arcade game	61	$3.3 * 10^9$	18	–	13	12	−33%	–	−8%
Model Trans.	88	$1.63 * 10^{13}$	25	40	24	24	−4%	−40%	0%
Coche Ecologico	94	23200000	93	92	90	90	−3%	−2%	0%
UP estructural	97	$> 10^9$	36	–	37	33	−8%	–	−10%
Printers	172	$> 10^9$	182	–	180	180	−1%	–	0%

TABLE 7.6 – Comparaison de Pacogen, SPLCATool ; MosoPolite et CASA en terme de nombre de configurations *pairwise*.
Paramètres : énumération = first_fail ; sort
Métriques : nombre de *features* (♯F) et nombre de configurations (♯Conf) pour les modèles de *features*

Le tableau 7.2.2 présente les temps de résolution des trois approches pour les 17 modèles présentés dans le tableau 7.2.2. Pour deux modèles Pacogen a un temps de résolution plus faible que les deux autres approches. Ces deux modèles sont les plus petits en nombre de *features*, le problème combinatoire est donc plus petit à résoudre. Pour tous les autres modèles, l'approche proposée par SPLCATool est la plus rapide. Pacogen est plus rapide que CASA sur huit modèles et est plus lent sur sept modèles. Il n'est pas possible de déterminer qui de Pacogen ou CASA est le plus rapide. Dans certain cas Pacogen est bien plus rapide que CASA, par exemple dans le cas de Web Portal, Pacogen est 29 fois plus rapide. Dans d'autres cas c'est CASA qui est le plus rapide : par exemple pour le modèle Doc Generation CASA est 184 fois plus rapide que Pacogen.

7.2.2.1 Comparaison exhaustive de Pacogen, SPLCATool et CASA

Dans cette section nous comparons exhaustivement les trois outils : Pacogen, SPLCATool et CASA. Pour déterminer les paramètres de classe 1 de Pacogen, nous utilisons le protocole expérimental suivant. Une première résolution de Pacogen sans minimisation est réalisée sur les 224 modèles de *features*. Cette première résolution fournit deux informations : la taille de la première solution sans minimisation (la deuxième des deux méthodes qui sont présentées section 7.1.2.1) ainsi que le temps nécessaire pour obtenir cette première solution. Ces deux paramètres sont utilisés lors d'une deuxième résolution ou nous imposons une minimisation d'une durée égale à quatre fois le temps de résolution de la première. Le choix de la valeur 4 est arbitraire.

Comparaison exhaustive de Pacogen et SPLCATool

La comparaison entre Pacogen et SPLCATool sur les 224 modèles de *features* montre que Pacogen génère un nombre de configurations plus petit pour 79 % des 224 modèles. Pour 2% des modèles, SPLCATool fournit de meilleurs résultats que Pacogen. Pour ces 2 % Pacogen n'est pas capable de fournir de meilleure solution ou équivalente à celle de SPLCATool car le temps impartit par le protocole expérimental n'est pas assez important. Si le temps de minimisation avait été plus élevé, Pacogen aurait été capable de fournir des résultats équivalents à ceux de SPLCATool. Dans 18% des cas, les résultats sont identiques. Pour certains de ces modèles, Pacogen permet d'obtenir des configurations jusqu'à 60% plus petites que celles obtenues par SPLCATool. La figure 7.4 présente les différences en nombre de configurations entre les deux outils pour les 224 modèles sélectionnés. L'axe des abscisses représente la différence en nombre de configurations, en pourcentage. Ces différences sont groupées en neuf classes. Une valeur négative signifie que Pacogen génère moins de configurations que SPLCATool, tandis qu'une valeur positive signifie que SPLCATool obtient de meilleurs résultats. La colonne de la classe 0 % représente les modèles pour lesquels Pacogen et SPLCATool ont rendu le même résultat.

Pour toutes ces expériences, le temps de résolution moyen est de 132 074 ms et le temps médian est de 9 855 ms pour Pacogen. SPLCATool possède un temps de réso-

Modèle de *features*	Pacogen (ms)	Casa (ms)	SPCATool (ms)
Car PL	50	212	340
Aircraft	600	206	971
Movie App	70	192	520
Search Engine	400	297	596
Stack PL	280	4 244	273
Connector	830	1 633	803
Fame	55 350	1 450	574
Smart Home	1 110	4 217	974
Inventory	476 390	28 846	400
Sienna	6 510	6 078	1 693
Doc generation	184 440	1 7336	734
Web Portal	5 940	174 459	2 270
Arcrade Game	23 830	6840	2 559
Model Transformation	472 010	1 0012 246	3 165
Coche Ecologico	1 015 940	175 751	4 724
UP Estructural	565 850	607 557	3 320
Printers	11 662 080	783 253	15 306

TABLE 7.7 – Comparaison des temps de résolution en ms des outils Pacogen, SPLCATool et CASA. En rouge, temps de résolution pour lesquels Pacogen est le plus lent. En orange, temps de résolution pour lesquels Pacogen est plus rapide qu'une approche et moins rapide que l'autre. En vert, temps de résolution pour lesquels Pacogen est le plus rapide

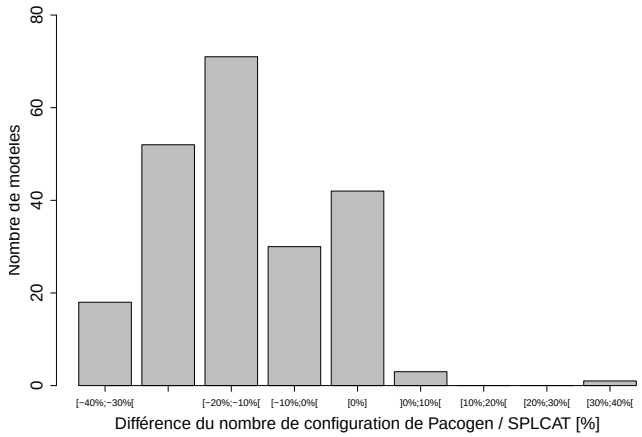


FIGURE 7.4 – Différence du nombre de configurations trouvées par Pacogen et SPLCATool (en pourcentage) pour 224 modèles. Paramètres : labelling = first_fail ; tri des paires ; Fonction de minimisation 1 ; Taille de matrice = 200.

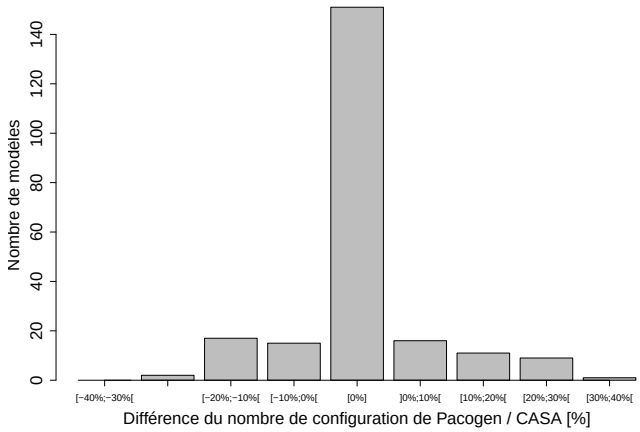


FIGURE 7.5 – Différence du nombre de configurations trouvées par Pacogen et SPLCATool (en pourcentage) pour 224 modèles. Paramètres : labelling = first_fail ; tri des paires ; Fonction de minimisation 1 ; Taille de matrice = 200.

lution moyen de 611 ms et un temps de résolution médian de 326 ms. Cet outil est bien plus rapide que Pacogen. Cette rapidité s'explique par l'absence de processus de minimisation dans le processus de résolution de SPLCATool.

Comparaison exhaustive de Pacogen et CASA

La comparaison exhaustive de Pacogen et CASA montre que pour 17 % des modèles CASA retourne un nombre de configurations plus petit. Ces configurations sont en moyenne 13 % de configurations en moins. Sur 15 % des modèles Pacogen retourne un nombre de configurations plus petit que CASA. Ces configurations ont un moyenne 11 % de configurations en moins. Le graphique présenté figure 7.5 résume ces résultats. L'axe des abscisses représente la différence en nombre de configurations, en pourcentage. Ces différences sont groupées en neuf classes. Une valeur négative signifie que Pacogen génère moins de configurations que CASA, tandis qu'une valeur positive signifie que CASA obtient de meilleurs résultats. La colonne de la classe 0 % représente les modèles pour lesquels Pacogen et CASA ont rendu le même résultat

Pour ces expériences, le temps de résolution moyen est de 132 074 ms et le temps médian est de 9 855 ms pour Pacogen. Pour CASA le temps de résolution moyen est de 101 564 ms et le temps médian est de 718 ms. Pacogen est plus lent que CASA car notre approche utilise un processus de minimisation. Ce temps de minimisation est supérieur au temps nécessaire pour trouver une solution, et est utilisé pour explorer l'espace des solutions. CASA lui retourne une solution et n'utilise pas de processus de minimisation. C'est pour cette raison que Pacogen est plus lent.

Comparaison exhaustive de Pacogen et MosoPolite

La comparaison entre Pacogen et MosoPolite ne peut se faire que sur sept modèles, car ce sont les seuls résultats que nous avons pu nous procurer. Elle montre que Pacogen génère un nombre de configurations plus petit sur six des modèles et un nombre de configurations équivalent pour un seul modèle. Les résultats sont présentés table 7.2.2.

7.3 Discussion

Cette section présente plusieurs méthodes pour la sélection des valeurs adéquates pour deux paramètres critiques : la taille de la matrice solution et le temps de minimisation. À l'aide des expérimentations réalisées sur 224 modèles *features* différents, nous avons identifié empiriquement une combinaison optimale de paramètres de classe 2 : le tri des paires, la stratégie d'énumération et la fonction de coût. Cette combinaison optimale est la suivante : *first-fail*, la profondeur du père pour le tri des paires et la fonction de coût numéro 1. Nous avons ensuite évalué Pacogen par rapport aux autres outils de l'état de l'art. Dans un premier temps nous avons comparé Pacogen, SPLCATool et CASA en utilisant les résultats retournés par CASA. Cette évaluation nous a permis de montrer que Pacogen rend, dans 100 % des cas un nombre de configurations inférieur ou égal au nombre de configurations retourné par les deux autres outils. Parmi ces trois outils, SPLACTool est le plus rapide. Les temps de résolution pour les outils CASA et Pacogen ne permettent pas de conclure si un des deux outils est plus rapide que l'autre.

Nous avons ensuite comparé exhaustivement Pacogen, SPLCATool et CASA sur 224 modèles de *features* différents. Cette évaluation s'est faite en utilisant seulement les résultats fournis par Pacogen. Nous avons montré que Pacogen fournit des résultats meilleurs ou égaux que deux des approches existantes : dans 98% des modèles pour SPLCATool et dans 100 % des cas Pacogen est meilleur que MosoPolite, et nous avons montré que les résultats retournés par Pacogen sont proches de ceux retournés par CASA. Si nous avions accordé un temps de minimisation plus élevé nous aurions obtenu des résultats meilleurs en terme de nombre de configurations.

Une des faiblesses de notre approche réside dans le temps de minimisation. L'arbre de recherche étant grand il faut parfois un temps de minimisation très important pour trouver une solution optimale. Cependant, en l'absence de contrat de temps, la solution retournée à la fin du processus de minimisation par Pacogen sera la plus petite.

Chacun des outils évalués à ses points forts et ses points faibles : SPLCATool est très rapide, mais génère un nombre de configurations plus élevé. CASA génère un nombre de configurations plus petit, mais est moins rapide, de plus il ne prend pas directement en entrée des modèles de *features*, une transformation de modèles doit être réalisée pour transformer un modèle de *features* en un modèle CASA. Pacogen, lui, génère un

nombre de configurations plus petit mais nécessite d'être paramétré finement : taille de la matrice, temps de minimisation... et, est généralement plus lent (en fonction du temps de minimisation alloué).

De part leurs caractéristiques différentes, ces trois outils peuvent être utilisés dans des contextes différents. Ainsi, l'outil SPLCATool, qui est plus rapide, mais qui génère un nombre de configurations plus élevé est adapté à des phases de tests unitaires ou les configurations sélectionnées font office de données de test. Le fait d'ajouter un test ne va pas coûter de trop en temps de test. Au contraire, dans les cas ou les modèles de *features* représentent des architectures à composants, l'utilisation de Pacogen pourrait être plus intéressante. En effet, dans le cas de ce type d'architecture, ce sont des configurations de test qui sont extraites, il faut alors tester ces configurations. Si le temps nécessaire pour tester une configuration est d'une semaine, alors il est pertinent de consacrer plus de temps à la sélection des configurations afin d'obtenir le plus petit nombre possible de configurations de test et donc de réduire le temps de test.

Chapitre 8

Pairwise et industrie : Application

Dans ce chapitre, nous étudions comment la technique de sélection de configurations de test qui respectent le critère *pairwise* peut s'inscrire dans un cadre industriel. Nous étudions l'applicabilité de cette technique sur les deux projets de test présentés chapitre 2 : le projet de test du logiciel BIEW de Orange et le projet de test du logiciel TéléPrésense de Cisco.

8.1 Applicabilité dans le secteur industriel

Dans le chapitre 6 nous montrons que la technique de test combinatoire basé sur le critère *pairwise* permet d'extraire un petit échantillon de configurations, alors que cet espace de configurations peut en contenir parfois plusieurs millions. Ces résultats montrent que le pairwise est une solution pertinente et efficace pour résoudre le problème de sélection des configurations de test. Cependant, les résultats expérimentaux présentés chapitre 7 sont exclusivement basés sur des modèles académiques et ne s'intéressent qu'à une dimension purement quantitative : le nombre de configurations obtenu. Aucune analyse n'est réalisée pour étudier la capacité de ces configurations à mettre en évidence des défauts dans le logiciel. De plus, les travaux existants [JHF+12b, YC06] qui s'intéressent à l'utilisation du *pairwise* n'étudient pas l'intégration de cette technique dans les processus industriels, les gains potentiels... Dans ce chapitre, nous étudions l'application du critère *pairwise* dans un cadre industriel, nous évaluons l'utilisation de Pacogen selon plusieurs critères :

- le gain de temps dû à l'utilisation d'un modèle de *features*
- la pertinence des configurations de test sélectionnées dans le cas du projet de Test Biew.

8.2 *Pairwise* et le logiciel BIEW

Le projet BIEW est un projet de test d'une durée de cinq ans, qui a occupé en moyenne une équipe de six testeurs. L'étude à posteriori de ce projet de test nous a permis d'identifier plusieurs difficultés. Dans cette section, nous revenons sur ce projet

de test et nous le revisitons en utilisant une approche *pairwise*. L'analyse réalisée s'est faite en marge du projet, les configurations extraites n'ont pas été directement utilisées telles quelles dans le cadre de ce projet.

8.2.1 Problématiques

Dans ce projet de test, les testeurs font face à deux problèmes majeurs : une absence de représentation explicite de la variabilité et une difficulté à choisir les configurations de test.

8.2.1.1 Représentation de la variabilité

Les testeurs n'ont pas de vision globale du système et ont des difficultés à le modéliser. Par exemple, ils ne peuvent pas modéliser les incompatibilités entre des éléments qui composent l'environnement. De plus, certains choix réalisés ne reflètent pas la réalité : par exemple, les configurations de test sélectionnées par les testeurs ne peuvent pas avoir plus de deux navigateurs en même temps. Ce choix est critiquable car un peu décalé de la réalité : de nombreux utilisateurs possèdent en général deux, voire trois navigateurs différents.

8.2.1.2 Sélection des configurations de tests

Face à la taille du problème de test : des millions d'environnements possibles composés de multiples éléments, les testeurs ont défini leur propre méthode de test, cette méthode est décrite dans le chapitre 2. Un des points faibles de la méthodologie employée est que les testeurs n'étaient pas capable de capitaliser les configurations de test. Des configurations de test identiques étaient régulièrement mises en place, pour une même campagne de test. Les configurations de test étaient sélectionnées sans discussion entre les testeurs et utilisées telles quelles.

Les travaux présentés précédemment dans cette thèse mettent en avant deux outils qui permettent de répondre à ces deux problématiques : l'utilisation des modèles de *features* pour capturer les environnements de tests et le critère *pairwise* permettant la sélection de configurations de test.

8.2.2 Application

Dans cette section nous présentons comment le modèle de *features* a été réalisé et comment il est utilisé pour extraire un ensemble de configurations de test.

8.2.2.1 Réalisation du modèle de *features*

Nous avons nous même réalisé le modèle de *features*. Pour concevoir le modèle nous avons utilisé plusieurs sources d'informations : les spécifications environnementales, qui contiennent l'ensemble des informations relatives aux environnements de test : Système

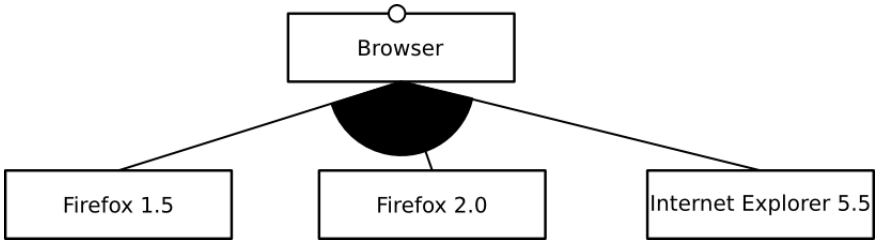


FIGURE 8.1 – Extrait du modèle de *features* du projet de test Biew

FIREFOX 1.5	FIREFOX 2	INTERNET EXPLORER 5.5
0	0	0
1	0	0
0	1	0
0	0	1

TABLE 8.1 – Configurations partielles de test considérées pour le mode de calcul des configurations chapitre 6 ; 0 : feature non sélectionnée ; 1 : feature sélectionnée

d'exploitation, clés 3G... et la connaissance informelle des testeurs qui contient des informations relatives aux incompatibilités, par exemple celle entre le périphérique 3G ZTE ROHS et Windows 2000. La réalisation du modèle de *features* s'est faite en environ vingt heures. Chaque catégorie environnementale a été transformée en feature. Les éléments appartenant à chaque catégorie environnementale ont ensuite été ajoutés en tant que fils de cette feature. Ainsi la catégorie environnementale Navigateur permet de créer le nœud présenté figure 8.1

Le modèle obtenu est de 64 *features* et mène à 2 975 520 configurations possibles. Ce chiffre est plus important que le nombre de configurations calculé dans le chapitre 6 (1 920 000). Cette différence s'explique par le mode de calcul utilisé. Dans le chapitre 6, le nombre de configurations résulte du produit du nombre d'éléments de chaque **catégorie environnementale** (nombre de systèmes d'exploitation $\times$ le nombre de clés $3G$ $\times$ le nombre de navigateurs ...). Cependant, ce mode calcul ne prend pas en compte les combinaisons de deux ou trois navigateurs ou les incompatibilités. L'utilisation d'un modèle de *features* permet de considérer ces cas. Dans le modèle réalisé, les *features* de type navigateur sont reliées par un opérateur *Or* (figure 8.1) qui autorise ces arrangements. Le nombre de configurations entre les navigateurs n'est alors plus de 4 mais de 8. Les configurations considérées par les deux modélisations sont présentées dans les tables 8.1 et 8.2.

8.2.2.2 Génération des configurations pairwise

Pour réaliser l'extraction des configurations de test, nous utilisons Pacogen avec les paramètres suivants : *first-fail*, avec une matrice de taille 300, la fonction de minimisa-

Firefox 1.5	Firefox 2	Internet Explorer 5.5
0	0	0
1	0	0
0	1	0
0	0	1
1	1	0
0	1	1
1	0	1
1	1	1

Table 8.2 – Configurations partielles de test considérées pour le mode de calcul des configurations basé sur le modèle de *features* ; 0 : feature non sélectionnée ; 1 : feature sélectionnée

tion 1 et un temps de minimisation de 3 heures. L'utilisation du critère *pairwise* sur le modèle de *features* du logiciel BIEW permet d'extraire 254 configurations de test. Ce nombre de configurations est relativement important par rapport aux résultats expérimentaux présentés dans le tableau 7.2.2 où la taille des configurations extraites était plus faible. Nous expliquons ce résultat par la structure du modèle de *features*. Ce modèle est composé de multiples nœuds xor : entre les systèmes d'exploitation, ou encore entre les éléments permettant de se connecter en mobilité (ceux appartenant à la catégorie Mobile chapitre 2). La présence de nœuds xor provoque une augmentation du nombre de configurations nécessaire pour atteindre le critère *pairwise*. Par exemple, pour couvrir pairwise entre les catégories Système d'exploitation (cinq systèmes d'exploitation différents) et Mobile (25 périphériques mobiles), deux catégories où les *features* filles sont reliées pas une relation xor, il faut $5 \times 25 = 125$ configurations.

8.2.3 Pertinence de l'approche

Dans ce projet industriel, l'ensemble des éléments produits durant le projet est disponible : cas de test, configurations de test, verdicts... Tous ces documents nous permettent d'évaluer l'intérêt d'appliquer une approche *pairwise*. Nous évaluons cette approche selon deux dimensions : d'un point de vue quantitatif, en comparant le nombre d'environnements obtenus avec *pairwise* et le nombre d'environnements mis en place durant le projet de test ; et d'un point de vue qualitatif, en étudiant si les environnements sélectionnés par *pairwise* sont des environnements susceptibles de mettre en évidence des défauts dans le logiciel BIEW.

8.2.3.1 Bénéfices de l'utilisation d'un modèle de *features*

L'utilisation du modèle de *features* est source de plusieurs bénéfices pour les testeurs. Elle permet de quantifier le volume de test demandé par Orange : tester plus de deux millions d'environnements. De plus, l'utilisation d'un modèle de variabilité permet

aux équipes de test de sélectionner des configurations plus proches de la réalité. Ce modèle de variabilité permet de capitaliser la connaissance des testeurs, et il est ainsi possible de formaliser des incompatibilités matérielles ou logicielles grâce à l'utilisation des contraintes mutex.

8.2.3.2 Dimension quantitative

D'un point de vue quantitatif, l'application de l'approche *pairwise* dans le cadre du projet BIEW est pertinente. Les testeurs, dans le projet BIEW ont dû mettre en place 390 environnements différents. Parmi ces 390 environnements de test, 149 ont été identifiés comme redondants : ces environnements ont été mis en place plusieurs fois pour exécuter des cas de test différents. L'utilisation de Pacogen pour appliquer le critère *pairwise* sur ce projet de test permet de sélectionner 254 environnements de test, soit 45 % de moins que le nombre d'environnements utilisés dans le projet industriel.
De plus, l'utilisation du critère *pairwise* permet une bien meilleure couverture des éléments composant l'environnement. Lors du projet de test BIEW les environnements mis en place sont des environnements partiels qui impliquent seulement un système d'exploitation et un autre élément qui appartient à une autre caractéristique environnementale (par exemple : browser, VPN, Navigateur...). Ces configurations partielles ne sont pas représentatives des configurations de l'utilisateur final : elles sont incomplètes et manquent de diversité. Pour mesurer cette diversité, nous considérons les paires d'éléments contenues dans les configurations. On appelle une paire d'éléments un couple d'éléments issus chacun d'une catégorie environnementale. Par exemple si l'on considère la configuration suivante :

```
(Windows XP, Firefox 2, Sagem 706 A)
```

celle ci est composée de trois paires d'éléments :

```
(Windows XP, Firefox 2),
(Firefox 2, Sagem 706 A),
(Windows XP,  Sagem 706 A)
```

Dans ce projet de test, les configurations choisies couvrent 285 paires, sur un total de 6 832 paires possibles, soit une couverture de 4 %. L'utilisation du critère *pairwise* permet d'atteindre une couverture de 100 % des paires. L'ensemble de configurations de test est alors plus diversifié.

Il est intéressant de s'interroger sur la pertinence de l'utilisation du *pairwise* en émettant l'hypothèse que les ingénieurs ont été capables de supprimer les environnements redondants. Si l'on admet que le processus de test n'admet aucun environnement redondant, le nombre d'environnements de test utilisé est alors de 241, contre 254 si l'on applique le critère *pairwise*. L'approche *pairwise* nécessite alors de tester 5 % de configurations en plus par rapport à l'approche originale. Néanmoins, ce chiffre est à mettre en balance avec les gains offerts par l'utilisation de Pacogen :

- la sélection des configurations résulte d'un processus automatique, tandis que dans le projet de test les configurations sont choisies manuellement
- les configurations choisies sont partielles, tandis qu'avec Pacogen, celles-ci sont complètes
- les configurations sélectionnées sont plus représentatives de la diversité de l'environnement (100 % des paires d'éléments couvertes contre 4 %)
- le critère *pairwise* fournit une information objective sur la couverture des environnements de test (toutes les interactions *pairwise*) tandis que le processus utilisé ne permet pas d'obtenir cette information.

Au regard des points cités précédemment, il est raisonnable de penser que l'utilisation de Pacogen dans le cadre de ce projet reste pertinente même si le nombre de configurations à tester dans le cas d'un processus de test optimisé est supérieur de 5 % .

8.2.3.3 Dimension qualitative

Pour évaluer la pertinence des configurations sélectionnées, nous avons choisi de voir si les configurations extraites par Pacogen pouvaient mettre en évidence des défauts dans l'application. Comme il nous est impossible d'exécuter manuellement les centaines des tests sur les 254 configurations de tests, nous utilisons une autre approche. Pour réaliser cette étude, nous nous sommes basés sur les éléments produits par les testeurs durant le projet de test : les configurations sélectionnées et les verdicts des tests exécutés. Basé sur ces éléments nous exécutons le processus suivant : pour chaque configuration mise en place par les testeurs durant le projet de test, nous regardons si celle-ci est bien présente dans les configurations fournies par Pacogen. Si celle-ci est couverte par Pacogen, nous en déduisons que dans un processus basé sur Pacogen, celle-ci aurait été testée et les défaut auraient été détectés.

L'analyse des configurations utilisées par les testeurs lors du projet BIEW montre que toutes ces configurations sont bien couvertes par les configurations obtenues après l'utilisation de Pacogen. Nous pouvons en déduire que l'utilisation de la technique *pairwise*, dans le cadre du projet BIEW ne provoque pas une diminution des défauts détectés. Néanmoins, il nous est impossible de savoir si l'utilisation du critère *pairwise* aurait pu permettre de détecter d'autres défauts. Pour obtenir une telle information, il aurait été nécessaire de réexecuter tous les cas de test dans toutes les configurations extraites.

8.2.4 Intégration et limite

L'intégration d'une approche basée sur *pairwise* dans le processus de test actuel nécessite certaines évolutions. Dans le processus de test actuel, les matrices de variabilité permettent de maintenir une traçabilité entre cas de test, exigences et configurations de test. L'adoption d'une approche *pairwise* permettrait de s'affranchir de l'utilisation de ces matrices : le choix des configurations serait fait en amont et il ne serait plus nécessaire d'utiliser ces matrices de variabilité. Les configurations de test seraient identifiées au plus tôt et le temps associé à la réalisation des matrices de variabilités serait économisé.

Cependant en supprimant ces matrices, le lien entre cas de tests et configurations de test est perdu. Il est donc nécessaire de trouver une moyen de maintenir la traçabilité entre les cas de tests de les *features* composant les configurations de façon à pouvoir rapidement identifier quels cas de test exécuter dans chaque configuration.

8.3 Pairwise et le logiciel Téléprésence

Dans cette section nous présentons les résultats de l'application d'une approche de test basée sur l'utilisation de Pacogen dans le contexte du logiciel TéléPrésence de Cisco.

8.3.1 Problématiques

La validation du logiciel TéléPrésence par les ingénieurs de Cisco est source de multiples défis. Nous les présentons dans cette section.

8.3.1.1 Des activités de test de tailles variables, bornées dans le temps

Le logiciel TéléPrésence est en constante évolution, offrant ainsi aux utilisateurs de nouvelles fonctionnalités. Ces évolutions logicielles sont souvent de tailles variables. Le principal problème des équipes de test est de suivre le rythme imposé pour valider en temps et en heure les différentes évolutions logicielles.

8.3.1.2 Des configurations de test invalides

TéléPrésence est une application configurable : un produit résulte d'une sélection de *features*. Des contraintes lient ces *features*. Une partie des activités de test résident dans la sélection de ces *features* afin d'obtenir des configurations testables. Il est apparu que les testeurs avaient parfois du mal à sélectionner des configurations respectant les contraintes entre les *features*. La sélection de configurations invalides provoque alors des défaillances lors des tests. Il est difficile pour les ingénieurs d'identifier que les défaillances ont pour origine une mauvaise configuration, ce qui peut engendrer des retards dans les projets de test.

8.3.1.3 Une couverture de test insuffisante

La sélection des configurations de test résulte d'une procédure manuelle. Cette procédure n'est basée sur aucun critère objectif. Cette absence de critères a deux conséquences : une non-formalisation du processus de test, ce qui peut être source de difficultés en cas de départs dans les équipes de test, et un déséquilibre dans les fonctionnalités testées. Certaines parties sont trop testées, car trop souvent sélectionnées par les testeurs tandis que d'autres parties de l'application ne le sont pas assez.

8.3.2 Application

En réponse à ces défis, nous étudions l'impact qu'aurait l'utilisation des modèles de *features* et de l'utilisation de Pairwise sur les activités de test du logiciel TéléPrésence.

8.3.2.1 Réalisation du modèle de *features*

Le temps passé pour réaliser le modèle de *features* dans sa première version a été estimé à environ 54 heures de travail. Une fois la première version réalisée, l'ajout de nouvelles *features* est beaucoup plus rapide et prend tout au plus quelques heures. Le modèle réalisé est composé de 169 *features* et peut aboutir à la création de plus de 10^9 configurations.

8.3.2.2 Génération des configurations *pairwise*

Pour réaliser l'extraction des configurations de test, nous avons utilisé Pacogen. Nous l'avons paramétré de la façon suivante : *first-fail*, avec une matrice de taille 100, la fonction de minimisation numéro 1 et un temps de minimisation de 13 heures. L'utilisation du critère *pairwise* sur le modèle de *features* de TéléPrésence permet d'extraire 35 configurations de test. Ce nombre de configurations est bien plus faible que celui obtenu avec le logiciel BIEW, car les contraintes *xor* sont beaucoup moins présentes dans ce modèle : 51 *features* sont obligatoires, 91 optionnelles et seulement 8 groupes *xor*.

8.3.3 Pertinence de l'approche

Pour mesurer la pertinence de l'approche dans le cadre du projet de test TéléPrésence, nous avons étudié l'impact de l'utilisation des modèles de *features* et de la sélection de configurations de test qui respectent le critère pairwise.

8.3.3.1 Dimension quantitative

L'utilisation de Pacogen dans le cadre du projet TéléPrésence a été la source de plusieurs améliorations :

Temps de sélection des configurations réduit
L'utilisation de Pacogen a permis en 13 heures d'extraire 35 configurations de test valides. Lors d'un processus manuel, le temps nécessaire pour la sélection des configurations de test par les ingénieurs est de 87 heures. L'utilisation de Pacogen permet de réduire de 85 % le temps de sélection des configurations. De plus, comme le processus est automatisé, nous pouvons considérer que le besoin effectif en activité humaine n'est que de deux heures (ces activités consistent à lancer Pacogen et à récupérer les configurations), ce qui permet d'économiser 87 heures de travail, soit plus deux semaines entières sur une base de 40 heures de travail par semaine.

Plus petit nombre de configurations de test

Dans le cadre de la validation du logiciel TéléPrésence, les ingénieurs ont sélectionné manuellement 87 configurations de tests. L'utilisation de Pacogen permet la sélection de 35 configurations, soit une réduction de 59.7%.

Couverture de toutes les paires

Une analyse des configurations de test sélectionnées manuellement par les ingénieurs montre que ces configurations couvrent 505 paires. Le nombre total de paires possibles dans le modèle de *features* est de 53195. Une approche manuelle ne couvre que 0.1 % des paires. L'utilisation du pairwise permet de choisir des configurations de tests couvrant bien plus d'environnements de test.

8.3.3.2 Dimension qualitative

L'analyse des rapports de test a mis en évidence que 10 % des défauts remontés étaient dus à de mauvaises configurations. L'utilisation de Pacogen permet de supprimer ces configurations erronées et donc de réduire le nombre de défauts détectés dus à une mauvaise configuration. Cette approche permet d'économiser un temps certain, car les testeurs peuvent être sûrs que le défaut détecté vient bien du logiciel et non pas d'une mauvaise configuration. Lors de discussions avec les différents acteurs du projet de test, ceux-ci nous ont fait part de leurs ressentis par rapport à l'utilisation d'une telle approche. Selon ces personnes, les configurations sélectionnées sont pertinentes. De plus,l'idée de systématiser le processus de sélection des configurations est attrayante pour les testeurs car elle permet une formalisation des activités de test.

8.3.4 Conclusion

L'application du critère *pairwise* dans le cadre du projet TéléPrésence a de multiples bénéfices sur ce projet de test :

- une formalisation du processus de test basé sur un critère précis
- une économie de deux semaines de travail d'ingénieur
- une sélection de configurations valides qui permet d'éliminer 11 % des défauts (ceux ayant pour origine une erreur dans la configuration)
- une réduction du nombre de configurations à tester de 85 %
- Une meilleure couverture des configurations grâce à une couverture de 100 % des paires.

8.4 Résumé

Bien que ces deux projets de test visent à valider deux applications différentes, les problématiques à traiter restent communes : une grande diversité des configurations de tests, des problématiques de modélisation des configurations de test... La table 8.2 présente les chiffres clés de ces deux projets.

Projet de test	BIEW	TéléPrésence
Nombre de *features*	64	169
Nombre de configurations	2.975.520	$> 10^9$
Projet de test orignal		
Nombre de configurations de test	390	87
Pourcentage de paires couvertes	4 %	0.1 %
Approche Pairwise		
Nombre de configurations de test	254	35
Pourcentage de paires couvertes	100 %	100 %

FIGURE 8.2 – Résumé des projets de test des deux cas industriels

Dans les deux cas, les organisations font face à une combinatoire élevée, le nombre potentiel d'environnements de test est supérieur à 1 million ce qui empêche l'utilisation d'une approche exhaustive. Face à cette combinatoire, les testeurs ont fait le choix de sélectionner manuellement les configurations de test. Cette sélection se fait à l'aide de critères non objectifs. Cette approche a pour conséquence un choix de configurations de test peu représentatif de l'ensemble des configurations possibles : seulement 4 % des paires sont couvertes dans le cadre du projet de test BIEW et 0.1 % pour le projet de test TéléPrésence. L'application de la technique *pairwise* dans les deux projets de test permet d'améliorer les processus de test sur différents points :

- une réduction des configurations à tester
- un meilleur échantillonnage de l'espace des configurations de test avec 100 % des paires couvertes.

Dans le cadre du projet BIEW, une l'étude du projet de test nous permet de conclure que les configurations qui respectent le critère *pairwise* sont en mesure de mettre en évidence tous les défauts rencontrés durant le projet de test. Il nous a été impossible de réaliser une étude similaire sur le projet de test du logiciel TéléPrésence car nous n'avons pas eu accès aux données nécessaires.

8.5 Discussion

Dans ce chapitre nous étudions l'application d'une méthodologie de test basée sur Pacogen dans le cadre de deux projets industriels BIEW et TéléPrésence. Nous montrons que la sélection de configurations de test *pairwise* apporte de multiples bénéfices dans les projets de test :

- une rationalisation de la variabilité grâce à l'utilisation de modèles de *features*
- une diminution du nombre de configurations à tester
- des configurations plus représentatives de l'espace des configurations possibles
- une qualité (nombre de défauts détectés) préservée.

Dans cette section nous discutons de ces bénéfices et nous les pondérons en mettant en avant leurs limites et les problèmes qu'ils peuvent lever.

L'utilisation des modèles de *features* pour les deux projets de test apporte plusieurs bénéfices. Dans le cadre du projet de test BIEW, le modèle de *features* permet de quantifier le nombre d'environnements de test et de formaliser les relations entre les différentes *features* comme les incompatibilités. Dans le cadre du projet de test TéléPrésence, l'utilisation du modèle de *features* permet d'éviter de mettre en place des configurations de test erronées. Il est difficile pour les ingénieurs chargés des activités de test de savoir si une erreur a pour origine une mauvaise configuration. En capturant ces mauvaises configurations, via l'utilisation d'un modèle de *features*, ce problème disparaît. Néanmoins, l'utilisation des modèles de *features* dans des projets de test industriels nécessite plusieurs investissements : il faut enseigner le formalisme aux ingénieurs, mettre en place les outils permettant de gérer ces modèles de *features* (comme pure variant [1] ou familiar [2]) puis éliciter cette variabilité, c'est-à-dire créer les modèles de *features*. Toutes ces étapes ont un coût qu'il faut prendre en compte lors de la mise en place de cette approche. Pour le projet de test BIEW, la création du modèle de *features* a été simple, car la variabilité était explicitée dans un fichier et les personnes chargées du projet étaient disponibles pour répondre aux questions. Dans le cas du projet de test de TéléPrésence, la réalisation du modèle de *features* a été plus complexe : le produit existe depuis plusieurs années et les points de variations sont bien plus importants. Il a fallu 57 heures pour réaliser la première version, puis les mises à jour du modèle ont nécessité entre 9 et 15 heures.

La diminution des configurations à tester peut être un objectif pertinent lorsque le coût de test d'une configuration est élevé ; si le temps pour tester est élevé ou si la phase de test nécessite une mobilisation de ressources humaines ou matérielles importante. Cependant, l'intérêt d'avoir un nombre de configurations minimum peut diminuer lorsque que le coût de test d'une configuration est très faible voire quasi nul. Dans ce cas, l'intérêt de minimiser ce nombre de configurations de test diminue : économiser quelques configurations de test n'a plus vraiment d'intérêt quand le coût de test d'une configuration est quasi nul.

La sélection des configurations de tests à l'aide du critère *pairwise* permet la sélection systématique de configurations plus hétérogènes et plus représentatives de l'espace des configurations possibles. Comme la sélection est automatisée selon un critère précis, les configurations sélectionnées ne souffrent pas d'un biais résultant d'un processus de sélection manuel. Il est possible que les configurations choisies lors d'un processus manuel ne couvrent pas toutes les *features* du système ou que ces configurations soient très similaires : un ensemble de *features* identiques se retrouve dans toutes les configurations et il y a seulement des variations mineures entre les configurations (peu de changement en terme de *features* sélectionnées ou non sélectionnées entre deux

1. http ://www.pure-systems.com/pure_variants.49.0.html
2. http ://familiar-project.github.io/

configurations). Les testeurs du projet Téléprésence ont souffert de ce biais en sélectionnant des configurations trop similaires, ce qui a eu pour conséquence un déséquilibre entre les *features* testées : certaines *features* sont trop testées tandis que d'autres ne le sont pas assez. Avoir 100 % des paires couvertes permet d'avoir bon échantillonnage de l'espace de test.

Il est légitime de s'interroger sur la pertinence du critère toutes les paires couvertes dans le cadre d'un projet de test industriel. De multiples études scientifiques se sont attachées à montrer que les techniques *pairwise* sont efficaces pour la sélection de configurations de test, néanmoins ces études ne laissent pas la parole aux industriels du test. En fonction de la criticité de l'application et du budget alloué, le critère *pairwise* n'est pas toujours pertinent. L'utilisation du *pairwise* pour tester un produit qui a potentiellement plusieurs millions de configurations, mais dont seulement une dizaine seront réellement utilisées est discutable. Il est peut-être plus intéressant de ne tester que ces configurations identifiées. Dans d'autres projets de test, les ingénieurs en charge de la validation pourraient ne vouloir couvrir que le critère 1-*wise*, c'est à dire extraire un ensemble de configurations ou toutes les *features* sont au moins présente une fois. La couverture de toutes les paires est donc un critère de test qui permet de montrer la présence de défauts, mais n'est pas forcément une solution de test à adopter de façon systématique.

L'étude approfondie du projet de test BIEW permet de montrer que la **qualité logicielle est conservée** : aucun des défauts détectés dans le projet de test original n'aurait été oublié avec une approche *pairwise*. Cependant pour s'assurer de la pertinence de l'utilisation de Pacogen dans un cas industriel, il aurait été nécessaire de réexecuter l'ensemble de cas de test dans les configurations sélectionnées, puis de comparer les temps passés sur les deux projets et les défauts détectés. Il nous était impossible de réexécuter ces cas de test, car la charge de travail nécessaire était trop importante. Il nous est donc difficile d'évaluer complètement l'impact d'une approche de test combinatoire dans un projet industriel. Certaines dimensions restent à évaluer : adaptation des processus de test, nombre de défauts détectés, coût d'apprentissage et coût des activités de test.

L'utilisation d'une approche *pairwise* dans l'industrie lève aussi un problème pour le projet BIEW. Dans le projet BIEW, les configurations sélectionnées sont des configurations de test. Une fois les configurations établies, il est nécessaire d'exécuter les cas de test. Dans le projet de test original, les **matrices de variabilité** permettaient d'associer à des éléments de l'environnement des cas de test à exécuter. En mettant en place une approche de sélection automatique des configurations de test, le lien cas de test et configurations de test est perdu. Il faut donc fournir au testeur des moyens de rétablir cette traçabilité. De cette façon pour chaque configuration de test sélectionnée par Pacogen, un ensemble de cas de tests à exécuter y est associé.

Troisième partie

Conclusion et Perspectives

Chapitre 9

Conclusion et Perspectives

9.1 Conclusion

La gestion de la variabilité lors des activités de test est une problématique courante dans les organisations. Les entreprises Orange et Cisco ont toutes les deux rencontré des difficultés pour gérer cette variabilité. L'application BIEW développée par Orange devait être validée sur près de 2 million d'environnements d'exécution différents. Le logiciel configurable TéléPrésence de Ciscon, lui, présentait plus de 1 milliard de configurations possibles. Ces deux organisations ont donc dû mettre en place des méthodes de test pour gérer ces nombreuses configurations de test. Dans cette thèse nous avons proposé une approche pour traiter les problèmes que cause la variabilité pour les activités de validation des systèmes configurables. Dans ce document, nous avons présenté trois contributions qui permettent de répondre à trois problématiques.

La première problématique rencontrée par les industriels réside dans la représentation de la variabilité. Dans cette thèse, nous avons proposé une implémentation de la sémantique des modèles de *features* à l'aide des techniques de programmation par contraintes. Cette implémentation utilise un ensemble de contraintes qui capturent la sémantique des modèles de *feature*s. Ainsi, cette représentation à l'aide de contraintes permet de raisonner sur les modèles de *feature*s et d'effectuer plusieurs analyses de l'état de l'art [BSRC10] comme compter le nombre de configurations, identifier les *features* centrales... Plusieurs transformations ont été réalisées pour que le moteur à contraintes puisse utiliser des modèles de *features* réalisés avec d'autres outils. Le moteur à contraintes qui a été développé peut lire des modèles de *features* instances du métamodèle proposé par Perrouin & Al. [PKGJ08] ou des modèles issus du dépôt en ligne de modèle de *features* SPLOT [MBC09].

Par la suite, nous avons traité le problème de la sélection des configurations à tester. Les modèles de *features* peuvent représenter plusieurs millions de configurations. Il est en général difficile de toutes les tester à cause de leur grand nombre. Nous avons proposé une technique qui permet de sélectionner un ensemble de configurations à tester

et qui respecte le critère *pairwise*. Cette approche repose sur l'utilisation d'une nouvelle contrainte appelée *pairwise*. La sélection de configurations de test qui respectent le critère *pairwise* est un domaine de recherche actuel. Plusieurs approches ont été proposées pour traiter ce problème. Les travaux de recherches de ces dernières années ont abouti à la création de deux outils : SPLCAtool [JHF11] et MosoPolite [OMR10]. L'approche que nous avons proposé se base sur l'utilisation des techniques de programmation par contraintes et se distingue des approches existantes en deux points. Notre approche est reproductible et permet de minimiser le nombre de configuration obtenu afin d'atteindre le plus petit nombre de configuration possible. L'approche à contraintes que nous avons proposé est paramétrable : il est possible d'utiliser les informations structurelles du modèle de *features*, de changer les stratégies d'énumérations et les fonctions de minimisation. Ces paramètres permettent à Pacogen de retourner des solutions de petites tailles. Pour déterminer la combinaison de paramètres optimale nous évaluons les différentes combinaisons de paramètres sur 224 modèles de *features* issus du dépôt en ligne SPLOT. Une fois la combinaison de paramètre optimale déterminée nous comparons notre approche aux deux outils de l'état de l'art. Les expériences sont réalisées sur 224 modèles de *features*. La comparaison des tailles des configurations retournées par notre approche et les tailles de configurations retournée par SPLCATool montre que dans 98% des cas notre approche retourne un nombre de configurations inférieur ou égal aux résultats retournés par SPLCATool. Pour les 3% restants, notre approche, dans le temps impartis, n'a pas pu rendre des configurations de tailles équivalentes à celles de SPLCATool. La comparaison avec MosoPolite montre que l'approche à contraintes retourne systématiquement un nombre de configurations plus petit. L'ensemble des développement réalisés est embarqué dans un produit logiciel appelé Pacogen.

Finalement, nous avons étudié l'applicabilité d'une telle technique de test dans le cas des deux projets industriels. Nous avons étudié l'impact causé par l'utilisation des modèles de *features* et des techniques de test combinatoire. Nous avons montré qu'une stratégie de test basée sur l'utilisation de modèle de *features* et l'extraction de configurations de tests qui respectent le critère *Pairwise* est pertinente. Pour les deux projets de test industriels : le projet de test du logiciel BIEW et le projet de test du logiciel Téléprésence, nous avons évalué le temps nécessaire pour réaliser le modèle de *features*. Nous avons montré qu'il est possible de réaliser un modèle de *features* en quelques heures, et, qu'une fois réalisés, ces modèles permettent un gain de temps lors des activités de configurations et permettent de capitaliser la connaissance des testeurs. La techniques de sélection des configurations de test *pairwise* permet de passer de près de 3 000 000 de configurations à tester de façon exhaustive à 254 pour le projet BIEW, et de plus de 10^9 à 35 pour le projet de test du logiciel TéléPresence. Nous avons comparé les nombres de configurations sélectionnés par Pacogen aux nombres de configurations de test utilisées dans les projets de tests. Pour le projet de test BIEW, le nombre de configurations à tester passe de 390 dans le processus actuel à 254, tandis que dans le cas du projet de test TéléPrésence le nombre de configurations de test passe de 87 à 35. Nous avons montré que le nombre de configurations sélectionné avec la technique *pairwise* est plus faible. L'étude approfondie des configurations de test sélectionnées avec

le critère *pairwise* montre que si celles-ci avaient été utilisées dans un des projets de test, elles auraient détecté tous les défauts qui ont été trouvés par les testeurs. De plus, les discussions avec les différents acteurs des projets de test confirment que l'approche basée sur les modèles de *features* est pertinente.

9.2 Perspectives

De nombreux travaux restent à mener dans le domaine de la sélection de configurations de test pour les systèmes configurables. Dans cette section sont présentées les perspectives de recherche identifiées.

9.2.1 N-wise

Les travaux présentés dans cette thèse se sont concentrés sur la sélection de configurations de test qui respectent le critère *pairwise*. Néanmoins, ce critère de sélection de configurations de test peut être généralisé pour proposer plusieurs degrés de couverture. Un de nos objectifs est de généraliser les algorithmes présentés afin de pouvoir proposer à l'utilisateur la sélection de configurations de test qui respectent le critère N-wise.

Une des conséquences de l'augmentation du degré de test est l'augmentation du nombre de configurations de test. Plus le degré de test est élevé, plus les interactions entre *features* testées sont nombreuses, et plus le nombre de configurations de test est important. Pour mieux maîtriser cette augmentation du nombre de configurations, nous envisageons d'offrir au testeur la capacité de sélectionner des configurations de test qui mélangent différents degrés de test. Ainsi, le testeur pourra définir des configurations de test où toutes les *features* seront testées avec le degré *pairwise*, et où un sous-ensemble de *features* sera testé en 3-wise. Cette capacité à mélanger les critères va permettre de prendre en compte les connaissances qu'ont les testeurs du système en testant plus exhaustivement les parties critiques de l'application et plus légèrement les autres.

9.2.2 Complétion des configurations à tester

Il n'est jamais facile de faire évoluer un processus bien établi dans une organisation. Il faut convaincre les personnes, les former et adapter les outils.

L'évolution d'un processus de test va aussi faire face aux mêmes difficultés. La transition vers un nouveau processus de test doit se faire progressivement, sans remettre totalement en cause les processus établi. Une des solutions possible est de proposer une approche qui prend en compte les configurations de test courantes, et qui va les compléter pour atteindre le critère *pairwise* visé. En prenant en compte les configurations existantes, il est possible de proposer un ensemble de configurations de test complémentaires aux configurations existantes afin d'atteindre le critère *pairwise*.

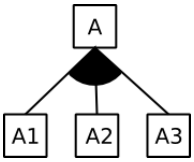


FIGURE 9.1 – Modèle de *features* Simple

9.2.3 Gestion des attributs

Les discussions avec différents partenaires industriels ont mis en évidence un besoin d'ajouter des attributs sur les *features* du modèle de *features*. L'ajout d'attributs sur les *features* permet ainsi de quantifier le prix ou le poids d'une configuration. Des travaux de recherches [BTRC05, CBH11] ont déjà été réalisés pour utiliser ces attributs. Nous envisageons d'étendre le modèle à contraintes pour prendre en compte ces attributs, et les utiliser lors de la sélection de configurations de test qui respectent le critère *pairwise*. Une fois le modèle à contraintes étendu nous envisageons d'ajouter des attributs aux modèles de *features* des cas industriels BIEW et Cisco. Dans le projet de test BIEW nous comptons attacher 2 types d'attributs : la difficulté à tester le composant et sa probabilité de déclencher un défaut. La difficulté à tester le composant sera définie par les testeurs et permettra de formaliser leurs connaissances du système. Ainsi, les testeurs pourront identifier les composants difficiles à installer ou dont les pilotes ne sont pas toujours à jours. La modélisation de la probabilité de déclencher un défaut sera définie par le rapport du nombre de tests en échec par le nombre de tests total pour modéliser la probabilité d'erreur. Grâce à ce type d'attribut les testeurs pourront sélectionner les configurations les plus à-même d'être sources de défauts.

La programmation par contrainte sur des variables à domaines finis se prête à ce genre de raisonnement. Un des défis levé par l'introduction des attributs dans les modèles de *features* réside dans la nécessité de préciser la sémantique à appliquer pour raisonner sur ces modèles attribués. Ainsi, en fonction de la nature de l'attribut la façon de raisonner peut changer.

La figure 9.1 présente un modèle de feature simple, composé d'un père et de trois *features* filles. Si toutes les *features* possèdent un attribut poids P, alors le poids de la configuration P_A se calcule de cette façon :

$$P_A = \mathbf{1}_{\{A_1>0\}} \times P_{A_1} + \mathbf{1}_{\{A_2>0\}} \times P_{A_2} + \mathbf{1}_{\{A_3>0\}} \times P_{A_3} \tag{9.1}$$

Où $\mathbf{1}_{\{A_i>0\}}$ est la fonction indicatrice telle que $\mathbf{1}_{\{A_i>0\}} = 0$ si Ai n'est pas sélectionnée dans la configuration et $\mathbf{1}_{\{A_i>0\}} = 1$ si A_i est sélectionnée.

Si les attributs sont des probabilités d'erreurs alors la probabilité P_A que le système est un comportement fautif est égale à :

$$P_A = 1 - ((1 - \mathbf{1}_{\{A_1>0\}} \times P_{A_1}) \times (1 - \mathbf{1}_{\{A_2>0\}} \times P_{A_2}) \times (1 - \mathbf{1}_{\{A_3>0\}} \times P_{A_3})) \tag{9.2}$$

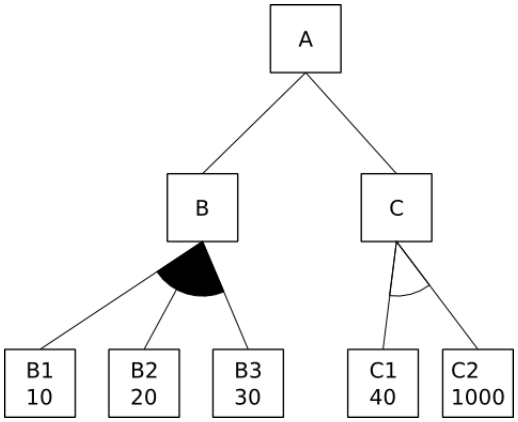


FIGURE 9.2 – Modèle de *features* attribué

Où P_{A_i}, la probabilité d'erreur d'une *feature*, et
$((1 - \mathbf{1}_{\{A_1>0\}} \times P_{A_1}) \times (1 - \mathbf{1}_{\{A_2>0\}} \times P_{A_2}) \times (1 - \mathbf{1}_{\{A_3>0\}} \times P_{A_3}))$ la probabilité que le système fonctionne correctement.

En fonction de la nature des attributs, les façons de raisonner diffèrent. Le système à contraintes doit pouvoir gérer ces différents attributs. Nous avons présenté deux types d'attributs : des attributs de poids et des attributs représentant des probabilités d'erreur. Il est possible que des attributs d'autres natures nécessitent une autre sémantique. Pour pouvoir prendre en compte ces nouveaux types d'attributs, il faut que le modèle à contraintes développé dispose de mécanismes d'extensions.

Le raisonnement sur les modèles attribués est plus complexe que sur les modèles de *features* sans attributs. Avec un modèle de *features* attribué le raisonnement peut se faire dans deux sens. À partir d'une configuration (sélection de *features*), il est possible d'obtenir son coût global ou à partir d'une contrainte sur le coût et effectuer des choix sur les *features* du modèle. L'exemple ci dessous illustre ces deux mécanismes de raisonnements. Le modèle de feature présenté figure 9.2 est un modèle attribué, les *features* B1,B2,B3,C1 et C2, possèdent toutes un coût (le numéro situé sous le nom de la *feature*). On suppose que le coût total est égal à la somme des coûts des *features* sélectionnées.

Le premier raisonnement possible avec ce modèle de feature consiste à sélectionner un ensemble de *features*. Si les *features* A,B,C et $C1$ sont sélectionnées, le système à contraintes permet de déduire que le coût est compris entre deux valeurs : $Cout \in [50, 70]$. Si la configuration est complète, alors le système à contraintes doit donner le coût de la configuration. Par exemple si les *features* A, B, B1, C et C1 sont sélectionnées, la configuration aura alors un coût total de $10 + 40 = 50$.
Le deuxième mécanisme de raisonnement doit permettre d'effecteur le raisonnement suivant : à partir d'une contrainte sur le coût, le modèle à contraintes doit être capable de

sélectionner ou de dé-sélectionner des *features* automatiquement grâce aux mécanismes de propagation des contraintes. Par exemple, si la contrainte suivante est posée : $Cout < 80$ alors le système à contraintes doit déduire que la *feature* $C1$ est sélectionnée et que la *feature* $C2$ ne l'est pas. En effet, la *feature* C doit être présente dans la configuration (C étant relié à A pour un lien and). C étant sélectionnée, soit la *feature* $C1$ soit la *feature* $C2$ doit être sélectionnée. Seule la *feature* $C1$ respecte la contrainte de coût, elle doit donc être sélectionnée.

Bibliographie

[Ach11] Mathieu Acher. *Managing Multiple Feature Models : Foundations, Language and Applications*. PhD thesis, PhD thesis, University of Nice Sophia Antipolis, 2011.

[ACLF10] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. Comparing approaches to implement feature model composition. In *Modelling Foundations and Applications*, pages 3–19. Springer, 2010.

[ACLF11] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B France. Slicing feature models. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 424–427. IEEE Computer Society, 2011.

[ACLF13] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. Familiar : A domain-specific language for large scale management of feature models. *Science of Computer Programming (SCP) Special issue on programming languages*, page 55, 2013. (in press).

[AM03] Memon AM and Soffa ML. Regression testing of guis. In *European Software Engineering Conference (ESEC)*, page 118–127, 2003.

[Bat05] Don Batory. Feature models, grammars, and propositional formula. *Software Product Lines, Lecture Notes in Computer Science*, 3714(3) :7–20, 2005.

[BCT04] D Benavides, Ruiz A Cortés, and P Trinidad. Coping with automatic reasoning on software product lines. 2004.

[BRCT05] David Benavides, Antonio Ruiz-Cortes, and Pablo Trinidad. Using constraint programming to reason on feature models. In *The Seventeenth International Conference on Software Engineering and Knowledge Engineering, SEKE*, volume 2005, pages 677–682, 2005.

[Bry86] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8) :677–691, 1986.

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models : A detailed literature review. *Information Systems*, (35) :615–636, 2010.

[BSTRC06] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. Using java csp solvers in the automated analyses of feature models. In *Ge-*

nerative and Transformational Techniques in Software Engineering, pages 399–408. Springer, 2006.

[BTRC05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. In *Advanced Information Systems Engineering*, pages 491–503. Springer, 2005.

[BV05] K.Z. Bell and M.A. Vouk. On effectiveness of pairwise methodology for testing network-centric software. In *Int. Conf. on Information and Communications Technology*, pages 221–235, 2005.

[CBH11] Andreas Classen, Quentin Boucher, and Patrick Heymans. A text-based approach to feature modelling : Syntax and semantics of tvl. *Science of Computer Programming*, 76(12) :1130–1143, 2011.

[CCL03] Myra B Cohen, Charles J Colbourn, and Alan CH Ling. Augmenting simulated annealing to build interaction test suites. In *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, pages 394–405. IEEE, 2003.

[CCT04] Charles J Colbourn, Myra B Cohen, and Renée C Turban. A deterministic density algorithm for pairwise interaction coverage. In *Proc. of the IASTED Intl. Conference on Software Engineering*, volume 41, pages 242–252. Citeseer, 2004.

[CDG97] Fredman ML Cohen DM, Dalal SR and Patton GC. The aetg system : An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7) :437–444, 1997.

[CE00] K. Czarnecki and U.W. Eisenecker. *Generative Programming : Methods, Techniques, and Applications*. Addison–Wesley, 2000.

[CGMC03] Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. Constructing test suites for interaction testing. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 38–48, Washington, DC, USA, 2003. IEEE Computer Society.

[Chv79] Vasek Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, 4(3) :233–235, 1979.

[COC97] M. Carlsson, G. Ottosson, and B. Carlson. An Open-Ended Finite Domain Constraint Solver. In *Proc. Programming Languages : Implementations, Logics, and Programs*, 1997.

[Cze06] Jacek Czerwonka. Pairwise testing in the real world : Practical extensions to test-case scenarios. In *Proceedings of 24th Pacific Northwest Software Quality Conference, Citeseer*, pages 419–430, 2006.

[GCD09] Brady J Garvin, Myra B Cohen, and Matthew B Dwyer. An improved meta-heuristic search for constrained interaction testing. In *Search Based Software Engineering, 2009 1st International Symposium on*, pages 13–22. IEEE, 2009.

[GCD11] BradyJ. Garvin, MyraB. Cohen, and MatthewB. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering*, 16(1) :61–102, 2011.

[GHB12] Arnaud Gotlieb, Aymeric Hervieu, and Benoit Baudry. Minimum pairwise coverage using constraint programming techniques. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 773–774. IEEE, 2012.

[GOA05] Mats Grindal, Jeff Offutt, and Sten F Andler. Combination testing strategies : a survey. *Software Testing, Verification and Reliability*, 15(3) :167–199, 2005.

[HBG11] Aymeric Hervieu, Benoit Baudry, and Arnaud Gotlieb. Pacogen : Automatic generation of pairwise test configurations from feature models. In *Proc. of Int. Symp. on Soft. Reliability Engineering (ISSRE'11)*, Hiroshima, Japon, 2011.

[HBG12] Aymeric Hervieu, Benoit Baudry, and Arnaud Gotlieb. Managing execution environment variability during software testing : An industrial experience. In Brian Nielsen and Carsten Weise, editors, *Testing Software and Systems - 24th IFIP WG 6.1 International Conference, ICTSS 2012*, volume 7641 of *Lecture Notes in Computer Science*, pages 24–38. Springer, 2012.

[HPSS06] Brahim Hnich, Steven D Prestwich, Evgeny Selensky, and Barbara M Smith. Constraint models for the covering test problem. *Constraints*, 11(2-3) :199–219, 2006.

[ISO05] BSEN ISO. Iec 17025 : 2005 general requirements for the competence of testing and calibration laboratories. *International Standards Organisation, Geneva*, 2005.

[JHF11] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. Properties of realistic feature models make combinatorial testing of product lines feasible. In *Model Driven Engineering Languages and Systems*, pages 638–652. Springer, 2011.

[JHF12a] Martin Fagereng Johansen, Oystein 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*, SPLC '12, pages 46–55, New York, NY, USA, 2012. ACM.

[JHF+12b] Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Erik Carlson, Jan Endresen, and Tormod Wien. A technique for agile and automatic interaction testing for product lines. In *Testing Software and Systems*, pages 39–54. Springer, 2012.

[JRL08] Narendra Jussien, G Rochart, and X Lorca. The choco constraint programming solver. In *CPAIOR'08 workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, 2008.

[Kat73] G.O.H. Katona. Two applications (for search theory and truth functions) of sperner type theorems. *Periodica Mathematica Hungarica*, 3(1-2) :19–26, 1973.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, November 1990.

[KOD10a] Ahmet Serkan Karatas, Halit Oguztüzün, and Ali H. Dogru. Global constraints on feature models. In *Principles and Practice of Constraint Programming - CP 2010 - CP 2010, St. Andrews, Scotland, UK, Sep. 6-10, 2010. LNCS 6308*, pages 537–551, 2010.

[KOD10b] Ahmet Serkan Karatas, Halit Oguztüzün, and Ali H. Dogru. Mapping extended feature models to constraint logic programming over finite domains. In *Software Product Lines : Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. - LNCS 6287*, pages 286–299, 2010.

[KS73] D. Kleitman and J. Spencer. Families of k-independent sets. *Discrete Mathematics*, 6 :255–262, 1973.

[KTS+09] Christian Kastner, Thomas Thum, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. Featureide : A tool framework for feature-oriented software development. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 611–614. IEEE, 2009.

[KW04] D. Richard Kuhn and Dolores D. Wallace. Software fault interactions and implications for software testing. *IEEE Trans. on Software Eng.*, 30(6) :418 – 421, 2004.

[LBP10] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7 :59–64, 2010.

[LKK+08] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. Ipog/ipog-d : efficient test generation for multi-way combinatorial testing. *Softw. Test., Verif. Reliab.*, 18(3) :125–148, 2008.

[Man02] Mike Mannion. Using first-order logic for product line model validation. In *Software Product Lines*, pages 176–187. Springer, 2002.

[MBC09] Marcilio Mendonca, Moises Branco, and Donald Cowan. Splot : software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 761–762. ACM, 2009.

[MC04] Mike Mannion and Javier Camara. Theorem proving for product line model verification. In *Software Product-Family Engineering*, pages 211–224. Springer, 2004.

[MFJF11] Oystein Haugen Martin Fagereng Johansen and Franck Fleurey. Properties of realistic feature models make combinatorial testing of product lines feasible. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, pages 638–652, 2011.

[MGSH13] Dusica Marijan, Arnaud Gotlieb, Sagar Sen, and Aymeric Hervieu. Practical pairwise testing for software product lines. In *Proceedings of the 17th International Software Product Line Conference*, pages 227–235. ACM, 2013.

[MJPL92] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing conflicts : a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1–3) :161 – 205, 1992.

[MP07] Andreas Metzger and Klaus Pohl. Variability management in software product line engineering. In *Companion to the proceedings of the 29th International Conference on Software Engineering*, ICSE COMPANION '07, pages 186–187, Washington, DC, USA, 2007. IEEE Computer Society.

[MSD+12a] Raúl Mazo, Camille Salinesi, Daniel Diaz, Olfa Djebbi, and Alberto Lora-Michiels. Constraints : The heart of domain and application engineering in the product lines engineering strategy. *International Journal of Information System Modeling and Design (IJISMD)*, 3(2) :33–68, 2012.

[MSD+12b] Raúl Mazo, Camille Salinesi, Daniel Diaz, et al. Variamos : a tool for product line driven systems engineering with a constraint based approach. In *24th International Conference on Advanced Information Systems Engineering (CAiSE Forum'12)*, 2012.

[MWC09] Marcilio Mendonca, Andrzej Wasowski, and Krzysztof Czarnecki. Sat-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*, pages 231–240. Carnegie Mellon University, 2009.

[MWCC08] Marcilio Mendonca, Andrzej Wasowski, Krzysztof Czarnecki, and Donald Cowan. Efficient compilation techniques for large scale feature models. In *Proceedings of the 7th international conference on Generative programming and component engineering*, pages 13–22. ACM, 2008.

[Nur04] Kari J Nurmela. Upper bounds for covering arrays by tabu search. *Discrete applied mathematics*, 138(1) :143–152, 2004.

[OMR10] Sebastian Oster, F. Markert, and P. Ritter. Automated incremental pairwise testing of software product lines. In *Software Product Line Conference (SPLC'10)*, 2010.

[Par76] David Lorge Parnas. On the design and development of program families. *Software Engineering, IEEE Transactions on*, (1) :1–9, 1976.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J van der Linden. *Software product line engineering : foundations, principles, and techniques.* Springer, 2005.

[PH12] Gilles Perrouin and Patrick Heymans. Bypassing the combinatorial explosion : Us-ing similarity to generate and prioritize t-wise test suites for large software product lines. 2012.

[PKGJ08] Gilles Perrouin, Jacques Klein, Nicolas Guelfi, and Jean-Marc Jézéquel. Reconciling automation and flexibility in product derivation. In *Software Product Line Conference (SPLC'08)*, pages 339–348, Limerick, Ireland, September 2008. IEEE Computer Society.

[POS+12] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. Le Traon. Pairwise testing for software product lines : comparison of two approaches. *Software Quality Journal*, 20(3-4) :605–643, 2012.

[PSK+10] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *International Conference on Software Testing (ICST'10)*, Paris, France, 2010.

[SHTB07] P.Y. Schobbens, P. Heymans, J.C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2) :456–479, 2007.

[SLB+10] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. The variability model of the linux kernel. *VaMoS*, 10 :45–51, 2010.

[SLB+11] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. Reverse engineering feature models. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 461–470. IEEE, 2011.

[SM99] Brett Stevens and Eric Mendelsohn. New recursive methods for transversal covers. *Journal of combinatorial designs*, 7(3) :185–203, 1999.

[SMDD10] Camille Salinesi, Raul Mazo, Daniel Diaz, and Olfa Djebbi. Using integer constraint solving in reuse based requirements engineering. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*, pages 243–251. IEEE, 2010.

[Sta01] John Stardom. *Metaheuristics and the search for covering and packing arrays*. PhD thesis, Simon Fraser University, 2001.

[STBF11] Itai Segall, Rachel Tzoref-Brill, and Eitan Farchi. Using binary decision diagrams for combinatorial test design. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 254–264. ACM, 2011.

[TA00] Yu-Wen Tung and Wafa S Aldiwan. Automating test case generation for the new generation mission software system. In *Aerospace Conference Proceedings, 2000 IEEE*, volume 1, pages 431–437. IEEE, 2000.

[TBRC+08] P. Trinidad, D. Benavides, A. Ruiz-Cortés, S. Segura, and A.Jimenez. Fama framework. In *12th Software Product Lines Conference (SPLC)*, 2008.

[TKESug] T. Thum, C. Kastner, S. Erdweg, and N. Siegmund. Abstract features in feature modeling. In *Software Product Line Conference (SPLC), 2011 15th International*, pages 191–200, Aug.

[TL02] Kuo-Chung Tai and Yu Lei. A test generation strategy for pairwise testing. *Software Engineering, IEEE Transactions on*, 28(1) :109–111, 2002.

[TRC09] Pablo Trinidad and Antonio Ruiz-Cortés. Abductive reasoning and automated analysis of feature models : how are they connected. In *Third International Workshop on Variability Modelling of Software-Intensive Systems. Proceedings*, pages 145–153, 2009.

[Tsa93] Edward Tsang. Foundations of constraint satisfaction, 1993.

[VGBS01] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 45–54. IEEE, 2001.

[YC06] Porter A. Yilmaz C, Cohen MB. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, 31(1) :2–34, 2006.

[ZZM04] Wei Zhang, Haiyan Zhao, and Hong Mei. A propositional logic-based method for verification of feature models. In *Formal Methods and Software Engineering*, pages 115–130. Springer, 2004.

Table des figures

Résumé

Aujourd'hui, les éditeurs logiciels ne conçoivent, développent et ne maintiennent plus leur offre logicielle avec comme cible un client unique. Au contraire, les offres logicielles sont conçues pour cibler plusieurs entités. Par conséquent, ces applications doivent s'intégrer dans des environnements différents et s'adapter aux besoins des clients. Ainsi, les produits logiciels développés ne sont plus des programmes uniques, mais des familles de produits [Par76]. Les systèmes configurables facilitent la création de ces familles de produits. Grâce à eux il est possible de créer un produit logiciel en sélectionnant les fonctionnalités qui seront intégrées. Cependant, la validation de ces systèmes est une tâche complexe. Un système configurable peut générer plusieurs millions de configurations possibles. Il ne s'agit donc plus de valider un seul et unique produit, mais un ensemble de produits. Cet important nombre de configurations est un problème pour les personnes chargées de la validation. Nous proposons trois contributions qui visent à mieux répondre aux problématiques liées à la variabilité lors des projets de test.

- une présentation détaillée de deux projets de test industriels faisant face à des problématiques de variabilité issus de deux entreprises : Cisco et Orange.
- une méthode originale basée sur les techniques de programmation par contraintes pour extraire des configurations de test qui respectent le critère Pairwise à partir d'un modèle explicite de la variabilité.
- une comparaison de cette approche par rapport aux techniques de l'état de l'art et une étude de l'application de cette technique de test sur deux projets de tests industriels.

Abstract

Nowadays, software companies develop and maintain their software for several clients. Consequently, these applications have to be integrated in heterogenous context and adapt to the user requriements. All these products are sharing commonalities but also differ in certain point due to business specific constraints. Configurable systems facilitate the creation of these product families. With them it is possible to create a software product by selecting the features that will be integrated, thus, the creation of a product is greatly simplified. However, the validation of these systems is a complex task. A configurable system can generate millions of possible configurations. Thus, validation process doesn't consist in validating a single product but in validating a set of products. This large number of configurations is a problem for those responsible of the validation. In this thesis we propose three contributions that aim to solve issues raised by variability during test projects :

- A detailled presentation of two industrial test projects coping tat variaibility issues
- An original methodology based on constraint programming techniques to select test configurations that respect pairwise criteria from a feature model
- An exhaustive comparison of this approach with the existing approches and a detailled study of the application of a such techniques on the two industrials projects.