REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE MOHAMED BOUDIAF - M'SILA
FACULTE DES MATHEMATIQUES ET DE L'INFORMATIQUE

DEPARTEMENT D'INFORMATIQUE

MEMOIRE de fin d'étude

Présenté pour l'obtention du diplôme de MASTER

Domaine : Mathématiques et Informatique

Filière : Informatique

Spécialité : Outils et Méthodes pour L'Informatique Décisionnelle

Par: DILMI RAMZI

SUJET

Application de la programmation par contraintes au problème d'optimisation du plan du test logiciel

Soutenu publiquement le : 12/06/2017 devant le jury composé de :

Mme. MELIOUH Amel	**Université de M'sila**	**Président**
M. KHETTAF Abdelouahab	**Université de M'sila**	**Rapporteur**
M. BARKAT Abdelbasset	**Université de M'sila**	**Examinateur**

Promotion : 2016 /2017

Dédicace

Je dédie ce travail :

A la plus belle créature que Dieu a créée sur terre, A cette source de tendresse, de patience et de générosité.

A mes parents ***DILMI AHMED SAADI*** et ***DILMI YAMENA***.

A mes frères et sœurs ***HICHAM***, ***YOUNES***, ***KHEIREDDINE***, ***MANAR***, et ***ROKIA***.

Vous vous êtes dépensés pour moi sans compter. En reconnaissance de tous les sacrifices consentis par tous et chacun pour me permettre d'atteindre cette étape de ma vie. Avec toute ma tendresse.

Un grand merci à mon superviseur ***MAA. Khettaf Abdelouahab*** pour leur soutien, et conseille.

Merci beaucoup et spécialement pour ***CH.Bissam*** et mes amis pour leur aide et soutien.

A tous les étudiants de la promotion 2016/2017

Option : **OMID** « **O**utils et **M**éthode de l'informatique **D**écisionnel ».

A tous ceux qui, par un mot, m'ont donné la force de continuer …

DILMI RAMZI

Remerciements

Je rends grâce, avant tout, à Dieu Tout-Puissant pour la volonté, la santé et la patience qu'Il m'a données durant ces années d'études afin que je puisse arriver à cet aboutissement.

Ma reconnaissance la plus profonde et ma gratitude la plus sincère à mon directeur de mémoire Monsieur ***KHETTAF Abdelouahab***, Maitre-assistant A à l'université de Msila, Je le remercie pour ses conseils judicieux, la grande confiance qu'il m'a accordée et pour la vision innovante de la recherche qu'il a toujours su m'inculquer.

Je tiens à remercier tous les membres de la faculté de math et informatique, pour leur aide et leur bonne humeur. J'ai trouvé une ambiance conviviale qui me fut précieux.

Je tiens à remercier tous ceux qui m'ont aidé de près ou de loin afin que je puisse achever ce travail.

Enfin je voudrai remercier ma famille et tous mes amis ainsi qui grâce à leur soutient morale et que tous ceux qui ont contribuent à la réussite de ce mémoire.

TABLE DES MATIERES

INTRODUCTION GENERAL - 1 -

CHAPITRE 01: GENERALITE SUR TEST DE LOGICIES

1. Introduction - 3 -
2. Test de logiciel - 3 -
2.1. Processus de test - 4 -
2.1.1. Sélection - 4 -
2.1.2. Exécution - 4 -
2.1.3. Analyse (Verdict) - 4 -
2.1.4. Évaluation - 4 -
2.2. Critères de test - 5 -
3. Type des tests logiciel - 5 -
3.1. Les tests en « boîte noire » - 6 -
3.1.1. Les avantages - 6 -
3.1.2. Les inconvénients - 7 -
3.2. Les tests en « boite blanche » - 7 -
3.2.1. Les avantages - 8 -
3.2.2. Les inconvénients - 8 -
3.3. Les tests en « boîte grise » - 9 -
4. Résumé pour mieux comprendre - 10 -
5. Conclusion - 11 -

CHAPITRE 02 : PROGRAMMATION PAR CONTRAINTES

1. Introduction - 13 -
2. Problème de satisfaction de contraintes - 13 -
3. Modélisation d’un problème par des contraintes - 13 -
3.1. Instanciation - 14 -
3.2. Affectation - 14 -
3.3. Affectation partielle ou totale - 14 -
3.4. Satisfaire ou violer une contrainte - 15 -
3.5. Affectation consistante et inconsistante - 15 -
3.6. Solution d'un CSP - 15 -

4. **Problème d'optimisation sous contraintes** - 16 -
5. **Exemples de solution d'un CSP** - 16 -
5.1. **le problème des n reines** - 16 -
5.2. **Coloriage d'une carte** - 17 -
6. **Quelques algorithmes de résolution de CSP** - 18 -
6.1. **L'algorithme « génère et teste»** - 19 -
6.1.1. Principe de l'algorithme « génère et teste» - 19 -
6.1.2. Algorithme « génère et teste» : - 19 -
6.1.3. Exemple de trace d'exécution de « génère et teste» - 20 -
6.1.4. Inconvénients - 20 -
6.1.5. Améliorations - 20 -
6.2. **L'algorithme « simple retour-arrière»** - 21 -
6.2.1. Principe de l'algorithme « simple retour-arrière» : - 21 -
6.2.2. Exemple de trace d'exécution de SimpleRetourArrière - 21 -
6.2.3. Avantages - 22 -
6.2.4. Inconvénients - 22 -
7. **Solveurs de contraintes** - 22 -
8. **Conclusion** - 23 -

CHAPITRE 03 : MODELISATION ET IMPLEMENTATION

1. **Introduction** - 25 -
2. **Test dynamique** - 25 -
3. **Test structurel** - 25 -
3.1. **Graphe de Contrôle** - 26 -
3.2. **Critères de couverture du graphe de Contrôle** - 27 -
3.2.1. Le critère tous les chemins - 28 -
3.2.2. Le critère toutes les instructions (tous les nœuds) - 28 -
3.2.3. Le critère tous les enchaînements - 28 -
3.3. **Prédicat d'un chemin** - 29 -
3.4. **Prédicats et CSP** - 30 -
4. **Implémentation sur un solveur de contrainte:** - 32 -
4.1. **Présentation du langage prolog :** - 32 -
4.2. **Les bases du langage :** - 32 -
4.2.1. Les commentaires - 32 -

4.2.2. Les variables : ..... - 33 -
4.2.3. Prédicats : ..... - 33 -
4.2.4. Liste : ..... - 33 -
4.2.5. Les contraintes sous GNU-Prolg : ..... - 33 -
4.3. Implémentation du CSP(X,D,C) ..... - 35 -
4.3.1. CSP non simplifié ..... - 35 -
4.3.2. CSP simplifié ..... - 38 -
5. Conclusion ..... - 39 -
CONCLUSION GENERALE ..... - 40 -

LISTE DES FIGURES

CHAPITRE 01: GENERALITE SUR TEST DE LOGICIES

Figure 1. 1 Test fonctionnel basé sur les spécifications [25] .... - 7 -
Figure 1. 2 Test structurel : basé sur l'analyse du programme [24] .... - 9 -

CHAPITRE 02 : PROGRAMMATION PAR CONTRAINTES

Figure 2. 1 Exemples d'affectation : 4 reines .... - 17 -
Figure 2. 2 Exemples de coloriage d'une carte avec CSP........ - 17 -
Figure 2. 3 Recherche de solution de ce CSP par algorithme GET .... - 20 -
Figure 2. 4 Exécution de SimpleRetourArrière [2]........ - 22 -

CHAPITRE 03 : MODELISATION ET IMPLEMENTATION

Figure 3. 1 Graphe de contrôle du programme PGCD........ - 27 -
Figure 3. 2 Test structurel= tester ce que fait chaque élément du programme .... - 27 -
Figure 3. 3 code de résolution du CSP par GNU-Prolog........ - 36 -
Figure 3. 4 Exemple d'exécution de résolution du CSP par GNU-Prolog sans indiquer les entrées........ - 37 -
Figure 3. 5 Exemple d'exécution de résolution du CSP par GNU-Prolog en indiquant les entrées .... - 37 -
Figure 3. 6 code de résolution du CSP par GNU-Prolog........ - 38 -
Figure 3. 7 Exemple d'exécution de résolution du CSP simplifié par GNU-Prolog en indiquant les entrées - 38 -
Figure 3. 8 Exemple d'exécution de résolution du CSP' simplifié par GNU-Prolog sans indiquer les entrées- 39 -

INTRODUCTION GENERAL

En termes de fondements, de développements et d'applications, la Programmation par Contraintes et le Test de logiciel sont deux domaines scientifiques qui ont a priori peu de choses en commun.

En bref, la Programmation par Contraintes est un paradigme permettant la résolution de problèmes combinatoires difficiles issus par exemple d'applications de planification ou d'ordonnancement de tâches, tandis que le Test de logiciel est un ensemble de techniques, méthodes et processus visant à évaluer la correction des programmes informatiques. Et pourtant, depuis une vingtaine d'années maintenant, plusieurs ponts solides ont été édifiés entre ces deux domaines. La terminologie "Test à Base de Contraintes" a ainsi été forgée pour décrire ce nouveau champ de recherche et d'applications.

Parmi les différentes méthodes de test, on peut distinguer deux grandes familles : le test fonctionnel qui s'appuie sur la spécification des programmes, et le test structurel qui est basé sur l'analyse du code source. La complémentarité de ces deux approches est aujourd'hui bien admise. Le test structurel est tout particulièrement requis pour les logiciels critiques qui doivent satisfaire certains critères imposés par les organismes normatifs.

La génération automatique de tests à l'aide de la programmation par contraintes passe par la traduction du programme sous test (ou de son modèle formel) et du critère de couverture choisi en un problème de résolution de contraintes. Ensuite, un solveur de contraintes résout les contraintes et fournit un cas de test, c'est-à-dire des valeurs pour les variables d'entrée, qui peuvent être accompagnées d'un oracle décrivant le comportement attendu du programme sur ces entrées..

Notre mémoire se compose en trois chapitres organisés comme suite :

Dans le chapitre un on va présenter et étudier les notions sur le test logiciels, y compris les techniques de test connues. Concernant le deuxième chapitre, on présentera des notions sur la programmation par contraintes en indiquant comment modéliser un problème décisionnel par un CSP. En fin on terminera par la modélisation d'un code source d'un programme par CSP et on implémentera le CSP en GNU-prolog, en précisant les résultats de test.

CHAPITRE 01

GENERALITE SUR TEST DE LOGICIELS

1. Introduction

L'objectif du test n'est pas de garantir un fonctionnement parfait dans toutes les situations ce qui est généralement impossible, mais de garantir un degré de confiance dans le fonctionnement du logiciel. Ce degré est détermine par le domaine où le logiciel est utilisé et la fonctionnalité est considérée. Il est ´évident que les secteurs critiques mettant en jeu des sommes importantes ou des vies humaines tels le secteur des transports ou bancaires nécessitent un degré supérieur à celui des loisirs par exemple où une défaillance aurait des conséquences moindres.

Ce chapitre va nous permettre d'introduire des notions et des définitions concernant le test de logiciel. Nous nous intéresserons ici au vocabulaire de base du test ainsi qu'aux techniques abordées dans ce manuscrit et les techniques structurelles et fonctionnelles.

2. Test de logiciel

La qualité d'un logiciel est un ensemble de propriétés et caractéristiques d'un produit ou service qui lui donnent l'aptitude à satisfaire des besoins explicites ou implicites et la sûreté de fonctionnement est une de ces propriétés les plus importantes.

La sûreté de fonctionnement exprime la confiance qu'on peut apporter sur le service délivré par un logiciel. L'établissement de cette confiance se fait par la vérification et la validation de l'exactitude du logiciel, entre autres, par un processus de recherche et d'élimination des fautes.

Une faute est définie comme étant à l'origine d'une erreur, où celle-ci représente un état du logiciel susceptible de provoquer une défaillance.

Les moyens proposés pour l'élimination des fautes vont de la preuve formelle au test de logiciel.

Le test de logiciel est une activité dynamique de vérification et de validation de programmes.

Le test pour la vérification suppose l'existence d'une spécification explicite et complète et vise à montrer que l'implémentation lui est conforme.

Par exemple, le test de conformité dans le domaine des protocoles est un moyen de vérification. Son but est de s'assurer qu'une implantation d'un protocole est conforme aux exigences formulées dans sa spécification. Le test pour la validation, en revanche, repose sur une spécification partielle voire implicite et vise, tout autant, à provoquer des défaillances qu'à conclure que le programme se comporte bien conformément à ce qu'on attend de lui.[1]

2.1. Processus de test

Le test consiste donc à exécuter un logiciel en lui fournissant des données en entrée ainsi qu'à décider si sa réaction est celle attendue. [1]

Cette définition, ainsi que la nécessité de décider quand l'activité de test est complétée, mettent en évidence quatre étapes dans un processus de test.[1]

2.1.1. Sélection

L'étape de sélection de données de test consiste à choisir des valeurs pour les entrées du logiciel.

Nous considérons, dans le cadre de ce travail, qu'une donnée de test est une instance du produit cartésien des domaines des paramètres d'entrée du programme et une séquence de test est une suite de données de test consécutives.

La constitution manuelle de données de test efficaces est une activité très difficile, coûtante en temps et dépendante de l'erreur humaine. La sélection automatique réduit les efforts du testeur, qui devront se porter non plus sur la conception des données de test, mais sur la description des objectifs du test. Le terme de génération est utilisé pour parler de la sélection automatique de données de test.[1]

2.1.2. Exécution

Cette étape consiste à exécuter le programme sous test, en lui soumettant les données de test choisies pendant l'étape de sélection. Le test est statique si l'ensemble des données de test est constitué avant l'exécution du programme. Par contre, les données sont constituées au fur et à mesure de l'exécution du programme, le test ici est dynamique. [1]

2.1.3. Analyse (Verdict)

Après l'exécution du programme avec des données de test, il est nécessaire de décider la réussite ou non du test en comparant les résultats obtenus avec ceux attendus.[1]

2.1.4. Évaluation

La décision d'arrêt du test nécessite l'évaluation de la qualité du test effectué, une notion fortement liée à l'objectif fixé pour le test.[1]

2.2. Critères de test

Un critère est un ensemble de propriétés et de conditions qu'un ensemble de données de test doit satisfaire pour atteindre un objectif. Selon que ces propriétés et conditions portent sur la sélection ou l'évaluation des données de test, un critère peut être :

- un critère de sélection, permettant de construire, a priori, les données de test.
- un critère d'adéquation, permettant d'évaluer, a posteriori, la qualité des données de tests ; il est appelé critère d'arrêt lorsqu'il est utilisé pour décider de l'arrêt du test.

Un critère de sélection de données de test peut être défini à partir d'une information venant du programme ou d'une spécification de celui-ci. Quand les données de test sont constituées à partir d'un modèle qui est une abstraction du programme sous test, par exemple le graphe de contrôle ou de données, on parle de test structurel ou « boîte blanche ». Dans le test boîte blanche, la structure interne (code) du programme est connue et utilisé pour définir un critère de sélection. A l'opposé, on parle de test fonctionnel ou « boîte noire » quand la structure interne est inconnue ou ignorée pour la sélection. Les données de test sont alors obtenues à partir de spécifications précisant le domaine d'entrée, les fonctions accomplies et/ou des propriétés du programme sous test.[1]

Les travaux présentés portent sur la formalisation des différents critères structurels, basé sur le flot de contrôle d'un programme, en utilisant la notation Z.

La formalisation inclue les critères bien connus qui sont basés sur la couverture des instructions, décisions, conditions et leurs combinaisons ainsi que d'autres, moins connus, comme la couverture des prédicats. Cette formalisation des critères est importante pour enlever les ambiguïtés liées à une description informelle et permet, entre autres, de définir précisément l'inclusion d'un critère dans un autre. Ces relations d'inclusion peuvent être utilisées comme une mesure quantitative de la qualité du test effectué avant de décider de l'arrêt du test et facilitent l'introduction de nouveaux critères.[1]

3. Type des tests logiciel

Lorsqu’un logiciel ou une application sont créés, il est vital de réaliser plusieurs types de tests pour s’assurer que le produit est fini, complet, sécurisé et efficace. Pour réaliser ces tests, plusieurs méthodes sont possibles : les tests en « boîte noire », en « boîte blanche » et en « boîte grise ». Chacune de ces méthodes offre des possibilités différentes, que nous allons exposer dans ce mémoire.[5]

3.1. Les tests en « boîte noire »

Les tests en « boite noire » consistent à examiner uniquement les fonctionnalités d'une application, c'est-à-dire si elle fait ce qu'elle est censée faire, peu importe comment elle le fait. Sa structure et son fonctionnement interne ne sont pas étudiés. Le testeur doit donc savoir quel est le rôle du système et de ses fonctionnalités, mais ignore ses mécanismes internes. Il a un profil uniquement « utilisateur ».[5]

Ainsi, cette méthode sert à vérifier, après la finalisation d'un projet, si un logiciel ou une application fonctionne bien et sert efficacement ses utilisateurs. En général, les testeurs sont à la recherche de fonctions incorrectes ou manquantes, d'erreurs d'interface, de performance, d'initialisation et de fin de programme, ou bien encore d'erreurs dans les structures de données ou d'accès aux bases de données externes.[5]

Pour cela, ils préparent des scénarios calqués sur les différents chemins utilisateur possibles sur le système testé. Toutes les fonctionnalités doivent être prises en compte, pour qu'une fois tous les tests effectués, elles aient toutes été éprouvées. Les tests consistent à suivre un scenario, et à vérifier pour chaque fonctionnalité que les entrées (inputs) valides sont acceptées que celles non valides qui sont refusées, et bien entendu, qu'à chaque fois, le résultat (sortie ou output) attendu est bien obtenu. C'est ce que l'on appelle la méthode « trial and error » (essais et erreurs).[5]

3.1.1. Les avantages

- Simplicité : ces tests sont simples à réaliser, car on se concentre sur les entrées et les résultats. Le testeur n'a pas besoin d'apprendre à connaître le fonctionnement interne du système ou son code source, qui n'est pas accessible. Cette méthode est donc également non intrusive.
- Rapidité : en raison du peu de connaissances nécessaires sur le système, le temps de préparation des tests est très court. Les scénarios sont relativement rapides à créer et à tester, puisqu'ils suivent les chemins utilisateurs, qui sont relativement peu nombreux selon la taille du système.
- Impartialité : on est ici dans une optique « utilisateur » et non « développeur ». Les résultats du test sont impartiaux : le système marche, ou il ne marche pas. Il n'y a pas de contestation possible, comme par exemple sur l'utilisation de tel processus plutôt qu'un autre selon l'opinion du développeur.[5]

3.1.2. Les inconvénients

- Superficialité : étant donné que le code n'est pas étudié, ces tests ne permettent pas de voir, en cas de problème, quelles parties précises du code sont en cause. De plus, les testeurs peuvent passer à côté de problèmes ou vulnérabilités sous-jacentes. Certains problèmes sont également difficilement repérables avec cette méthode, comme par exemple ceux liés à la cryptographie, ou à des aléas de mauvaise qualité. C'est donc l'un des tests les moins exhaustifs.
- Redondance : si d'autres tests sont effectués, il est possible que celui-ci perde grandement de son intérêt, puisque son champ d'action a tendance à être inclus dans celui d'autres tests.[5]

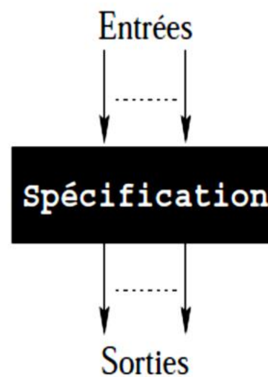

Figure 1. 1Test fonctionnel basé sur les spécifications [25]

3.2. Les tests en « boite blanche »

Les tests en « boîte blanche » (tests structurels) consistent à examiner le fonctionnement d'une application et sa structure interne, ses processus plutôt que ses fonctionnalités ici testent l'ensemble des composants internes du logiciel ou de l'application par l'intermédiaire du code source, principale base de travail du testeur.[5]

Ce type de test qui nous intéresse pour notre travail dans ce mémoire (nous le détaillerons dans le chapitre 3). Pour réaliser un test en « boîte blanche », ce dernier doit donc avoir des compétences de programmation, afin de comprendre le code qu'il étudie. Il doit également avoir une vue globale du fonctionnement de l'application, des éléments qui la composent, et naturellement de son code source. Contrairement aux tests en « boîte noire », le testeur ici à un profil développeur, et non pas utilisateur.[5]

En effectuant un test en « boîte blanche », on voit en effet quelle ligne de code est appelée pour chaque fonctionnalité. Cela permet de tester le flux de données ainsi que la gestion des exceptions et des erreurs. On s'intéresse également à la dépendance des ressources, ainsi qu'à la logique interne et justesse du code. C'est pourquoi ces tests sont surtout utiles pendant le développement d'une application, même s'ils peuvent être effectués durant de nombreuses phases de la vie d'un projet. La méthode en « boîte blanche » peut être appliquée pour les tests unitaires (majoritairement), les tests d'intégration et les tests système.[5]

La méthode en « boîte blanche » utilise des scénarios de test, créés par le testeur selon ce qu'il a appris du code source de l'environnement. L'objectif est qu'en testant l'ensemble de ces scénarios, toutes les lignes de code soient vérifiées. Ce qui est regardé, c'est le processus effectué par l'application après une entrée (input) pour obtenir un résultat. On ne fait que vérifier si le code produit les résultats espérés.[5]

3.2.1. Les avantages

- Anticipation : effectuer ces tests au cours du développement d'un programme permet de repérer des points bloquants qui pourraient se transformer en erreurs ou problèmes dans le futur (par exemple lors d'une montée en version, ou même lors de l'intégration du composant testé dans le système principal).
- Optimisation : étant donné qu'il travaille sur le code, le testeur peut également profiter de son accès pour optimiser le code, pour apporter de meilleures performances au système étudié (sans parler de sécurité…).
- Exhaustivité : étant donné que le testeur travaille sur le code, il est possible de vérifier intégralement ce dernier. C'est le type de test qui permet s'il est bien fait de tester l'ensemble du système sans rien laissé passer. Il permet de repérer des bugs et vulnérabilités cachées intentionnellement (comme par exemple des portes dérobées).[5]

3.2.2. Les inconvénients

- Complexité : ces tests nécessitent des compétences en programmation, et une connaissance accrue du système étudié.
- Durée : dû par la longueur du code source étudié, ces tests peuvent être très longs.

- Industrialisation : pour réaliser des tests en « boîte blanche », il est nécessaire de se munir d'outils tels que des analyseurs de code, des débogueurs… Cela peut avoir un impact négatif sur les performances du système, voire même impacter les résultats.
- Cadrage : il peut être très compliqué de cadrer le projet. Le code source d'un programme est souvent très long, il peut donc être difficile de déterminer ce qui est testé, ce qui peut être mis de côté… En effet, il n'est pas toujours réaliste de tout tester, ce qui prendrait trop de temps. Il est également possible que le testeur ne se rende pas compte qu'une fonctionnalité prévue dans le programme n'y a pas été intégrée. Il n'est donc pas dans le scope du testeur de vérifier si tout est là : il ne fait que tester ce qui est effectivement présent dans le code.
- Intrusion : cette méthode est très intrusive. Il peut en effet être risqué de laisser son code à la vue d'une personne externe à son entreprise : il y a des risques de casse, de vol, voire même d'intégration de portes dérobées… Choisissez donc toujours des testeurs professionnels ![5]

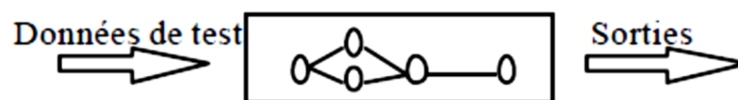

Figure 1. 2Test structurel : basé sur l'analyse du programme [24]

3.3. Les tests en « boîte grise »

Les tests en « boîte grise » compilent ces deux précédentes approches : ils éprouvent à la fois les fonctionnalités et le fonctionnement d'un système. C'est-à-dire qu'un testeur va par exemple donner une entrée (input) à un système, vérifier que la sortie obtenue est celle attendue, et vérifier par quel processus ce résultat a été obtenu.[5]

Dans ce type de tests, le testeur connaît le rôle du système, ses fonctionnalités, et il a également une connaissance bien que relativement limitée de ses mécanismes internes (en particulier la structure des données internes et les algorithmes utilisés). Attention cependant, il n'a pas accès au code source ![5]

Ces tests peuvent difficilement être effectués pendant la phase de développement du projet, car elle implique des tests sur les fonctionnalités du programme : celui-ci doit déjà être dans un état proche de final pour que ces tests puissent être pertinents. En effet, pendant les tests en « boîte grise », ce sont surtout des techniques de « boîte noire » qui sont utilisées, puisque le code n'est pas accessible. Cependant, les scénarios sont orientés pour jouer sur les processus sous-jacents, et ainsi les tester également.[5]

Bien entendu, la méthode « boîte grise » combine surtout les avantages des méthodes « boîte blanche » et « boîte noire ». On peut cependant noter deux gros bénéfices de cette technique :

- Impartialité : les tests en « boîte grise » gardent une démarcation entre les développeurs et le testeur, puisque ce dernier n'étudie pas le code source et peut s'appuyer sur les résultats obtenus en testant l'interface utilisateur.
- Intelligence : en connaissant la structure interne du programme, un testeur peut créer des scénarios plus variés et intelligents, afin d'être certain de tester toutes les fonctionnalités mais également tous les processus correspondants du programme.

En parallèle, l'un des inconvénients les plus importants de ces tests est le suivant :

- Non exhaustivité : étant donné que le code source n'est pas accessible, il est impossible avec des tests en « boîte grise » d'espérer avoir une couverture complète du programme.

4. Résumé pour mieux comprendre

Une analogie est souvent utilisée pour différencier ces techniques, en comparant le système testé à une voiture.

- En méthode « boîte noire », on vérifie que la voiture fonctionne en allumant les lumières, en klaxonnant et en tournant la clé pour que le moteur s'allume. Si tout se passe comme prévu, la voiture fonctionne.
- En méthode « boîte blanche », on emmène la voiture chez le garagiste, qui regarde le moteur ainsi que toutes les autres parties (mécaniques comme électriques) de la voiture. Si elle est en bon état, elle fonctionne.
- En méthode « boîte grise », on emmène la voiture chez le garagiste, et en tournant la clé dans la serrure, on vérifie que le moteur s'allume, et le garagiste observe en même temps le moteur pour s'assurer qu'il démarre bien selon le bon processus.

5. Conclusion

Dans ce chapitre, nous avons donné un bref aperçu sur test de logiciel et quelques types de test, en indiquant le test qui nous intéresse pour notre travail qui est le test en boite blanche, mais afin de réaliser notre travail on a besoin de prendre quelques idées sur la programmation par contraintes.

CHAPITRE : 02

PROGRAMMATION PAR CONTRAINTES

1. Introduction

Nous présentons brièvement les principes de la programmation par contraintes pour la résolution de problèmes de satisfaction de contraintes et d'optimisation sous contraintes. Nous insisterons sur le rôle crucial joué par les stratégies de recherche notamment par le biais des heuristiques de sélection de variable et de valeur. Nous rappellerons aussi quelques algorithmes de recherche. Avant de conclure, nous passerons en revue les principaux solveurs rencontrés lors de nos recherches.

2. Problème de satisfaction de contraintes

Un problème de satisfaction de contraintes CSP (Constraint Solving Problem) est un problème modélisé sous la forme d'un ensemble de contraintes posées sur des variables. [21]

Une contrainte est une propriété entre différentes inconnues (les variables) qui doit être vérifiée. Chaque variable prend ses valeurs dans un ensemble donné (fini ou non) que l'on appelle domaine. Selon le domaine considéré, il peut être décrit soit sous la forme d'un ensemble de valeurs soit par un intervalle ou une union d'intervalles. Dans ce mémoire, nous ne considérons que les domaines finis (Finite Domain). Ainsi, une contrainte peut être vue comme une restriction des valeurs que peuvent prendre simultanément les variables. Il existe différentes caractérisations d'une contrainte (linéaire, relationnelles, booléennes, etc.), pour plus d'informations, nous renvoyons le lecteur à [3, 4,7].

Une contrainte peut également être définie en intension par :

- une équation mathématique ($x+y \leq 4$),
- une relation logique simple ($x \leq 1 \Rightarrow y /= 2$) ,
- ou même par un prédicat portant sur n variables (allDifferent (x1, . . . ,xn)).

Une telle contrainte est appelée contrainte globale. La définition d'une nouvelle contrainte est aisée puisque la seule opération indispensable est le test de consistance qui détermine si une instanciation de ces variables satisfait la contrainte.

3. Modélisation d'un problème par des contraintes

Un problème de satisfaction de contraintes (CSP) est défini formellement par un triple (X , D, C) représentant respectivement un ensemble fini de variables X , une fonction D affectant à chaque variable $x \in X$ son domaine $D(x)$ et un ensemble fini de contraintes C. Le domaine initial d'une variable $D_0(x)$ représente l'ensemble de valeurs auxquelles la variable x peut être instanciée avant le début de la résolution. Chaque contrainte $c \in C$ est une

relation multidirectionnelle portant sur un sous-ensemble de variables noté var(c) $\subseteq X$ restreignant les valeurs que ces variables peuvent prendre simultanément. [2]

Exemple : soit le CSP

- $X = \{a, b, c, d\}$
- $D = \{D_a, D_b, D_c, D_d\}$ avec
  $D_a = D_b = D_d = \{1, 2,3, 4, 5\}$
  $D_c = \{0, 1\}$
- $C = \{c_1, c_2, c_3\}$ avec
  $c_1 : a \neq b$
  $c_2 : c \neq d$
  $c_3 : a + c < b$

3.1. Instanciation

Une instanciation $x \leftarrow v$ consiste à affecter à une variable x une valeur v appartenant à son domaine D(x). À chaque étape de la résolution, une affectation partielle A est définie comme l'ensemble des domaines courants de toutes les variables. Le domaine courant D(x) d'une variable x est toujours un sous-ensemble de son domaine initial $D_0(x)$ (inclusion non stricte). On note var (A) l'ensemble des variables instanciées, c'est-à-dire dont le domaine est réduit à un élément.

3.2. Affectation

On appelle affectation le fait d'instancier certaines variables par des valeurs (évidemment prises dans les domaines des variables). On notera $A = \{(X_1, V_1), (X_2, V_2), ..., (X_r, V_r)\}$ l'affectation qui instancie la variable X_1 par la valeur V_1, la variable X_2 par la valeur V_2, ..., et la variable X_r par la valeur V_r.

Par exemple, sur le CSP précédent,

A = {(a, 1), (b,1), (c, 1)} est l'affectation qui instancie a à 1 et b à 1.

3.3. Affectation partielle ou totale

Une affectation est dite partielle si var (A) $\subseteq X$ ou totale si var (A) = X.

Une affectation A' restreint une affectation partielle A, noté $A' \subseteq A$, si le domaine de n'importe quelle variable x dans A' est un sous-ensemble de son domaine dans A.

Les valeurs minimum et maximum du domaine d'une variable entière x sont notées respectivement min(x) et max(x). Les notations $min(x) \leftarrow v$ et $max(x) \leftarrow v$ représentent

respectivement les réductions du domaine à $D(x) \cap [v, +\infty[$ et $D(x) \cap]-\infty, v]$. La notation $x \,/\!\!\leftarrow v$ indique la suppression de la valeur v du domaine D(x). [2]

Sur le CSP précédent,

A = {(a, 1), (b,1), (c, 1)} est une affectation partielle

A = {(a,1), (b,1), (c,1), (d,1)} est une affectation totale

3.4. Satisfaire ou violer une contrainte

Une affectation viole une contrainte si toutes ses variables sont instanciées et que la relation associée à la contrainte n'est pas vérifiée (satisfait).

3.5. Affectation consistante et inconsistante

Une affectation est consistante si elle ne viole aucune contrainte et inconsistante dans le cas contraire.

Sur le CSP précédent,

- l'affectation partielle A = {(c,1),(d,0)} est consistante car elle ne viole aucune
- contrainte l'affectation totale A = {(c,1), (d,0) ,(a,3), (b,5)}est consistante car elle ne viole aucune contrainte
- l'affectation partielle A = {(a; 1); (b; 1)} est inconsistante car elle viole la contrainte c_1

3.6. Solution d'un CSP

Une solution d'un CSP est donc une affectation totale consistante.

Sur le CSP précédent,

L'affectation totale : A = {(c, 1), (d, 0), (a, 3); (b, 5)} est consistante, il s'agit donc d'une solution du CSP.

Résoudre un CSP consiste à exhiber une unique solution ou à montrer qu'aucune solution n'existe (le problème est irréalisable). Prouver la consistance d'un CSP est un problème NP-complet mais exhiber une solution est un problème NP-difficile dans le cas général. On peut également être intéressé par la détermination de plusieurs ou toutes les solutions d'un problème. [2,4]

4. Problème d'optimisation sous contraintes

Dans de nombreux cas, des relations de préférence entre les solutions d'un CSP existent eu égard aux critères fixés par le décideur. L'optimisation consiste alors à rechercher des solutions optimales d'un CSP au regard des relations de préférence du décideur. Nous nous intéresserons uniquement à des problèmes d'optimisation combinatoire monocritère, c'est-à-dire lorsque l'ensemble des solutions est discret et qu'il y a un critère d'optimalité unique. Ce critère d'optimalité est généralement associé à la maximisation/- minimisation d'une fonction objectif d'un sous-ensemble de variables vers les entiers. Un problème d'optimisation sous contraintes (COP) est un CSP augmenté d'une fonction objective f.

Cette fonction est souvent modélisée par une variable dont le domaine est défini par les bornes supérieures et inférieures de f. [2]

5. Exemples de solution d'un CSP

5.1. le problème des n reines

Au cours de ce chapitre, nous illustrerons divers concepts sur le problème des n reines. Le but de ce problème, inspiré du jeu d'échec, est de placer n reines sur un échiquier de dimension n × n de manière à ce qu'aucune ne soit en prise. Deux reines sont en prises si elles sont sur la même ligne, la même colonne ou la même diagonale. Un échiquier classique comporte 8 lignes et 8 colonnes. Ce problème désormais classique est devenu une référence de mesure de performance des systèmes grâce à un énoncé simple masquant sa difficulté.

Un modèle classique sans contraintes globales est défini dans les formules 2.1, 2.2 et 2.3. En observant que deux reines ne peuvent pas être placées sur la même colonne, on peut imposer que la reine i soit sur la colonne i. Ainsi, la variable l_i de domaine {1, . . ., n} représente la ligne où est placée la reine dans la colonne i. Les contraintes (2.1) imposent que les reines soient sur des lignes différentes alors que les contraintes (2.2) et (2.3) imposent que deux reines soient placées sur des diagonales différentes. [4]

$$l_i \neq l_j \qquad 1 \leq i < j \leq n \qquad (2.1)$$

$$l_i \neq l_j + (j - i) \qquad 1 \leq i < j \leq n \qquad (2.2)$$

$$l_i \neq l_j - (j - i) \qquad 1 \leq i < j \leq n \qquad (2.3)$$

Les contraintes précédentes montrent plusieurs affectations partielles pour le problème des 4 reines dans lesquelles l^{er} placement d'une reine sur l'échiquier est représenté par un cercle dans la case correspondante. Les reines sont ordonnées de gauche à droite et les positions de haut en bas.

On peut observer en figure 2.1 que l'affectation partielle $\{l_1 \leftarrow 1, l_2 \leftarrow 3, l_3 \leftarrow 2\}$ n'est pas consistante car elle viole une des contraintes (2.2). Au contraire, les affectations totales $\{l_1 \leftarrow 2, l_2 \leftarrow 4, l_3 \leftarrow 1, l_4 \leftarrow 3\}$ et $\{l_1 \leftarrow 3, l_2 \leftarrow 1, l_3 \leftarrow 4, l_4 \leftarrow 2\}$ sont les deux solutions symétriques des 4 reines. Si l'on définit un COP à partir du problème des n reines en définissant la fonction objectif min (l_1), alors le problème admet une unique solution optimale : $\{l_1 \leftarrow 2, l_2 \leftarrow 4, l_3 \leftarrow 1, l_4 \leftarrow 3\}$.

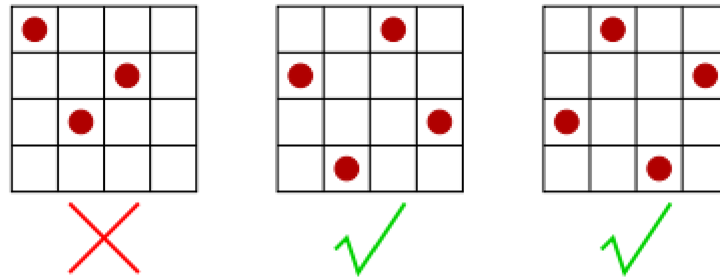

Figure 2. 1Exemples d'affectation : 4 reines

5.2. Coloriage d'une carte

Pour modéliser ce problème sous la forme d'un CSP, il s'agit d'identifier les variables (les inconnues du problème), les domaines de valeur de ces variables, et les contraintes existant entre ces variables.

Ici, ce que l'on doit déterminer (nos inconnues), c'est la couleur de chaque région. On aura donc une variable pour chaque région, chacune de ces variables pouvant prendre comme valeur une des 4 couleurs.

Les contraintes spécifient que deux régions voisines ne doivent pas être de la même couleur.[3]

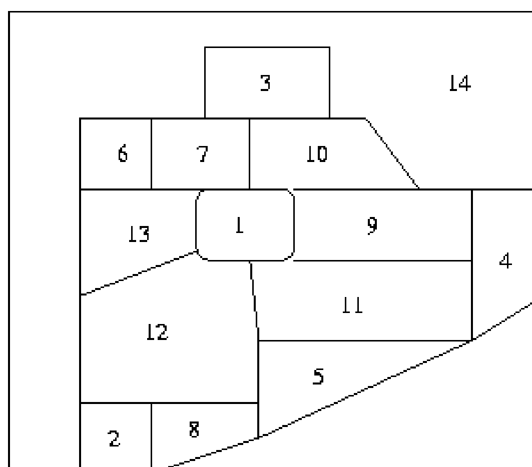

Figure 2. 2Exemples de coloriage d'une carte avec CSP

On définit le CSP (X, D, C) tel que

- $X = \{X_1, X_2, ..., X_{14}\}$
  (On associe une variable X_i différente par région i à colorier.)
- Pour tout X_i élément de X, $D(X_i)$ = {bleu, rouge, vert, jaune}
  (Chaque région peut être coloriée avec une des 4 couleurs.)
- C = {$X_i \neq X_j$ / X_i et X_j sont 2 variables de X correspondant à des régions voisines}(2 régions voisines doivent être de couleurs différentes.)

Pour être plus précis, on peut définir explicitement les relations de voisinage entre régions, par exemple à l'aide d'un prédicat voisines/2, tel que voisines(X, Y) soit vrai si X et Y sont deux régions voisines. Ce prédicat peut être défini en extension, en listant l'ensemble des couples de régions ayant une frontière en commun :

voisines(X,Y) <=> (X,Y) élément-de {(1,7), (1,9), (1,10), (1,11), (1,12), (1,13), (2,8), (2,12), (2,14), (3,7), (3,10), (3,14), (4,9), (4,11), (4,14), (5,8), (5,11), (5,12), (6,7), (6,13), (6,14), (7,1), (7,3), (7,6), (7,10), (7,13), (7,14), (8,2), (8,5), (8,12), (9,1), (9,4), (9,10), (9,11), (10,1), (10,3), (10,7), (10,9), (10,14), (11,1), (11,4), (11,5), (11,9), (11,12), (12,1), (12,2), (12,5), (12,8), (12,11), (12,13), (12,14), (13,1), (13,6), (13,7), (13,12), (13,14), (14,2), (14,3), (14,4), (14,6), (14,7), (14,10), (14,12), (14,13)}

On peut alors définir l'ensemble des contraintes C de la façon suivante :

C = {$X_i \neq X_j$ / X_i et X_j sont 2 variables différentes de X et voisines (X_i,X_j) = vrai }

Ce problème de coloriage d'une carte est un cas particulier du problème du coloriage des sommets d'un graphe (deux sommets adjacents du graphe doivent toujours être de couleurs différentes). De nombreux problèmes "réels" se ramènent à ce problème de coloriage d'un graphe : problème des examens, d'emploi du temps, de classification.

6. Quelques algorithmes de résolution de CSP

Les algorithmes que nous allons étudier permettent de rechercher une solution à un CSP (n'importe laquelle, c'est-à-dire la première que l'on trouve). Suivant les applications, « résoudre un CSP» peut signifier autre chose que chercher simplement une solution. En particulier, il peut s'agir de chercher la « meilleure» solution selon un critère donné. Par exemple, pour le problème du coloriage d'une carte, on peut chercher la solution qui utilise le moins de couleurs possibles,... Ces problèmes d'optimisation sous contraintes, où l'on cherche à optimiser une fonction objective donnée tout en satisfaisant toutes les contraintes, peuvent

être résolus en explorant l'ensemble des affectations possibles selon la stratégie de « Séparation & Evaluation » (Branch&Bound) bien connue en recherche opérationnelle. [4]

6.1. L'algorithme « génère et teste»

6.1.1. Principe de l'algorithme « génère et teste»

La façon la plus simple de résoudre un CSP sur les domaines finis consiste à énumérer toutes les affectations totales possibles jusqu'àen trouver une qui satisfait toutes les contraintes. Ce principe est repris dans la fonction récursive « génèreEtTeste (A, (X, D, C))» décrite ci-dessous. Dans cette fonction, A contient une affectation partielle et (X, D, C) décrit le CSP à résoudre (au premier appel de cette fonction, l'affectation partielle A sera vide). La fonction retourne vrai si on peut étendre l'affectation partielle A en une affectation totale consistante (une solution), et faux sinon. [4]

6.1.2. Algorithme « génère et teste» :

```
fonction GET(A,(X,D,C)) : booléen
début
    sitoutes les variables de X sont affectées alors
        siA est consistante alors
            retourner VRAI
        sinon
            retourner FAUX
        finsi
    sinon
        choisir une variable Xi de X qui n’est pas encore affectée
        pour toute valeur Vi appartenant à Difaire
            si GET(A ∪ {(Xi ,Vi )}, (X,D,C)) = VRAI alors
                retourner VRAI
            finsi
        fin pour
        retourner FAUX
     finsi
fin
```

6.1.3. Exemple de trace d'exécution de « génère et teste»

$X = \{X_1, X_2, X_3, X_4\}$

$D = \{D_1, D_2, D_3, D_4\}$ avec $D_1 = D_2 = D_3 = D_4 = \{0, 1\}$

$C = \{X_1 \neq X_2, X_3 \neq X_4, X_1 + X_3 < X_2\}$

Recherche de solutions de ce CSP par l'algorithme GET

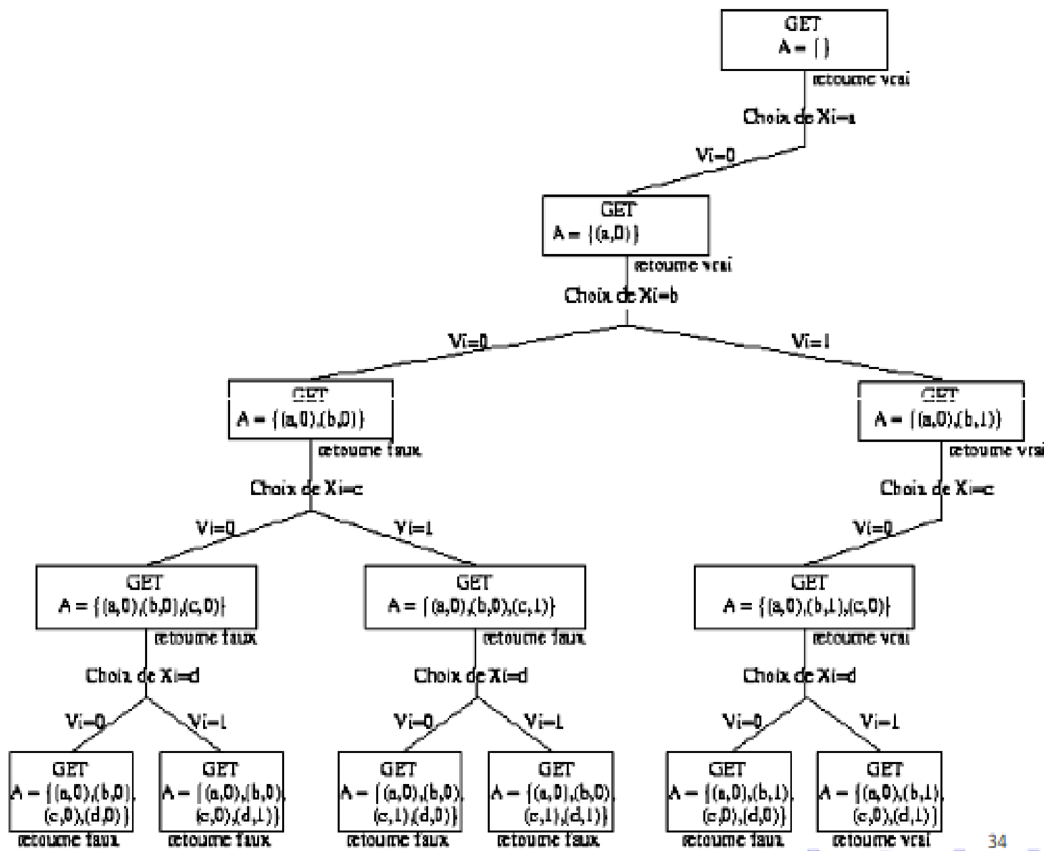

Figure 2. 3Recherche de solution de ce CSP par algorithme GET

6.1.4. Inconvénients

- Sur l'espace de recherche
- Ensemble des affectations complètes
- Inconsistance découverte au dernier moment
- Croissance exponentielle de la taille de l'espace de recherche

6.1.5. Améliorations

- Ne développer que des affectations partielles consistantes
- Réduire la taille des domaines

6.2. L'algorithme « simple retour-arrière»

6.2.1. Principe de l'algorithme « simple retour-arrière» :

Une première façon d'améliorer l'algorithme « génère et teste» consiste à tester au fur et à mesure de la construction de l'affectation partielle sa consistance : dès lors qu'une affectation partielle est inconsistante, il est inutile de chercher à la compléter. Dans ce cas, on retourne en arrière (« backtrack» en anglais) jusqu'à la plus récente instanciation partielle consistante que l'on peut étendre en affectant une autre valeur à la dernière variable affectée.

1.1.1. Algorithme « simple retour-arrière» [4]

fonction SRA(A,(X,D,C)) : **booléen**

début

    siA n'est pas consistante **alors**

        retourner FAUX

    finsi

    si toutes les variables de X sont affectées **alors**

        retourner VRAI

        sinon

        choisir une variable X_i de X qui n'est pas encore affectée

        pour toute valeur Vi appartenant à Di **faire**

            si SRA($A \cup \{(X_i, V_i)\}$, (X,D,C)) = VRAI **alors**

                retourner VRAI

            finsi

        fin pour

        retourner FAUX

    finsi

fin

6.2.2. Exemple de trace d'exécution de SimpleRetourArrière

Considérons le problème des 4 reines.

$X = \{X_1, X_2, X_3, X_4\}$

$D = \{D_1, D_2, D_3, D_4\}$ avec $D_1 = D_2 = D_3 = D_4 = \{1, 2, 3, 4\}$

$C = \{c_1, c_2, c_3\}$ avec

$c_1 = \{X_i \neq X_j, i \neq j, i \in \{1, 2, 3, 4\}, j \in \{1, 2, 3, 4\}\}$

$c_2 = \{X_i + i \neq X_j + j, i \neq j, i \in \{1, 2, 3, 4\}, j \in \{1, 2, 3, 4\}\}$

$c_3 = \{X_i - i \neq X_j - j, i \neq j, i \in \{1, 2, 3,4\}, j \in \{1, 2, 3,4\}\}$

L'enchainement des appels successifs à la fonction SRA peut être représenté par l'arbre suivant (Figure 2.7) (chaque nœud correspond à un appel de la fonction, l'échiquier dessiné à chaque nœud décrit l'affectation partielle en cours). [4]

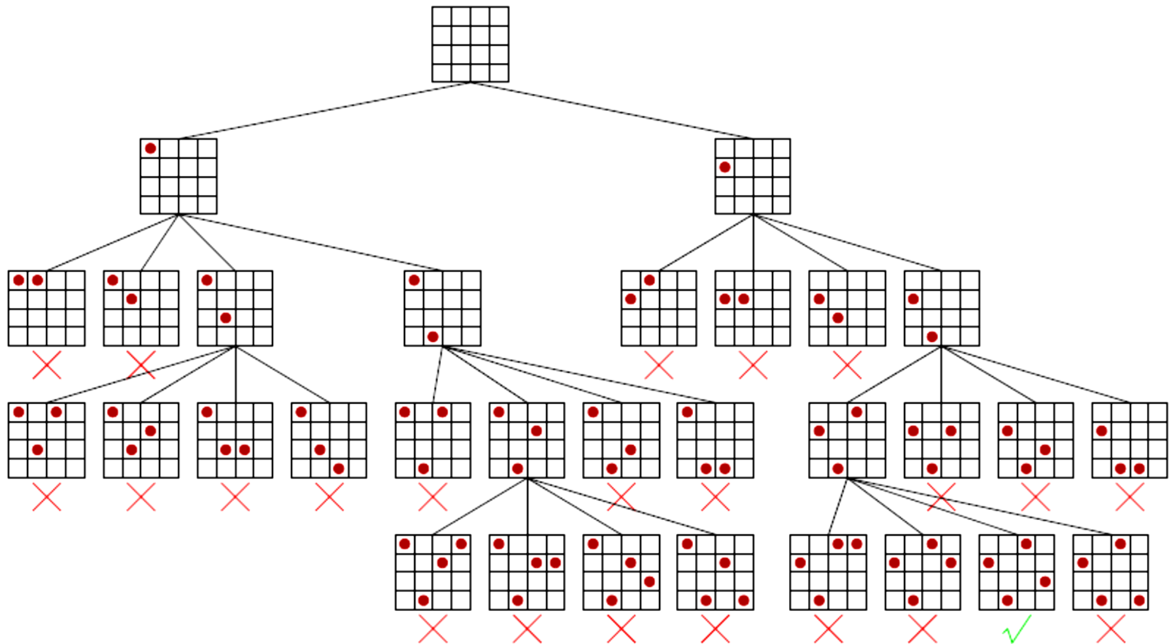

Figure 2. 4Exécution de SimpleRetourArrière[2]

6.2.3. Avantages

- Réduction de l'espace de recherche
- Améliore GET en espace et en temps d'exécution

6.2.4. Inconvénients

- Pas d'identification des causes de conflit
- Redondance
- Détection tardive des conflits

7. Solveurs de contraintes

Nous citons quelques outils de programmation par contraintes. Historiquement, les premiers solveurs étaient des extensions du langage déclaratif de programmation logique Prolog [6], dont l'algorithme d'unification est une clé. De nos jours, les extensions de Prolog les plus populaires sont GNU prolog, ECLiPSe et SICStus Prolog

Le principe d'un solveur de contraintes peut se résumer en quatre étapes principales : [25]

1. simplifier le problème donnée, puis stocker les contraintes non encore satisfaites dans un magasin de contraintes (constraint-store) jusqu'à l'affectation de variables qui pourrait la faire de nouveau progresser.
2. choisir puis instancier une variable selon la ou les heuristiques implémentées
3. L'affectation d'une variable « réveille » la propagation des contraintes concernées, qui peuvent disparaître (lorsqu'elles sont résolues), ou réveiller/créer d'autres contraintes, ou amener à un échec de la résolution (domaine vide ou échec lors de l'unification).
4. - si toutes les variables sont instanciées alors s'arrêter sur un succès
   - si il y a un échec alors opérer un backtracking
   - si il y a un échec et aucun backtrackingn'est possible alors s'arrêter sur un échec
   - sinon retourner à l'étape 2.

8. Conclusion

Nous avons décrit dans ce chapitre les principes de la programmation par contraintes (PPC). Cette approche déclarative permet de résoudre des problèmes combinatoires variés. Les utilisateurs décrivent leur problème en posant des contraintes sur les valeurs que peuvent prendre les différentes variables composant le problème. Un solveur de contraintes est alors chargé de calculer les solutions du problème. Le processus de résolution exacte de problèmes de satisfaction de contraintes permet de générer efficacement les solutions d'un problème grâce à la combinaison des techniques de filtrage et des algorithmes de recherche.

CHAPITRE : 03

MODELISATION ET IMPLEMENTATION

1. Introduction

Il existe de nombreuses méthodes de tests. Ces méthodes se répartissent en deux familles : les méthodes de test statique et les méthodes de test dynamique.

- Le test statique est appliqué sur une description du programme ou directement sur le texte du programme mais sans exécuter ce dernier.
- Le test dynamique est la technique de test la plus répandue. Elle repose sur l'exécution du programme sur un sous-ensemble des données et à la vérification que le résultat attendu est satisfaisant. Les domaines des entrées d'un programme étant généralement très grands, il n'est pas réaliste d'effectuer du test exhaustif c'est-à-dire une exécution sur toutes les entrées possibles. Il faut donc sélectionner un sous-ensemble pertinent, c'est-à-dire un sous-ensemble de données le plus efficace pour trouver des erreurs dans le programme.

2. Test dynamique

Le test dynamique se décompose en quatre étapes [26] :

1. la **sélection** qui consiste à choisir judicieusement un sous-ensemble d'entrées, que l'on appelle jeu de tests, parmi toutes les entrées possibles.
2. **l'exécution** qui consiste à faire exécuter le programme sur les entrées choisies à l'étape précédente.
3. **l'analyse**(Verdict) qui consiste à déterminer si un test a échoué ou non. Pour cela, on s'intéresse aux sorties du programme et on vérifie si elles sont conformes ou non aux résultats attendus. Si l'objectif du test est de vérifier un système alors on considère que le test a échoué dès qu'une erreur est révélée, par contre, si son objectif est de mettre en évidence un maximum d'erreurs, le test est un échec si aucune erreur n'est révélée.
4. **l'évaluation** de la qualité des tests exécutés qui permet entre autres de décider ou non de l'arrêt du test.

Le test dynamique à son tour se subdivise en deux tests, test aléatoire et test structurel (boite blanche vue dans le chapitre 1). Dans ce mémoire, nous nous intéressons au test structurel.

3. Test structurel

L'objectif du test structurel est de tester comment le programme fait ce qu'il a à faire. La sélection des données d'entrée se fait à l'aide d'une description de la structure du programme qui peut être le graphe de contrôle ou le flot de données [8].

Il existe différents types d'approches pour générer des données de test [25]:

- L'approche orientée chemin qui consiste à sélectionner un chemin du programme qui atteint l'élément choisi (instruction, enchaînement, etc.) puis de calculer les entrées qui permettent l'exécution de ce chemin.
- L'approche orientée but qui consiste à générer des données d'entrées qui permettent d'atteindre l'élément choisi (instruction, enchaînement, etc.) sans s'occuper du chemin emprunté. L'étape de sélection des chemins est inexistante.

Dans ce mémoire, nous nous intéressons aux méthodes orientées chemin.

3.1. Graphe de Contrôle

Un graphe de contrôle est constitué d'arcs et de sommets.

- Les sommets sont les blocs d'instructions indivisibles maximaux et les prédicats qui apparaissent dans les instructions conditionnelles ou les boucles,
- Les arcs correspondent aux transferts de contrôle possibles entre ces sommets.

Un bloc d'instructions indivisible maximal est suite d'instructions élémentaires telle que le contrôle passe consécutivement de l'une à l'autre sans choix possible (pas de While, IfThenElse, IfThen, etc.).

Les instructions conditionnelles IfThenElseseront représentées par trois sommets et deux arcs, les instructions IfThen par deux sommets et deux arcs et les boucles conduiront à un circuit dans le graphe.

Les blocs d'instructions indivisibles maximaux sont représentés par des sommets carrés et les prédicats par des sommets ronds.

Un chemin **complet** dans un graphe de contrôle est un chemin du graphe (suite de sommets ou d'arcs) qui va du point d'entrée du graphe à son point de sortie.

Un chemin **élémentaire** est un chemin complet du graphe tel que chaque arc n'est emprunté au plus qu'une seule fois.

La **longueur** d'un chemin est définie par le nombre d'arcs (resp. sommets) qui le composent.

Exemple :

E : intpgcd(u :int,v :int)

i,t : int ;

P1 : while (u>0){

P2 : if (v>u)

B1 : then { t:=u ; u:=v ; v:=t} ;

B2 : u:=u-v ;}

S : return(v)

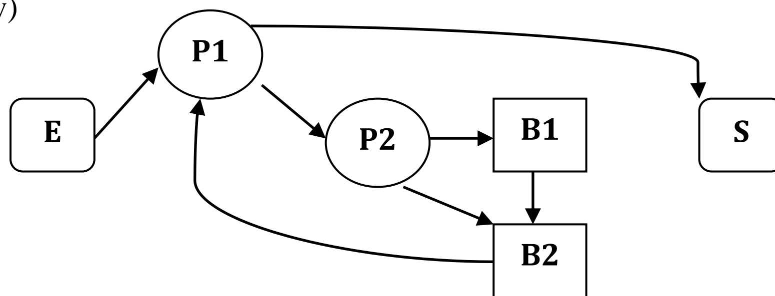

Figure 3. 1Graphe de contrôle du programme PGCD

Le graphe de contrôle de la figure 3.1 a2 chemins élémentaires qui vont du sommet entrant E au sommet sortant S :

- **E. P1.S.**
- **E.P1.P2.B1.B2.P1.S**

3.2. Critères de couverture du graphe de Contrôle

Dans l'étape de sélection, La sélection se fait à l'aide de critères de couverture. C'est également le critère de couverture qui détermine le nombre de tests à effectuer.

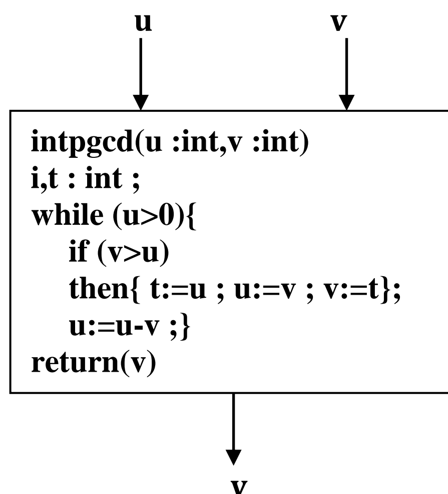

Figure 3. 2Test structurel= tester ce que fait chaque élément du programme

Les principaux critères de couverture structurelle qui se basent sur le graphe de contrôle sont le critère "tous les chemins, le critère toutes les instructions, c'est-à-dire les sommets carrés du graphe de contrôle, et le critère tous les enchaînements.

3.2.1. Le critère tous les chemins

Ce critère impose de tester au moins une fois tous les chemins exécutables entre le nœud d'entrée du graphe de contrôle et tous les nœuds de sortie. Ce critère, à priori simple, devient très vite trop coûteux et surtout impraticable dès lors qu'un programme contient une boucle qui implique l'existence d'une infinité de chemins à tester. Il est donc obligatoire d'y ajouter de restrictions (deux restrictions possibles):

1. Tous les chemins limite-intérieure (ou i-chemins) : demande d'itérer chaque boucle au plus une fois, puis dans un deuxième essai au plus deux fois, puis dans un $i^{ème}$ essai au plus i fois. Cette restriction permet ainsi de réduire le nombre de chemins pendant le test.

Dans l'exemple précédent (PGCD) :

- le critère toutes les 1-chemins satisfait par le chemin : **E.P1.P2.B1.B2.P1.S**
- le critère toutes les 2-chemins satisfait par le chemin : **E.P1.P2.B1.B2.P1. P2.B1.B2.P1.S**

2. Tous les chemins de taille bornée : on se limite à ne tester que les chemins de longueur inférieure ou égale à n.

3.2.2. Le critère toutes les instructions (tous les nœuds)

Ce critère exige d'exécuter au moins une fois chaque instruction : ce qui revient à passer au moins une fois par chaque sommet du graphe de contrôle. Il est relativement facile à mettre en œuvre car il existe toujours au moins un sous-ensemble fini de chemins qui le satisfait, et ce, que le programme contienne des boucles ou non.

Dans l'exemple précédent (PGCD), le critère toutes les instructions satisfait par le chemin : **E.P1.P2.B1.B2.P1.S**

3.2.3. Le critère tous les enchaînements

Ce critère exige d'emprunter au moins une fois chaque enchaînement possible d'instructions : ce qui revient à passer au moins une fois par chaque arc du graphe de contrôle.

C'est le critère le plus souvent utilisé en pratique car il est moins coûteux que "tous les chemins" et il assure une meilleure détection que "toutes les instructions".

Dans l'exemple (PGCD), le critère "tous les enchaînements" impose de couvrir les enchaînements suivants :

- passer dans la branche Then de P1 puis par la branche Then de P2
- passer dans la branche Else de P1

3.3. Prédicat d'un chemin

On peut considérer un chemin du programme comme une suite dont les éléments sont soit :

- une affectation
- un prédicat de conditionnelle ou de boucle (correspondant à l'exécution de la branche Then ou du corps de la boucle)
- la négation d'un prédicat de conditionnelle ou de boucle (correspondant à l'exécution de la branche Else, la sortie d'une conditionnelle incomplète ou d'une boucle)

Le prédicat d'un chemin est une condition sur les données d'entrée qui provoque l'exécution de ce chemin. Classiquement, la construction du prédicat de chemin se fait de la manière suivante [25] :

- On initialise la construction en considérant que toute variable x a une valeur courante initiale notée x_0
- Toute affectation x := expr sera transformée en une égalité $x_{i+1} = \widetilde{expr}$ où i est l'indice de la valeur courante de x et $\widetilde{expr}$ est l'expression expr où tout variable a été instanciée par sa valeur courante. La valeur courante de x devient x_{i+1}
- Tout prédicat P (resp. négation de prédicat non P) d'une conditionnelle ou d'une boucle est remplacé par le prédicat $\widetilde{P}$ (resp. $\widetilde{\neg P}$) qui est le prédicat (resp. non P) où toutes les variables ont été remplacées par leur valeur courante.

Le prédicat final est la conjonction de toutes ces formules.

Dans le programme pgcd précédent, si l'on exprime le prédicat du chemin **E.P1.S,** c'est-à-dire du chemin ne passant pas par la boucle, cela donne :

predicat(u_0,v_0,E.P1.S) := (u_0> 0)

Si l'on exprime le prédicat du chemin passant deux fois par la boucle, et une fois dans la branche Then (par exemple au premier passage dans la boucle), cela donne :

predicat(u_0,v_0,E.P1.P2.B1.B2.P1.P2.B2.P1.S) :=$(u_0>0)and(v_0>u_0)and(t_0=u_0)$ $and(u_1=v_0)and(v_1=t_0)and(u_2=u_1-v_1)$ $and(u_2>0)and(\neg(v_1>u_2))$ $and(u_3=u_2-v_1)and(\neg(u_3>0))$ (3.1)

où

– $u_0 > 0$ correspond au premier passage dans le corps de la boucle

– $v_0 > u_0$ correspond au passage dans la branche Then

– $u_2 > 0$ correspond au second passage dans le corps de la boucle

– $\neg (v_1 > u_2)$ correspond au non passage dans la branche Then

– $\neg (u_3 > 0)$ correspond à la sortie de la boucle

Ce prédicat peut s'exprimer également qu'avec les variables d'entrée :

Prédicat (u_0, v_0, E.P 1.P 2.B1.B2.P 1.P 2.B2.P 1.S) :=$(u_0 > 0)$ and $(v_0 > u_0)$ and $(v_0 - u_0 > 0)$ and $(\neg (u_0 > v_0 - u_0))$ and $(\neg (v_0 - u_0 - u_0 > 0))$ (3.2)

Où

- $(u_0 > 0)$: donne $\mathbf{(u_0 > 0)}$;
- $(v_0 > u_0)$: donne $\mathbf{(v_0 > u_0)}$;
- $(t_0 = u_0)$ et $(u_1 = v_0)$ et $(v_1 = t_0)$ et $(u_2 = u_1 - v_1)$ et $(u_2 > 0)$: donnent $\mathbf{(v_0 - u_0 > 0)}$
- $(t_0 = u_0)$ et$(v_1 = t_0)$et $(u_1 = v_0)$ et $(u_2 = u_1 - v_1)$ et $(\neg(v_1 > u_2))$ donnent $\mathbf{(\neg (u_0 > v_0 - u_0))}$
- $(\neg(u_3 > 0))$ et $(u_3 = u_2 - v_1)$ et $(u_2 = u_1 - v_1)$ et $(u_1 = v_0)$ et $(t_0 = u_0)$ et $(v_1 = t_0)$ donnent $\mathbf{(\neg (v_0 - u_0 - u_0 > 0))}$ou $(v_0 \leq 2u_0)$

Le prédicat (3.2) se simplifie en :

Prédicat (u_0, v_0, E.P1.P2.B1.B2.P1.P2.B2.P1.S) := $(u_0 > 0)$ and $(v_0 = 2u_0)$ (3.3)

Où

- $(u_0 > 0)$: donne $\mathbf{(u_0 > 0)}$;
- $(v_0 > u_0)$ e**t**$\mathbf{(v_0 \leq 2u_0)}$ donne $\mathbf{(v_0 = 2u_0)}$.

3.4. Prédicats et CSP

Si on reprend le prédicat (3.1) :

Le CSP(X,D, C) associé est :

- $X = \{u_0, v_0, t_0, u_1, v_1, u_2, u_3\}$ (3.4)
- $D = \{D_{u0};D_{v0} ;D_{t0} ;D_{u1} ;D_{v1} ;D_{u2} ;D_{u3}\}$ avec $\forall D_i \in D$, $D_i = IN$ (3.5)
- $C = \{c1, c2, c3, c4, c5, c6, c7, c8, c9, c10\}$ avec (3.6)
  - c1 : $u_0 > 0$
  - c2 : $v_0 > u_0$
  - c3 : $t_0 = u_0$
  - c4 : $u_1 = v_0$
  - c5 : $v_1 = t_0$
  - c6 : $u_2 = u_1 - v_1$

- $c7 : u_2 > 0$
- $c8 : \neg(v_1 > u_2)$
- $c9 : u_3 = u_2 - v_1$
- $c10 : \neg(u_3 > 0)$

L'ensemble des entiers ne pouvant être totalement représenté en machine, lors de l'utilisation d'un programme de résolution de contraintes, on est amené _a réduire cet ensemble à l'intervalle [minInt…maxInt] où les valeurs minInt et maxInt représentent respectivement le plus petit et le plus grand entier représentables par la machine.
Les domaines définis dans (3.5) s'écrivent comme suit :

$D = \{D_{u0};D_{v0} ;D_{t0} ;D_{u1} ;D_{v1} ;D_{u2} ;D_{u3}\}$ avec

- $D_{u0} = D_{t0} = D_{u2} = D_{v1} = [1\ldots maxInt/2]$
- $D_{u1} = D_{v0} = [2\ldots maxInt]$
- $D_{u3} = \{0\}$

Le CSP précédent peut s'écrit après simplification comme suit :

CSP'(X,D, C)

- $X = \{u_0, v_0, t_0, u_1, v_1, u_2, u_3\}$ (3.7)
- $D = \{D_{u0};D_{v0} ;D_{t0} ;D_{u1} ;D_{v1} ;D_{u2} ;D_{u3}\}$ avec (3.8)
  $D_{u0} = D_{t0} = D_{u2} = D_{v1} = [1\ldots maxInt/2]$
  $D_{u1} = D_{v0} = [2\ldots maxInt]$
  $D_{u3} = \{0\}$
- $C = \{c1, c2, c3, c4, c5, c6, c7, c8, c9, c10\}$ avec (3.9)
  - $c3 : t_0 = u_0$
  - $c11 : u_0 = v_1$
  - $c12 : u_0 = u_2$
  - $c13 : v_0 = 2 * u_0$
  - $c14 : u_1 = 2 * u_0$

où

- c1 et c2 sont éliminés par les domaines (3.8)
- c3 et c5 donnent c11
- c9 et c11 et $D_{u3} = \{0\}$ donnent c12
- c7 et c6 et c5 et c4 et c8 donnent c13

– c4 et c13 donnent c14

Donc on deux CSPs :

- CSP(X,D,C) (3.10)

  avec

  X défini par (3.4)

  D défini par (3.8)

  C défini par (3.6)

- CSP'(X,D,C) (3.11)

  avec

  X' défini par (3.4)

  D' défini par (3.8)

  C' défini par (3.9)

4. Implémentation sur un solveur de contrainte:

On a utilisé le GNU-prolog pour implémenter le CSP et CSP' vues dans section 3.4.

4.1. Présentation du langage prolog :

Prolog est un langage utilisé dans les domaines de l'Intelligence Artificielle et la Programmation Logique avec Contraintes. Sa syntaxe et son principe de fonctionnement sont radicalement différents de langages impératifs tels que C ou Java. Le raisonnement se rapproche plus de langages fonctionnels tels que Caml ou Lisp. Pourtant, Prolog n'est pas un langage fonctionnel. Prolog est le premier langage de programmation logique.[6]

Il existe différents outils pour vous permettre de programmer en Prolog :

- SWI Prolog : Possède un débuggeur graphique ainsi que plusieurs solveurs de contraintes
- GNU Prolog : Propose un solveur de contraintes sur domaine fini
- Sicstus Prolog (payant) : Possède des extensions en plus, dont plusieurs solveurs de contraintes

4.2. Les bases du langage :

4.2.1. Les commentaires

Les commentaires en Prolog se présentent sous deux formes :

- les commentaires multi lignes, comme en C, compris entre /* et */

  /* Ceci est un commentaire sur

plusieurs lignes */

- les commentaires sur une seule ligne, introduits par le symbole *%*

  % Un commentaire en fin de ligne

4.2.2. Les variables :

En Prolog, les variables sont représentées par un identificateur commençant par une lettre majuscule (exemple : *X, Machin, Liste*, ...).Tant qu'on n'affecte pas de valeur à une variable, on parle de « variable libre ». Une fois qu'une valeur est affectée, on parle de « variable liée ». On affecte une valeur à une variable par ce que l'on appelle le principe d'« unification ».[6]

4.2.3. Prédicats :

En Prolog, on ne parle pas de fonction, mais de « prédicat ». Un prédicat s'organise en « clauses ».

Une clause ne retourne pas de valeur : soit elle s'exécute, soit elle échoue. On affecte les valeurs des variables passées en paramètres par « unification ». Par convention, les variables en entrée sont placées à gauche et les variables en sortie sont placées à droite, mais l'unification peut se faire dans les deux sens.[6]

4.2.4. Liste :

En Prolog, on ne connait que deux choses d'une liste :

- son premier élément (la tête, ou *head*)
- le reste de la liste (la queue, ou *tail*)

Le parcours d'une liste se fait de manière récursive sur la queue.

La liste vide se note : **[]**.

Une liste constituée d'au moins un élément se note **[T|Q]**. Ici, **T** représente la tête et **Q** représente la queue. C'est aussi grâce à **[T|Q]** que l'on ajoute un élément en début de liste (par unification).

4.2.5. Les contraintes sous GNU-Prolg :

- variables sur les domaines finis

  A. **désignation** : fd_domain/3

  - **schéma d'appel** :fd_domain(+liste_de_variables_FD,+entier,+entier) ou fd_domain(?variable_FD,+entier,+entier)
  - **description** :fd_domain(L,Min,Max) ou fd_domain(X,Min,Max)
  - **exemple** : modélisation CSP du problèmes des 4 reines : fd_domain([X1,X2,X3,X4],1,4)

ou

fd_domain(X1,1,4), fd_domain(X2,1,4), fd_domain(X3,1,4), fd_domain(X4,1,4)

B. **désignation** : fd_domain/2

- **schéma d'appel** :fd_domain(+liste_de_variables_FD,+liste_d_entiers) ou fd_domain(variable_FD,+liste_d_entiers)
- **description** :fd_domain(L,Valeurs) ou fd_domain(X,Valeurs) (Valeurs : liste de valeurs)
- **exemple** : fd_domain([X1,X2,X3,X4],[1,2,3,4])

➢ Contraintes arithmétiques : Expr1 et Expr2 : deux expressions arithmétiques

A. consistance partielle

- égalité : Expr1 # = Expr2
- inégalité : Expr1 #\ = Expr2
- inférieur : Expr1 # < Expr2
- inférieur ou égal : Expr1 # =< Expr2
- supérieur : Expr1 # > Expr2
- supérieur ou égal : Expr1 # >= Expr2

B. consistance total

- égalité : Expr1 # = # Expr2
- inégalité : Expr1 #\ = # Expr2
- inférieur : Expr1 # < # Expr2
- inférieur ou égal : Expr1 # =< # Expr2
- supérieur : Expr1 # > # Expr2
- supérieur ou égal : Expr1 # >= # Expr2

➢ contraintes booléennes : Expr1 et Expr2 : deux expressions booléennes

- équivalence : Expr1 # <=> Expr2
- Non équivalence : Expr1 #\ <=> Expr2
- non : Expr1 #\ Expr2
- ou : Expr1 #\/ Expr2
- ou exclusif : Expr1 # # Expr2
- et : Expr1 #/\ Expr2

- Résolution
  - **désignation** : fd _labeling/1
  - **schéma** d'appel :ld_labeling(+liste_de_variables_FD)
  - **description** :fd_labeling(L) : affecte une valeur à chaque variable FD de L afin de satisfaire toutes les contraintes
  - **exemple :** Problème des 4 reines :
    - déclaration des 4 variables X1, X2, X3 et X4
    - écriture des contraintes sur X1, X2, X3 et X4
    - rechercher d'une solution : fd_labeling([X1,X2,X3,X4])
- Structure générale d'un programme simple (par contraintes)

  Dans le cas le plus simple, un programme pour résoudre une contrainte s'écrit en trois parties :
    - définir les domaines des variables
    - décrire les contraintes
    - labeling

    exemple :

```
probleme([X,Y,Z]) :- fd_domain(X,0,5), fd_domain([Y,Z],3,7),
X+Y #< 2*Z, fd_labeling([X,Y,Z]).


| ?- probleme(L).
L = [0,3,3] ? ;
L = [0,3,4] ? ;
```

4.3. Implémentation du CSP(X,D,C)

4.3.1. CSP non simplifié

pgcd2 : est le nom du prédicat avec critère tous les 2-chemins (passage deux fois dans le boucle While)

```
pgcd2(entrée1=U0,entrée2 = V0,U1,U2,U3,T0,sortie = V1):-
% on prend max_integer = 512
fd_domain([U0,U2,V1,T0],1,256),
fd_domain([U1,V0],2,512),
fd_domain(U3,[0]),
%c1 : u0 > 0
U0 #> 0,
%c2 : V0 > U0
V0 #> U0,
%c3 : t0 = u0
T0 #= U0,
%c4 : u1 = v0
U1 #= V0,
%c5 : v1 = t0
V1 #= T0,
%c6 : u2 = u1 - v1
U2 #= (U1 - V1),
%c7 : u2 > 0
U2 #> 0,
%c8 : not(v1 > u2)
#\(V1 #> U2),
%c9 : u3 = u2 - v1
U3 #= (U2 - V1),
%c10 : not(u3 > 0)
#\(U3 #> 0),
%c11 : u0 = v1
U0 #= V1,
%c12 : u0 = u2
U0 #= U2,
%c13 : v0 = (2 * u0)
V0 #= (2 * U0),
%c14 : u1 = 2 * u0
U1 #= (2 * U0),
fd_labeling([U0,U1,U2,U3,V0,V1,T0]).
```

Figure 3. 3 code de résolution du CSP par GNU-Prolog

```
GNU Prolog console
File  Edit  Terminal  Prolog  Help
| ?- fd_max_integer(512).

no
| ?- pgcd2(U0,U1,U2,U3,V0,V1,T0).

T0 = (sortie=1)
U0 = (entrée1=1)
U1 = (entrée2=2)
U2 = 2
U3 = 1
V0 = 0
V1 = 1 ? ;

T0 = (sortie=2)
U0 = (entrée1=2)
U1 = (entrée2=4)
U2 = 4
U3 = 2
V0 = 0
V1 = 2 ? ;

T0 = (sortie=3)
U0 = (entrée1=3)
U1 = (entrée2=6)
U2 = 6
U3 = 3
V0 = 0
V1 = 3 ?
```

Figure 3. 4 Exemple d'exécution de résolution du CSP par GNU-Prolog sans indiquer les entrées

```
GNU Prolog console
File  Edit  Terminal  Prolog  Help
| ?- pgcd2(entrée1=3,entrée2 = 6,U1,U2,U3,T0,V1).

T0 = 3
U1 = 6
U2 = 3
U3 = 0
V1 = (sortie=3)

yes
| ?- pgcd2(entrée1=3,entrée2 = 5,U1,U2,U3,T0,V1).

no
| ?- pgcd2(entrée1=3,entrée2 = 9,U1,U2,U3,T0,V1).

no
| ?-
```

Figure 3. 5Exemple d'exécution de résolution du CSP par GNU-Prolog en indiquant les entrées

4.3.2. CSP simplifié

pgcd2_s : est le nom du prédicat simplifié avec critère tous les 2-chemins (passage deux fois dans le boucle While)

```
pgcd2_s(entrée1=U0,entrée2 = V0,U1,U2,U3,T0,sortie = V1):-
% on prend max_integer = 512
fd_domain([U0,U2,V1,T0],1,256),
fd_domain([U1,V0],2,512),
fd_domain(U3,[0]),
%c3 : t0 = u0
T0 #= U0,
%c11 : u0 = v1
U0 #= V1,
%c12 : u0 = u2
U0 #= U2,
%c13 : v0  =2* u0
V0  #=2* U0,
%c14 : u1  =2* u0
U1  #=2* U0,
fd_labeling([U0,U1,U2,U3,V0,V1,T0]).
```

Figure 3. 6 code de résolution du CSP par GNU-Prolog

```
GNU Prolog console
File  Edit  Terminal  Prolog  Help
| ?- pgcd2_s(entrée1=3,entrée2 = 5,U1,U2,U3,T0,V1).

no
| ?- pgcd2_s(entrée1=3,entrée2 = 6,U1,U2,U3,T0,V1).

T0 = 3
U1 = 6
U2 = 3
U3 = 0
V1 = (sortie=3)

(31 ms) yes
| ?- pgcd2_s(entrée1=3,entrée2 = 9,U1,U2,U3,T0,V1).

no
| ?-
```

Figure 3. 7 Exemple d'exécution de résolution du CSP simplifié par GNU-Prolog en indiquant les entrées

GNU Prolog console

File Edit Terminal Prolog Help

```
| ?- pgcd2_s(U0,U1,U2,U3,V0,V1,T0).

T0 = (sortie=1)
U0 = (entrée1=1)
U1 = (entrée2=2)
U2 = 2
U3 = 1
V0 = 0
V1 = 1 ? ;

T0 = (sortie=2)
U0 = (entrée1=2)
U1 = (entrée2=4)
U2 = 4
U3 = 2
V0 = 0
V1 = 2 ? ;

T0 = (sortie=3)
U0 = (entrée1=3)
U1 = (entrée2=6)
U2 = 6
U3 = 3
V0 = 0
V1 = 3 ?
```

Figure 3. 8 Exemple d'exécution de résolution du CSP' simplifié par GNU-Prolog sans indiquer les entrées

On voie bien que les résultats des deux codes pour CSP et CSP' sont les mêmes, où les figures 3.4 et 3.8 représentent des requêtes avec des entrées inconnues (u et v), dans ce cas prolog donne plusieurs résultats avec les données de test et ce en appuyant sur la touche point-virgule (;) au clavier.

Concernant les figures 3.5 et 3.7, on indique les entrées (u et v) et on attend au prolog d'indiquer si ces deux entrées ont un pgcd ou non, si oui, il affiche (Yes) les données de test.

5. Conclusion

Dans ce chapitre on a essayé de modéliser un code source d'un programme par un CSP en basant sur les critères de couvertures de graphe de contrôle représentant de code source, et en fin on a implémenter le modèle obtenu en GNU-prolog.

CONCLUSION GENERALE

Au cours de ce mémoire, nous avons présenté les différents types de test de logiciels. Nous avons aussi pu formaliser d'une manière concrète avec l'une des méthodes informatiques de modélisation des problèmes décisionnels sous contraintes, en utilisant le solveur de contraintes GNU-Prolog. Cette approche déclarative permet de résoudre des problèmes combinatoires variés. Les utilisateurs décrivent leur problème en posant des contraintes sur les valeurs que peuvent prendre les différentes variables composant le problème.

Nous avons réalisé un test de logiciel en basant sur un CSP permettant de générer les données de test selon les chemins d'instruction.

Ces travaux ouvrent diverses perspectives : améliorer le parcours du graphe de flot de contrôle, étendre le langage traité, ne générer que des entrées réalistes, prendre en compte des spécifications, améliorer le traitement des appels de méthode, étendre l'approche pour couvrir d'autres critères, augmenter la puissance de déduction des opérateurs, et permettre une approche orientée but.

Enfin, nous espérons que notre travail puisse apporter l'aide et la satisfaction des développeurs et que ce travail a fait l'objet d'une expérience intéressante, qui nous a permis d'améliorer nos connaissances et nos compétences dans le domaine de la programmation par contraintes et le test de logiciels.

BIBLIOGRAPHIE

[1] Besnik SELJIMI ; Test de logiciel synchrone avec PLC, thèse de doctorat ; université Joseph Fourier Grenoble 1, 2009.

[2] http://www.i3s.unice.fr/~malapert/thesis.html, Arnauld Malapert, Techniques d'ordonnancement d'atelier et de fourn sur la programmation par contraintes

[3] www.liris.cnrs.frhttp://liris.cnrs.fr/christine.solnon/Site-PPC/session2/e-miage-ppc-sess2.htm

[4] www.memoireonline.comhttp://www.memoireonline.com/07/15/9207/m_Resolution-dun-probleme-de-satisfaction-de-contraintes-sur-les-intervalles.html

[5] www.nbs-system.comhttps://www.nbs-system.com/blog/tests-en-boite-noire-grise-ou-blanche-quelles-differences.html

[6] www.pcaboche.developpez.com/http://pcaboche.developpez.com/article/prolog/ /

[7] Christine SOLNON, »AntColonyOptimization and ConstraintProgramming »,2010, chapitre 3 ,3.1 Whatis a constraint?

[8] Sami Taktak ; Test et Validation du Logiciel « Test Structurel » ; support de cours ; Centre d'Étude et De Recherche en Informatique et Communications Conservatoire National des Arts et Métiers

[9] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. Programmation et Vé-rication des Systèmes Réactifs : le langage LUSTRE. Technique et Science Informatique, 10(2), 1991.

[10] AbdesselamLakehal. Critères de Couverture Structurelle pour les Programmes Lustre. Thèse, Université Joseph Fourier, Grenoble, France, Septembre 2006.

[11] Bruno Marre and AgnèsArnould. Test SequencesGenerationfrom LUSTRE Descriptions :GATeL. In Automated Software Engineering, pages 229-237, 2000.

[12] C. Mazuet. Stratégies de Test pour des Programmes Synchrones - Application au Langage LUSTRE. Thèse, Toulouse, France, Decembre 1994.

[13] V. Papailiopoulou, B. Seljimi, and I. Parissis. Revisiting the Steam-Boiler Case StudywithLutess : Modeling for Automatic Test Generation. In 12th European Workshop on DependableComputing, Toulouse, France, Mai 2009.

[14] Matthieu Petit. Test statistique structurel par résolution de contraintes de choix probabiliste. Thèse, Université de Rennes I, Rennes, France, Juillet 2008.

[15] L. Py, B. Legeard, and B. Tatibouët. Évaluation de spécifications formelles en programmation logique avec contraintes ensemblistes - application à l'animation de spécications B. In AFADL, Grenoble, France, Janvier 2000.

[16] B. Seljimi and I. Parissis. Test de logiciels synchrones : apports de la programmation par contraintes. In Approches Formelles dans l'Assistance au Développement de Logiciels, Namur, Belgique, Juin 2007.

[17] BesnikSeljimi and IoannisParissis. Using CLP to AutomaticallyGenerateTest Sequences for Synchronous ProgramswithNumeric Inputs and Outputs.In 17th International Symposium on Software Reliability Engineering, pages105-116, Raleigh, USA, Novembre 2006.

[18] BesnikSeljimi and IoannisParissis. AutomaticGeneration of Test Data Generators for Synchronous Programs :Lutess V2. In Workshop on Domain Specific Approaches to Software Test Automation, pages 8-12, Dubrovnik, Croatia, Septembre 2007.

[19] R.A. DeMillo and A.J. Offut. Constraint-basedautomatic test data generation. IEEE Transaction on Software Engineering, 17(9) :900–910, septembre 1991.

[20] J.W. Duran and S.C. Ntafos. An evaluation of randomtesting. IEEE Transactions on Software Engineering, SE-10 :438–444, July 1984.

[21] S.-D. Gouraud. Application de la génération aléatoire de structures combinatoires au test de logiciel. In Approches Formelles dans l'Assistance au développement de Logiciels (AFADL), 2001.

[22] S.-D. Gouraud. AuGuSTe : un outil pour le test statistique. In Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL), pages 337–340, 2004.

[23] Alessandra Di Pierro and Herbert Wiklicky. An OperationalSemantics for Probanilistic Concurrent ConstraintProgramming. International Conference on Computer Langages - IEEE Cmputer Society Press, 1998.

[24] Arnaud Gotlieb ; Test structurel de programmesimpératifs ; support de cours master 1 ; IRISA / LANDE ; 2005/2006

[25] S.-D. Gouraud. AuGuSTe : Utilisation des Structures Combinatoires pour le Test Statistique, thèse de doctorat ; université de Paris ; 2005

[26] M.-C. Gaudel, B. Marre, F. Schlienger, and G. Bernot. Précis de génie logiciel. Masson, 1996. ISBN 2-225-85189-1.

ملخص

في هذه المذكرة ركزنا عل تحسين مشكلة التحقق من البرامج عن طريق برمجة القيود، عند إنشاء تطبيق أو برنامج فإنه يمر بعدة اختبارات منها نوعين أساسيين "اختبار هيكلي، اختبار وظيفي".

استعملنا مجموعة تعليمات برنامج "القاسم المشترك الأكبر" وقمنا بنمذجته عن طريق "برمجة مشاكل القيود" ، بعد النمذجة، استعملنا لغة البرمجة GNU-Prolog للتحقق من النتائج.

كلمات مفتاحية: "برمجة القيود"،"برمجة مشاكل القيود"، اختبار البرمجيات، البرلوغ، اختبار هيكلي.

Abstract

In our work we have focused on the optimization of a problem of software test with constraint programming. When a software or an application is created, it is vital to carry out several types of tests, there are two main types of tests: " structural test, and functional test ".

We have used a source code of a program "PGCD", for modeling them with CSP «constraints satisfaction problems » after modeling, wehave used the language GNU-prolog, for implementing the model obtained and testing the results.

Keywords :Constraint programming, CSP «constraints satisfaction problems», Software test, Prolog, structural test.

Résumé

Dans notre travail nous avons concentré sur l'optimisation d'un problème de test de logiciel avec la programmation par contraintes. Lorsqu'un logiciel ou une application sont créés, il est vital de réaliser plusieurs types de tests, il existe deux types principaux « test structurel, test fonctionnel ».

nous avons utilisé un code source d'un programme « PGCD » pour le modéliser par CSP « problèmes de satisfaction de contraintes » après la modélisation, on a utilisé le langage GNU-prolog pour implémenter le modèle obtenu et tester les résultats.

Mots-clés : Programmation par contraintes, CSP « problèmes de satisfaction de contraintes », Test logiciel, GNU-Prolog, test structurel.