UNIVERSITÉ DU QUÉBEC À MONTRÉAL

UNE IMPLÉMENTATION DE LA PROGRAMMATION PAR CONTRAT EN NIT, UN LANGAGE À OBJET

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

FLORIAN DELJARRY

DÉCEMBRE 2020

REMERCIEMENTS

Avant toute chose, je souhaite remercier Jean Privat, mon directeur de recherche et professeur, pour sa patience, son aide et ces enseignements, qu'il a su me prodiguer tout au long de cette maîtrise, ainsi que pour les innombrables cafés. Merci aussi à l'ensemble des professeurs du département informatique de l'UQAM que j'ai eu plaisir et le privilège de côtoyer.

Je tiens aussi à remercier ma compagne, Mélanie, qui a fait preuve d'un soutien indéfectible. Elle a su m'accompagner, me guider et surtout me supporter durant toutes les étapes de cette maîtrise. Merci aussi à l'ensemble de mes amis et camarades, Quentin, Lâm, Julien, Morganne, Oswald, Jehan, Aimeric et tous les autres que je ne puis citer qui ont su me faire oublier le temps de soirées cette maîtrise. Merci aussi à Adrien, qui m'a encouragé et rappelé de prendre des pauses durant toute la rédaction de ce mémoire.

Pour finir, je remercie mes parents qui m'ont toujours poussé, encouragé et soutenu tout au long de ma vie, et sans qui rien de tout cela n'aurait été possible. Même si des remerciements ne suffiront jamais à exprimer toute la gratitude que j'éprouve à leur égard.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

RÉSUMÉ

Le paradigme de programmation par contrat, ou plus connu sous le nom de *design by contrat*™ en anglais, se base sur le concept de contrat pour venir renforcer la fiabilité des systèmes logiciels. L'idée derrière cette méthodologie est de fournir au programmeur un moyen pour définir de façon programmative les obligations des différents éléments du langage telles que les méthodes, interfaces et classes afin de vérifier l'adéquation de l'implémentation envers sa spécification.

Ce mémoire fournit une étude de diverses approches d'implémentation (ISE Eiffel, dmd, Code Contracts, Cofoja, AspectJML et OpenJML). Nous abordons ainsi les différentes constituantes de ces implémentations tant sur le plan syntaxique et sémantique que sur la mise en œuvre pour procéder à la vérification, afin d'offrir la prise en charge de la programmation par contrat dans un langage à objet. Nous abordons ensuite une proposition d'implémentation pour le langage à objet Nit. Contrairement aux approches existantes, notre proposition met en œuvre une représentation basée sur l'utilisation de plusieurs points d'entrée. Cette approche permet ainsi de faire coexister plusieurs versions d'une même méthode avec un niveau de vérification différent. Le principal avantage est la possibilité de pouvoir déterminer statiquement la vérification nécessaire pour chaque site d'appel, permettant ainsi de diminuer l'impact dynamique.

L'ensemble de nos expérimentations réalisées sur un ensemble de programmes Nit, tendent à montrer que notre proposition offre un coût d'utilisation négligeable comparé à l'évaluation des conditions, environ 4%.

Mots-clés : programmation par contrat, compilation, langage à objet.

INTRODUCTION

Initialement introduit dans le langage Eiffel (ECMA International, 2006) et définit par B.Meyer (Meyer, 1992), la programmation par contrat est un paradigme de programmation permettant d'augmenter la fiabilité des systèmes logiciels. Pour y parvenir, la programmation par contrat ajoute la possibilité au langage de programmation de définir des entités appelées contrats. Basé sur la métaphore des contrats commerciaux, ces contrats permettent de régir les interactions entre les différents composants logiciels. Ces interactions mettent deux parties en oeuvre, un client et un fournisseur, le contrat permet ainsi de représenter les avantages et les exigences de chacune des parties.

Exprimé à l'aide de prédicats, souvent définit dans le même langage que le reste du programme, les clauses des contrats permettent d'exprimer de manière claire et non ambiguë les règles que doivent respecter chaque élément du programme.

Dans un contexte de programmation orienté objet, la programmation par contrat se caractérise par la présence de trois types de contrat. Les pré-conditions permettant de définir les exigences qui doivent être satisfaites à l'appel d'une méthode. Les post-conditions permettant de définir les exigences qui doivent être satisfaites après l'appel d'une méthode. Ainsi que les invariants de classe permettant de définir les contraintes appliquées aux objets.

De nos jours, nous trouvons d'autres langages que Eiffel qui viennent de manière native supporter la programmation par contrat, c'est notamment le cas avec des langages comme D (D Language Foundation, 2020) ou Ada (AdaCore, 2020). D'un autre côté nous trouvons aussi un ensemble d'outils externes permettant

d'apporter la programmation par contrat à des langages qui en sont dépourvus. C'est notamment le cas avec AspectJML (Rebêlo *et al.*, 2014), OpenJML (Cok, 2014) basé sur le langage de spécification JML (Leavens *et al.*, 1999) pour Java, Cofoja (Minh Lê, 2016) (Java), ou encore Code Contracts (Microsoft, 2018) (C#).

Cette maîtrise a pour objectif de venir répondre à deux problématiques qui sont : « Comment est mis en œuvre le support de la programmation dans les langages à objet existant ? » et « Comment mettre en oeuvre la programmation par contrat dans le langage Nit afin de prendre en charge les diverses spécificités et contraintes du langage ? ». De ces problématiques découle un ensemble d'objectifs. Tout d'abord, il est nécessaire d'étudier de manière générale les différents aspects de la programmation par contrat. Il est ensuite nécessaire de fournir une étude de l'ensemble des moyens utilisés dans la littérature pour apporter un support de méthodologie dans les langages à Objet, tant sur un plan syntaxique et sémantique que sur la représentation mise en oeuvre pour effectuer l'évaluation des contrats. Outre les objectifs d'étude, cette maîtrise a aussi pour ambition d'intégrer une prise en charge native et officielle de la programmation par contrat dans le langage Nit.

Ce mémoire apporte comme principales contributions, l'étude de différentes implémentations de la programmation par contrat pour des langages à objet compilé et statiquement typé comme c'est le cas pour Nit, ainsi que la mise en oeuvre d'une implémentation pour le langage Nit. Cette implémentation doit notamment prendre en compte des spécificités propres au langage Nit tel que le raffinement de classe (Privat et Ducournau, 2005). Dans ce mémoire, nous proposons aussi de nouvelles approches de résolution des problématiques propres à la programmation par contrat, notamment avec l'utilisation de facettes permettant d'adapter statiquement la vérification des contrats en fonction des règles de vérification. Nous proposons aussi une méthodologie de résolution des problèmes d'héritage

des contrats en se basant sur l'utilisation de la linéarisation de l'appel à *super*.

Dans le Chapitre 1, nous présentons tous d'abord les fondements et les avantages apportés par la programmation par contrat, nous effectuons une comparaison avec des approches de développement similaires, ainsi que présentons un ensemble de règles destiné à l'écriture de contrat d'un haut niveau de qualité. Le Chapitre 2 présente les différentes approches syntaxiques et sémantiques mises en oeuvre dans l'approche étudiée. Le Chapitre 3 se concentre sur la représentation interne mise en oeuvre par les différentes approches afin de résoudre des problématiques telles que la gestion de l'héritage ou le verrouillage de l'évaluation des contrats. Nous détaillons et discutons dans le Chapitre 4 la mise en oeuvre de notre proposition. Le Chapitre 5 aborde la validation en termes d'utilisation et de performance de notre proposition. Dans un premier temps afin de valider l'utilisation, nous procédons à la conversion d'assertions et à l'introduction de nouveaux contrats. Nous procédons ensuite à une comparaison afin de mesurer l'impact engendré sur les performances de ces modifications. Dans un second temps nous utilisons un micro-benchmark afin de mettre en perspective les performances de notre proposition comparée aux solutions étudiées.

CHAPITRE I

CONCEPTION ET PROGRAMMATION PAR CONTRAT

Ce chapitre présente la programmation par contrat et les concepts qui y sont associés. Dans la Section 1.1, nous abordons les différents concepts derrière la programmation par contrat. La Section 1.2, met en avant l'utilité de la mise en oeuvre d'une telle technique. La Section 1.3 quant à elle met en parallèle l'utilisation de la programmation par contrat par rapport à certaines techniques de développement tel que la programmation défensive ainsi que la mise en oeuvre de tests unitaires. La Section 1.4 aborde, ensuite, la représentation des contrats dans UML à l'aide du langage de contrainte OCL. Enfin, la Section 1.5 définit un ensemble de bonnes pratiques afin d'écrire des contrats avec un haut niveau de qualité.

1.1 Programmation par contrat

La programmation par contrat (*design by contract*™ DBC en anglais) est un paradigme de programmation défini par Meyer (Meyer, 1992). Initialement, la programmation par contrat a été introduite avec le langage de programmation Eiffel (ECMA International, 2006; Meyer, 1992) l'objectif principal étant d'aider à la mise en oeuvre de système logiciel fiable. Meyer (Meyer, 1997) définit la fiabilité comme la somme de deux paramètres, la justesse qui représente la capacité d'un logiciel à exécuter ses tâches telles que définies dans la spécification, et la

robustesse qui est la capacité à réagir de manière appropriée à des conditions non prévues dans la spécification. Ces définitions de justesse et de robustesse sont relatives. En effet, il est impossible de déterminer si un logiciel est fiable sans connaître les spécifications qu'il doit respecter. L'aspect des spécifications est alors primordial pour arriver à un logiciel fiable.

C'est sur cette notion que se base la programmation par contrat. En effet, le mécanisme clé de la programmation par contrat est la mise en oeuvre de ces spécifications sous la forme d'entités appelées *contrats*. Ce nom de *contrat* tire son origine de l'analogie avec les contrats qui régissent les relations commerciales entre un client et un fournisseur. Lorsqu'un client et un fournisseur signent un contrat, chaque partie s'engage à remplir les obligations qui lui incombent et en retour chaque partie attend un bénéfice. C'est exactement la même chose avec la programmation par contrat. Nous retrouvons un contrat qui permet de régir les interactions entre deux opérations, d'un côté nous avons l'opération appelante qui représente notre client et de l'autre l'opération appelée qui représente notre fournisseur. L'appelant doit ainsi satisfaire les conditions définies par l'opération appelée et en retour l'opération appelée doit satisfaire la condition pour laquelle elle a été appelée. Les contrats définissent ainsi la relation entre deux composants coopérants, le client et le fournisseur permettant ainsi d'exprimer les obligations et les bénéfices de chaque partie. Ces contrats se matérialisent par la mise en oeuvre de pré-conditions qui spécifient les conditions qui doivent être respectées par le client pour que le fournisseur exécute correctement sa tâche et de post-conditions qui spécifient les conditions qui doivent être respectées par le fournisseur après son opération.

Ce concept de pré/post-condition remonte aux travaux de Hoare (Hoare, 1969) avec le triplet de Hoare. Ces triplets décrivent comment l'exécution d'un morceau de code change l'état du système. Le triplet de Hoare est représenté sous la forme

suivante :

$$P\{C\}Q$$

Où **P** et **Q** sont des prédicats représentant l'état du système, et **C** représente un morceau de code (suite d'une ou plusieurs opérations). Nous avons **P**, notre pré-condition, qui caractérise le contexte initial dans lequel doit être exécuté notre morceau de code **C** et **Q**, notre post-condition, qui définit l'état final après l'exécution du morceau de code **C**.

Meyer (Meyer, 1992) définit trois types de contrat pour la programmation orientée objet :

- Les pré-conditions permettent de spécifier les conditions qui doivent être respectées pour que la méthode puisse être appelée. Les pré-conditions sont une obligation pour la méthode appelante et une garantie pour la méthode appelée.
- Les post-conditions permettent de spécifier les conditions qui doivent être respectées par l'appelé. Les post-conditions sont une obligation pour l'appelé et une garantie pour l'appelant.
- Les invariants de classe permettent de spécifier les contraintes d'intégrité des instances d'une classe. Les invariants de classe doivent être satisfaits par chaque instance, ils peuvent être vus à la fois comme l'association d'une pré-condition ainsi qu'une post-condition pour l'ensemble des méthodes de la classe.

1.2 Programmation par contrat : avantages

L'utilisation de la programmation par contrat permet entre autres de guider l'analyse, la conception, la documentation et le test lors du développement de logiciels (Meyer, 1997, p.389) (Mitchell *et al.*, 2001, p.137). Comme nous l'avons dé-

fini précédemment, cette méthodologie de programmation se base sur l'ajout de contrats définissant explicitement les spécifications des éléments auxquels ils sont rattachés : interfaces, classes et méthodes. Ces contrats offrent plusieurs bénéfices.

Tout d'abord les contrats offrent un excellent support à la documentation ; en effet, les programmeurs peuvent utiliser les informations fournies par les contrats afin de vérifier et de garantir l'exactitude des différents éléments de leurs programmes. De plus, si nous comparons les contrats à un autre support de documentation que sont les commentaires écrits en langage naturel, les contrats offrent deux avantages majeurs. Les contrats sont plus explicites. Effectivement, les contrats sont généralement écrits dans le même langage que le logiciel (ECMA International, 2006; Minh Lê, 2016; Abercrombie et Karaorman, 2002; D Language Foundation, 2020; Bartetzko *et al.*, 2001) ou bien à l'aide de langage formel (Leavens *et al.*, 1999; Rebêlo *et al.*, 2014; Cok, 2014) ; il n'y aura alors aucune ambiguïté pour comprendre la spécification sur laquelle est basée l'implémentation. Les contrats fournissent aussi une documentation plus fiable. En effet, les contrats peuvent être vérifiés à l'exécution ; il est alors possible de détecter la conformité du contrat et de l'implémentation de l'opération. Cela aura pour effet d'éviter que les contrats ne se retrouvent désynchronisés de la réalité de l'implémentation.

La programmation par contrat joue également un rôle majeur dans les phases de tests et de débogages. Lors de la vérification dynamique des contrats, si l'un d'entre eux n'est pas respecté cela signifiera la présence d'une défaillance dans la mise en oeuvre de l'implémentation. La violation d'une pré-condition explicite un problème dans la mise en oeuvre du client alors qu'une violation d'une post-condition ou d'un invariant de classe explicite un problème dans la mise en oeuvre du fournisseur. Notons aussi que la violation d'une pré-condition, d'une post-condition ou d'un invariant de classe peut éventuellement signifier une défaillance dans le contrat lui-même. Grâce à cette règle, le programmeur peut alors

facilement identifier de manière certaine l'élément en cause.

Les contrats offrent aussi un support idéal pour la mise en oeuvre de la programmation orienté objet. Effectivement, la mise en oeuvre de la programmation par contrat dans les langages à objet s'accompagne souvent de sémantique spécifique afin de prendre en charge l'héritage des classes. En effet, lors du développement de logiciels à l'aide de la programmation orientée objet, les programmeurs sont souvent amenés à venir étendre des éléments existants afin de les adapter à un nouveau contexte. Cette extension des fonctionnalités passe par la mise en oeuvre de l'héritage. La programmation par contrat définit ce lien d'héritage comme la matérialisation d'une sous-traitance (Meyer, 1992), où le travail initialement défini dans une super classe est délégué à une nouvelle implémentation fournie dans la sous-classe. Ce lien de sous-traitance doit alors respecter l'ensemble des spécifications initiales, cela dans le but de garantir que peu importe l'implémentation sélectionnée celle-ci respectera au moins les spécifications prévues à l'origine. Cette sémantique est traitée plus en détail dans la Section 2.2.5.

1.3 Programmation par contrat : méthode similaire

La programmation par contrat trouve des similitudes dans différentes pratiques de développement comme la programmation défensive ainsi que la mise en oeuvre de test unitaire.

1.3.1 Programmation par contrat et programmation défensive

L'objectif principal de la programmation dite défensive est d'encourager le programmeur à protéger chaque module logiciel par autant de contrôle que possible. Si nous comparons cette approche à la programmation par contrat, tous deux ont pour objectif final d'améliorer la fiabilité des systèmes logiciels. En effet, les deux approches reposent souvent sur un mécanisme d'assertion généralement dé-

finie avec des conditions identiques. Toutefois, Meyer (Meyer, 1992) évoque que la programmation défensive entraîne souvent une mise en oeuvre redondante des vérifications du côté du client et du fournisseur. Cette redondance introduit alors plus de code, ce qui a pour finalité d'augmenter la complexité totale du logiciel. Bien que similaire en apparence, la programmation par contrat se base sur le principe qu'un élément logiciel doit être considéré comme une implémentation destinée à satisfaire une spécification bien définie (Meyer, 1992). La programmation par contrat offre ainsi une méthodologie de développement fournissant une approche plus systématique pour traiter la mise en oeuvre des spécifications dans l'implémentation. La programmation par contrat se focalise ainsi sur l'aspect de vérification des spécifications, alors que la programmation dite défensive est utilisée dans le but de traiter des cas spécifiques. En effet, la programmation par contrat ne devrait jamais être utilisée comme un mécanisme permettant de prendre en charge des cas spécifiques. Comme le fait remarquer Liskov (Liskov et Guttag, 2001), les vérifications mises en oeuvre à l'aide de la programmation défensive ne devraient pas être désactivées à moins d'avoir prouvé que les erreurs ne peuvent jamais survenir, or la programmation par contrat a pour finalité d'être désactivée dans la version finale du logiciel (Meyer, 1997). L'utilisation de la programmation par contrat ne se substitue donc pas à l'utilisation de la programmation défensive et inversement.

1.3.2 Programmation par contrat et test unitaire

Tout comme la programmation par contrat, les tests unitaires sont un mécanisme permettant de s'assurer de la conformité de l'implémentation vis-à-vis de la spécification pour laquelle elle a été mise en oeuvre. De manière globale, les tests unitaires peuvent mettre en oeuvre de manière similaire les pré-conditions qui sont utilisées dans le but de vérifier l'exactitude entre les différents composants du programme et les post-conditions permettant de vérifier le comportement des

composants de manière individuel. Toutefois malgré des similitudes, ces deux mécanismes ont une porté et un objectif différent. Tout d'abord, les tests unitaires sont exécutés uniquement lors de la phase de tests, les contrats quant à eux font partie intégrante du code, ils sont vérifiés à chaque appel. Les contrats permettent ainsi de raisonner de manière globale sur un ensemble de possibilité, quant à eux les tests unitaires se concentrent sur un ensemble restreint de situations sélectionnées.

Toutefois comme le fait remarquer Meyer (Meyer, 1997, Chap.12) la mise en oeuvre de la programmation par contrat ne veut pas dire que l'implémentation est exempte de tout défaut, mais seulement que si un défaut survient le programmeur pourra identifier plus facilement l'élément qui en est à l'origine (client, fournisseur). C'est pour cela qu'il est nécessaire de venir mettre en oeuvre un ensemble de tests afin de vérifier le bon fonctionnement de l'implémentation.

1.4 Modélisation et contrat

Comme nous l'avons évoqué précédemment, la programmation par contrat est un outil qui aide à la mise en oeuvre de logiciels fiables. Lors du développement d'un logiciel plusieurs phases sont réalisées en amont de la mise en oeuvre de l'implémentation. Nous retrouvons notamment la phase d'analyse et de conception. La mise en oeuvre de ces phases se fait généralement à l'aide de langage de notation standardisé, l'un des langages les plus répandus est la notation UML (Unified Modeling Language, Langage de Modélisation Unifié en français). La notation UML est un langage visuel constitué d'un ensemble de diagrammes permettant de représenter les différents aspects du logiciel. L'un des diagrammes les plus utilisés lors du développement d'un logiciel, à l'aide d'un langage orienté objet, est le diagramme de classes. Il permet de présenter les classes et les interfaces des systèmes ainsi que les différentes relations entre celles-ci. UML inclut un mécanisme afin de

fournir au développeur le moyen d'exprimer les contraintes qui seront appliquées aux différents composants. Ces contraintes peuvent être formulées sous différentes formes : en langage naturel, grâce à un langage de programmation (exemple Java), à l'aide d'une notation mathématique ou en utilisant le langage *Object Constraint Language* aussi connu sous le nom d'OCL (Object Management Group Inc, 2012). Visuellement ces contraintes peuvent se présenter sous la forme d'une expression mise entre accolades *{...}* accrochée à un élément du modèle ou bien sous la forme d'un élément séparé Figure 1.1.

Le langage de contraintes OCL est un composant clé d'UML pour exprimer les restrictions sur un modèle ou un système (Warmer et Kleppe, 2003). Comme le montre Jilani, Iqbal, Khan et Usman (Jilani *et al.*, 2019) OCL a fait l'objet de nombreux travaux depuis sa création. Pour notre étude nous nous focalisons sur l'aspect de représentation des contrats à l'aide d'OCL sur un diagramme de classes. Pour ce faire nous introduisons l'exemple d'une classe *ListeChaine* qui représente la définition d'une liste simplement chaînée avec la mise en oeuvre des contrats (Figure 1.1). Cette classe introduit deux attributs, *premier_maillon* qui représente le maillon de tête de notre liste c'est-à-dire le dernier maillon ajouté à la liste et *taille* qui représente le nombre d'éléments dans notre liste, notons que l'attribut *taille* est manipulé à l'aide d'un mutateur privé et d'un assesseur public. La classe introduit aussi une requête *non_vide* afin de savoir si la liste contient des éléments ou non, cette requête définit une post-condition pour expliciter que le résultat retourné est en adéquation avec le fait que la taille soit supérieure à zéro. La classe *ListeChaine* introduit aussi deux commandes, *ajouter* et *depiler*. La première permet d'ajouter un élément à la liste, et définit deux post-conditions, une pour indiquer qu'après l'appel la taille sera supérieure de un comparée à la valeur avant l'exécution, et une autre pour définir que l'élément ajouté correspond bien au dernier élément stocké. L'autre commande *depiler* permet de faire l'opé-

ration inverse et de retirer le dernier élément ajouté à la liste et de retourner cet élément. Elle introduit une pré-condition qui définit que cette commande ne peut être appelée sur une liste vide, et deux post-conditions, une qui indique qu'après l'appel la taille sera inférieure de un comparée à la valeur avant l'exécution, et une autre qui indique que l'élément retourner est égal à la valeur du dernier élément avant l'exécution. La classe *ListeChaine* définit aussi un invariant de classe pour spécifier que les instances de cette classe ne peuvent pas avoir une taille inférieure à zéro.

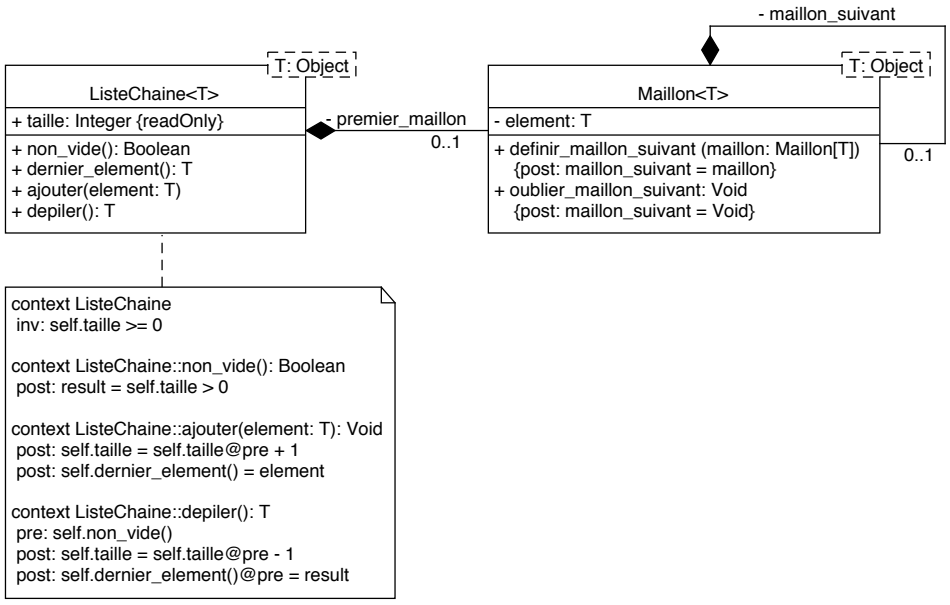

Figure 1.1 Diagramme de classe *ListeChaine* avec OCL.

Comme nous pouvons le voir dans la Figure 1.1, une expression OCL est toujours attachée à un contexte qui peut être soit un type soit une opération. Il permet de lier la contrainte à l'élément concerné. La définition de ce contexte peut se faire de manière explicite, identifiée à l'aide du mot-clé *context* comme présenté avec les contrats de la classe *ListeChaine* ou bien de manière implicite à l'aide

de crochets *{...}* accrochée à l'élément qui représente le contexte comme défini avec la commande *oublier_maillon_suivant* de la classe *maillon*. On retrouve ensuite différents stéréotypes représentant le type de contrainte dans le cas des contrats *inv*, *pre*, *post* qui représente respectivement les invariants de classe, les pré-conditions et les post-conditions. Ces stéréotypes peuvent être suivis par un nom de contrainte optionnel. On retrouve ensuite la condition de la contrainte exprimée sous la forme d'une expression booléenne. OCL offre différentes sémantiques additionnelles afin de pouvoir expliciter les différents types de condition. Dans les conditions introduites dans la Figure 1.1 nous retrouvons la possibilité de faire référence avec *result* au retour d'une méthode *@pre* qui permet de faire référence à une valeur d'un élément avant l'exécution de la méthode ainsi que *self* qui désigne l'objet auquel appartient le contexte. Les contrats définis à l'aide d'une condition utilisant *result* et *@pre* seront étudié plus en détail dans la Section 2.2.4.

1.5 Écriture de contrat

Les contrats ont pour but d'être utilisé comme méthode de support et de vérification à la mise en oeuvre de système logiciel. Il est alors primordial de définir des contrats avec un haut niveau de qualité afin de garantir leur pertinence. Une approche proposée par Hunt et Thomas dans leur livre (Hunt et Thomas, 2000, p. 140) est de définir les contrats de telle manière à définir des pré-conditions fortes et des post-conditions faibles. L'objectif étant de fournir une définition stricte des attendus tout en promettant le moins possible.

Avant d'aller plus loin dans l'analyse de la conception de contrats de qualité il est nécessaire de définir certains termes utilisés.

— Commande : représente les méthodes qui ont pour objectif d'effectuer des modifications sur l'objet sur lequel elles sont appelées. Selon Meyer (Meyer, 1997) une commande ne devrait jamais retourner de valeur.

— Requête : représente les méthodes qui ne doivent pas modifier d'objets visibles de l'extérieur elles doivent se contenter de retourner une valeur. Il existe deux types de requête :
  — Requête élémentaire : représente une requête de bas niveau, exemple pour une classe *ListeChaine* la requête élémentaire pour savoir si la liste n'est pas vide est $taille > 0$
  — Requête dérivé : représente une requête qui définit un concept de haut niveau, exemple pour une classe *ListeChaine* la requête dérivée pour savoir si la liste n'est pas vide est *non_vide*

Mitchell et McKin (Mitchell *et al.*, 2001) utilisent ces concepts pour définir un ensemble de six principes permettant la mise en oeuvre de contrat de qualité.

1. Séparer les commandes des requêtes afin de pouvoir exprimer les contrats uniquement à l'aide des requêtes dans le but d'éviter tout effet de bord. Ce principe se base sur le postulat que poser une question sur l'état d'un système ne devrait pas avoir d'effet observable sur ce dernier. Dans la Figure 1.1 la classe *ListeChaine* introduit une requête *non_vide* ainsi que deux commandes *ajouter* et *depiler*.
2. Définir des requêtes dérivées composées à l'aide de requêtes de base afin de représenter des concepts de plus hauts niveaux. L'objectif est de fournir au programmeur des moyens explicites et cohérents pour connaître l'état du système. La classe *ListeChaine* dans la Figure 1.1 introduit la requête dérivée *non_vide* pour savoir si la liste est vide.
3. Chaque requête dérivée doit définir une post-condition composée de l'ensemble des requêtes élémentaires et dérivées qu'elle représente. L'objectif est de garantir que la requête dérivée respecte bien l'ensemble des requêtes qu'elle étend. La requête dérivée *non_vide* introduit une post-condition

exprimée à l'aide de la requête de base *taille > 0*.

4. Chaque commande doit définir une post-condition qui détermine quelles requêtes élémentaires sont affectées par l'exécution de la commande. L'objectif est d'expliciter les effets visibles de chaque commande. La commande *ajouter* introduit deux post-conditions, une pour exprimer que la taille va augmenter, et une autre pour exprimer que la liste va contenir un nouvel élément.
5. Chaque requête ou commande doit définir une pré-condition adaptée pour définir quand il est possible de l'exécuter. La commande *depiler* définit une pré-condition pour informer que l'appel de cette commande ne peut être effectué sur une liste vide.
6. Chaque classe doit définir des invariants pour indiquer les propriétés qui doivent être maintenues. Notez que même si la vérification de l'invariant est présente dans la post-condition d'une commande il convient de laisser les deux vérifications afin de faciliter la compréhension de quelle commande affecte quoi. La classe *ListeChaine* introduit l'invariant *taille >= 0* afin de s'assurer que la taille de la liste ne puisse être inférieure à zéro.

1.6 Conclusion

En conclusion, la programmation par contrat est une méthodologie qui vise à améliorer la fiabilité des systèmes logiciels. Pour ce faire elle se concentre sur la mise en oeuvre de contrat de différents types, invariant de classe, pré-condition et post-condition permettant ainsi de représenter les spécifications des éléments (interface, classe, méthode) auxquels ils sont rattachés. Dans le prochain chapitre nous abordons plus en détail l'aspect syntaxique et sémantique des contrats à travers les différentes solutions étudiées.

CHAPITRE II

SYNTAXE ET SÉMANTIQUE DES CONTRATS

Ce chapitre fournit une vue globale sur la syntaxe et la sémantique des contrats dans un langage à objets. Dans la Section 2.1 nous abordons les différentes approches de mise en oeuvre syntaxique des contrats, notamment grâce à des mots-clés, commentaires, annotations, méthodes dédiées, ainsi que des aspects. La Section 2.2 quant à elle aborde la sémantique des différents types de contrats (invariant de classe, pré-condition, post-condition) ainsi que les sémantiques spécifiques associées aux post-conditions. Bien qu'un grand nombre de propositions de programmation par contrat existent dans la littérature (AdaCore, 2020) (Censi, Andrea, 2019) (Rust, 2020), nous avons choisi dans ce chapitre de se concentrer sur l'étude des langages Eiffel (ECMA International, 2006) et D (D Language Foundation, 2020) qui supportent nativement la programmation par contrat, ainsi que sur l'étude des implémentations : Code Contracts pour C# (Microsoft, 2018), OpenJML (Cok, 2014), AspectJML (Rebêlo *et al.*, 2014), Cofoja (Minh Lê, 2016) pour le langage Java. En effet, ces langages/implémentations à eux seuls permettent de fournir un aperçu global des différentes approches syntaxiques et sémantiques utilisées dans la littérature afin d'apporter un support de la programmation par contrat dans un langage à objet.

2.1 Syntaxe

Pour implémenter la programmation par contrat dans un langage à objet, il faut dans un premier temps définir la syntaxe de spécification des différents types de contrats. Nous identifions cinq approches possibles. Un contrat peut être spécifié à l'aide de mots-clés (2.1.1), de commentaires (2.1.2), d'annotations (2.1.3), de méthodes dédiées (2.1.4) ou bien à l'aide d'aspects (2.1.5). Le choix de l'approche dépend principalement du niveau d'ouverture du langage mais aussi de sa maturité.

2.1.1 Mots-clés

L'approche de définition des contrats par l'ajout de mots-clés est généralement mise en oeuvre par les langages ayant choisi de mettre en place la programmation par contrats dès leurs créations ou du moins dans les premiers temps de leurs phases de développement. En effet, cette approche passe par l'extension de la grammaire, en particulier par l'ajout de nouveaux mots réservés. Cet ajout de mots réservé peut avoir pour conséquence d'invalider du code existant, ce qui peut être problématique si le langage fait l'objet d'une forte utilisation. Nous retrouvons notamment, le langage Eiffel (ECMA International, 2006) ainsi que D (D Language Foundation, 2020), tous deux ayant choisi d'intégrer les contrats par le biais de mots-clés.

Dans le langage Eiffel les contrats sont désignés avec les mots-clés *require* pour désigner les pré-conditions et *ensure* pour désigner les post-conditions ainsi que *invariant* pour désigner les invariants de classe. On peut aussi noter la présence du mot-clé *old* afin de faire référence à une valeur évaluée en entrée dans les post-conditions mais cette spécificité sera détaillée plus en détail dans la Section 2.2.4.

La Figure 2.1 montre un exemple d'utilisation de la syntaxe pour spécifier des

contrats en Eiffel. Cet exemple est basé sur la Figure 1.1 de la page 12. Que cela soit pour les pré-conditions ou post-conditions ainsi que les invariants de classes, la syntaxe est identique. On retrouve le mot-clé désignant le type de contrat (*invariant*, *require*, *ensure*) suivi par un ensemble de conditions. La condition d'un contrat peut être explicitée à l'aide d'un identifiant optionnel. Cet identifiant est notamment utilisé afin de rendre les messages d'erreurs plus explicites. Cet identifiant est ensuite suivi par la clause du contrat, cette clause est mise en oeuvre sous la forme d'un prédicat.

```
1  class LISTE_CHAINE[T]
2  feature
3      taille: INTEGER
4      premier_maillon: MAILLON[T]
5
6      non_vide: BOOLEAN
7      do -- Corps de la méthode
8      ensure
9          non_vide: Result = (taille > 0)
10     end
11
12     depiler: T
13     require
14         liste_non_vide: non_vide
15     do -- Corps de la méthode
16     ensure
17         taille_decrement: taille = old taille - 1
18         retour: Result = old dernier_element
19     end
20
21     ajouter(element: T)
22     do -- Corps de la méthode
23     ensure
24         taille_increment: taille = old taille + 1
25         dernier_element = element
26     end
27 invariant
28     taille_valide: taille >= 0
29 end
```

Figure 2.1 Déclaration des contrats en Eiffel.

Dans la Figure 2.1 nous remarquons que la syntaxe de déclaration des contrats est similaire que cela soit pour les pré-conditions ou les post-conditions et invariants,

hormis l'utilisation de mots-clés différents. Toutefois, nous pouvons voir dans la Figure 2.2, qui représente la redéfinition de contrat lors de l'héritage, la présence de deux mots-clés à savoir *else* pour les pré-conditions et *then* pour les post-conditions et invariant de classe. Ces mots-clés ont été introduits dans le but de rendre explicite la présence de contrats hérités, et sont obligatoires lors de la redéfinition d'un contrat.

```
1  class NOUVELLE_PILE[T]
2  inherit LISTE_CHAINE[T] redefine depiler end
3  feature
4      depiler: T
5      require else
6          -- Nouveau contrat
7      do
8          Result := Precursor
9      ensure then
10         -- Nouveau contrat
11     end
12 end
```

Figure 2.2 Redéfinition de contrats hérités en Eiffel.

Le langage D (D Language Foundation, 2020) offre nativement un support de la programmation par contrat. La Figure 2.3 montre un exemple d'utilisation des contrats en D. Les contrats sont définis à l'aide des mots-clés *in* et *out* désignant respectivement la pré et post-condition et le mot-clé *invariant* désignant les invariants de classe. Le langage D offre deux syntaxes pour spécifier les clauses des contrats. La première similaire à celle d'Eiffel où la clause est explicitée sous la forme d'une expression retournant une valeur booléenne. La seconde se présente sous la forme d'un bloc complexe (similaire au corps d'une méthode). Cette deuxième syntaxe offre un gain non négligeable en termes d'expressivité, mais par la même occasion fournit au programmeur la possibilité de détourner les contrats afin de venir exécuter du code sans rapport avec la vérification de spécification. Outre ce problème, le fait de représenter les contrats à l'aide d'un bloc complexe

peut entraver la compréhension des spécifications qui sont à l'oeuvre. Ce qui réduit l'intérêt d'utiliser la programmation par contrat comme support à la documentation afin d'exprimer clairement les attendus et les bénéfices de l'appel d'une méthode.

```
1  class ListeChaine(T){
2      invariant(obtenir_taille() >= 0, "La taille doit être supérieure ou égale à 0");
3
4      private int taille = 0;
5      int obtenir_taille(){ return taille; }
6      private Maillon<T> premier_maillon;
7      T dernier_element(){ return premier_maillon.obtenir_element(); }
8
9      bool non_vide()
10     out (retour; retour == (obtenir_taille() > 0))
11     {/**/}
12
13     void ajouter(T element)
14     out (; dernier_element() == element)
15     {/**/}
16
17     T depiler()
18     in
19     {
20         assert(non_vide());
21     }
22     do
23     {/**/}
24 }
```

Figure 2.3 Déclaration des contrats en D.

2.1.2 Commentaire

La définition de contrat à l'aide de commentaires est sans doute le mécanisme de spécification le plus adaptable. Effectivement, l'ensemble des langages de programmation ont la possibilité d'intégrer des commentaires libres directement dans le code source. Cette méthode est notamment utilisée par JML. JML est un langage de spécification comportemental développé par Gary T. Leavens (Leavens *et al.*, 1999). Ce langage se base sur une syntaxe spécifique encapsulée dans des com-

mentaires Java. Cette syntaxe permet d'exprimer le comportement des éléments auxquels elle est rattachée. On y retrouve notamment les mots-clés *invariant*, *requires* ou *pre* (pré-condition), *ensures* ou *post* (post-condition) afin d'ajouter une spécification aux méthodes et aux classes (Leavens *et al.*, 2006). On retrouve aussi la possibilité de faire référence dans les post-conditions à une valeur évaluée en entrée avec *old*, ainsi que la possibilité de faire référence au résultat d'une fonction *result*. Notez que JML est uniquement un langage de spécification qui vient se greffer au langage hôte Java. Pour apporter la programmation par contrat au langage Java il sera alors nécessaire d'utiliser des outils prenant en charge le langage JML. Nous retrouvons notamment des outils tels qu'AspectJML (Rebêlo *et al.*, 2014), OpenJML (Cok, 2014; OpenJML, 2018) mettant en oeuvre les contrats du langage JML. La principale différence entre ces deux outils réside dans la mise en oeuvre de la vérification, cette partie est abordée plus en détail dans le Chapitre 3. Toutefois, il existe un grand nombre d'outils utilisant la syntaxe JML (Burdy *et al.*, 2005), chacun offrant des fonctionnalités et une mise en oeuvre différente. La Figure 2.4 représente la classe *ListeChaine* avec la mise en oeuvre des contrats à l'aide de la syntaxe JML. Outre la mise en oeuvre des contrats nous pouvons remarquer la présence du marqueur */*@ pure @*/* qui se réfère au principe de définition des requêtes (Section 1.5) et permet d'expliciter que la requête n'a pas d'effets visibles de l'extérieur. Notons aussi que de manière identique à Eiffel, lors de l'héritage de contrat il faudra indiquer de manière implicite la redéfinition de ceux-ci en utilisant le mot-clé *also*.

Toutefois, ce gain de flexibilité au niveau de la spécification entraîne un coût de développement conséquent afin notamment, d'identifier les commentaires mettant en oeuvre de potentiels contrats, ainsi que vérifier la validité des clauses, nécessitant ainsi le développement d'un analyseur lexical, syntaxique et sémantique.

```
1  class ListeChaine<T>{
2      //@ invariant obtenir_taille() >= 0;
3      private int taille = 0;
4      /*@ pure @*/ int obtenir_taille(){ return taille; }
5      private Maillon<T> premier_maillon;
6      /*@ pure @*/ T dernier_element(){ return premier_maillon.obtenir_element(); }
7
8      //@ ensures \result == (obtenir_taille() > 0);
9      /*@ pure @*/ boolean non_vide(){ return taille > 0; }
10
11     //@ requires non_vide();
12     /*@ pure @*/ T dernier_element(){/*...*/}
13
14     //@ ensures dernier_element() == element;
15     //@ ensures \old(obtenir_taille()) + 1 == obtenir_taille();
16     void ajouter(T element){/*...*/}
17
18     //@ requires non_vide();
19     //@ ensures \old(obtenir_taille()) - 1 == obtenir_taille();
20     //@ ensures \old(dernier_element()) == \result;
21     T depiler(){/*...*/}
22 }
```

Figure 2.4 Déclaration des contrats avec JML.

2.1.3 Annotation

Un autre moyen pour spécifier des contrats est d'utiliser le concept d'annotation. Ce concept offre la possibilité au programmeur d'ajouter des méta-données personnalisées aux éléments du langage, dans notre contexte les interfaces, classes et méthodes. Ce concept, d'annotation, disponible dans plusieurs langages, permet une grande flexibilité en matière d'extension de langage. Grâce à celui-ci, il est possible d'étendre notre langage sans pour autant changer la grammaire de celui-ci. Ce mécanisme est présent dans divers langages avec parfois des noms différents. On le retrouve notamment en Java sous le nom d'annotation, mais aussi en C++ (Cppreference, 2019) ainsi qu'en C# (Microsoft, 2018) sous le nom d'attribut. Dépendant du langage et du compilateur, cette méthode peut nécessiter plus ou moins de développement en fonction notamment des interfaces de programmation

d'applications fournies par les outils du langage.

Le cadre de développement *Contract For Java* (Cofoja) (Minh Lê, 2016) utilise les annotations pour déclarer les contrats. La Figure 2.5 reprend l'exemple de la classe *ListeChaine*.

```
1  @Invariant("obtenir_taille() >= 0")
2  class ListeChaine<T>{
3      private int taille = 0;
4      int obtenir_taille(){ return taille; }
5      private Maillon<T> premier_maillon;
6      T dernier_element(){ return premier_maillon.obtenir_element(); }
7
8      @Ensures("result == (obtenir_taille() > 0)")
9      boolean non_vide(){/*...*/}
10
11     @Ensures("dernier_element() == element")
12     @Ensures("old(obtenir_taille()) + 1 == obtenir_taille()")
13     void ajouter(T element){/*...*/}
14
15     @Requires("non_vide()")
16     @Ensures("old(obtenir_taille()) - 1 == obtenir_taille()")
17     @Ensures("old(dernier_element()) == result")
18     T depiler(){/*...*/}
19 }
```

Figure 2.5 Déclaration des contrats avec Cofoja.

Lors de la rédaction de ce mémoire, le standard C++20 (G. Dos Reis, 2018) aurait dû introduire les contrats à l'aide d'attribut sous la forme *[[expects :]] [[ensure :]]* suivit de la clause à respecter. Toutefois cette nouvelle fonctionnalité a été repoussée à une prochaine version du standard.

2.1.4 Méthode dédiée

L'approche de définition des contrats à l'aide de méthodes dédiés consiste à fournir l'implémentation du contrat dans une méthode séparée. Cette approche est notamment utilisée dans JContractor (Karaorman et Abercrombie, 2005; Abercrombie et Karaorman, 2002). La Figure 2.6 montre la déclaration de notre classe

ListeChaine avec le cadre de développement JContractor. Les contrats sont liés à la méthode à l'aide d'une convention de nommage spécifique. Pour lier une méthode à un contrat, il faut que le nom de la méthode contractuelle corresponde au nom de la méthode suivi du type de contrat (_ *Precondition*, _ *Postcondition*). Les invariants de classe, quant à eux, n'étant pas reliés à une méthode, seul le nom du contrat sera utilisé _ *Invariant*. Chaque méthode contractuelle doit retourner une valeur booléenne afin d'être évaluée dans une assertion.

```
1  public class ListeChaine<T> {
2      // Attribut utilisé pour faire référence aux valeurs avant l'exécution d'une méthode.
3      ListeChaine<T> OLD;
4
5      private int taille = 0;
6      int obtenir_taille(){return taille;}
7      private Maillon<T> premier_maillon;
8      T dernier_element(){ return premier_maillon.obtenir_element(); }
9
10     boolean _Invariant(){ return obtenir_taille() >= 0; }
11
12     boolean non_vide_Postcondition(boolean RESULT){
13         return RESULT == (obtenir_taille() >0);
14     }
15
16     boolean depiler_Precondition(){ return non_vide(); }
17
18     boolean depiler_Postcondition(T RESULT){
19         return (OLD.obtenir_taille() - 1 == obtenir_taille()) &&
20                 (OLD.dernier_element() == result);
21     }
22
23     boolean ajouter_Postcondition(T element){
24         return OLD.obtenir_taille() + 1 == obtenir_taille() &&
25                 dernier_element() == element;
26     }
27 }
```

Figure 2.6 Déclaration des contrats avec JContractor.

Cette technique offre plusieurs avantages. Les contrats étant directement mis en oeuvre sous la forme de méthode standard, il n'est alors pas nécessaire pour le programmeur d'apprendre un langage de spécification de contrat comme c'est le cas avec JML (Leavens *et al.*, 1999). Cette technique vient aussi simplifier la

mise en oeuvre de la vérification. Toujours grâce au fait que les contrats sont disponibles sous la forme de méthode standard, il n'est pas nécessaire de vérifier la validité syntaxique et sémantique des contrats, cette partie étant directement réalisée par le compilateur.

Toutefois, malgré ces avantages, cette technique souffre d'un manque d'association et de liaison entre les méthodes et les contrats. Sur une base de code importante il sera difficile d'identifier facilement les méthodes de contrat ainsi que de repérer les méthodes liées à des contrats. Cette problématique annule un des avantages qu'offre la programmation par contrat, qui est de fournir un support à la documentation en exprimant clairement les attendus et les bénéfices de l'appel d'une méthode.

2.1.5 Aspect

Une dernière méthode pour mettre en place la programmation par contrat est de s'appuyer sur la programmation par aspect. La programmation orientée aspect plus généralement appelé AOP pour *aspect-oriented programming* en anglais, est un paradigme de programmation (Kiczales *et al.*, 1997) défini comme étant un moyen de traiter les préoccupations transversales de manière globale. L'objectif étant de regrouper sous la forme d'entité appelée *aspect* une préoccupation transverse, permettant ainsi de diminuer la redondance de code, en fournissant une séparation des préoccupations ce qui a pour finalité d'augmenter la réutilisation. Un aspect est structuré en deux éléments principaux, un greffon (*advice)* qui définit le comportement que doit réaliser l'aspect, dans notre cas la vérification du respect d'un contrat, ainsi qu'un point de coupe (*pointcut*) qui définit l'endroit où doit être appliqué le greffon. La liaison est ensuite mise en oeuvre à l'aide d'un tisseur d'aspects qui introduira le greffon aux points de coupes définis. La Figure 2.7 définit un aspect pour traiter les pré-conditions (définit par le point

de coupe *before*) et les post-conditions (définit par le point de coupe *after*) de la méthode *depiler()* de la classe *ListChaine*.

```
1  @SuppressWarnings({"rawtypes"})
2  public aspect ListeChaineDepilerContract{
3      // Attributs utilisés pour stocker les valeurs avant l'exécution de la méthode.
4      private int oldTaille;
5      private Object oldElement;
6
7      public pointcut checkContract(ListeChaine liste):
8          !within(ListeChaineDepilerContract) &&
9          call(* ListeChaine.depiler())
10         && target(liste);
11
12     before(ListeChaine liste): checkContract(liste){
13         oldTaille = liste.obtenir_taille();
14         oldElement = liste.dernier_element();
15         assert(liste.non_vide());
16     }
17
18     after(ListeChaine liste)returning(Object result): checkContract(liste){
19         assert(oldElement == result);
20         assert(oldTaille - 1 == liste.obtenir_taille());
21     }
22 }
```

Figure 2.7 Déclaration des contrats avec AspectJ.

Comme le font remarquer Balzer et Eugster (Balzer *et al.*, 2006) la programmation par contrat est souvent associée à une préoccupation transverse. Cette association est notamment due au fait que lorsque l'on procède à la définition de contrats sur l'ensemble des méthodes d'une classe il n'est pas rare de venir définir plusieurs fois des contrats avec des conditions identiques, il est alors raisonnable de penser que de telles conditions pourraient être regroupées en une seule entité. Toutefois, ils soulignent que les aspects utilisés de manière directe avec des extensions telles qu'AspectJ comme présenté dans la Figure 2.7 ne peuvent remplir l'ensemble des bénéfices de la programmation par contrats. En effet, par nature les aspects sont séparés des classes qu'ils augmentent, ce qui restreint l'utilisation des contrats

comme aide à la documentation. Toutefois, des approches telles qu'AspectJML (Rebêlo *et al.*, 2014) permettent de résoudre cette problématique en mettant en oeuvre les aspects seulement dans l'implémentation. La partie déclaration utilisant alors la syntaxe JML présentée précédemment dans la Section 2.1.2.

2.2 Sémantique

Comme nous l'avons vu précédemment, la programmation par contrat se caractérise par la présence de pré-conditions, de post-conditions ainsi que d'invariants de classe. La définition de ces éléments met en oeuvre une sémantique particulière. Cette section vise à fournir une vue d'ensemble des différentes sémantiques mise en oeuvre dans les différentes implémentations étudiées.

2.2.1 Visibilité

Lorsqu'une méthode définit une pré-condition ainsi qu'une post-condition, elle vient spécifier respectivement les conditions préalables à son exécution, ainsi que le résultat de son exécution. En d'autres termes, une pré-condition et une post-condition permettent de vérifier un contexte d'exécution. Le contexte de l'exécution d'une méthode englobe l'ensemble des méthodes, attributs d'une classe ainsi que les paramètres qui lui sont fournis. Cela se traduit par la possibilité de faire référence dans la clause du contrat à l'ensemble des paramètres de la méthode auquel le contrat est rattaché, ainsi qu'à l'ensemble des attributs et méthodes disponibles dans la classe. Les invariants de classe permettent quant à eux de spécifier les contraintes d'intégrité des instances d'une classe, ils peuvent faire appel à l'ensemble des méthodes et attributs de cette même classe. Notons aussi que les post-conditions offrent la possibilité de faire référence à des valeurs avant l'exécution de la méthode souvent représentée à l'aide du mot *old*, ainsi que de pouvoir faire référence au résultat d'une fonction souvent représentée à l'aide du

mot *result*. La sémantique liée à ces éléments est détaillée plus en détail dans la Section 2.2.4.

Toutefois, certaines implémentations restreignent l'utilisation des méthodes et des attributs en fonction de leurs visibilités. L'approche proposée par Meyer (Meyer, 1997, p.357-358) et utilisée dans le langage Eiffel, est de définir une règle de visibilité. Cela se traduit par une restriction de visibilité uniquement pour les constituants de la clause de pré-conditions. Tout éléments, attributs et méthodes ayant une visibilité inférieure à la méthode ne pourront pas être utilisés dans la clause. La justification sous-jacente à ce choix de mise en oeuvre est qu'un client doit pouvoir s'assurer du respect des pré-conditions. En effet il en va de sa responsabilité de s'assurer que la méthode peut être appelée comme définie dans la Section 1.1. Cependant, ce choix peut poser problème pour les attributs. En effet, si la pré-condition se réfère à un attribut, il peut être dangereux d'autoriser la modification de celui-ci au client. Cette problématique ne touche pas le langage Eiffel puisque celui-ci interdit l'accès en écriture à l'extérieur de la classe.

Des propositions comme Code Contracts et JML vont plus loin dans la mise en place de restriction de visibilité. En effet, par défaut ces deux propositions définissent que, tous les membres mentionnés dans un contrat doivent être d'une visibilité égale ou supérieure à la méthode sur lequel le contrat est rattaché. Par défaut la visibilité d'un contrat sur une méthode pré-condition et post-condition est définie par la visibilité de la méthode associée, pour les invariants de classe la visibilité par défaut est de type public. Cependant, le problème évoqué précédemment de rendre un attribut public est présent dans les deux langages C# et Java. Pour ce faire, ces propositions offrent la possibilité d'utiliser un marqueur afin de changer la visibilité d'un attribut afin de pouvoir l'utiliser dans les spécifications. Nous retrouvons ainsi dans Code Contracts le marqueur *ContractPublicPropertyNameAttribute* destinés uniquement aux attributs, et en JML les marqueurs

spec_public ou *spec_protected* destinés à l'ensemble des éléments. Notons qu'en arrière les marqueurs proposés par JML utilisent le concept de variable de modèle présenté par Yoonsik (Cheon *et al.*, 2005) ainsi que les méthodes de modèle [1]. Toutefois, la mise en oeuvre de ces marqueurs ne fournit pas au client la possibilité de vérifier le respect de la condition. L'une des solutions possibles est d'utiliser un accesseur ou bien de mettre en oeuvre le principe 2 évoqué dans la Section 1.5 à savoir, élaborer des requêtes dérivées afin de représenter des concepts de plus haut niveau d'abstraction. Nous retrouvons un exemple dans la classe *ListeChaine* avec la requête dérivée *non_vide* qui permet de représenter le concept de l'état de la pile à savoir qu'elle n'est pas vide *taille* > 0.

Nous pouvons aussi noter que pour les langages offrant la possibilité d'introduire des méthodes statiques, les pré-conditions et post-conditions sur de telles méthodes doivent être composées uniquement d'éléments statiques eux aussi.

2.2.2 Politique d'évaluation

Lors de la mise en oeuvre de la vérification des contrats à l'exécution d'une méthode celle-ci suit un ordre spécifique qui est le suivant : invariant de classe, pré-condition, corps de la méthode, post-condition et invariant de classe. Cet ordre est sujet à plusieurs modifications en fonction de l'implémentation. Par exemple, pour la vérification des invariants, le compilateur ISE Eiffel (Eiffel, 2019) ajoute un mécanisme de vérification des clauses d'invariants uniquement lorsque l'appel n'est pas effectué sur l'instance courante. Meyer (Meyer, 1997) justifie ce choix par la nature même de la programmation orientée objet. En effet, la programmation objet se base sur l'utilisation d'instances qui sont amenées à évoluer par muta-

1. La représentation sous forme d'élément de modèle est destinée à fournir une abstraction entre l'implémentation et la spécification permettant ainsi à séparer le code destiné à être utilisé dans le programme et le code uniquement destiné à des fins de spécification.

tion or ces mutations sont souvent la résultante de plusieurs appels internes. Il se pourrait donc que l'objet se retrouve dans un état incohérent durant ces appels internes. Cette politique de vérification est notamment suivie par l'implémentation Cofoja (Minh Lê, 2016).

Cette décision a cependant un désavantage. Si les clauses d'invariants sont vérifiées uniquement lorsque l'appel n'est pas effectué sur l'instance courante, il peut être difficile d'identifier la méthode qui a rendu la clause d'invariants non valide. D'autres implémentations de programmation par contrat, notamment OpenJML (Cok, 2014), JContractor (Abercrombie et Karaorman, 2002) ou bien D (D Language Foundation, 2020) proposent d'effectuer la vérification à chaque appel de méthode même interne. Néanmoins celles-ci n'introduisent aucun garde-fou pour désactiver la vérification des invariants sur les éléments membres constituant la clause, ce qui aura pour conséquence de faire rentrer le programme dans une récursion infinie de vérification. Afin de remédier à ce problème, D (D Language Foundation, 2020) ou OpenJML (Cok, 2014) demandent au programmeur d'utiliser uniquement des attributs ainsi que des méthodes statiques pour expliciter les clauses d'invariants.

Le problème soulevé par les récursions infinies de vérification est un problème connu de la programmation par contrat et ne concerne pas uniquement les invariants de classe. En effet, lorsqu'un programmeur écrit un contrat il est facile de venir introduire des récursions infinies. Cette problématique est due à la possibilité d'écrire les clauses de contrat à l'aide de méthodes de requête. La Figure 2.8 illustre un tel exemple où une des méthodes introduit des contrats croisés où l'appel de *is_foo* dépend de l'appel de *is_ bar*. Dans cet exemple l'appel de la méthode *is_foo* ou *is_ bar* entraînera ainsi une récursion infinie.

Pour se protéger de cette problématique les implémentations mettent en oeuvre

```
1 fun is_foo: Bool is expect(is_bar) do #...#
2
3 fun is_bar: Bool is expect(is_foo) do #...#
```

Figure 2.8 Exemple de récursion infinie de vérification en Nit.

des garde-fous afin de désactiver la vérification des contrats. Outre le fait de provoquer des boucles infinies, Meyer (Meyer, 1997, p.403) justifie son choix de désactiver la vérification du fait que, lorsque l'on effectue la vérification d'une assertion à l'aide d'une méthode de requête il est trop tard pour effectuer l'évaluation des assertions de cette dite méthode. Il indique que le moment approprié pour une telle vérification est lorsque l'on décide d'appeler la méthode faisant l'objet du contrat explicité sous la forme d'une méthode de requête. La mise en oeuvre de ces gardes fous est étudiée en détail dans la Section 3.2.3.

Outre l'évaluation systématique de l'ensemble des contrats, des implémentations proposent des gestions plus fines de la vérification. C'est notamment le cas pour des implémentations comme D (D Language Foundation, 2020), ISE Eiffel (Eiffel, 2019) ou Cofoja (Minh Lê, 2016). Cette gestion peut se faire à différents niveaux de granularité. Le niveau le moins fin consiste à gérer l'activation par type de contrat sur l'ensemble du programme (D (D Language Foundation, 2020)). De manière plus fine Eiffel (Eiffel, 2019) propose grâce à un fichier de configuration (*.ecf*) de gérer pour chaque type l'activation individuelle des invariants de classe, pré-condition et post-condition. Cofoja (Minh Lê, 2016) quant à lui propose une gestion encore plus fine en offrant la gestion de l'activation des contrats sur les méthodes, classes et interfaces à l'aide de motif. Il convient de noter que désactiver les contrats pour un type spécifique n'empêche pas que ces contrats soient hérités et vérifiés par les sous-types. Toutefois, Cofoja (Minh Lê, 2016) propose à l'aide

d'un autre mécanisme (*black list*) de désactiver entièrement les contrats associés à un type spécifique pour que ceux-ci ne soient pas pris en considération dans l'ensemble des sous-types.

2.2.3 Constructeur

La tâche principale d'un constructeur étant d'initialiser un objet, la sémantique de mise en oeuvre des contrats est alors différente aux autres méthodes. En effet, comme nous l'avons évoqué les pré-conditions peuvent faire référence à des éléments membres de la classe méthode et attribut. Toutefois, l'objet n'étant pas encore initialisé il est impossible de venir faire référence à des éléments membres. En fonction des implémentations cette règle est plus ou moins explicite. C'est notamment le cas pour les implémentations se basant sur JML (Cok, 2014; Rebêlo *et al.*, 2014) où il est interdit de faire référence à l'instance courante dans les pré-conditions, de même en Eiffel.

Lors de la définition d'une post-condition, seule une restriction est appliqué quant à l'utilisation du *old* pour évaluer des éléments membres de la classe (méthode et attribut) avant l'exécution du constructeur. Cela s'explique pour la même raison qu'il n'est pas possible de faire référence à des éléments membres dans une pré-condition.

Notons aussi que les règles précédemment énoncées sur l'ordre de vérification des contrats sont différentes pour les constructeurs. En effet, lorsque l'on appelle un constructeur, les invariants ne sont pas vérifiés en entrée, toujours dû à l'initialisation.

2.2.4 Sémantique Old et Result

La majorité des implémentations étudiées offre aux post-conditions des mécanismes additionnels afin de fournir au programmeur la possibilité de définir des contraintes en faisant référence au retour des fonctions, ainsi que la possibilité de faire référence à des valeurs évaluées avant l'exécution de la méthode permettant ainsi d'expliciter les modifications entre l'état initial et l'état final.

L'ensemble des solutions étudiées permettent de résoudre la problématique des clauses sur le résultat des fonctions grâce à l'introduction d'une variable. Cette variable sera utilisée afin de stocker la valeur de retour. Généralement cette variable est appelée de manière explicite *result*. Certaines solutions comme c'est le cas dans le langage D (D Language Foundation, 2020), proposent de définir le nom de cette variable. Dans la Figure 2.3, nous pouvons remarquer à la ligne dix que le premier argument de la post-condition est dédié à cet effet. Dans notre exemple la variable est nommée *retour*. Notons que si cet argument est laissé vide le nom par défaut est *result*.

De manière générale, les outils étudiés fournissent la possibilité de faire référence à une valeur évaluée avant l'exécution d'une méthode, à l'exception faite du langage D. Cette fonctionnalité est exprimée *old* et peut se présenter sous la forme d'une variable, fonction ou mot-clé. La sémantique associée au *old* consiste à l'évaluation de l'expression qui lui est associée en amont de l'exécution de la méthode. Une fois évaluée cette valeur est ensuite stockée dans une variable, de telle sorte qu'elle puisse être utilisée lors de l'évaluation de la clause de la post-condition.

Il convient de noter que dépendamment de la nature du langage et de l'expression évaluée, l'élément stocké sera généralement une référence vers la valeur réelle. Or le fait de stocker une référence ne garantit pas que la valeur pointée par celle-ci ne sera pas modifiée lors de l'exécution de la méthode. Bien que généralement

les implémentations choisissent de laisser le contrôle de cette problématique au programmeur, l'implémentation Jass (Bartetzko *et al.*, 2001) pour Java, quant à elle, a imposé l'utilisation de la méthode *clone* afin de dupliquer l'instance. Le programmeur devra alors mettre en oeuvre une implémentation de cette méthode pour chaque type utilisé dans un contexte de *old*.

2.2.5 Héritage et sous-typage multiple

Comme évoqué dans la Section 1.2, la programmation par contrat offre un support à la mise en oeuvre des langages à objet, avec notamment une prise en charge de l'héritage. Basé sur un concept de réutilisabilité et d'adaptabilité, l'héritage offre notamment la possibilité au programmeur de redéfinir des fonctionnalités héritées dans le but de les spécialiser afin de satisfaire à des exigences spécifiques. De la même manière qu'il est possible de redéfinir l'implémentation de la méthode lors de l'héritage, il est aussi possible de redéfinir l'implémentation de ses contrats. Toutefois, la redéfinition des contrats suit une règle spécifique. Effectivement, comme l'ont défini Liskov et Wing (Liskov et Wing, 1994) une instance d'un type A doit pouvoir être remplacée par une instance de type B, tel que B sous-type de A, sans que cela ne modifie la cohérence du programme. En effet, grâce au polymorphisme le choix de l'implémentation à appeler sera effectué à l'exécution en fonction du type dynamique de l'instance qui sera associé à l'appel, il est alors primordial que peu importe le sous-type associé à l'appel l'implémentation sélectionnée les contrats initialement définis seront respectés.

L'exemple illustré par la Figure 2.9 va nous permettre d'illustrer nos propos. Nous avons une hiérarchie de classes où B est un sous-type de A. A introduit une méthode *foo* et B la spécialise. Un client peut donc utiliser deux implémentations de la méthode *foo* en fonction du type dynamique soit A ou B. Cet exemple va permettre d'identifier les différentes sémantiques d'héritages à l'oeuvre en fonction

du type de contrat : pré-condition, post-condition et invariant de classe.

```
1 class A
2     fun foo do #...#
3 end
4
5 class B
6     super A
7     fun foo do #...#
8 end
```

Figure 2.9 Relation d'héritage en Nit.

Les pré-conditions sont définies comme contravariantes. En effet, si la classe B vient renforcer les pré-conditions de la méthode *foo* cela voudrait dire que cette nouvelle spécialisation viendrait rendre certains appels qui étaient corrects du point de vue du contrat initialement défini dans A, incorrect du point de vue de la nouvelle spécialisation fournie par B. Or, cette situation ne respecte pas le principe de Liskov (Liskov et Wing, 1994). Pour que ce principe soit respecté la pré-condition de la méthode *foo* de B doit accepter l'ensemble des éléments qui avait été défini comme acceptable dans la spécification de la version initiale. La spécialisation d'une pré-condition peut alors uniquement être affaiblie, cet affaiblissement revient à dire que la nouvelle définition peut prendre une gamme d'entrée plus large.

Les post-conditions définissent la relation inverse et sont considérées comme covariants. En effet, si la classe B vient affaiblir les post-conditions de la méthode *foo* cela signifie que la méthode peut retourner un résultat non acceptable du point de vue du client. Or, comme énoncé précédemment, la spécialisation permet d'adapter la méthode à un nouveau contexte et non d'effectuer un travail complètement différent. Les post-conditions ne peuvent donc pas être affaiblies, mais

peuvent seulement être renforcées afin de venir restreindre l'ensemble des résultats acceptables.

L'héritage pour les invariants de classe se base sur le principe suivant, une instance de la classe B peut être considérée comme une instance particulière de A, il est donc nécessaire que l'ensemble des contraintes de cohérence qui s'applique à une instance de A s'applique aussi à une instance de B. C'est pourquoi les invariants de classe sont définis comme covariants de la même manière que les post-conditions. Les invariants de classe peuvent alors seulement être renforcés afin de garantir que l'état d'une instance de B sera acceptable du point de vue d'une instance de A. L'invariant d'une classe est alors représenté par la conjonction de l'ensemble des invariants hérités ainsi que de ceux définis dans celle-ci. L'invariant d'une sous classe est ainsi toujours plus fort ou égal aux invariants de chacun de ces parents.

Nous pouvons résumer la mise en oeuvre de la sémantique d'héritage des contrats de la manière suivante :

— La pré-condition d'une méthode peut se traduire par une disjonction entre l'ensemble des pré-conditions héritées.
— La post-condition d'une méthode peut se traduire par une conjonction entre l'ensemble des post-conditions héritées.
— L'invariant d'une classe peut se traduire par une conjonction entre l'ensemble des invariants hérités.

Grâce à cette représentation, nous pouvons garantir que lors de la mise en oeuvre de l'héritage, la clause totale d'une pré-condition sera équivalente ou plus faible que la spécification originale et que la clause totale d'une post-condition et de l'invariant de classe sera plus forte ou équivalente à la spécification originale.

Cette sémantique d'héritage est mise en oeuvre de manière identique dans l'en-

semble des implémentations étudiées. Toutefois, nous pouvons noter que certains choix de définition syntaxiques ne se prêtent pas à une mise en oeuvre systématique de cette sémantique. C'est notamment le cas avec la méthode utilisée par JContractor (Abercrombie et Karaorman, 2002; Karaorman et Abercrombie, 2005) qui est d'utiliser des méthodes dédiées ainsi que pour la définition des contrats à l'aide d'aspect, ce sera alors au programmeur de prendre en charge cette sémantique d'héritage. Nous pouvons aussi noter une différence d'interprétation lors de la redéfinition de méthode pour le langage D (D Language Foundation, 2020), où lorsqu'une méthode est redéfinie sans pré-condition cela signifie que toutes les valeurs en entrée sont maintenant considérées comme acceptables.

2.3 Conclusion

En conclusion, la programmation par contrats passe par la mise en oeuvre d'une syntaxe et d'une sémantique associée. Toutefois, celles-ci varient fortement en fonction des implémentations et du langage hôte où les contrats sont mis en oeuvre. Le Tableau 2.1 résume les différentes caractéristiques des implémentations étudiées dans ce chapitre. Bien qu'évoquées dans ce chapitre les implémentations Jass (Bartetzko *et al.*, 2001) ainsi que JContractor (Abercrombie et Karaorman, 2002) ne sont plus d'actualité, il est toutefois pertinent de les citer afin d'avoir un panel de comparaison plus large sur les différentes stratégies de réalisation.

	Jass	JContractor	OpenJML AspectJML	Cofoja	D	Eiffel	Code Contracts
Syntaxe							
Mots-clés (2.1.1)					×	×	
Commentaire (2.1.2)	×		×				
Annotation (2.1.3)				×			
Méthode (2.1.4)		×					×
Sémantique							
Politique de visibilité (2.2.1)			×			×	×
Vérouillage (2.2.2)				×		×	×
Configuration (2.2.2)				×	×	×	×
Résult (2.2.4)	×	×	×	×	×	×	×
Old (2.2.4)	×	×	×	×		×	×
Héritage (2.2.5)			×	×	×	×	×

Tableau 2.1 Résumé de la mise en oeuvre syntaxique et sémantique de la programmation par contrat dans les solutions étudiées.

CHAPITRE III

ÉTUDE DES APPROCHES ET IMPLÉMENTATIONS

Ce chapitre fait l'étude de la mise en oeuvre technique des implémentations suivantes : les compilateurs ISE Eiffel (Eiffel, 2019) et Liberty Eiffel (Liberty Eiffel, 2019), le compilateur DMD (D Language Foundation, 2020), le préprocesseur Code Contracts (Microsoft, 2018),les compilateurs OpenJML (Cok, 2014), AspectJML (Rebêlo *et al.*, 2014) et le préprocesseur Cofoja (Minh Lê, 2016). Dans un premier temps, la Section 3.1 aborde l'aspect d'analyse du code fourni par le programmeur afin de traiter les contrats. Nous identifions deux méthodes principales afin d'analyser et de générer du code source, à savoir l'utilisation de préprocesseur (3.1.1) et la définition d'un compilateur spécifique (3.1.2). Dans un second temps, la Section 3.2 détaille les différentes stratégies mises en oeuvre pour effectuer la vérification des contrats, avec l'étude de la représentation des contrats dans la Sous-Section 3.2.1, la Sous-Section 3.2.2 aborde la mise en oeuvre de la représentation des différentes règles d'héritage définies dans la Section 2.2.5, pour finir nous abordons dans la Sous-Section 3.2.3 le contrôle de l'évaluation des contrats.

3.1 Analyse et génération des contrats

3.1.1 Préprocesseur

Une des approches fréquemment utilisées pour la mise en oeuvre de la programmation par contrat est l'utilisation de préprocesseurs. L'objectif d'un préprocesseur est d'effectuer un pré-traitement sur le code source, le code source qui résulte de cette opération de pré-traitement est alors utilisé comme entrée pour une prochaine étape (autre préprocesseur, compilateur...). Dans notre cas cette opération de pré-traitement a pour mission de venir générer l'ensemble du code source de vérification de contrat basé sur les informations disponibles dans le code fourni par le programmeur. Les préprocesseurs peuvent se présenter sous la forme de programmes autonomes ou bien sous la forme d'entité à part entière du compilateur.

En fonction du langage et de l'environnement de développement nous trouvons plusieurs mécanismes pour réaliser des pré-traitements sur le code source fourni par l'utilisateur. C'est notamment le cas avec l'environnement de développement *Java Development Kit (JDK)* qui offre trois mécanismes afin de pré-traiter le code source :

— Les plugins de *javac* pouvant être utilisés pour analyser et générer du code source, il est à noter toutefois que cette fonctionnalité est propre au JDK et ne fait pas partie de la spécification Java SE.
— Les *doclet* utilisés par l'outil *javadoc* pour générer de la documentation. Bien qu'utilisé à des fins de documentation, il est possible d'utiliser les *doclet* pour générer du code Java standard comme c'est le cas avec l'outil expérimental DBCProxy (Eliasson, 2002).
— Les processeurs d'annotation (Annotation Processing) initialement introduit sous la forme d'un outil autonome dans JDK 5, il fait maintenant

partie intégrante de *javac*.

Bien que le choix du mécanisme de pré-traitement soit souvent guidé par un choix syntaxique (utilisation d'annotation, commentaire...) la majorité des mécanismes peuvent se substituer entre eux. Il convient de noter qu'en fonction du mécanisme certaines taches seront plus ou moins difficiles à réaliser dû à l'API offerte par chaque mécanisme.

L'outil Cofoja (*Contract For Java*) (Minh Lê, 2016) procède à la mise en oeuvre de la programmation par contrat à l'aide d'un préprocesseur. Effectivement, l'implémentation fournie par Cofoja repose en partie sur l'utilisation du mécanisme de pré-traitement des annotations (Annotation Processing). La Figure 3.1 illustre le processus de compilation mis en oeuvre par l'outil Cofoja (Minh Lê, 2016), se concentrant sur la représentation des étapes réalisées par le processeur d'annotation. Ce processus se déroule en trois étapes : création, génération et compilation des contrats.

En entrée du processeur d'annotation nous avons l'ensemble des contrats présents sous la forme de chaîne de caractères liés syntaxiquement aux éléments du langage (interface, classe et méthode) par le biais d'annotations. La première étape consiste à la création d'un modèle intermédiaire afin de lier les éléments annotés avec leur type de contrat associé. À cette étape du processeur, l'ensemble des conditions des contrats sont toujours représentées sous la forme d'une chaîne de caractères et non pas encore fait l'objet d'analyse afin de vérifier leur validité syntaxique et sémantique. La seconde étape consiste à la génération de classes alternatives aux classes originales, c'est à ce moment-là que les conditions sont analysées et traduites en code Java standard afin de générer l'ensemble de l'architecture de vérification des contrats. La dernière étape consiste à compiler les classes alternatives afin de générer le code octet Java. Il convient de noter qu'une fois la compilation terminée le

code dédié à la mise en oeuvre des contrats et le code fourni par le programmeur sont présents dans des fichiers séparés et ne sont nullement liés. La liaison entre ces deux codes pour rendre la vérification des contrats effective est effectuée à l'aide d'un *Java Agent* (Oracle, 2020a) qui procédera à la réécriture du code octet lors du chargement des classes. Cofoja (Minh Lê, 2016) propose toutefois une alternative à l'instrumentation dynamique, en proposant l'utilisation d'un mode *hors ligne*. Ce mode permet de générer le code octet Java instrumenté de manière statique.

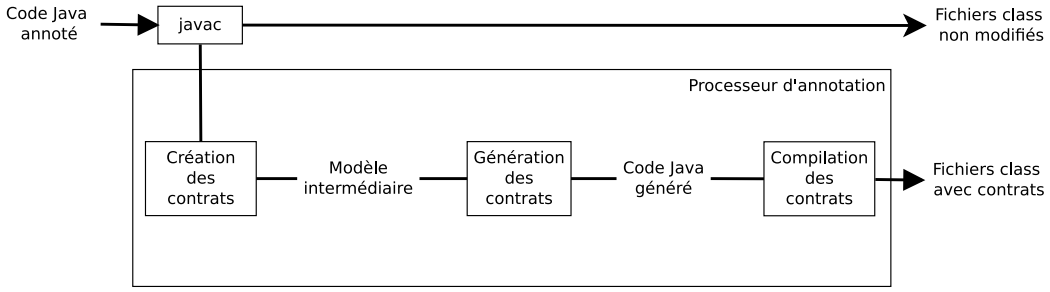

Figure 3.1 Étapes de compilation de Cofoja.

3.1.2 Compilateur

Une des approches pour prendre en charge la syntaxe mise en oeuvre pour la déclaration des contrats est le développement d'un compilateur spécifique. Ce procédé de prise en charge de la programmation par contrat est souvent utilisé lorsque les contrats font partie intégrante du langage. C'est notamment le cas pour le langage Eiffel ainsi que le langage D (D Language Foundation, 2020) qui intègrent la programmation par contrat nativement. Nous pouvons aussi noter que le langage Eiffel fait l'objet d'un standard ECMA-367 (ECMA International, 2006) qui régit les différents aspects du langage dont la définition des contrats. Tout compilateur voulant être conforme à ce standard devra alors prendre en charge cet aspect, c'est notamment le cas avec le compilateur *officiel* ISE Eiffel ou bien

avec le compilateur Liberty Eiffel (Liberty Eiffel, 2019) (basé sur *SmartEiffel*).

Dans la littérature nous trouvons des outils comme OpenJML (Cok, 2014; OpenJML, 2018) et AspectJML (Rebêlo *et al.*, 2014) qui mettent en oeuvre leur propre compilateur pour fournir la prise en charge des contrats dans le langage Java. Bien que ces deux compilateurs fournissent une implémentation pour prendre en charge le langage JML (Leavens *et al.*, 1999), chacun d'entre eux implémente leur propre analyseur syntaxique et sémantique ainsi que leur propre générateur de code octet Java. Outre le fait d'un coût de développement important dû aux multiples parties à mettre en oeuvre (analyseur lexical, syntaxique, sémantique et générateur de code), cette technique engendre aussi un coût de maintenance élevé indépendant à la prise en charge des contrats. Ce coût de maintenance est principalement dû à l'évolution du langage hôte. Effectivement si le langage hôte modifie sa syntaxe ou sémantique, le compilateur devra adapter sa structure afin de prendre en considération ces nouvelles modifications. Cette problématique impacte fortement les implémentations JML, c'est notamment le cas avec les outils OpenJML et AspectJML qui ne supportent que du code Java inférieur à la version 8.

3.2 Stratégie de vérification

3.2.1 Représentation et mise en oeuvre des contrats

Pour permettre la mise en oeuvre de la programmation par contrat il est essentiel de définir un point d'insertion où procéder à la vérification des contrats, ainsi qu'une représentation de cette vérification. Le point d'insertion correspond au moment où l'on procédera à la vérification des différents contrats. Cette insertion peut avoir lieu directement dans la méthode faisant l'objet de vérification. Cette approche est utilisée dans la majorité des implémentations étudiées (Eiffel, 2019) (Minh Lê, 2016) (Cok, 2014) (D Language Foundation, 2020) (Liberty Eiffel, 2019) (Microsoft, 2018). Une autre façon de faire consiste à déplacer l'injection de

la vérification directement au site d'appel comme c'est le cas dans Code Contracts [1] (Microsoft, 2018). Le principal avantage évoqué par Ben et Joseph Albahari (Albahari et Albahari, 2017) de cette dernière approche est d'offrir la possibilité de pouvoir distribuer une bibliothèque compilée sans la prise en charge des contrats, et de fournir malgré tout au programmeur utilisant cette bibliothèque la possibilité de mettre en oeuvre la vérification des contrats [2]. Toutefois cet avantage peut aussi être appliqué aux implémentations effectuant le lien entre le code fourni par le programmeur et la représentation des contrats lors du chargement des classes comme c'est le cas pour Cofoja (Minh Lê, 2016) et OpenJML (Cok, 2014).

D'autres approches utilisent des mécanismes d'encapsulation afin d'injecter le code de vérification des contrats. Cette approche d'injection, peut s'apparenter à la fois comme de l'injection dans la méthode et l'injection au site d'appel. Effectivement, l'encapsulation consiste :

- À définir une nouvelle méthode qui est chargée d'effectuer la vérification des contrats et d'appeler la propriété faisant l'objet de contrat,

- Et à modifier le site d'appel existant pour le diriger vers la nouvelle méthode introduite.

Cette approche est notamment utilisée par l'implémentation proposée par Eliasson (Eliasson, 2002) avec son outil DBCProxy pour Java. Cette réalisation se base sur l'utilisation des proxys dynamiques disponibles dans l'API du JDK (Oracle, 2020b). Avant toute chose un proxy est un patron de conception décrit par

---

1. Code Contracts permet d'appliquer les deux approches de point d'insertion en fonction de l'option sélectionnée par le programmeur (Albahari et Albahari, 2017)

2. Notons tous de même que pour que cette méthode fonctionne les contrats doivent être représentés dans un fichier séparé.

Gamma, Helm, Johnson et Vlissides (Gamma *et al.*, 1995, p.233) comme étant une technique permettant de contrôler l'accès à un objet à l'aide d'un médiateur appelé *proxy*. Ce patron permet notamment d'ajouter/modifier le comportement d'une classe existante. Pour y parvenir, le proxy et l'objet à contrôler doivent implémenter une même interface de sorte que le programmeur puisse utiliser le proxy de la même manière que la classe originale. Le proxy quant à lui possédera en arrière une instance de l'objet à contrôler. Lors de l'utilisation le proxy pourra alors transférer les appels à l'objet encapsulé en fournissant une logique supplémentaire. Un proxy dynamique se base sur le même principe à la différence que seule l'implémentation du proxy est générée dynamiquement à l'exécution grâce à l'utilisation de la réflexion. L'utilisation des proxys dynamiques est particulièrement adaptée à la programmation par contrat du fait que les contrats sont destinés à être vérifiés en entrée et en sortie de la méthode. Toutefois, l'utilisation des proxys dynamiques de l'API du JDK a pour principale limitation de dépendre de la définition d'une interface décrivant le comportement de la classe. En effet, la création de l'objet proxy se fait grâce à l'introspection des interfaces qu'implémente la classe, si la classe ne met pas en oeuvre d'interface il ne sera alors pas possible de venir générer le proxy. Cette limitation peut malgré tout être surmontée grâce notamment à l'utilisation de bibliothèque externe comme CGLIB proxy (Github, 2020) qui se base sur la classe de l'objet pour générer le proxy.

En ce qui concerne la représentation de la vérification des contrats celle-ci correspond à la mise en oeuvre de la vérification des conditions appliquées au contrat. Outre le fait de représenter l'évaluation des conditions à l'aide des mécanismes comme les assertions ou les exceptions, la représentation prend en compte la façon dont les mécanismes sont mis en oeuvre. L'approche standard consiste à l'intégration des mécanismes de vérification de l'ensemble des contrats (invariants de classes, pré-conditions et post-conditions incluant ceux hérités) directement au

point d'insertion. Cette représentation est notamment utilisée par le compilateur ISE Eiffel (Eiffel, 2019), DMD (D Language Foundation, 2020) ainsi que par le compilateur OpenJML (Cok, 2014). La Figure 3.2 montre la méthode *depiler* une fois les contrats mis en oeuvre à l'aide de cette approche.

```
1  fun depiler(): T
2  do
3      var old_taille = taille
4      var old_element = dernier_element
5      var result: T
6
7      assert self.taille >= 0 # Vérification invariant de classe
8      assert self.non_vide # Vérification pré-condition
9
10     # [...] Code de la méthode initiale
11     # L'expression de retour est remplacé par l'affectation de `result`
12
13     assert self.taille == old_taille - 1 # Vérification post-condition
14     assert result == old_element
15     assert self.taille >= 0 # Vérification invariant de classe
16     return result
17 end
```

Figure 3.2 Représentation de l'intégration du code de vérification des contrats dans une méthode.

D'autres approches tentent d'apporter une séparation entre le code de vérification du contrat et la méthode faisant l'objet de cette vérification grâce à la mise en place d'une représentation des contrats sous la forme de méthode dédiée. Chaque contrat vient ainsi introduire sa propre méthode de vérification. Cette approche est utilisée par Cofoja (Minh Lê, 2016). La Figure 3.3 montre la méthode *depiler* mettant en oeuvre les contrats à l'aide de méthodes dédiées à la vérification des contrats. Il convient aussi de noter que Cofoja (Minh Lê, 2016) effectue une

séparation pour chaque clause d'un contrat [3].

```
fun invariant do assert self.taille >= 0 # Vérification invariant de classe

fun pre_condition_depiler do assert self.non_vide #Vérification pré-condition

fun post_condition_depiler(result: T, old_taille: int, old_element: T)
do assert self.taille == old_taille - 1 # Vérification post-condition

fun depiler: T
do
    var old_taille = taille
    var old_element = dernier_element
    var result: T

    invariant
    pre_condition_depiler
    # [...] Code de la méthode initiale
    # L'expression de retour est remplacé par l'assignation de `result`

    post_condition_depiler(result, old_taille, old_element)
    invariant
    return result
end
```

Figure 3.3 Représentation de l'intégration du code de vérification dans des méthodes autonomes.

Comme évoqué dans la Section 2.1.5 les contrats peuvent aussi être représentés à l'aide d'aspect, c'est notamment le cas avec l'outil AspectJML (Rebêlo *et al.*, 2014) qui vient traduire la syntaxe JML en code aspect identique à celui précédemment introduit dans la Figure 2.7, ce code généré sera ensuite mis en oeuvre

3. Exemple si le programmeur introduit deux pré-conditions sur une méthode alors Cofoja introduit deux méthodes de vérification séparée, une pour évaluer la première clause et l'autre pour la seconde.

par le moteur d'aspect AspectJ.

Tout comme le choix syntaxique et sémantique, le choix de la représentation dépend fortement du langage hôte auquel est appliquée la programmation par contrat. Effectivement, certaines implémentations doivent mettre en place des mécanismes afin de contourner les limitations dues au langage, c'est notamment le cas avec la mise en oeuvre de code dans les interfaces. En effet, faire vivre les contrats sous la forme de méthode de vérification autonome pose des questions quant à la définition de contrat dans les interfaces. Toutefois, cette problématique n'est plus réellement présente de nos jours avec la possibilité d'implémenter des méthodes par défaut dans les interfaces, comme c'est le cas en Java depuis la version 1.8 et en C# depuis la version 8. On retrouve cependant dans la littérature des solutions comme Cofoja (Minh Lê, 2016), OpenJML (Cok, 2014) ou Code Contracts (Microsoft, 2018) qui mettent en oeuvre des implémentations avant l'ajout du mécanisme de définition d'une implémentation par défaut dans les interfaces. La solution utilisée par OpenJML (Cok, 2014) est de déporter l'implémentation des méthodes de vérification dans les classes qui implémenteront l'interface. Code Contracts (Microsoft, 2018) quant à lui demande au programmeur de définir une classe qui implémente l'interface en fournissant l'ensemble des contrats, l'implémentation des méthodes est ensuite utilisée lors de la phase d'injection des contrats où celui-ci est dupliqué à chaque point d'injection. D'un autre côté Cofoja (Minh Lê, 2016) introduit une classe utilitaire qui vient implémenter l'interface. Cette classe a pour objectif de contenir l'ensemble des méthodes de vérification des contrats introduit dans l'interface. Cette technique nécessite néanmoins une adaptation des méthodes de vérification insérées dans cette classe d'aide. En effet celles-ci sont définies comme statiques afin de pouvoir être appelé sans instanciation, et comprennent un paramètre supplémentaire qui représente l'objet concret sur lequel est appliqué la vérification du contrat.

En ce qui concerne la représentation des valeurs évaluées avant l'exécution d'une méthode et de la référence au retour d'une fonction dans les post-conditions, celles-ci sont mises généralement en oeuvre à l'aide de variable locale. Cette variable locale est ensuite utilisée directement dans le code de vérification ou passée en argument à la méthode de vérification. Toutefois, Cofoja (Minh Lê, 2016) met en oeuvre une séparation du mécanisme permettant aux post-conditions de faire référence à des valeurs évaluées avant l'exécution d'une méthode *old*, Section 2.2.4. Pour ce faire, Cofoja introduit pour chaque utilisation de l'expression *old* une nouvelle méthode permettant d'évaluer l'expression fournie. Cette méthode est introduite dans le même contexte et possède la même signature que la méthode faisant l'objet d'une post-condition à l'exception du paramètre de retour qui lui est défini en fonction du type de l'expression évaluée.

3.2.2 Représentation de l'héritage

Comme évoqué dans la Section 2.2.5 la prise en compte de l'héritage est l'un des éléments clés de la programmation par contrat dans le paradigme objet. Pour rappel, lors de la présence d'héritage, la pré-condition d'une méthode doit être équivalente à la disjonction de l'ensemble des pré-conditions héritées, et les post-conditions et les invariants de classe équivalant à la conjonction de l'ensemble des post-conditions et des invariants de classe héritées.

Indépendamment de la représentation des contrats, la prise en considération de l'héritage se traduit par une mise en oeuvre indentique qui consiste à combiner l'ensemble des vérifications définit dans les parents. Cette combinaison se caractérise par la duplication systématique de la représentation de la vérification de chaque contrat hérité. Le résultat de cette combinaison est inséré, dépendamment de l'approche de représentation, soit directement au point de vérification (cite d'appel ou méthode faisant l'objet des contrats) ou bien dans une méthode

de vérification dédiée. On retrouve cette approche dans OpenJML (Cok, 2014), ISE Eiffel [4] (Eiffel, 2019), DMD (D Language Foundation, 2020), Code Contracts (Microsoft, 2018).

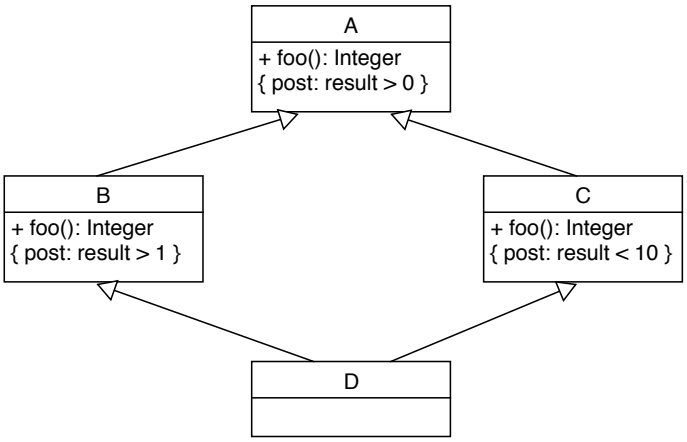

Figure 3.4 Hiérarchie de classe en diamant.

Toutefois, il convient de noter que lorsque nous représentons les contrats à l'aide de méthodes dédiées, un conflit peut apparaître lors de l'héritage multiple. Prenons l'exemple illustré par la Figure 3.4 qui représente une hiérarchie de classes en diamant, où la classe A est héritée par les classes B et C qui sont elles-mêmes hérités par la classe D. La classe A introduit une fonction *foo* retournant une valeur entière, cette fonction définit une post-condition afin de restreindre les valeurs acceptables en sortie. Cette fonction est par la suite redéfinie par les classes B et C qui introduisent à leur tour une nouvelle restriction sur la sortie de celle-ci. Si nous raisonnons avec des instances de classe B et C, la détermination de la méthode de vérification est évidente.

---

4. ISE Eiffel procède avec cette approche seulement pour les pré-conditions et post-conditions, les invariants sont quant à eux vérifiés à l'aide d'une méthode procédant à un appel récursif sur chaque type parent

Lorsque nous sommes en présence d'instance de type D, déterminer quelle méthode de vérification appelée est problématique, effectivement, nous pouvons appeler la méthode de vérification de B ou C. Pour résoudre ce conflit une solution est de redéfinir de façon systématique la méthode de vérification même si aucun nouveau contrat n'est introduit dans la classe. Cette redéfinition peut mettre ainsi en oeuvre le code de vérification des méthodes de vérification parente. Cette méthode est notamment utilisée par Cofoja (Minh Lê, 2016). Toutefois cette solution pose une problématique de double évaluation des contrats. Reprenons l'exemple 3.4, si nous appelons la méthode de vérification de B celle-ci va mettre en oeuvre la vérification définit dans A et B, de même si nous appelons la méthode de vérification de C nous vérifierons A et C. Si maintenant nous appelons la méthode de vérification de D, celle-ci met en oeuvre la vérification définit dans C et B, sauf que toutes deux mettent en oeuvre la même vérification de A, ce qui provoquera une double évaluation de la condition définie dans la classe A. Une solution mise en oeuvre par Eiffel (Eiffel, 2019) pour la vérification des invariants de classe, est d'effectuer la vérification avec une liste de travail. Chaque fois qu'une vérification est effectuée une nouvelle entrée est ajoutée à la liste pour signifier que le contrat a déjà fait l'objet d'une vérification.

Une autre piste de solution serait d'utiliser un appel à la prochaine définition (appel à *super*) déterminé grâce à la linéarisation. La linéarisation consiste à déterminer l'ordre de résolution des méthodes, pour exemple dans la Figure 3.4 l'ordre de résolution de la classe D pourrait être [D, B, C, A]. Ainsi si nous procédons à l'appel à *super* sur la méthode de vérification redéfinie dans la classe D, celle-ci appellera la définition de la classe B qui à son tour appellera la définition de la classe C jusqu'à arriver à l'introduction de la propriété dans la classe A. Toutefois, comme le font remarquer Ducournau et Privat (Ducournau et Privat, 2011) la prise en charge de l'appel dans les langages n'est pas identique, il faut

ainsi nécessairement prendre en considération les spécificités du langage hôte, afin de fournir une implémentation adaptée.

3.2.3 Verrouillage des contrats

Pour rappel dans la Section 2.2.2 nous avons abordé la définition de la sémantique de vérification des contrats. Celle-ci consiste à ne pas évaluer un contrat lorsque le programme est déjà dans l'évaluation d'un contrat, et ne pas évaluer les invariants de classe sur l'instance courante. Outre l'aspect logique de cette sémantique défini par Meyer (Meyer, 1992), celle-ci permet avant tout d'éviter la problématique de récursion infinie sous-jacente à la possibilité de définition des clauses de contrat à l'aide de méthodes de requête.

Certaines implémentations comme D (D Language Foundation, 2020) ainsi qu'AspectJML (Rebêlo *et al.*, 2014) et OpenJML (Cok, 2014) ont fait le choix de ne pas mettre en oeuvre cette sémantique de verrouillage. Toutefois afin d'éviter les boucles infinies il est conseillé aux programmeurs de faire attention à ne pas introduire de contrat croisé et de seulement définir des invariants à l'aide de méthodes statiques, d'attributs ou en utilisant des méthodes de modèle (Cheon *et al.*, 2005) comme c'est le cas notamment pour les implémentations basées sur JML AspectJML et OpenJML.

Pour les implémentations qui ont choisi de mettre en oeuvre cette sémantique de verrouillage, notamment Liberty Eiffel. (Liberty Eiffel, 2019) ISE Eiffel (Eiffel, 2019), Code Contracts (Microsoft, 2018) ou encore Cofoja (Minh Lê, 2016) celles-ci se basent sur l'utilisation d'une variable ou méthode permettant de connaître l'état de la vérification.

Cofoja (Minh Lê, 2016) introduit une classe d'assistance qui a pour objectif de venir contrôler l'évaluation des contrats. Ce contrôle passe par la récupération pour

chaque méthode d'un contexte de contrat, qui est défini à l'aide de l'identifiant de fil d'exécution actuel, du type de la classe déclarante ainsi que de la signature de la méthode. Grâce à cela il est possible d'identifier chaque appel de méthode de manière fine afin de vérifier s'il est possible d'effectuer l'évaluation d'un contrat, et de savoir si la vérification des invariants de classes doit être appelée. En d'autres termes cette mise en oeuvre est comparable à la définition d'une pile d'exécution parallèle propre à l'évaluation des contrats.

En ce qui concerne l'implémentation EiffelStudio (Eiffel, 2019), celle-ci procède à la définition d'une variable booléenne globale au programme avec pour particularité qu'en fonction du contexte (réalisation d'un programme avec plusieurs fils d'exécution ou non) la variable est définie comme ayant une valeur propre à chaque fil d'exécution afin de garantir que l'évaluation d'un contrat dans un fil d'exécution n'empêche pas l'évaluation dans un autre fil d'exécution indépendant. Cependant, à la différence de l'implémentation proposée par Cofoja (Minh Lê, 2016) qui utilise plusieurs informations (identifiant du fil d'exécution, type de la classe et la signature de la méthode) pour permettre de réaliser les deux sémantiques de verrouillage (invariant sur self et verrouillage des contrats lors de l'évaluation), la variable mise en oeuvre par EiffelStudio permet seulement de garantir que les contrats ne peuvent être évalués lors de l'évaluation d'un autre contrat. Pour garantir que les invariants seront seulement vérifiés lorsque l'appel d'une méthode n'est pas effectué sur l'instance courante, un second attribut global similaire au premier est injecté, permettant de représenter cette information. Pour ce faire, chaque fois que l'on détecte que l'appel est imbriqué on vient définir la valeur de la variable à *vrai* ce qui indiquera que l'invariant de classe peut être vérifié.

Bien que mettant en oeuvre un verrouillage lors de l'évaluation, les implémentations de Liberty Eiffel (Liberty Eiffel, 2019) et Code Contracts (Microsoft, 2018)

sont quelque peu différentes. Liberty Eiffel a choisi de représenter l'état du système à l'aide d'une variable statique dans la méthode. Celle-ci évite que les contrats d'une méthode ne soient évalués de nouveau, ce qui garantit que le programme ne rentrera pas dans une récursion infinie. Cette sémantique définit ainsi que chaque méthode exécutera une fois ces contrats sans distinction. Code Contracts quant à lui utilise une méthode utilitaire pour obtenir le niveau de profondeur de récursion propre à chaque appel de contrat. La profondeur de vérification pour chaque contrat est définie à quatre par défaut, ce qui signifie que chaque appel de contrat pourra engendrer dans le pire des cas la vérification de trois niveaux de sous-contrats.

Le code illustré par la Figure 3.5 introduit la définition d'une pré-condition croisée entre *foo()* et *bar()*. Ce code permet d'exhiber les effets sur l'évaluation des différentes mises en oeuvre du verrouillage des contrats. Les Figures 3.6, 3.7 et 3.8 quant à elles représentent respectivement les résultats de l'exécution de ce code pour les implémentations Cofoja (Minh Lê, 2016), ISE Eiffel (Eiffel, 2019), Liberty Eiffel (Liberty Eiffel, 2019) et Code Contracts (Microsoft, 2018).

```
class A
    fun foo(f : Int, b : Int): Bool is
        expect(self.bar(f, b + 1))
    do
        print "foo({f})"
        return true
    end

    fun bar(f : Int, b : Int): Bool is
        expect(self.foo(f + 1, b))
    do
        print "bar({b})"
        return true
    end
end

var a = new A
a.foo(1,1)
```

Figure 3.5 Exemple politique de vérification avec contrat croisé.

```
bar(2) # Résultat du contrat de foo
foo(1) # Résultat de l'appel de foo
```

Figure 3.6 Résultat de la politique de verrouillage Cofoja et ISE Eiffel.

```
foo(2) # Résultat du contrat de bar
bar(2) # Résultat du contrat de foo
foo(1) # Résultat de l'appel de foo
```

Figure 3.7 Résultat de la politique de verrouillage Liberty Eiffel.

```
bar(4) # Résultat du contrat de foo niveau de récursion 4
foo(3) # Résultat du contrat de bar niveau de récursion 3
bar(3) # Résultat du contrat de foo niveau de récursion 3
foo(2) # Résultat du contrat de bar niveau de récursion 2
bar(2) # Résultat du contrat de foo niveau de récursion 2
foo(1) # Résultat de l'appel de niveau de récursion 1
```

Figure 3.8 Résultat de la politique de verrouillage Code Contracts.

3.3 Conclusion

Pour conclure, ce chapitre a permis de répondre à la question « *Comment mettre en oeuvre la vérification de la programmation par contrat dans un langage à objet ?* », en abordant les différentes parties qui constituaient cette réalisation. Le Tableau 3.1 permet de résumer les différentes approches mises en oeuvre par solution étudiée.

	dmd	ISE Eiffel	Liberty Eiffel	Cofoja	OpenJML	AspectJML	Code Contracts
Analyse et génération (3.1)							
Compilateur	×	×	×		×	×	
Préprocesseur				×			×
Point de vérification (3.2.1)							
Site d'appel						×	×
Méthode	×	×	×	×	×	×	×
Représentation des contrats (3.2.1)							
Code de vérification	×	×	×		×		×
Méthode dédiée		×		×			
Aspect						×	
Verrouillage des contrats (3.2.3)							
Attribut global		×		×			
Variable statique par méthode			×				
Méthode de calcul de profondeur							×

Tableau 3.1 Résumé de la mise en oeuvre de la programmation par contrat dans les solutions étudiées.

CHAPITRE IV

MISE EN ŒUVRE EN NIT

Dans les chapitres précédents nous avons évoqué les mécanismes pour intégrer la programmation par contrat dans un langage à Objet. Ce chapitre va se concentrer sur la mise en oeuvre de la programmation par contrat dans le langage Nit. Dans un premier temps les Sections 4.1 et 4.2 présentent et définissent les concepts préalables à la mise en oeuvre de notre proposition. Dans un second temps nous présentons dans la Section 4.3 une vue d'ensemble de notre proposition. Pour finir les Sections 4.4 , 4.5 et 4.6 se concentrent sur des aspects bien précis de notre solution tels que la structure et la déclaration des contrats.

4.1 Le langage Nit

Nit est un langage de programmation orienté objet statiquement typé. Il est l'évolution du langage PRM anciennement développé au LIRMM, à Montpellier, France. Ce langage est développé dans un cadre académique, et a pour vocation de fournir un cadre d'étude à la mise en pratique et à l'étude de nouveaux concepts de programmation et de technique de compilation. Nit met en oeuvre un grand nombre de concepts. On retrouve notamment : l'héritage multiple, la généricité, les types virtuels et le raffinement de classe (Privat et Ducournau, 2005).

Un programme Nit se caractérise par un ensemble de modules. Chaque module

peut être considéré comme une entité autonome et offre la possibilité au programmeur de déclarer un ensemble d'interfaces et de classes ainsi que son propre point d'entrée (*main*). Un module peut importer d'autres modules. Grâce à cette importation le programmeur peut alors profiter de l'ensemble des classes interfaces ainsi que des importations effectuées dans ce module, afin d'utiliser, de sous-classer ou raffiner l'ensemble des classes et interfaces.

Le raffinement (Privat et Ducournau, 2005) est l'un des concepts central du langage Nit, il permet au programmeur de venir modifier la mise en oeuvre des éléments existants. Grâce au raffinement il est ainsi possible de modifier l'implémentation de méthodes, d'étendre les classes par l'ajout de classe parente et/ou de propriétés (attribut et méthode).

Nit propose aussi un concept d'annotation comme présenté dans la section 2.1.3. Ce mécanisme a pour principal objectif de marquer les éléments du langage afin de leur associer une sémantique additionnelle. On retrouve notamment leur utilisation pour la définition des méthodes abstraites grâce au mot-clé *is* suivit de l'annotation *abstract*.

4.2 Architecture de Nit

Afin d'étayer l'ensemble des choix réalisés, il convient de détailler plus en profondeur le langage Nit. Dans un premier temps nous allons nous intéresser à la structure des outils permettant d'exécuter un programme écrit en Nit. En effet, l'objectif final est de fournir une proposition disponible sur l'ensemble de ces outils. Les deux outils principaux sont le compilateur (nitc) et l'interpréteur (nit). La Figure 4.1 permet d'illustrer le processus de compilation/interprétation actuel. Comme on peut le constater, ces deux entités partagent la même face avant (*front-end*) qui a pour but d'analyser le code source.

La face avant est découpée en trois étapes principales :

— Analyse lexicale : identifie les jetons dans le code source. En sortie de cette analyse nous avons une liste de jetons.
— Analyse syntaxique : permet de vérifier la validité syntaxique du programme. On cherche à définir si le texte source appartient au langage décrit par la grammaire. En sortie de cette étape nous obtenons un arbre syntaxique abstrait (AST).
— Analyse sémantique : celle-ci est découpée en plusieurs analyses. On retrouve notamment l'analyse de types, l'analyse de la portée des identifiants, etc. Le résultat de cette étape est l'obtention d'un arbre syntaxique abstrait (AST) augmenté ainsi que l'obtention du modèle objet représentant les entités (modules, classes, méthodes ...) de notre programme, ainsi que leurs relations.

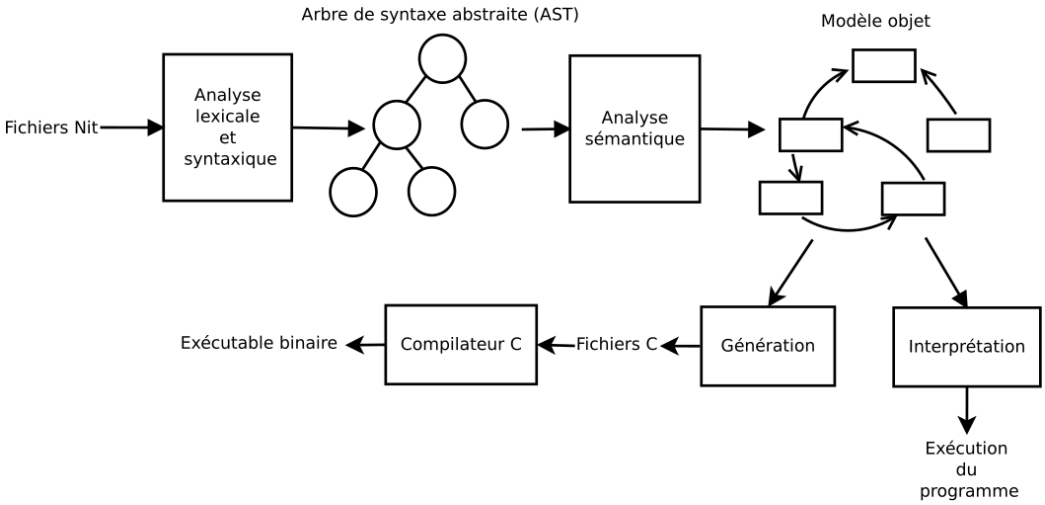

Figure 4.1 Processus de compilation/interprétation de Nit.

Pour la suite nous allons nous focaliser sur l'analyse sémantique. Cette analyse est structurée de manière modulaire grâce à différentes entités appelées phases. Chacune d'entre elles a un objectif précis. L'ensemble de ces phases est représenté

à l'aide d'un ensemble partiellement ordonné. Cette structure permet de gérer les potentielles dépendances entre toutes les phases et ainsi effectuer l'exécution de manière adéquate. En effet, certaines phases sont nécessaires pour l'exécution d'autres, c'est notamment le cas entre la phase de typage qui doit par exemple être exécutée seulement une fois que l'ensemble des phases de construction du modèle on était effectués. Cet ensemble est notamment constitué de phases destinées à la vérification comme c'est le cas pour vérifier la validité des annotations utilisées. Outre des phases de vérification, nous pouvons aussi trouver des phases permettant d'étendre le langage, comme c'est notamment le cas avec les phases mettant en oeuvre l'implémentation automatique de la sérialisation (Laferrière, 2012) ou la gestion des méthodes destinées à être exécutées dans un autre fil d'exécution (Chanoir, 2017).

4.3 Vue d'ensemble de notre proposition

Dans la section précédente nous avons détaillé le processus de compilation/interprétation actuel représenté par la Figure 4.1. La Figure 4.2 quant à elle fait un zoom sur les modifications apportées au processus de compilation/interprétation de Nit. Notre proposition vient étendre l'analyse sémantique grâce à l'ajout d'une phase spécifique à la prise en charge de la programmation par contrat. Cette phase s'effectue en deux temps. Dans un premier temps, nous analysons l'ensemble des méthodes, mutateurs, interfaces et classes pour savoir s'ils introduisent des contrats à l'aide d'annotations spécifiques, si c'est le cas nous générons l'ensemble du code d'implémentation. Le second temps de la phase est dédié au tissage des sites d'appels. L'objectif est de venir diriger les appels originaux sans vérification des contrats vers notre infrastructure de vérifications générées dans l'étape précédente. En sortie de cette phase nous obtenons un modèle objet ainsi qu'un arbre de syntaxe abstraite étendu pour prendre en charge la programmation par contrats.

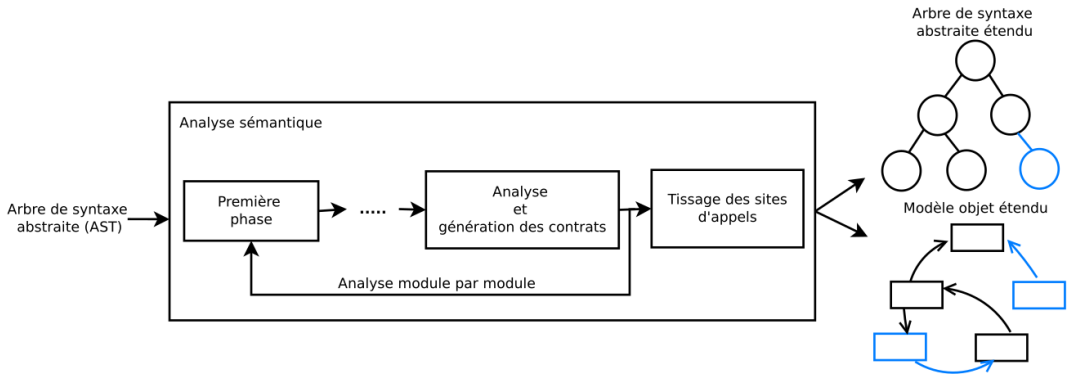

Figure 4.2 Processus d'analyse et de génération des contrats en Nit.

Les Figures 4.3 et 4.4, présentent respectivement la déclaration des classes *Maillon* et *ListeChaine* en utilisant notre proposition.

```
1  class Maillon[T]
2      private var element: T
3      private var maillon_suivant: nullable Maillon[T]
4
5      fun obtenir_element: T do return element
6
7      fun obtenir_maillon_suivant: nullable Maillon[T]
8      do
9          return maillon_suivant
10     end
11
12     fun definir_maillon_suivant(maillon : Maillon[T])
13     is
14         ensure(self.obtenir_maillon_suivant == maillon) # Post-condition
15     do
16         maillon_suivant = maillon
17     end
18
19     fun oublier_maillon_suivant
20     is
21         ensure(self.obtenir_maillon_suivant == null) # Post-condition
22     do
23         maillon_suivant = null
24     end
25 end
```

Figure 4.3 Déclaration de la classe *Maillon* en Nit.

```
class ListeChaine[T]
    invariant( self.taille >= 0 )

    var taille: Int = 0
    private var premier_maillon: nullable Maillon[T]

    fun non_vide: Bool
    is
        ensure(result == (taille > 0)) # Post-condition
    do
        return taille > 0
    end

    fun dernier_element: T
    is
        expect(self.non_vide) # Pré-condition
    do
        return premier_maillon.obtenir_element
    end

    fun ajouter(element : T)
    is
        ensure(dernier_element == element) # Post-condition
        ensure(old(taille) + 1 == taille) # Post-condition
    do
        var nouveau_maillon = new Maillon[T](element, premier_maillon)
        self.premier_maillon = nouveau_maillon
        self.taille = self.taille + 1
    end

    fun depiler: T
    is
        expect(self.non_vide) # Pré-condition
        ensure(old(taille) - 1 == taille) # Post-condition
        ensure(old(dernier_element) == result) # Post-condition
    do
        var ancien_element = self.premier_maillon.obtenir_element
        premier_maillon = self.premier_maillon.obtenir_maillon_suivant
        taille = taille - 1
        return ancien_element
    end
end
```

Figure 4.4 Déclaration de la classe *ListeChaine* en Nit.

4.4 Déclaration des annotations

Dans le Chapitre 2 nous avons abordé les différentes manières pour spécifier un contrat, à savoir, l'utilisation de mots-clés, commentaire, méthode dédiée et an-

notation. Lors de la phase d'analyse, il a été choisi d'utiliser les annotations pour procéder à la déclaration des contrats. Ce choix a été effectué pour les raisons suivantes :

— L'utilisation des annotations a un coût syntaxique nul. En effet, dans notre cas, le langage Nit offre déjà la possibilité d'ajouter des méta-informations à l'ensemble des éléments du langage nécessaire aux contrats (interface, classe, méthode et mutateur).
— Les annotations et les éléments qui la composent sont déjà validés sur le plan lexical et syntaxique. En effet, les éléments qui composent l'annotation doivent être écrits directement en code Nit valide, ils ont donc leur propre représentation dans l'AST, ce qui facilite l'analyse du contenu de l'annotation ainsi que l'infrastructure de vérification.

Dans notre proposition nous retrouvons trois annotations pour déclarer les différents types de contrats. Nous avons, les pré-conditions représentées à l'aide de l'annotation *expect*, les post-conditions avec l'annotation *ensure* et les invariants de classe déclarés à l'aide de *invariant*. L'ensemble des annotations sont illustrées avec la Figure 4.4.

L'ensemble des annotations sont structurées de manière identique. On retrouve ainsi une annotation (*expect*, *ensure*, *invariant*) suivie de la clause du contrat en argument. Cette clause doit être exprimée sous la forme d'une expression booléenne. Pour exprimer la clause du contrat, le programmeur peut utiliser l'ensemble des opérations disponible sur l'instance courante sans restrictions de visibilité, ainsi qu'utiliser l'ensemble des paramètres de la méthode. Il convient aussi de noter que les clauses des post-conditions *ensure()* offrent la possibilité de faire référence aux résultats d'une fonction grâce à l'utilisation du mot-clé *result* et de faire référence à la valeur d'une expression évaluée avant l'exécution de la méthode à l'aide du

mot-clé *old(expression_ a_ évaluer)*.

Notre proposition offre aussi différentes manières pour représenter la conjonction de plusieurs clauses. Il est possible de les découper en les séparant par des virgules *expect(clause_ un, clause_ deux)*, il est aussi possible d'utiliser plusieurs fois la même annotation *expect(clause_ un), expect(clause_ deux)*. Ces deux syntaxes auront pour effet de produire un résultat identique à avoir écrit la conjonction des deux conditions soit *expect(clause_ un and clause_ deux)*. L'objectif derrière cette mise en oeuvre est de faciliter la relecture, par le programmeur, des différentes conditions lors de la mise en oeuvre de contrat complexe.

Une autre annotation a aussi été introduite, *no_ contract*. L'objectif de cette annotation est de permettre au programmeur de supprimer la relation d'héritage des contrats. Cette annotation a dû être ajoutée pour éviter les erreurs lors de l'utilisation de la programmation par contrat dans un contexte de programmation parallèle. Effectivement, si nous prenons l'exemple de la méthode *depiler* celle-ci définit une pré-condition comme quoi la liste ne doit pas être vide afin que celle-ci puisse être appelée. Dans un contexte de programmation parallèle cette vérification devient quasiment impossible à effectuer pour le client car ce qui est valide au moment de la vérification peut avoir été rendu invalide par un autre client juste avant l'appel. Cette problématique est connue et est appelée *concurent paradox* (paradoxe de concurrence) par Meyer (Meyer, 1997) avec notamment pour solution la définition de condition d'attente (Nienaltowski *et al.*, 2009).

4.5 Structure des contrats

La Figure 4.5 représente la classe *ListeChaine* une fois la phase des contrats exécutés. Afin de pouvoir prendre en charge la programmation par contrat, notre proposition procède à l'ajout d'une infrastructure composé, d'une méthode de vérification dédiée pour chaque contrat, de nouveaux points d'entrée procédant à

l'appel des méthodes de vérification, une structure permettant la prise en charge du verrouillage de l'évaluation, ainsi qu'une structure additionnelle pour prendre en charge les références à *old* dans les post-conditions.

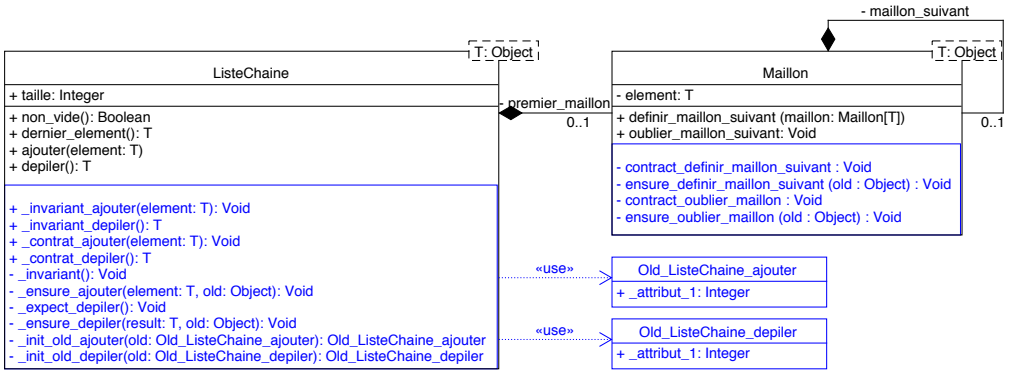

Figure 4.5 Représentation de l'exemple *ListeChaine* une fois compilé en Nit.

4.5.1 Méthode de vérification

Dans la Figure 4.5 nous pouvons identifier des méthodes nommées de la manière suivante ; *_ensure_*, *_expect_*, *_invariant*. Ces méthodes sont des procédures et permettent d'évaluer la clause qui est fournie aux annotations des contrats associés. Chaque type de contrat (pré-condition, post-condition et invariant) possède une méthode de vérification qui lui est propre. La signature de ces procédures est déterminée en fonction de l'élément auquel elle est rattachée.

— Les procédures de vérification des invariants ont une signature vide.
— La signature des procédures de vérification des pré-conditions a une signature équivalente à celle de la méthode à laquelle elle est rattachée.
— La signature des procédures de vérification des post-conditions est quant à elle composée de l'ensemble des paramètres de la méthode auquel elle est rattachée, d'un paramètre dédié à l'utilisation des *old* (abordé plus en détail

dans la Section 4.5.1) ainsi qu'un paramètre utilisé pour faire référence au retour de la fonction *result*.

Prenons l'exemple de la méthode *depiler* de la classe *ListeChaine*. Cette méthode introduit deux types de contrats ; pré-condition et post-condition. Lors de la compilation deux méthodes d'évaluation, une pour la pré-condition et l'autre pour la post-condition, sont ainsi créées comme présentées dans la Figure 4.6. Le corps de ces méthodes de vérification est uniquement constitué d'une assertion *assert* qui prend comme condition l'argument qui est fourni à l'annotation comme on peut le voir dans la Figure 4.6.

```
1  # Méthode de vérification des invariants de classe
2  redef fun _invariant do
3      assert taille >= 0
4      super
5  end
6
7  # Méthode de vérification de la pré-condition
8  fun _expect_depiler do assert non_vide
9
10 # Méthode de vérification de la post-condition
11 fun _ensure_depiler(result: T, old: Old_Pile_depiler) do
12     assert taille == (old.taille - 1)
13     assert result == old.dernier_element
14 end
```

Figure 4.6 Représentation de la méthode *depiler* avec les méthodes de vérification et de la méthode de vérification des invariants de classe.

Pour la procédure de vérification des invariants de classe celle-ci est composée comme nous pouvons le voir dans la Figure 4.6 de l'utilisation d'une assertion suivit d'un appel *super*. En effet, vu que chaque raffinement et spécialisation peut introduire des invariants de classe, il devient difficile de maintenir une structure

d'invariant pour chaque interface/classe. Afin de résoudre cette problématique notre proposition se base sur l'utilisation de l'appel à *super* et de sa linéarisation comme présentée dans la Section 3.2.2. Chaque procédure de vérification des invariants de classe vient ainsi évaluer les clauses qui lui sont propres et déléguer les vérifications des invariants hérités aux implémentations parentes. Il convient de noter que la procédure de vérification des invariants de la classe racine *Object* est fournie avec une implémentation vide. Grâce à cette mise en oeuvre, on évite l'introduction de plusieurs procédures de vérification distincte ce qui facilite grandement la prise en charge de l'héritage multiple.

La Figure 4.7 montre la représentation des procédures de vérification des pré-conditions et post-conditions lors de la présence d'héritages et de raffinements. Nous pouvons ainsi remarquer que les deux procédures se basent sur le principe d'appel à *super*.

La procédure d'évaluation des pré-conditions en-capsule l'appel à *super* dans une structure conditionnelle *if* avec comme condition la négation de la nouvelle clause fournie lors de la redéfinition. Cette représentation permet ainsi de respecter les règles d'héritage introduites dans la Section 2.2.5, indiquant qu'une pré-condition doit être égale à la disjonction de l'ensemble des pré-conditions héritées. En effet, si l'évaluation de la nouvelle condition échoue, nous déléguerons les vérifications des pré-conditions héritées aux implémentations parentes. Le corps de la méthode d'introduction étant mis en oeuvre avec une assertion, nous sommes sûrs que si le contrat initial n'est pas respecté, alors l'exécution du programme se terminera avec une erreur indiquant à l'utilisateur que la pré-condition n'est pas satisfaite. Notons que nous avons choisi d'utiliser la négation *not* pour représenter la condition du *if* dans le but de simplifier la structure du code généré.

En ce qui concerne la procédure d'évaluation des post-conditions, celle-ci est iden-

tique aux invariants présentés précédemment, la redéfinition de la méthode est composée d'un appel à *super* permettant de déléguer les vérifications des post-conditions héritées aux implémentations parentes suivit d'une expression *assert* pour évaluer la nouvelle clause. Cette représentation permet ainsi de garantir que la nouvelle post-condition sera égale à la conjonction de l'ensemble des post-conditions parentes.

```
1  class NouvelleListeChaine[T]
2      super ListeChaine[T]
3      invariant(nouvelle_expression_booleenne)
4
5      redef fun depiler: T is expect(), ensure()
6
7      redef fun _invariant do
8          assert nouvelle_expression_booleenne
9          super
10     end
11
12     redef fun expect_depiler do
13         if not nouvelle_expression_booleenne then super
14     end
15
16     redef fun ensure_depiler(result: T, old: Old_Pile_depiler) do
17         super
18         assert nouvelle_expression_booleenne
19     end
20 end
```

Figure 4.7 Représentation de la méthode *depiler* avec l'héritage des contrats.

Post-condition *result*

Comme nous pouvons le voir dans la Figure 4.4, précédemment introduite la fonction *non_vide* définit une post-condition qui fait référence aux résultats de retour. Notre proposition met en place la capture du résultat d'une fonction grâce

à l'ajout d'un paramètre *result* à la signature de la méthode de vérification. La Figure 4.8 montre le code de la méthode de vérification.

```
1 fun non_vide: Bool is ensure(result == (taille > 0))
2
3 # Méthode de vérification de la post-condition
4 fun _ensure_non_vide(result: Bool) do assert result == (taille > 0)
```

Figure 4.8 Méthode de vérification avec référence au résultat.

Post-condition *old*

La référence à une expression évaluée avant l'exécution d'une méthode dans les post-conditions est rendue possible grâce à l'ajout de plusieurs éléments :

- Une classe représentant l'ensemble des valeurs *old* évaluée dans une post-condition.
- Une méthode permettant l'initialisation d'une instance de cette classe
- Un paramètre dans la signature de la méthode de vérification des post-conditions permettant de fournir à notre méthode de vérification de l'instance de la classe de représentation.

La nouvelle classe introduite est uniquement constituée d'attributs, chacun permet de stocker la valeur d'expression associée à un *old*. La Figure 4.9 montre la représentation de cette classe pour la méthode *depiler*.

Une fois la classe de représentation créée, une fonction d'initialisation est injectée afin de venir initialiser l'ensemble des attributs de l'instance de notre classe de représentation. Cette fonction a des paramètres identiques à la méthode auxquels est rattachée la post-condition, ce qui permet aussi d'utiliser les arguments de la méthode dans un contexte de *old*. Cette méthode définit aussi un paramètre

```
class Old_ListeChaine_depiler[T]
    var attribut_1: Int # Représente old(taille)
    var attribut_2: T # Représente old(dernier_element)
end
```

Figure 4.9 Classe de représentation des expressions évaluées en tant que *old()*.

d'entrée et de retour de type *old*. Lors de l'introduction, cette fonction se contente de venir initialiser les attributs de la classe et de retourner l'instance créée. Cependant, lors d'une redéfinition cette méthode effectuera d'abord un appel à la définition parente (*super*) afin d'agréger notre objet avec l'ensemble des attributs initialisés qui seront utilisés par l'ensemble des clauses de la procédure de vérification des post-conditions. La Figure 4.10 montre la représentation de cette fonction pour la méthode *depiler*, la Figure 4.11 quant à elle montre la redéfinition de la méthode d'initialisation des attributs en présence d'héritage.

```
fun _init_old_Pile_depiler(Old: Old_Pile_depiler): Old_Pile_depiler
do
    # Initialisation des attributs
    old.attribut_1 = taille
    old.attribut_2 = dernier_element
    return old
end
```

Figure 4.10 Fonction d'initialisation de l'objet *old*.

Notons que le principal désavantage de cette technique de représentation est le coût d'instanciation. En effet, une instanciation systématique à chaque appel entraînerait un coût conséquent en termes de mémoire et de temps. Pour éviter cela, nous avons intégré une option *--keep-old-instance* qui permet de conserver l'instance de *old* en tant qu'attribut de la classe. Notons toutefois que dans un

```
redef fun _init_old_Pile_depiler(old: Old_Pile_depiler): Old_Pile_depiler
do
    super
    # Initialisation des attributs utilisés dans la redéfinition
    #...#
    return old
end
```

Figure 4.11 Fonction d'initialisation de l'objet *old* en présence d'héritage.

contexte d'application parallèle, l'utilisation de cette technique n'est pas conseillée afin que les évaluations ne rentrent pas en conflit.

4.5.2 Point d'entrée

Afin de modifier le moins possible le code source fourni par l'utilisateur, notre proposition introduit de nouveaux points d'entrée dédiés aux contrats pour les méthodes assujetties à des contrats, la méthode initiale n'est alors nullement modifiée. Ces points d'entrée, que nous appelons aussi « facettes », permettent de diminuer l'impact dynamique des contrats. Effectivement, le choix du point d'entrée peut être déterminé lors de la compilation en fonction des politiques de vérification.

Il existe deux types de points d'entrée. Les points d'entrée destinés à la vérification des pré-conditions et des post-conditions et les points d'entrée destinés à la vérification des invariants. Cette séparation entre les points d'entrée est effectuée afin de pouvoir appliquer le principe évoqué par Meyer (Meyer, 1997) qui indique que la vérification des clauses d'invariants doit uniquement être effectuée lorsque l'appel n'est pas effectué sur l'instance courante. Comme nous avons pu le voir dans la Section 3.2.3 la majorité des solutions réalise cette dissociation entre appels sur l'instance courant ou non à l'aide d'une valeur booléenne. Toutefois, cette

approche a un coût lors de l'exécution, or cette information peut être déterminée de manière statique. Grâce à cette séparation lors de la phase de tissage des contrats il sera possible de diriger les appels sur l'instance courante directement vers le point d'entrée vérifiant les pré-conditions et post-conditions.

La Figure 4.12 montre les différents points d'entrée générés pour la méthode *depiler* de la classe *ListeChaine*. On retrouve le point d'entrée de vérification des pré-conditions et post-conditions représentées par les méthodes *_contrat_depiler*. Ce type de point d'entrée est en charge de la mise en place des différents éléments des contrats, capturer le retour de la fonction, appeler l'initialisation des *old* afin de les fournir en argument à la post-condition. On retrouve aussi les points d'entrée des invariants qui sont introduits à l'ensemble des propriétés (introduites et hérités) lors de la définition d'un invariant *_invariant_depiler*. Bien que la Figure 4.12 montre uniquement les points d'entrée pour la méthode *depiler*, le même traitement est appliqué à l'ensemble des méthodes.

Il convient de noter qu'en présence d'héritages et de redéfinition de contrat, seul le point d'entrée de vérification des pré-conditions et des post-conditions peut se trouver redéfini. Cette redéfinition a lieu lorsque le programmeur introduit un nouveau type de contrat qui n'avait pas été défini précédemment (par exemple l'ajout d'une post-condition). Nous pouvons aussi noter qu'une redéfinition sera effectuée si le programmeur introduit un (*old*) dans une post-condition, et que celle-ci n'en faisait pas l'objet auparavant. Cette redéfinition permet ainsi de mettre en oeuvre l'initialisation de la classe de représentation (Section 4.5.1).

4.5.3 Verrouillage

Comme nous l'avons abordé dans la Section 2.2.2 un problème subsiste dans l'expression d'une assertion à l'aide d'une méthode. Ce problème est la possibilité d'obtenir une récursion infinie. Pour éviter ce problème nous avons encapsulé

```
fun _depiler_contrat: T do
    # Création et initialisation de l'objet Old
    var old = _init_old_Pile_depiler(new old_Pile_depiler)
    # Évaluation de la pré-condition
    _expect_depiler()
    # Appel de la méthode d'origine
    var result = depiler
    # Évaluation de la post-condition
    _ensure_depiler(result, old)
    return result
end

fun _depiler_invariant: T do
    # Appel du point d'entrée avec les pré-conditions et post-conditions
    var result = _depiler_contract
    # Évaluation des invariants
    _invariant()
    return result
end
```

Figure 4.12 Points d'entrée de vérification des contrats pour la méthode *depiler* de la classe *ListeChaine*.

l'appel des procédures de vérification dans une structure conditionnelle. Cette structure aura pour but de vérifier une variable globale *in_assertion* qui représente l'état du système à savoir en évaluation ou non. Ce verrouillage est introduit de manière native dans le code généré par le compilateur *nitc* et introduit à l'aide d'un attribut propre à l'interpréteur *nit*. La Figure 4.13 représente une vue générale du mécanisme de verrouillage.

Il convient de noter que pour prendre en considération des problématiques d'évaluation des contrats lors de la mise en place d'un programme utilisant la programmation parallèle à l'aide de différent fils d'exécutions dans le compilateur, cet attribut est défini de manière à ce que sa valeur soit locale à chaque fil d'exé-

```
if not in_assertion then
    in_assertion = true
    # Appel de la méthode de vérification du contrat
    in_assertion = false
end
```

Figure 4.13 Verrouillage de l'évaluation des contrats en Nit.

cution. Ce choix permet de garantir que si un contrat est en cours d'évaluation dans un fil d'exécution celle-ci n'empêchera pas l'évaluation de contrats dans un autre fil d'exécution.

4.6 Phase de contrat

La prise en charge des contrats en Nit se fait à l'aide d'une phase dédiée qui intervient sur l'ensemble des modules du programme. Cette phase vient s'exécuter en deux temps. Dans un premier temps nous visitons chaque module afin de générer l'infrastructure de vérification des contrats. Cette étape de construction est réalisée après le passage des phases de génération du modèle objet et du typage. En se plaçant à ce moment on profite de l'ensemble des informations nécessaires à la construction des contrats. En effet, la génération des contrats effectue une liaison entre les méthodes annotées par des contrats, et avec l'implémentation de ces contrats (méthode de vérification, point d'entrée, classe d'assistance) il est alors nécessaire d'avoir le modèle objet afin d'effectuer cette liaison. Le typage est quant à lui nécessaire afin de correctement définir les types des différents éléments générés comme les attributs de la classe utilisée pour conserver les *old* ainsi que les signatures des différentes méthodes. Il permet aussi de rapporter les erreurs dans les contrats le cas échéant.

Une fois cette étape terminée nous procédons à l'analyse de tous les envois de

messages pour définir si les propriétés ciblées font l'objet d'un contrat. Cette étape consiste à une visite de chaque site d'appel afin de déterminer si nous devons modifier la propriété ciblée pour la remplacer par le point d'entrée adéquat (pré-condition et post-condition ou invariant de classe).

```
1  module a
2  class A
3      fun foo: Int do return 1
4  end
5
6  fun bar(a: A) do
7      a.foo
8  end
9
10 module b
11 class B
12     super A
13     redef fun foo
14     is
15         ensure(result > 100)
16     do
17         return 2
18     end
19 end
20
21 bar(new B)
```

Figure 4.14 Exemple de problématique de tissage vers les points d'entrées.

La Figure 4.14 permet d'illustrer pourquoi l'étape de tissage est réalisée après l'analyse de l'ensemble des modules. Dans cet exemple nous avons, le module *a* qui instruit une classe avec deux méthodes, *foo* qui retourne *1* et *bar* qui fait appel à la méthode *foo*. Le module *b* vient introduire une classe *B* qui spécialise la classe *A*. La classe *B* vient redéfinir la méthode *foo* afin d'ajouter une post-condition sur le retour de celle-ci et fournit une implémentation adaptée à cette nouvelle

restriction. Or dans ce cas, si nous procédions à une analyse module par module de l'ensemble des appels à *foo* dans le module *a*, le site d'appel ne ferait jamais appel au point d'entrée de vérification des contrats de *foo*. Or comme la nouvelle implémentation de *foo* ne retourne pas un résultat supérieur à 100 celle-ci est considérée comme non conforme à la spécification et devrait être détecté il est donc essentiel de passer par le point d'entrée de vérification.

4.6.1 Configuration des contrats

Afin de fournir au programmeur une gestion plus fine de la mise en oeuvre des contrats, notre solution propose un ensemble d'options disponibles pour le compilateur *nitc* et l'interpréteur *nit*. Le programmeur a ainsi la possibilité de venir :

- Activer l'ensemble des contrats sur tout le programme *--full-contract*.
- Désactiver la vérification des pré-conditions et des post-conditions lorsque l'appel est effectué sur l'instance courante *--no-self-contract*.
- Activer la vérification des invariants de classe en entrée [1] *--in-out-invariant*
- Activer l'évaluation de l'ensemble des pré-conditions, post-conditions ou invariants de classe *--all-expect*, *--all-ensure --all-invariant*.

Par défaut si le programmeur ne fournit aucune directive la mise en oeuvre utilisera le mode *par défaut*. Ce mode a pour objectif de fournir le meilleur compromis en termes de performance et de vérification, pour ce faire, il met en oeuvre la stratégie suivante :

---

1. Par défaut la vérification des invariants de classe est seulement effectuée en sortie de la méthode. Ce choix est dû au fait qu'il est impossible de modifier les attributs de l'extérieur de la classe sans passer par des mutateurs qui font l'objet de la vérification des invariants de classe en sortie, ce qui signifie qu'il est peu probable que l'invariant de classe ne soit pas respecté en entrée.

— Les pré-conditions et post-conditions sont vérifiées de manière systématique à chaque appel dans le module principal.
— Les invariants de classe sont uniquement vérifiés en sortie.
— Dans les modules directement importés par le module principal seules les pré-conditions sont vérifiées.
— Les post-conditions et invariants de classe ne sont pas vérifiés dans les modules importés.

4.7 Conclusion

Dans ce chapitre nous avons abordé notre proposition de mise en oeuvre de la programmation par contrat dans le langage Nit. Sur le plan syntaxique notre proposition introduit les contrats à l'aide d'annotations spécifiques permettant d'expliciter les différents types de contrats : invariant de classe (*invariant*), pré-condition (*expect*) et post-condition (*ensure*). La génération du code de vérification des contrats est quant à elle divisée en deux étapes. D'un côté, nous avons les méthodes de vérification dédiée à chaque type de contrat et permettant de vérifier les clauses du contrat et de mettre en oeuvre la politique de prise en charge de l'héritage qu'il soit simple ou multiple. De l'autre, nous avons deux points d'entrée pour chaque méthode, un dédié aux pré-conditions et post-conditions et l'autre dédié à la prise en charge des invariants de classe. Ces points d'entrée sont aussi la responsabilité de la construction des *old*, du verrouillage de la vérification des contrats ainsi que de la capture du résultat de la fonction. Pour finir, notre solution ajoute différentes options afin de gérer de manière plus fine la mise en oeuvre des contrats. Le Tableau 4.1 vient reprendre les Tableaux 2.1 et 3.1 en y ajoutant notre proposition.

	Nit	dmd	ISE Eiffel	Liberty Eiffel	Cofoja	OpenJML	AspectJML	Code contracts
Syntaxe (2.1)								
Mots-clés		×	×	×				
Commentaire						×	×	
Annotation	×				×			
Méthode								×
Sémantique (2.2)								
Politique de visibilité			×	×		×	×	×
Verrouillage	×		×	×	×			×
Configuration	×	×	×	×	×			×
Result	×	×	×	×	×	×	×	×
Old	×		×	×	×	×	×	×
Héritage	×	×	×	×	×	×	×	×
Analyse et génération (3.1)								
Compilateur	×	×	×	×		×	×	
Préprocesseur					×			×
Point de vérification (3.2.1)								
Site d'appel							×	×
Méthode		×	×	×	×	×	×	×
Point d'entrée dédié	×							
Représentation des contrats (3.2.1)								
Code de vérification		×	×	×		×		×
Méthode dédiée	×		×	×	×			
Aspect							×	
Verrouillage des contrats (3.2.3)								
Attribut global	×		×		×			
Variable statique par méthode				×				
Méthode de calcul de profondeur								×

Tableau 4.1 Résumé de notre proposition face aux implémentations étudiées.

CHAPITRE V

VALIDATIONS EXPÉRIMENTALES

Dans ce dernier Chapitre, nous présentons une évaluation de notre proposition. Cette évaluation permet de répondre aux différentes questions évoquées dans la Section 5.1. Nous abordons ensuite dans la Section 5.2 la faisabilité de la définition des contrats dans la bibliothèque des collections du langage Nit et dans le compilateur *nitc*. Nous comparons dans la Section 5.3 l'impact sur le temps d'exécution de différents programmes Nit. Nous évaluons ensuite dans la Section 5.4 les performances de notre proposition face aux implémentations étudiées (dmd, ISE Eiffel, Cofoja, Code Contracts, OpenJML et AspectJML).

L'ensemble des résultats évoqués dans ce chapitre ont été obtenus à l'aide d'une machine ayant les caractéristiques suivantes : processeur Intel I7-4700HQ 2.40GHz 64Bits, 16Go de mémoire vive avec comme système d'exploitation Ubuntu 20.04.1 LTS. Chaque résultat représente la moyenne en seconde entre les cinq exécutions consécutives de la même expérimentation. Le temps d'exécution est déterminé à l'aide de la commande *GNU time* en prenant seulement en compte le temps utilisateur.

5.1 Questions d'expérimentations

Les différentes expérimentations réalisées sur notre proposition ont pour objectif de venir évaluer le coût de notre proposition. Nous allons ainsi dans les prochaines sections chercher à répondre aux questions suivantes :

1. Notre spécification des contrats est-elle utilisable dans du code Nit existant ?
2. L'évaluation des contrats prend t-elle plus de temps que l'utilisation d'assertion ?
3. Quelle partie de l'infrastructure représente le coût le plus élevé ? Dans l'infrastructure de notre proposition nous identifions deux potentielles sources de coût à savoir : l'envoi de messages (engendré par les points d'entrée et l'appel des méthodes de vérifications) et le verrouillage des contrats.
4. La politique d'utilisation par défaut (Section 4.6.1) des contrats est-elle pertinente en termes de coût d'utilisation et de vérification ?
5. Le coût général de notre proposition est-il acceptable au vu d'autres implémentations de programmation par contrat ?

5.2 Intégration des contrats en Nit

La validation de notre proposition passe par l'intégration des contrats dans la bibliothèque standard des collections du langage Nit, et dans le compilateur *Nitc*. L'intégration des contrats est effectuée en deux phases, la première consiste au remplacement d'expressions d'assertion utilisées à des fins de pré-condition et post-condition par leurs contrats respectifs. La seconde phase consiste à définir de nouveaux contrats en fonction des tests unitaires et de la documentation des méthodes.

5.2.1 Définition des contracts

La bibliothèque des collections du langage Nit est décomposée en 9 modules introduisant 13 interfaces et 22 classes (dont 4 abstraites). L'ensemble de ces classes/interfaces introduisent 151 méthodes. Avant l'introduction des contrats, 31 assertions étaient présentes dans le code. La procédure de remplacement a remplacé 26 assertions par l'utilisation de 20 pré-conditions et 1 post-condition. En ce qui concerne le compilateur *nitc* nous avons procédé au remplacement des assertions dans les paquets destinés à réaliser l'analyse sémantique et la génération du code [1]. Ensemble, ces paquets introduisent 91 classes, dont 14 abstraites, qui définissent 739 méthodes. Au total 176 assertions étaient utilisées, dont 49 ont pu être remplacés par l'utilisation de 47 pré-conditions et 2 post-conditions.

La Figure 5.1 introduit un exemple de remplacement d'une assertion pour la méthode *first* de l'interface *Collection*.

```
1 # Retourne le premier élément d'une liste.
2 fun first: E is expect(not_empty)
3 do
4     assert not_empty
5     #....#
6 end
```

Figure 5.1 Exemple d'assertion remplacé par des contrats dans la bibliothèque des collections et *nitc*.

Comme nous pouvons le voir les assertions ont principalement été remplacées

1. Notons, qu'une grande partie du code de l'analyse lexicale et syntaxique n'a pas fait l'objet de contrats car celui-ci est généré de manière automatique par SableCC (Gagnon et Hendren, 1998).

par l'utilisation de pré-condition. Effectivement, les assertions remplacées par des pré-conditions sont utilisées dans le but de provoquer un échec rapide (*fail-fast*) et sont souvent représentées avec des conditions peu coûteuses à évaluer, comme c'est le cas notamment pour vérifier que l'index fourni à une méthode d'ajout est correct. D'un autre côté les assertions sont très rarement utilisées comme des post-conditions dues généralement à leur coût d'évaluation plus élevé [2]. L'exemple typique est l'évaluation qu'un objet a correctement été ajouté à une liste. Effectivement, les assertions n'offrent aucun mécanisme de gestion fin [3] comme c'est le cas avec la programmation par contrat, il était alors peu pertinent d'introduire un grand nombre d'assertions potentiellement coûteuses à l'exécution.

En ce qui concerne l'ajout de nouveaux contrats, nous avons introduit 30 contrats dont 5 pré-conditions et 25 post-conditions dans la bibliothèque des collections. Pour le compilateur nous avons introduit 29 contrats dont 15 pré-conditions et 13 post-conditions. Notons toutefois qu'il serait possible d'ajouter un plus grand nombre de pré-conditions et post-conditions dans le compilateur avec une étude plus approfondie sur l'ensemble des méthodes. La Figure 5.2 introduit un exemple d'introduction de contrat pour la méthode *detach* de la classe *ANode*.

```
1 # Détache un noeud d'ast de son parent.
2 fun detach is ensure(parent == null)
```

Figure 5.2 Exemple de contrat ajouté dans *nitc*.

Le Tableau 5.1 permet de résumer le nombre de contrats remplacés et ajoutés.

---

2. Ou à une confiance inébranlable du développeur quant au bon fonctionnement de son implémentation.

3. À part l'activation ou désactivation totale sur l'ensemble du projet.

	Remplacé		Ajouté	
	Pré-condition	Post-condition	Pré-condtion	Post-condition
Bibliothèque des collections	20	1	5	25
nitc	47	2	15	13
Total	67	3	20	38

Tableau 5.1 Récapitulatifs des contrats définis.

5.2.2 Conclusion et discussion

Grâce à l'intégration des contrats dans la bibliothèque standard des collections du langage Nit, et dans le compilateur *Nitc*, nous avons pu répondre de manière positive à notre première interrogation (« Notre spécification des contrats est-elle utilisable dans du code Nit existant ? »).

L'introduction de ces contrats permet d'apporter un renforcement de la vérification, notamment dû à la mise en oeuvre systématique de la vérification des contrats dans l'ensemble des redéfinitions. Outre ce renforcement, l'introduction de contrat avec l'utilisation de méthode d'abstraction a pu mettre en évidence des erreurs dans les implémentations de certaines classes. C'était notamment le cas pour la classe *Bytes* qui représente une *array* d'octet où l'implémentation générique de la méthode *has* ne prenait pas correctement en compte les nouvelles spécificités de la classe.

À cela, nous pouvons aussi rajouter que la définition des contrats a aussi permis de mettre en évidence des écarts entre les commentaires et l'implémentation réelle.

5.2.3 Menaces à la validité

Toutefois, ces conclusions sont à mettre en perspective. En effet, la migration et l'ajout de contrat ont été réalisés dans un périmètre restreint tant en termes de contributeurs que de paquets impactés par ces modifications (seulement deux *nitc*, *Collection*). Une intégration à plus large échelle avec un plus grand nombre de paquets ainsi que de contributeurs aurait peut-être fourni des résultats différents, notamment sur l'expressivité ou sur le renforcement apporté par notre proposition.

5.3 Mesure des performances de la proposition

Dans cette section nous mesurons l'impact sur le temps d'exécution de l'introduction des contrats sur différents programmes Nit.

5.3.1 Mise en oeuvre de l'expérimentation

Pour étudier les effets de la mise en oeuvre des contrats dans la bibliothèque des collections et le compilateur, afin de répondre aux questions une à trois (5.1), nous procédons à l'analyse de l'exécution de différents programmes Nit :

— *nitc* Compilateur du langage Nit. Exécuté avec le mode de compilation séparée --*separate* et global --*global*.
— *nitdoc* Générateur de pages HTML de documentation à partir de fichiers source Nit.
— *nitlight* Génère la représentation HTML mise en valeur d'un fichier source Nit.
— *nitmd* Traduit les fichiers au format *Markdown* vers un format *html* ou *man*.
— *jwrapper* Générateur de classes externes Nit encapsulant le comportement d'API écrite en Java.

— *puzzle* Résolveur du problème n-puzzle.
— *queens* Résolveur du problème n-dames.

Nous avons choisi un ensemble de programmes qui font partie intégrante de la liste d'outil disponible pour le langage Nit. Ces outils ont pour avantage de ne pas être des programmes triviaux et permettent ainsi de se faire une réelle idée du coût engendré par les contrats. Nous retrouvons notamment l'un des projets les plus conséquents réalisés à l'aide du langage Nit à savoir le compilateur (*nitc*). Notons aussi la présence de deux démonstrateurs les résolveurs de problème *puzzle* et *queens*. Ce choix est dû à leur structure de données. En effet, en interne ces programmes utilisent des *Collections*, ce qui permet d'évaluer de manière intéressante l'impact sur des programmes utilisant principalement des appels à une bibliothèque qui met en œuvre des contrats.

Chaque programme est exécuté avec les scénarios suivants :

— Version d'origine avec les assertions.
— Remplacement des assertions par l'utilisation de contrat en activant l'ensemble des contrats.
— Utilisation des contrats (assertions remplacées et nouveau contrat) avec les politiques :
  — Désactivation de l'ensemble des contrats.
  — Activation de l'ensemble des pré-conditions et désactivation des post-conditions.
  — Politique par défaut voir Section 4.6.1 pour plus de détails.
  — Activation de l'ensemble des post-conditions et désactivation des pré-conditions.
  — Activation de l'ensemble des contrats avec une condition définit à *vrai*.

Notons que nous avons évalué mais ne présentons pas le scénario avec l'ensemble

des contrats activé dû au fait que celui-ci correspond au temps total de l'ensemble des post-conditions activé additionné à l'écart entre la version avec l'ensemble des pré-conditions activé et la version sans contrat [4]. Notons aussi que l'évaluation avec l'utilisation de toutes les pré-conditions activées, fournira un temps d'exécution quasiment similaire à la politique par défaut pour l'ensemble des programmes. Effectivement, la bibliothèque des collections est systématiquement considérée comme un import direct implicite [5] ce qui signifie que par défaut seul les pré-conditions seront vérifiées. Toutefois dans le contexte du compilateur, le temps devrait être différent vu que le compilateur introduit des post-conditions dans le module principal et sera donc vérifié avec la politique par défaut.

5.3.2 Résultat

La Figure 5.3 représente le temps d'exécution de l'ensemble des scénarios introduit précédemment. Concernant le questionnement *« L'évaluation des contrats prend t-elle plus de temps que l'utilisation d'assertion ? »* nous pouvons répondre oui mais celui-ci est négligeable. En effet, si nous prenons le détail pour le programme *nitc* en mode compilation séparé fourni par le Tableau 5.2 nous observons un écart de 3,7% entre le temps d'exécution avec assertion et le temps d'exécution avec les contrats, ce que l'on peut qualifier comme acceptable. Toutefois, cet écart prend aussi en considération la mise en oeuvre de l'héritage. En effet, les assertions n'étaient pas appliquées dans toutes les redéfinitions, le fait de les avoir convertis en contrat à l'introduction de la méthode a donc engendré des vérifications qui n'étaient pas effectuées auparavant. Pour mesurer de manière plus précise l'impact

---

4. temps avec l'ensemble des contrats activé = (temps avec pré-condition - temps sans contrat) + temps avec post-condition.

5. Exception faite lorsque le module principal importe un paquet qui importe lui-même la bibliothèque des collections.

de la vérification de contrat dû à l'héritage, nous avons procédé à l'exécution du programme avec les assertions remplacées par les contrats mais avec des conditions à *vrai*. Grâce à cette mesure, nous avons identifié que sur les 100 millions d'appels supplémentaires entre la version avec assertion et celle avec contrat, 20 millions d'appels sont dus à la prise en charge de l'héritage [6]. Cet ajout de vérification engendre ainsi un coût sur le temps d'exécution de 0,9%. On peut donc conclure que le coût supplémentaire de notre proposition dû au verrouillage et à l'envoi de messages est d'environ 2,8%.

Concernant la question *« Quelle partie de l'infrastructure représente le coût le plus élevé ? »* nous pouvons voir que le coût principal de notre proposition concerne la structure de verrouillage. En effet, entre le scénario avec l'ensemble des contrats activé avec des conditions à *vrai* et celui où nous effectuons uniquement le travail de verrouillage nous observons seulement un écart d'environ 2%. Nous pouvons donc conclure que l'écart de 13% entre la version sans contrat et la version avec les conditions évaluées à *vrai*, vient du verrouillage. Notons toutefois qu'au vu de la structure de notre implémentation il nous est difficile de mesurer uniquement le coût de verrouillage. Ce que nous mesurons en réalité lorsque nous parlons de version avec uniquement l'infrastructure de verrouillage c'est le coût de l'infrastructure plus le coût de l'appel au point d'entrée de vérification des contrats. La différence entre les deux mesures représente seulement le coût d'envoi de messages vers la méthode de vérification. Toutefois malgré l'ajout du coût de l'appel au point d'entrée dans la mesure de la structure de verrouillage, nous pouvons raisonnablement penser que celui-ci est dans le même ordre de grandeur que le coût de l'appel à la méthode de vérification soit aux alentours de 2%. Ce qui

---

6. Nombre d'appels supplémentaires dus à l'héritage = (nombre d'appels totaux - nombre d'appels avec condition à vrais) - (nombre d'appels avec assertion - nombre d'appels sans contrat ni assertion).

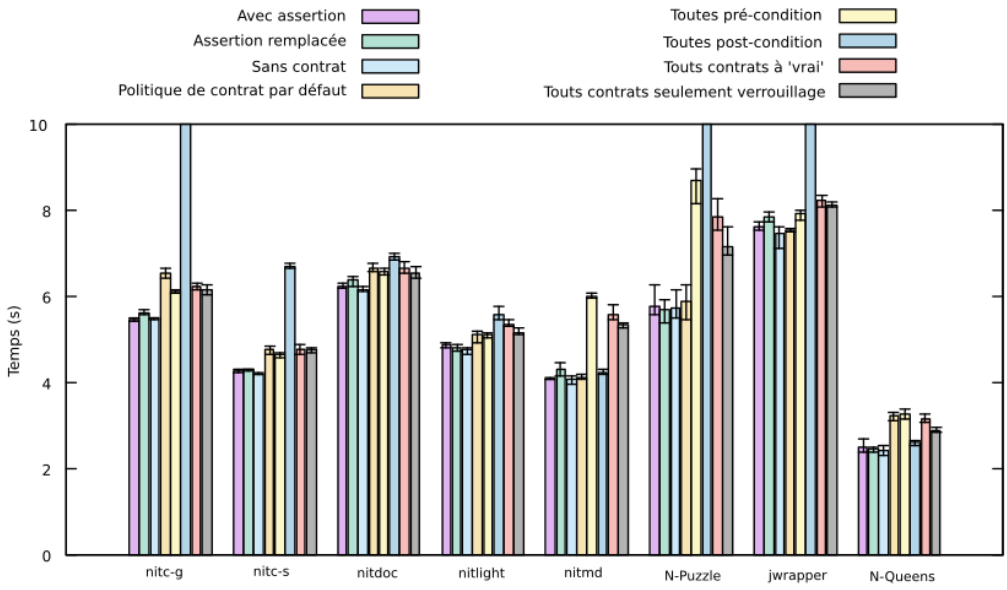

Figure 5.3 Résultats des expérimentations sur des programmes Nit.

représenterait un coût pour la structure de verrouillage d'environ 9%. Il convient de noter que le coût de l'envoi de messages pourrait augmenter avec la définition d'invariant de classe qui engendrerait l'apparition de deux points d'entrée pour l'ensemble des méthodes.

Pour la question « *La politique d'utilisation par défaut des contrats est-elle pertinente en termes de coût d'utilisation et de vérification ?* », nous pouvons répondre de manière affirmative. En effet, concernant le temps d'exécution nous sommes à +16% comparé à l'utilisation d'un programme sans assertion ce qui nous semble raisonnable. Déterminer la pertinence en termes de vérification est plus dur à prouver. Toutefois en nous basant sur le nombre de contrats vérifiés nous remar-

	Temps d'exécution (écart politique par défaut)	Nombre d'appel		
		Total d'appels du programme	Pré-conditions	Post-conditions
Assertion	4,32s	$\approx 12,5G$	0	0
Contrat	4,48s	$\approx 12,6G$	8 896 882	2 775
Ajout et remplacement avec contrat				
Pas de contrat	4,27s (−13,56%)	$\approx 12,3G$	0	0
Contrat par défaut	4,94s (0%)	$\approx 15,1G$	66 662 223	11 059 388
Pré-condition	4,8s (−2,83%)	$\approx 14,6G$	66 662 223	0
Post-condition	6,86s (38,86%)	$\approx 30,4G$	0	36 698 114
Condition vrai	4,91s (−0,6%)	$\approx 15G$	66 662 223	36 698 114
Structure de vérouillage	4,82s (−2,43%)	$\approx 14,2G$	66 662 223	36 698 114

Tableau 5.2 Détail de l'expérimentation sur le compilateur *nitc* en compilation séparée.

quons que celui-ci est égal à l'ensemble des pré-conditions plus une partie des post-conditions. En termes de chiffre, sur le nombre total des appels de contrat (77 721 611), nous en vérifions 11 059 388 qui n'étaient pas évalués soit une augmentation de 17%. Nous pouvons donc conclure que la politique que nous avons définie par défaut semble raisonnable et pertinente à l'utilisation de programme réel.

De façon plus générale, si nous comparons le temps avec l'ensemble des post-conditions avec le temps de l'ensemble des pré-conditions, nous pouvons valider l'hypothèse, évoquée dans la Section 5.2.1, que la mise en oeuvre de l'évaluation de post-condition a un coût d'évaluation plus élevé que les post-conditions. Cela

est principalement dû à l'ajout de post-condition pour vérifier qu'un élément a bien été ajouté à une liste. Effectivement, cette post-condition vient parcourir l'ensemble de la liste à chaque ajout pour valider l'insertion or cette itération a un coût important. On peut notamment le voir dans les outils comme *nitc*, *N-Puzzle*, *jwrapper* qui font un grand nombre de manipulations sur les listes.

Notons que dans la Figure 5.3 nous pouvons voir une particularité entre la version avec la politique par défaut et toutes les pré-conditions évaluées. Comme énoncé dans la Section 5.3.1 le temps de la version par défaut devrait être égal ou supérieur à la version avec l'ensemble des pré-conditions activées. Or dans le programme *nitmd* et *N-puzzle* le temps avec la politique par défaut est inférieur. Cela est dû au fait que ces deux programmes n'importent pas directement le paquet *core*, aucun site appel ne sera alors tissé vers les points d'entrées de vérification des contrats.

5.3.3 Conclusion et discussion

Dans les sections précédentes, l'objectif était de valider la mise en oeuvre des contrats dans un contexte d'utilisation réel avec des programmes en Nit. Nous avons ainsi pu répondre à une grande partie de nos questions de recherche. En conclusion à ces interrogations, nous pouvons dire, que la mise en oeuvre du remplacement des assertions engendre un surcoût d'environ 3% principalement engendré par la structure du verrouillage de l'évaluation des contrats. Lors de la mise en oeuvre de cette vérification nous avons ainsi introduit 128 contrats répartis dans la bibliothèque des collections, et dans le compilateur *nitc*. Grâce à cet ensemble de contrats nous avons ainsi pu tester la politique par défaut, et ainsi pouvoir affirmer que l'utilisation de cette politique offre un bon compromis comparé à l'utilisation des pré-conditions, post-conditions, seules ou bien toutes activées.

5.3.4 Menaces à la validité

L'ensemble des résultats obtenus dans le cadre du remplacement des assertions par l'utilisation des contrats sont toutefois à mettre en perspective. En effet, comme nous l'avons vu dans le Chapitre 4, notre proposition de programmation par contrat comprend plusieurs aspects dont certains ne rentrent pas en compte dans les résultats obtenus. C'est le cas notamment de la prise en charge des invariants de classe. Effectivement, le remplacement des assertions a uniquement introduit des pré-conditions et des post-conditions, le coût éventuel de l'envoi de messages supplémentaires engendré par l'ajout d'invariant de classe n'est ainsi pas mesuré. À cela, nous pouvons rajouter que les post-conditions n'utilisent pas les *old* ce qui pourrait être un facteur de ralentissement de notre proposition avec notamment l'instanciation d'une classe d'assistance pour stocker les valeurs évaluées [7] ou bien dû à l'envoi de messages supplémentaires engendré par l'initialisation des valeurs de l'instance.

5.4 Microbenchmarks

Tout au long de ce mémoire nous avons étudié diverses implémentations de programmation par contrats, l'objectif de cette section est d'effectuer un microbenchmark afin de mettre en perspective les performances de ces différentes implémentations, et ainsi pouvoir comparer notre mise en oeuvre avec l'existant. Il convient toutefois de noter que ces implémentations sont destinées à différents langages (C#, Java, Eiffel, D, Nit) ayant chacune des particularités distinctes [8]. Ces particularités peuvent ainsi avoir des impacts sur les performances mesurées. C'est

---

7. Bien que ce dernier puisse être minimisé par l'utilisation de l'option *--keep-old-instance*.

8. Comme l'utilisation de machine virtuelle procédant à l'optimisation du code dynamiquement (Java, C#), ou bien l'utilisation de ramasse-miette permettant de gérer la mémoire...

pour cela que l'information la plus pertinente concerne l'écart en termes de temps d'exécution pour le scénario sans et avec contrat.

5.4.1 Outils

Voici la liste de l'ensemble des outils, leur version ainsi que les options que nous allons utiliser :

- *nitc* dans la version *v0.8-3701-g3279c5d6c*[9]. Les options de compilations utilisées sont *--semi-global* et *--keep-old-instance*
- *dmd* dans sa version *DMD64 v2.090.1*.
- *ISE EiffelStudio* le compilateur intégré à l'outil Eiffel Studio version *ISE EiffelStudio 19.05.10.3187 GPL Edition - linux-x86-64*. Utilisé avec l'option *-finalize*.
- *OpenJML* version *0.8.46-20200505* exécuté avec *OpenJDK Runtime* version *1.8.0_265*. Pour la compilation nous utilisons les options *-rac* et *-racCompileToJavaAssert*.
- *AspectJML* version *1.8.0* exécuté avec *OpenJDK Runtime* version *1.8.0_265*.
- *Cofoja* version *1.3* utilisé avec le compilateur *javac* version *1.8.0_265*. Exécuté avec *OpenJDK Runtime* version *1.8.0_265*
- *mcs* le compilateur pour Mono C# version *6.8.0.105*. Le programme est ensuite exécuté avec *Mono JIT compiler* version *6.8.0.105*.

Notons que nous allouons 1Go de mémoire à l'exécution de *OpenJDK Runtime*.

5.4.2 Description du Micro-benchmark

N'ayant pas trouvé d'implémentation dédiée à l'évaluation des performances des contrats, nous avons décidé de mettre en oeuvre l'exemple de la classe *ListeChaine*

9. Disponible à l'adresse suivante : `https://github.com/Delja/nit/tree/rendu_memoire`.

introduit dans le Chapitre 1 avec la Figure 1.1, dans l'ensemble des langages. Cet exemple a pour avantage de ne dépendre d'aucune spécificité propre à un langage. Effectivement, l'implémentation ne dépend d'aucune bibliothèque spécifique ce qui permet de fournir une implémentation très similaire entre chaque langage.

Le microbenchmark consiste à l'ajout de *50 000 000* d'éléments dans une instance de la classe *ListeChaine*. Si le nombre d'insertions actuelles est pair nous appelons la méthode *dernier_ element*, à l'inverse si celui-ci est impair nous vérifions si la liste est vide et si cela n'est pas le cas nous appelons la méthode *depiler*. La Figure 5.4 illustre la mise en oeuvre du code en Nit pour le microbenchmark.

```
1  for x in [0..nombre_iteration[ do
2      liste.ajouter(x)
3      if x%2 == 0 then
4          liste.dernier_element
5      else
6          if liste.non_vide then
7              liste.depiler
8          end
9      end
10 end
```

Figure 5.4 Code du Micro-benchmark en Nit.

Il convient aussi de noter que l'implémentation de la liste chaînée pour le langage *D* diffère des autres afin de pouvoir prendre en charge les *old*. Pour prendre en charge cette fonctionnalité nous avons ajouté un attribut *OLD_ TAILLE* et *OLD_ ELEMENT* assigné à chaque entré dans le corps de la méthode.

5.4.3 Résultats

Le digramme de la Figure 5.5 représente le temps d'exécution de l'ensemble des implémentations avec et sans l'utilisation des contrats. Concernant la dernière

question d'expérimentation, *« Le coût général de notre proposition est-il acceptable au vu d'autres implémentations de programmation par contrat ? »* nous pouvons répondre que oui. En effet, si nous prenons le détail fourni par le Tableau 5.3, nous remarquons que notre proposition même si n'ayant pas le coût le plus faible comparé à des implémentations comme celles mises en oeuvre par les compilateurs *dmd* et *ISE Eiffel* reste acceptable au vu d'autres implémentations comme *Code Contract* ou bien *Cojofa*. L'écart entre notre implémentation et celle proposées par *ISE Eiffel* est principalement dû à l'envoi de messages engendré par les différents point d'entrée, ainsi que les appels engendrés par l'utilisation d'une instance d'assistance pour conserver les *old*. Quant à lui l'écart entre notre proposition et celle mise en oeuvre par le compilateur *dmd* est dû au verrouillage de la vérification des contrats, ainsi qu'à l'envoi de messages. Notons toutefois que l'option *--keep-old-instance* permet de diminuer significativement le coût des *old* en supprimant le coût lié à l'instanciation systématique de la classe d'assistance.

Ces résultats sont toutefois à mettre en perspective. Effectivement, dans celui-ci, nous remarquons que l'implémentation de *dmd* est plus performante. Toutefois comme nous l'avons vu dans le Chapitre 3 *dmd* ne met pas en oeuvre de verrouillage sur l'évaluation des contrats ; ce qui dans un contexte où les contrats seraient plus coûteux à l'évaluation provoquerait une forte baisse de performance.

	dmd	nitc	ISE Eiffel	Cofoja	OpenJML	AspectJML	Code Contracts
Sans contrat	4,06s	3,78s	12,6s	12,15s	12,2s	11,70s	4,22s
Avec contrat	5,08s	5,9s	17,91s	30,18s	45,89s	457,91s	9,62s
Écart (%)	+25,1	+52,1	+42,1	+148,4	+276,1	+3813,8	+127,9

Tableau 5.3 Résumé des performances des implémentations étudiées.

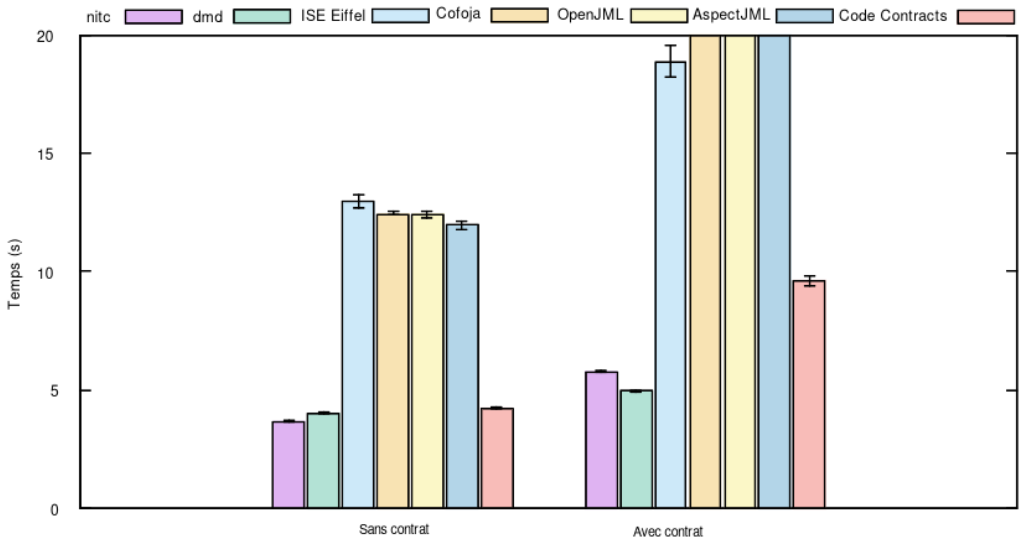

Figure 5.5 Résultats du Micro-benchmark entre les différentes implémentations.

5.4.4 Conclusion et discussion

Bien que notre solution ne fournisse pas le coût d'utilisation le plus faible celle-ci reste tout de même acceptable comparé à une grande partie des implémentations (*Cofoja*, *Code Contracts*, *OpenJML* et *AscpectJML*). Toutefois, nous pensons que dans un contexte réel notre proposition pourrait s'avérer plus avantagée. En effet, dans le cas de notre microbenchmark la majorité des sites d'appels mettent en oeuvre la vérification des invariants et sont donc systématiquement dirigés vers les points d'entrée adéquats, ce qui engendre un coût d'envoi de messages supplémentaire. Dans un contexte où un plus grand nombre d'appels seraient effectués sur l'instance courante, notre proposition avec la détermination statique

de la vérification des invariants profiterait d'un gain non négligeable face à des implémentations mettant en oeuvre cette détermination dynamiquement.

Il convient de noter, que même si nous effectuons la comparaison avec Code Contracts (Microsoft, 2018), il n'est plus conseillé d'utiliser cet outil dans de nouveaux projets, pour la raison que celui-ci n'est plus maintenu.

5.4.5 Menaces à la validité

Comme nous l'avons évoqué, les résultats obtenus sont dépendants des implémentations, mais aussi fortement des langages. En effet, outre le fait de tester la mise en oeuvre des contrats, ce microbenchmark est fortement impacté par la gestion de la mémoire effectuée dans les différents langages. Effectivement, l'ajout d'un nouvel élément dans notre liste chaînée engendre la création d'une nouvelle instance de maillon pour chaque ajout. Les langages utilisant des ramasse-miettes sont ainsi pénalisés par l'appel systématique de celui-ci afin d'essayer de libérer de la mémoire, ce qui est notamment le cas pour Java. Ainsi, l'allocation de plus ou moins de mémoire à l'exécution de la machine virtuelle peut impacter de manière significative les résultats.

CONCLUSION

Dans ce mémoire nous avons abordé les différentes phases de la mise en oeuvre de la programmation par contrat dans un langage orienté objet. Nous avons commencé dans un premier temps par aborder la programmation par contrat, en présentant ces avantages ainsi qu'en la comparant à des approches comme la programmation défensive. Nous avons par la suite étudié la syntaxe, la sémantique ainsi que la mise en oeuvre concrète d'implémentation de programmation par contrat. Bien qu'ayant abordé plusieurs implémentations, les principales étudiées dans ce mémoire ont été Cofoja, ISE Eiffel, LibertyEiffel, AspectJML, OpenJML ainsi que Code Contracts. Nous avons ensuite abordé les différentes spécificités de notre proposition d'implémentation pour le langage Nit. En proposant notamment une nouvelle approche pour prendre en charge les spécificités de la programmation par contrat, en utilisant notamment le concept de point d'entrée (facette), permettant de faire coexister plusieurs versions d'une méthode avec des niveaux de vérification différents (évaluation de l'ensemble des contrats de la méthode, évaluation des pré-conditions et post-conditions, aucune évaluation). Ces points d'entrée permettent d'apporter une grande flexibilité à notre approche. En effet, dans le mode de compilation par défaut que nous proposons, les niveaux de vérification coexistent ensemble, ce qui permet de gérer la liaison des contrats de manière plus fine tout en réduisant le surcôut à l'exécution. Notons que l'un des avantages de notre technique est la détermination statique des points d'entrée de vérification.

Dans la dernière partie de ce travail, nous avons pu réaliser une comparaison de performance entre l'utilisation d'assertions et de contrats, ainsi que comparer notre

proposition aux diverses implémentations étudiées. Nous avons pu en conclure, que comparé à l'utilisation d'assertions notre proposition d'implémentation n'entraîne pas de surcoût significatif (environ 4%). Ce coût étant principalement dû à la mise en oeuvre du verrouillage de l'évaluation. Bien que notre proposition n'offre pas le rapport en termes de coût d'utilisation le plus faible sur notre microbenchmark, elle en reste toutefois acceptable aux vues d'autres implémentations. Nous pensons aussi que dans des conditions réelles notre proposition pourrait s'avérer plus avantagée.

Pour conclure, dans un contexte de travaux futurs, nous pouvons entrevoir des pistes d'amélioration de notre proposition notamment dans l'adaptation de celle-ci a un mode de compilation globale pouvant par exemple insérer le code de verrouillage seulement quand celui-ci est réellement nécessaire. De plus, nous pouvons raisonnablement penser qu'avec une étude approfondie de l'ensemble des bibliothèques du langage Nit, il serait possible d'introduire des contrats sur l'ensemble. Cela permettrait ainsi de grandement améliorer leur qualité globale. En outre, comme nous l'avons précisé dans la section 5.2.1, avec une connaissance plus approfondie du fonctionnement des méthodes constituant le compilateur *nitc* il serait possible d'augmenter significativement le nombre total de contrat dans l'ensemble. Pour finir, nous pouvons aussi voir des pistes de recherche concernant le support de l'analyse statique des contrats avec notamment l'utilisation d'un solveur de contraintes, comme nous pouvons le trouver dans des solutions telles qu'OpenJML (Cok, 2014) ou GNATprove (Guitton *et al.*, 2011).

RÉFÉRENCES

Abercrombie, P. et Karaorman, M. (2002). Jcontractor : Bytecode instrumentation techniques for implementing design by contract in java. *Electronic Notes in Theoretical Computer Science*, *70*(4), 55–79.

AdaCore (2020). Design by contracts. Consulté : 2020-02-21. Récupéré de `https://learn.adacore.com/courses/intro-to-ada/chapters/contracts.html`

Albahari, J. et Albahari, B. (2017). *C 7.0 in a Nutshell : The Definitive Reference* (1st éd.). O'Reilly Media, Inc.

Balzer, S., Eugster, P. T. et Meyer, B. (2006). Can aspects implement contracts ? Dans N. Guelfi et A. Savidis (dir.). *Rapid Integration of Software Engineering Techniques*, 145–157., Berlin, Heidelberg. Springer Berlin Heidelberg.

Bartetzko, D., Fischer, C., Möller, M. et Wehrheim, H. (2001). Jass — java with assertions. *Electronic Notes in Theoretical Computer Science*, *55*(2), 103–117.

Burdy, L., Cheon, Y., Cok, D. R., Ernst, M. D., Kiniry, J. R., Leavens, G. T., Leino, K. R. M. et Poll, E. (2005). An overview of jml tools and applications. *International Journal on Software Tools for Technology Transfer*, *7*(3), 212–232.

Censi, Andrea (2019). Pycontracts. Consulté : 2019-10-02. Récupéré de `https://pypi.org/project/PyContracts/`

Chanoir, R. (2017). Cellulonit, une implémentation du modèle d'acteur en nit.

Cheon, Y., Leavens, G., Sitaraman, M. et Edwards, S. (2005). Model variables : Cleanly supporting abstraction in design by contract. *Softw., Pract. Exper.*, *35*, 583–599. `http://dx.doi.org/10.1002/spe.649`

Cok, D. R. (2014). Openjml : Software verification for java 7 using jml, openjdk, and eclipse. *Electronic Proceedings in Theoretical Computer Science*, *149*(Proc. F-IDE 2014), 79–92. Récupéré de `https://doaj.org/article/e1a4736cc6b34c74b9f11ee95c991daf`

Cppreference (2019). cppreference attribute specifier sequence. Consulté : 2020-03-21. Récupéré de https://en.cppreference.com/w/cpp/language/attributes

D Language Foundation (2020). Langage d. Consulté : 2020-04-21. Récupéré de https://dlang.org/

Ducournau, R. et Privat, J. (2011). Metamodeling semantics of multiple inheritance. *Science of Computer Programming*, *76*(7), 555 – 586. Récupéré de http://www.sciencedirect.com/science/article/pii/S0167642310001905

ECMA International (2006). Standard ecma-367 eiffel 2nd edition : Analysis, design and programming language. Consulté : 2019-09-21. Récupéré de https://www.ecma-international.org/publications/standards/Ecma-367.htm

Eiffel (2019). Ise eiffelstudio. Consulté : 2019-03-21. Récupéré de https://www.eiffel.org/doc/eiffelstudio/Compiler

Eliasson, A. (2002). Implement design by contract for java using dynamic proxies. Consulté : 2019-09-21. Récupéré de https://www.javaworld.com/article/2074026/implement-design-by-contract-for-java-using-dynamic-proxies.html

G. Dos Reis, J. D. Garcia, J. L. A. M. N. M. B. S. (2018). Support forcontract based programming in c++. Récupéré de https://isocpp.org/files/papers/p0542r4.html

Gagnon, E. et Hendren, L. (1998). Sablecc an object-oriented compiler framework. *Proceedings of TOOLS 1998*. http://dx.doi.org/10.1109/TOOLS.1998.711009

Gamma, E., Helm, R., Johnson, R. et Vlissides, J. (1995). *Design patterns elements of reusable object-oriented software*. Addison-Wesley professional computing series. Reading, Mass. ; Don Mills, Ont : Addison-Wesley.

Github (2020). Cglib. Consulté : 2019-11-02. Récupéré de https://github.com/cglib/cglib

Guitton, J., Kanig, J. et Moy, Y. (2011). Why hi-lite ada ?

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, *12*(10), 576–580. Récupéré de https://doi.org/10.1145/363235.363259

Hunt, A. et Thomas, D. (2000). *The Pragmatic Programmer : From Journeyman to Master*. USA : Addison-Wesley Longman Publishing Co., Inc.

Jilani, A. A., Iqbal, M. Z., Khan, M. U. et Usman, M. (2019). Chapter three - advances in applications of object constraint language for software engineering. In A. M. Memon (dir.), *Advances in computers*, volume 112 chapitre 3, 135–184. Elsevier

Karaorman, M. et Abercrombie, P. (2005). Jcontractor : Introducing design-by-contract to java using reflective bytecode instrumentation. *Formal Methods in System Design*, *27*(3), 275–312.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. et Irwin, J. (1997). Aspect-oriented programming. In M. Aksit et S. Matsuoka (dir.), *ECOOP'97 - Object-Oriented Programming*, volume 1241 220–242. Springer, Berlin, Heidelberg.

Laferrière, A. (2012). Interface native pour nit.

Leavens, G. T., Baker, A. L. et Ruby, C. (1999). Jml : A notation for detailed design. In H. Kilov, B. Rumpe, et W. Harvey (dir.), *Behavioral Specifications of Businesses and Systems* chapitre 12, 175–188. Kluwer Academic Publishers.

Leavens, G. T., Baker, A. L. et Ruby, C. (2006). Preliminary design of jml : A behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, *31*(3), 1–38. Récupéré de `https://doi.org/10.1145/1127878.1127884`

Liberty Eiffel (2019). Compilateur liberty eiffel. Consulté : 2019-04-21. Récupéré de `https://www.liberty-eiffel.org/`

Liskov, B. et Guttag, J. (2001). *Program Development in Java - Abstraction, Specification, and Object-Oriented Design.* Addison-Wesley Professional.

Liskov, B. H. et Wing, J. M. (1994). A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, *16*(6), 1811–1841. Récupéré de `https://doi.org/10.1145/197320.197383`

Meyer, B. (1992). Applying'design by contract'. *Computer*, *25*(10), 40–51.

Meyer, B. (1997). *Object-oriented software construction* (2nd éd.). Upper Saddle River, N.J : Prentice-Hall.

Microsoft (2018). Code contracts. Consulté : 2019-12-15. Récupéré de `https://www.microsoft.com/en-us/research/project/code-contracts/`

Microsoft (2018). Writing custom attributes. Consulté : 2020-03-21. Récupéré de `https://docs.microsoft.com/en-us/dotnet/standard/attributes/writing-custom-attributes`

Minh Lê, N. (2016). Contracts for java cofoja. Consulté : 2020-03-21. Récupéré

de https://github.com/nhatminhle/cofoja

Mitchell, R., McKim, J. et Meyer, B. (2001). *Design by Contract, by Example*. USA : Addison Wesley Longman Publishing Co., Inc.

Nienaltowski, P., Meyer, B. et Ostroff, J. S. (2009). Contracts for concurrency. *Form. Asp. Comput.*, *21*(4), 305–318. Récupéré de https://doi.org/10.1007/s00165-007-0063-2

Object Management Group Inc (2012). Omg object constraint language (ocl) 2.3.1. Consulté : 2020-04-21. Récupéré de https://www.omg.org/spec/OCL/2.3.1/PDF

OpenJML (2018). Openjml. Consulté : 2020-06-03. Récupéré de https://www.openjml.org/

Oracle (2020a). Java agent. Consulté : 2020-04-02. Récupéré de https://docs.oracle.com/en/java/javase/14/docs/api/java.instrument/java/lang/instrument/package-summary.html

Oracle (2020b). Java dynamic proxy. Consulté : 2019-11-02. Récupéré de https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html

Privat, J. et Ducournau, R. (2005). Raffinement de classes dans les langages à objets statiquement typés. *Obj. Logiciel Base données Réseaux*, *11*, 17–32.

Rebêlo, H., Leavens, G. T., Bagherzadeh, M., Rajan, H., Lima, R., Zimmerman, D. M., Cornélio, M. et Thüm, T. (2014). Aspectjml : Modular specification and runtime checking for crosscutting contracts. Dans *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, p. 157–168., New York, NY, USA. Association for Computing Machinery. Récupéré de https://doi.org/10.1145/2577080.2577084

Rust (2020). Rust contract. Consulté : 2020-02-04. Récupéré de https://docs.rs/contracts/0.6.0/contracts/

Warmer, J. et Kleppe, A. (2003). *The Object Constraint Language : Getting Your Models Ready for MDA* (2 éd.). USA : Addison-Wesley Longman Publishing Co., Inc.