Rapport de stage : Interprétation abstraite orientée objet, contrats et invariants d'objets

Michaël Monerau

Juillet - Octobre 2009

Sous la direction de **Manuel Fähndrich**

à Microsoft Research, Redmond, WA, USA

Microsoft Research

Table des matières

1 Le cadre du stage **2**
1.1 Merci! . . . 2
1.2 Microsoft Research . . . 2
1.3 Place dans mon cursus . . . 2

2 Contenu scientifique **3**
2.1 CodeContracts & Clousot . . . 3
2.1.1 L'environnement . . . 3
2.1.2 Mon travail dans ce cadre . . . 4
2.1.3 L'implémentation . . . 5
2.2 Les contrats inférés . . . 6
2.2.1 Le contexte . . . 6
2.2.2 Stockage des contrats inférés . . . 6
2.2.3 Utilité des contrats . . . 8
2.2.4 Validité des contrats . . . 9
2.3 Invariants d'objets . . . 10
2.3.1 Intégration d'une analyse d'objets dans Clousot . . . 10
2.3.2 Analyse des champs `readonly` . . . 11

3 Expérience et conclusions **13**

1 Le cadre du stage

1.1 Merci !

Je tiens à remercier Manuel Fähndrich pour m'avoir encadré dans ce stage avec disponibilité et enthousiasme. C'était très motivant et agréable. Je remercie également Francesco Logozzo pour son aide et sa disponibilité sans faille. Plus globalement, je remercie toutes les autres personnes avec qui j'ai pu discuter ou travailler à MSR et les autres stagiaires (hey Jean !). Tous ont contribué à me faire passer un été génial, c'était un vrai plaisir de travailler au sein de cette équipe. Et merci tout particulièrement à Radhia et Patrick Cousot pour les discussions toujours agréables et intéressantes.

1.2 Microsoft Research

Microsoft est le géant du software que tout le monde connaît, basé à Redmond, WA aux États-Unis. La compagnie emploie près de 100 000 personnes au total. Sur le campus de Redmond, il y a plus de 30 000 employés, ce qui représente le cœur de la firme.

La principale mission de Microsoft est d'éditer des solutions logicielles pour permettre d'utiliser l'outil informatique de manière naturelle et efficace.

Dans le cadre de cette mission, rester à la pointe de l'innovation et de la recherche représente un enjeu de taille. C'est pourquoi Microsoft a une filière de recherche très importante. Microsoft Research regroupe près de 1 000 chercheurs dans des domaines aussi variés que les langages de programmation, la vision, le hardware, la cryptographie, des sujets purement théoriques, etc.

Le financement de cette filière Microsoft Research est très confortable et permet un travail serein dans toutes les considérations matérielles.

J'ai travaillé dans l'équipe RiSE (*Research in Software Engineering*), et plus précisément dans l'équipe "*Programming Languages and Analysis*" dirigée par Manuel Fähndrich (http ://research.microsoft.com/en-us/groups/pla).

1.3 Place dans mon cursus

J'ai un L3/M1 de Mathématiques & Informatique, ainsi que l'agrégation de Mathématiques, option Informatique.

En cette année scolaire 2009-2010, j'ai décidé de prendre une année de césure afin de travailler en recherche dans le domaine de l'interprétation abstraite. Ce stage chez Microsoft constitue la première étape de ce projet.

2 Contenu scientifique

2.1 CodeContracts & Clousot

2.1.1 L'environnement

CodeContracts est un projet visant globalement à fournir au programmeur des outils efficaces d'analyse statique et dynamique de code.

La partie analyse statique est basée sur l'interprétation abstraite et s'applique à tout binaire .NET compilé. C'est-à-dire que l'analyse peut être exécutée sur n'importe quelle librairie ou exécutable provenant d'un langage .NET (C#, Managed C++, VB .NET, ...).

En effet, le framework .NET fonctionne de la manière suivante : le code source est écrit dans n'importe quel langage .NET, puis un compilateur transforme ce code en code "MSIL" qui peut être assimilé à du code assembleur pour une machine virtuelle .NET, qui, elle, se charge d'exécuter le code. Un des intérêts de MSIL par rapport à un langage assembleur classique est qu'il conserve des informations sémantiques comme les définitions des classes, des types, etc.

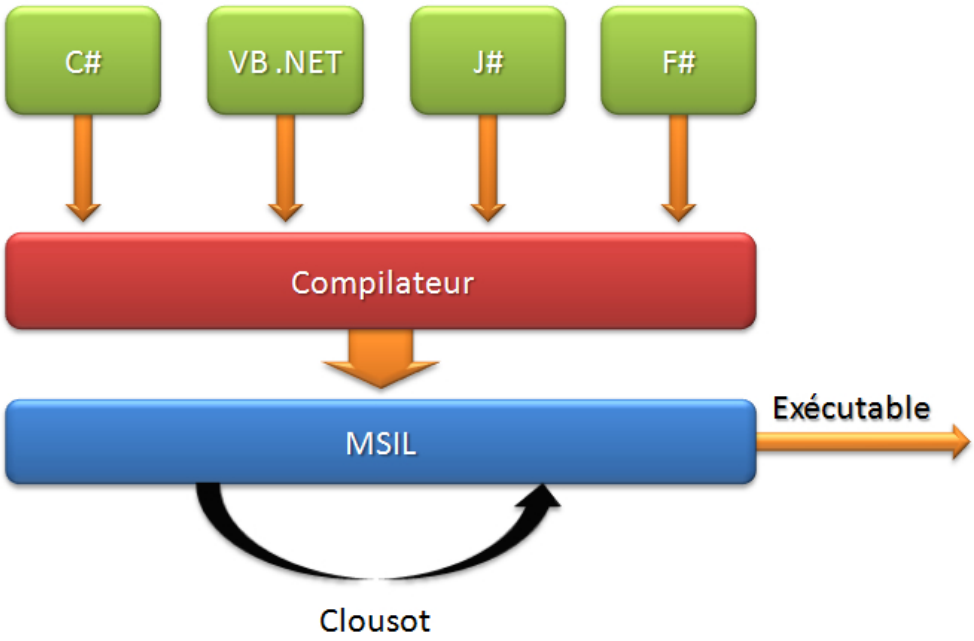

FIGURE 1 – Organisation de .NET Framework et analyse statique de Clousot

C'est pourquoi faire l'analyse statique au niveau MSIL est si intéressant : cela permet de cibler tous les langages .NET, et de plus on accède au code après compilation ce qui évite d'avoir à vérifier le compilateur en lui-même. D'autre part, on dispose déjà d'outils pour extraire la sémantique du code MSIL.

Le déroulement de l'analyse se fait en plusieurs étapes.

Premièrement, au moment de l'écriture du code, une librairie .NET permet au programmeur d'insérer des annotations dans le code (sous forme de vrai code, pas des commentaires, ce qui permet de tester sémantiquement ces annotations) : pré-conditions et post-conditions sur les méthodes, invariants d'objets pour les classes, assertions, assomptions.

Après compilation (avec le compilateur habituel correspondant au langage utilisé) et si CodeContracts est activé, ces annotations sont propagées dans l'assembly MSIL afin de permettre une analyse ultérieure (statique et/ou à l'exécution).

Clousot, l'analyseur statique intervient alors : il exécute plusieurs analyses sur le code (vérification qu'il n'y a pas de déréférencement de références nulles, que les accès aux tableaux ne débordent pas, etc.) en prenant en compte les annotations du programmeur et en inférant automatiquement un maximum d'autres. Au bout du compte, on obtient une liste de warnings (resp. erreurs) selon les résultats, qui pointent vers les endroits du code où un bug est possible (resp. présent).

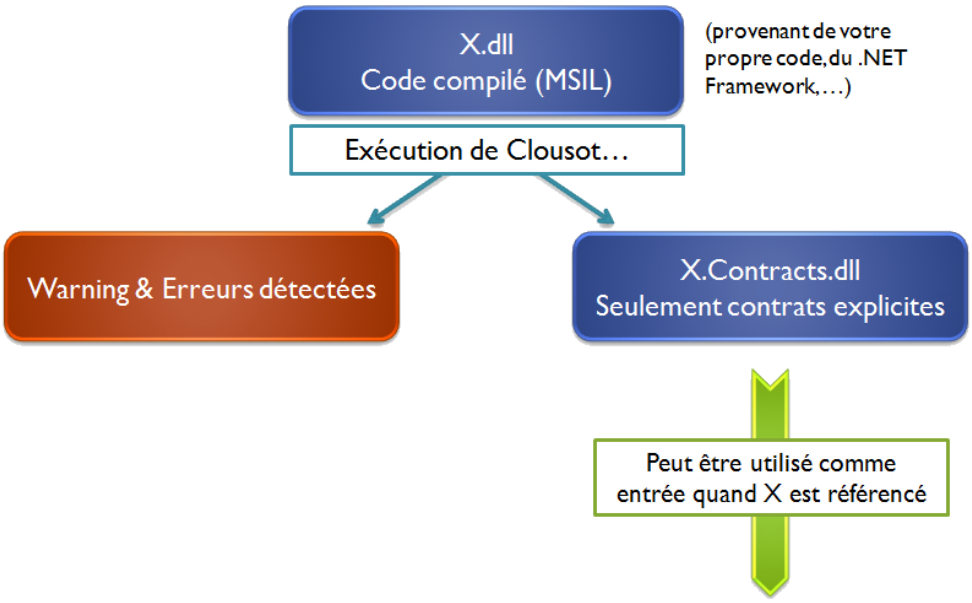

FIGURE 2 – Workflow de Clousot, l'analyseur statique au cœur du projet CodeContracts

Comme tout ceci repose sur la théorie de l'interprétation abstraite, on est garanti que l'analyse est correcte (aux erreurs humaines de programmation dans l'analyseur près) mais il peut éventuellement y avoir des avertissements superflus du fait d'un manque de précision dans l'inférence.

Si l'analyse dynamique est activée, toutes les assertions, pré et post conditions sont vérifiées en direct à l'exécution et terminent le programme si elles ne sont pas respectées.

2.1.2 Mon travail dans ce cadre

J'ai réalisé deux projets durant mon stage. Le premier vise à permettre de réutiliser les résultats d'une analyse, le deuxième vise à inférer des invariants d'objets. Voyons cela plus précisément.

1er projet : si un assembly A référence un autre assembly B, on veut importer les contrats de B dans l'analyse de A. Avant mon stage, on ne pouvait importer que les contrats écrits explicitement par le programmeur dans le code de B parce que tous les contrats inférés ne vivaient qu'en mémoire lors de l'analyse de B.

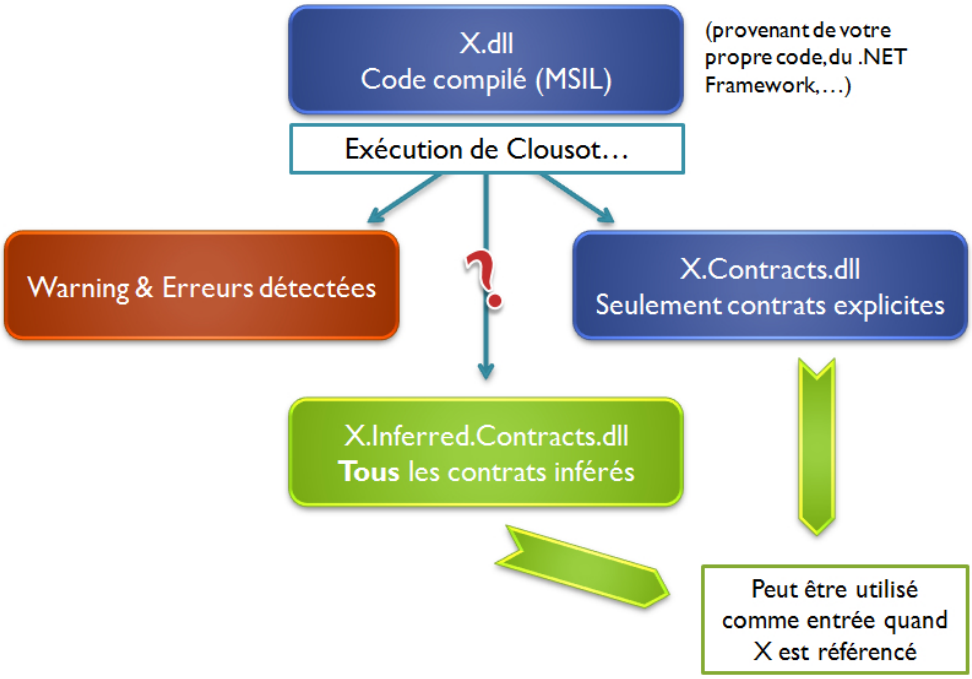

FIGURE 3 – Mon premier projet : Workflow de Clousot, on voudrait réutiliser les contrats inférés

Mon premier projet a donc eu pour but de rendre les contrats *inférés lors de l'analyse de B* visibles par A. Comme on peut s'y attendre, le caractère automatique de l'inférence génère beaucoup plus de contrats que ce qu'on peut écrire à la main. En particulier, une telle fonctionnalité est très utile pour les assemblies système : on peut lancer une analyse sur les DLLs systèmes souvent référencées puis utiliser les résultats pour toutes les futures analyses qui en dépendent.

De plus, la théorie de l'interprétation abstraite permettant de choisir la précision d'analyse de manière très pratique, on peut imaginer lancer des analyses très précises mais longues sur les DLLs souvent référencées, puis stocker les résultats une bonne fois pour toutes.

2ème projet : l'objectif était d'intégrer des analyses d'objets dans Clousot afin d'inférer des invariants d'objets. En effet jusque là, l'analyseur n'inférait que des contrats sur les méthodes de classes.

2.1.3 L'implémentation

L'intégralité du projet CodeContracts est implémentée en C#. Nous nous intéresserons ici seulement à l'interpréteur abstrait.

La lecture du code MSIL se fait à travers la librairie Microsoft CCI (disponible sur internet). La structure du code est ensuite abstraite pour que l'analyseur puisse manipuler des objets de moins bas niveau que le code MSIL directement.

Les analyses de méthodes, indépendantes et utilisant chacune leur propre domaine abstrait, sont exécutées séquentiellement. L'ordre d'analyse des méthodes suit les contraintes imposées par le graphe des appels.

Le projet est scindé en différents modules, indépendants les uns des autres qui limitent les dépendances logiques entre les éléments orthogonaux de l'implémentation.

Donnons à présent une idée de l'échelle. Le code du projet CodeContracts occupe plusieurs centaines de milliers de lignes, et Clousot représente à lui seul entre 50 000 et 100 000 lignes.

2.2 Les contrats inférés

2.2.1 Le contexte

Lorsqu'il écrit son code, le programmeur peut fournir des contrats lui-même (sous la forme de ce qu'on peut voir en figure 5 bien plus bas). On s'attend à ce qu'il fournisse à l'analyseur les contrats non-triviaux qui sont trop complexes pour être déduits automatiquement. De façon complémentaire, l'analyseur doit, pour sa part, déduire un maximum de contrats pour qu'un maximum d'assertions puissent être prouvées (et ainsi éviter des avertissements inutiles).

Pour avoir un ordre d'idée, le nombre de contrats déduits sur une DLL de grande échelle (par exemple System.dll, provenant du .NET Framework) est de quelques dizaines de milliers (cf. chiffres exacts plus bas en figure 6).

Dans ce contexte, il se pose quelques questions naturelles sur ces contrats inférés :

1. Comment les stocker pour les réutiliser dans une autre analyse ? C'est ce qu'on a déjà vu en Figure 3.
2. Sont-ils tous utiles ? Dans le cas contraire, il serait intéressant de supprimer les contrats inutiles afin d'accélérer l'analyse.
3. Sont-ils tous valides ? On ne veut évidemment pas faire de raisonnements faux.

Voyons tour à tour comment répondre à ces inquiétudes.

2.2.2 Stockage des contrats inférés

Le stockage des contrats inférés n'est pas si évident qu'il pourrait y paraître. En effet, on ne veut pas stocker simplement le contrat en lui-même (une représentation sous forme de chaîne de caractères par exemple), mais bien des informations sémantiques associées (il serait inefficace et compliqué de réinterpréter chaque contrat à l'importation).

Principe

La solution que nous avons retenue pour réaliser cette tâche est de repasser par du code C# : l'idée est que, pour chaque assembly analysé, on génère tout le code C# de l'assembly, sauf que le corps de chaque méthode contient seulement les contrats inférés.

Une fois qu'on dispose de ce code, on peut le compiler simplement avec le compilateur habituel, et on obtient gratuitement une DLL contenant tous les contrats inférés. À présent, on peut fournir cette DLL de contrats inférés en entrée lors de l'analyse d'éléments qui dépendent de notre assembly de référence (Figure 4 ci-dessous).

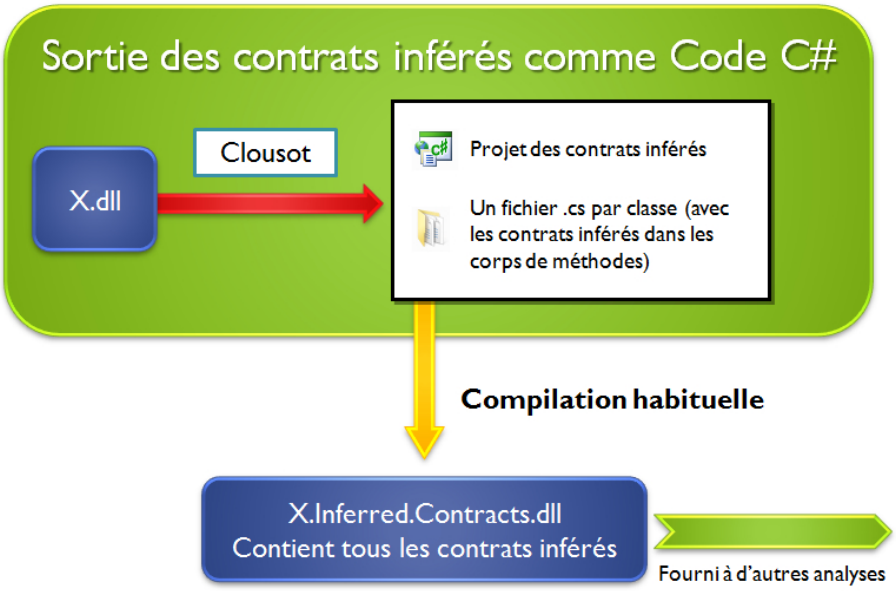

FIGURE 4 – Stocker les contrats inférés sous forme de code C#

Avantages et intérêts

Cette solution, outre de remplir l'objectif fixé, présente beaucoup d'autres avantages comme nous allons le voir.

Citons d'ores et déjà le fait que le code produit est tout à fait lisible. Il permet donc une édition humaine qui peut s'avérer très intéressante. En effet, générer une "*bibliothèque de contrats*" pour les DLLs système est évidemment d'une très grande importance pour l'utilité de CodeContracts. Jusque là, la seule solution pour générer ces contrats étaient de les écrire à la main sous forme de code C#. Il n'est pas besoin de préciser à quel point cette tâche est fastidieuse et longue. Ici, on obtient déjà le squelette avec tous les contrats inférés et on n'a qu'à ajouter simplement ceux qu'on désire, comme le montre l'exemple en figure 5 plus bas.

De plus, on obtient une vérification sémantique gratuite des contrats (car effectuée par le compilateur C#) qui est très utile. En effet, il y a tellement de raisons pour un contrat généré automatiquement qu'il ne se compile pas (il suffit de violer la moindre petite règle C#), qu'il faut faire un vrai effort pour le rendre *compilable*. Par conséquent, un contrat qui compile a de très bonnes chances d'être effectivement voulu. C'est ce que nous avons constaté dans les faits.

Enfin, une fois les contrats stockés, on peut itérer l'analyse en fournissant en entrée les contrats inférés jusqu'à ce qu'on ne déduise rien de plus. Cependant, à cause de petits problèmes indépendants dans Clousot, je n'ai pas pu faire de mesure exacte de l'efficacité d'une telle approche.

```
public class ProtectedProviderSettings : ConfigurationElement
{
  #region Object Invariants
  [ContractInvariantMethod]
  protected void InferredObjectInvariant()
  {
    Contract.Invariant(this._propProviders != null);
  }
  #endregion

  #region Methods and constructors
  public ProtectedProviderSettings ()
  {
      Contract.Ensures(Contract.Result<System.Collections.Specialized.NameValue
      Contract.Ensures(this._PropertyNameCollection != null);
      Contract.Ensures(Contract.Result<System.Collections.Specialized.NameValue
  }
  #endregion

  Properties and indexers

  Fields
}
```

FIGURE 5 – Sortie typique de l'outil *OutputPrettyC#* : le squelette de l'assembly mais avec tous les contrats (et invariants, cf. projet 2) inférés lors de l'analyse. Ici provenant d'une analyse de System.Configuration.dll

Mon travail

Pour réaliser ce projet, il fallait intercepter les contrats générés dans Clousot et les "nettoyer" (cf. plus bas). Indépendamment, j'ai programmé une librairie permettant de produire du code C# valide (et joli!), et l'ai utilisée en particulier pour générer tout le code C# voulu.

Au bout du compte, tout a fonctionné correctement. Les DLLs les plus importantes du framework .NET menaient à un code généré qui compile sans erreur, et qui contient tous les contrats inférés intéressants.

2.2.3 Utilité des contrats

Comme évoqué ci-dessus, cette approche fournit une vérification sémantique gratuitement. Cette vérification représentait la première "mise à l'épreuve" des contrats fournis par Clousot. En effet, jusque là, les contrats étaient utilisés de manière interne et simplement affichés à l'écran. Il n'y avait pas vraiment de moyen de vérifier qu'ils voulaient bien *dire quelque chose*.

Nous avons beaucoup appris de cette nouvelle approche.

La programmation sur la plateforme .NET est résolument orientée objet. Cependant, l'analyseur statique n'est pas intégralement orienté objet. Bien entendu, la structure objet du code est prise en compte lors de l'abstraction du code MSIL, mais nous nous sommes aperçus qu'à certains endroits il était possible de faire des optimisations supplémentaires.

La première chose qui est ressortie des tests de compilation des contrats inférés est qu'il y avait une bonne proportion de contrats qui n'avaient pas d'utilité pour l'analyse (et qui menaient à des erreurs de compilation).

Typiquement, on trouvait dans le corps d'une méthode des contrats sur un champ privé d'une autre classe, inaccessible depuis ce point. Un tel contrat ne peut être d'aucune utilité

localement puisque le champ référé ne pourra jamais être accédé. Il peut donc être ignoré sans perte d'information. Attention cependant : ce contrat pourrait servir dans l'analyse "interne" de l'autre classe, il convient donc ne pas le supprimer complètement partout.

Cela conduit à faire une distinction de principe entre les contrats : certains sont des contrats "d'interface", ie. ils concernent des membres publics et donnent des informations pour une méthode utilisant un objet de ce type ; d'autres sont des contrats "internes" à une classe et permettent simplement l'inférence de plus de contrats publics.

La compilation du code C# produit par notre nouvel outil a également permis de détecter des contrats n'ayant pas de sens simplement à cause de problèmes de types. Essentiellement des problèmes `signed`/`unsigned`, et surtout des opérations arithmétiques entre booléens et entiers (les booléens étant simplement des valeurs 0 et 1 dans le code MSIL à certains endroits) ou entre `enum` et entiers. Ces contrats ne faisaient pas sens ou nécessitaient des ajustements qui ont rendu l'analyse plus robuste.

La compilation a aussi permis de détecter des contrats qui n'étaient pas faux, mais triviaux (l'adresse d'un champ de classe n'est pas nulle... c'est vrai mais c'est aussi une tautologie, autant l'ignorer).

Une fois ces améliorations et ces filtrages implémentés dans Clousot, le nombre de contrats produits a été réduit de manière drastique, menant à une accélération très significative du processus d'analyse, comme en témoignent les chiffres suivants :

	System. Configuration.dll		System.dll		System. Deployment.dll		System. Design.dll	
	Avant	**Après**	**Avant**	**Après**	**Avant**	**Après**	**Avant**	**Après**
Temps d'analyse	2'50"	1'30"	20'40"	18'30"	2'50"	1'10"	14'20"	11'20"
Post-conditions	3397	1469	27149	13866	3287	1596	19458	10372
Pré-conditions	849	642	8349	7883	1323	924	11132	10603
Nb. de métodes	1921		15497		2212		12922	

FIGURE 6 – Résultats : contrats inutiles en moins, meilleurs temps d'exécution

Bien entendu, l'important n'est pas de limiter le nombre de contrats, mais bien de le faire sans perdre de pouvoir déductif. Et en effet, on voyait bien qu'il n'y avait pas moins d'assertions prouvées après avoir enlevé tous ces contrats inutiles qu'avant, ce qui valide a posteriori notre approche.

2.2.4 Validité des contrats

Comme nous l'avons déjà dit, l'analyseur statique se base sur la théorie de l'interprétation abstraite. Cette théorie garantit que les contrats inférés sont *corrects*. On en manque peut-être par manque de précision, mais tout ce qui est inféré est heureusement valide.

Cependant, la pratique est différente : l'implémentation de l'analyseur est humaine et contient donc fatalement des erreurs de programmation. Comment les détecter ?

Souvent, ces erreurs n'apparaissent que dans des cas bien particuliers (parce que les cas courants sont testés sur les petits exemples manipulés par le développeur). Et le paradoxe est que sur les grandes DLLs où on pourrait voir les erreurs, il y a trop de contrats pour qu'on puisse les examiner un par un à la main.

Notre solution de recompiler les contrats s'est montrée utile dans la détection de telles erreurs. Comme dit plus haut, il y a tellement de raisons pour un contrat de ne pas compiler que des erreurs dans l'analyseur ont de grandes chances de mener à un contrat qui ne compile pas.

Évidemment, ce n'est ni un argument solide ni une preuve, mais notre approche nous a permis d'effectivement détecter des bugs dans l'analyseur, que nous avons pu corriger par la suite.

Essentiellement, nous avons détecté que des pré-conditions étaient faussement trop généralisées, que la construction du control flow comportait des bugs menant à de fausses contradictions.

2.3 Invariants d'objets

La deuxième partie de mon stage a été l'implémentation dans Clousot d'une analyse pour inférer des invariants d'objets.

Jusque là, Clousot n'inférait que des contrats concernant les méthodes mais pas d'invariant d'objets. Ma tâche a donc été d'ajouter cette fonctionnalité au framework. J'ai ensuite utilisé ce nouvel environnement pour implémenter une analyse particulière (et très basique).

2.3.1 Intégration d'une analyse d'objets dans Clousot

Je ne m'attarderai pas sur la partie programmation de l'implémentation effective du principe d'analyse d'objets dans le design de Clousot, mais il est intéressant de mentionner un problème que nous ne soupçonnions pas (à ce point) au début.

Problème d'impact mémoire

Afin de réaliser une analyse d'objets, il est indispensable de disposer des résultats des analyses de chaque méthode de la classe concernée.

Si plusieurs analyses sont exécutées simultanément, il faut donc conserver en mémoire plusieurs états abstraits pour chaque méthode jusqu'à pouvoir lancer l'analyse de classe. Les états abstraits (un pour chaque domaine abstrait mis en jeu) peuvent être volumineux en mémoire, et ce même pour des méthodes de taille raisonnable.

Clairement, il n'est pas possible d'exécuter toutes les analyses de classe après toutes les méthodes, puisqu'il faudrait alors garder les résultats de chaque analyse de chaque méthode simultanément en mémoire - c'est trop.

On peut donc penser qu'on va faire les analyses de classe "à la volée" : après l'analyse de chaque méthode, si toutes les autres méthodes de sa classe ont déjà été analysées, on lance les analyses de classe correspondantes (et on peut supprimer de la mémoire toutes les informations sur les méthodes après cela).

Mais les contraintes du graphe d'appels induisent un ordre particulier sur l'ordre d'analyse des méthodes. Et en pratique, si on n'y fait pas attention, les méthodes d'une classe donnée peuvent être analysées à des moments inutilement espacés, obligeant à garder les informations sur leur classe pendant longtemps.

Sur de grosses DLLs, avec quelques milliers de classes, nous avons mesuré qu'on ouvrait l'analyse de plus de 800 classes (et certainement plus si nous n'avions pas rencontré des *Out Of Memory* à ce moment-là) en attendant de voir toutes les autres méthodes de ces classes. L'impact mémoire était de plusieurs Go et n'est clairement pas acceptable.

Solution

Il apparaissait ainsi clair qu'il fallait réorganiser l'ordre d'analyse des méthodes. En effet, beaucoup de méthodes n'ont pas de contrainte particulière entre elles, et il convient donc de rassembler un maximum les analyses de méthodes d'une même classe.

Pour cela, on construit en plus du graphe d'appels entre méthodes $\mathcal{G}_m$ un graphe d'appels entre classes $\mathcal{G}_c$, ie. un graphe où les nœuds sont les classes et où il y a un lien d'une classe c_1 vers une classe c_2 si une méthode de c_1 appelle une méthode de c_2 (qui n'est rien d'autre que $\mathcal{G}_m$ quotienté par la relation $m_1 \sim m_2 \iff classe(m_1) = classe(m_2)$).

Jusque là, seul l'ordre topologique des composantes fortement connexes de $\mathcal{G}_m$, qu'on note $<_m$, était utilisé pour choisir l'ordre d'analyse des méthodes. Mais l'ordre à l'intérieur des composantes fortement connexes de $\mathcal{G}_m$ était quelconque.

Ici, on améliore celà en utilisant maintenant $\mathcal{G}_c$ pour regrouper les méthodes selon leur classe lorsque c'est permis par $\mathcal{G}_m$.

Précisément, on considère les composantes fortement connexes de $\mathcal{G}_c$ dans l'ordre topologique, puis on utilise $<_m$ au sein de chacune d'elles pour l'ordre d'analyse. Typiquement, ces composantes ne comportent qu'un élément et donc on analyse souvent toutes les méthodes d'une classe au même moment. Ainsi, on minimise le temps où l'on a à conserver les informations sur les méthodes, et on voit facilement qu'on ne perd pas les contraintes imposées par $\mathcal{G}_m$.

Résultats

Cette approche n'a aucun impact perceptible sur le temps d'exécution (on demande seulement la construction et un parcours de $\mathcal{G}_c$, avec quelques tris selon l'ordre $<_m$).

Mais elle permet de ne plus souffrir du problème de mémoire : sur les mêmes DLLs que ci-dessus, le nombre maximal de classes ouvertes à un moment donné ne dépasse pas 25 (ie. le cardinal de la plus grosse composante fortement connexe de $\mathcal{G}_c$, et l'impact mémoire est donc tout à fait raisonnable.

2.3.2 Analyse des champs `readonly`

L'analyse d'invariants d'objets que j'ai implémentée se basait sur le fait que, dans les langages .NET qui le supportent (ce qui inclut C#), les champs d'une classe déclarés comme `readonly` ne peuvent être modifiés que dans le constructeur.

Principe

Ainsi, prenons une classe $\mathcal{C}$ fixée et notons $(f_i)_i$ ses champs `readonly` (on suppose évidemment qu'il y en a un, sinon l'analyse ne dit rien). On suppose également qu'on réalise une seule analyse, liée à un domaine abstrait fixé.

À présent, notons $(C_j)_j$ les différents constructeurs de $\mathcal{C}$, et $\widehat{f_{i,j}}$ l'état abstrait du champ f_i à la sortie du constructeur C_j. Alors on peut déduire l'invariant de classe concernant f_i :

$$\widehat{\mathsf{Inv}}_i \quad = \quad \bigsqcup_j \widehat{f_{i,j}}$$

Ainsi, on a un ensemble $(\widehat{\mathsf{Inv}}_i)_i$ d'invariants concernant la classe $\mathcal{C}$.

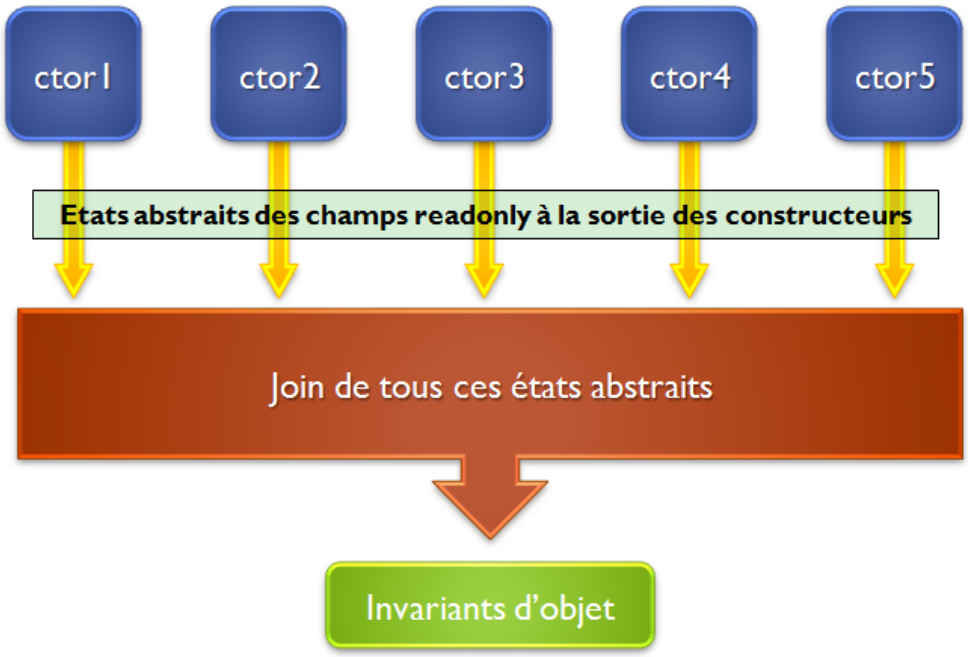

FIGURE 7 – Inférence des invariants à propos des champs `readonly`

Reste ensuite à appliquer ceci indépendamment pour chaque analyse avec chacune leur propre domaine abstrait.

Notons qu'ici on a besoin seulement des informations d'analyse concernant les constructeurs et pas les autres méthodes, ce qui peut permettre d'économiser la mémoire (on lance l'analyse après que tous les constructeurs soient analysés et on ne stocke même pas les informations à propos des autres méthodes si aucune autre analyse n'en a besoin).

Résultats

Sans surprise, les performances de cette analyse sont très bonnes en pratique une fois l'amélioration de l'ordre d'analyse effectuée, puisqu'elle fait simplement quelques *join* sur des états abstraits.

Cependant, il se trouve que cette approche naïve des invariants d'objets ne donne malheureusement pas de très bons résultats pratiques dans le sens où elle permet d'inférer seulement très peu d'invariants. En effet, même sur les grosses DLLs avec plusieurs milliers de classes, on ne dépasse guère les 10 invariants déduits.

Cela peut s'expliquer de plusieurs manières, parmi lesquelles nous semblent les plus probables : les champs `readonly` ne sont pas très utilisés dans les DLLs système ou bien les analyses utilisées ici ne déduisaient rien sur ceux-ci et/ou les idiomes utilisés concernant les champs `readonly` sont trop compliqués pour être captés par l'analyseur en premier lieu.

Une des façons d'améliorer significativement les performances de cette analyse serait d'*inférer* le caractère `readonly` des champs au lieu d'attendre l'information du programmeur.

Nous n'avons pas vraiment creusé la question. L'intéressant était surtout d'implémenter la *possibilité* d'avoir des analyses d'invariants d'objets dans Clousot. L'analyse particulière réalisée ici n'est que le premier pas de l'utilisation de ce nouveau cadre.

3 Expérience et conclusions

Nous avons vu tour à tour les deux projets que j'ai réalisés pendant mon stage à MSR.

Le premier visait à recompiler les contrats inférés lors d'une analyse afin de les réutiliser, ce qui a mené à une vérification sémantique très intéressante et bénéfique. C'était en fait la première fois que les contrats étaient vraiment *regardés*.

Il reste forcément des ajustements à faire sur la génération de code C# à partir du code MSIL, au fil des cas particuliers qui seront rencontrés par la suite. Mais l'outil développé fonctionne correctement même sur les plus importantes DLLs du framework .NET et fournit un code C# qui compile *out-of-the-box*.

Mon second projet visait à introduire dans Clousot des analyses d'objets. Après avoir fait ceci, j'ai implémenté une première analyse d'objets naïve dans ce nouveau cadre.

L'étape suivante consiste maintenant à implémenter sur cette base des analyses d'objets plus sophistiquées qui donneront de meilleurs résultats et auront un meilleur pouvoir déductif.