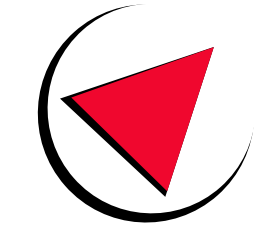

L I F C

Université de Franche-Comté

Mémoire, Master 2 Recherche

Domaines réalistes pour la génération de tests unitaires dirigés par les contrats

par Ivan Enderlin

encadré par

Frédéric Dadeau ⋆ Maître de conférences à l'université de Franche-Comté

soutenu le 30 aout 2011 *devant le jury*

Fabrice Bouquet ⋆ Professeur à l'université de Franche-Comté
Pierre-Cyrille Héam ⋆ Professeur à l'université de Franche-Comté
Alain Giorgetti ⋆ Maître de conférences à l'université de Franche-Comté

Table des matières

Table des figures

Table des exemples

"In God we trust, for the rest we test"

Première partie

Contexte

Chapitre 1

Introduction

Sommaire

À travers les années, le test est devenu le moyen principal pour valider un logiciel. Des techniques de développement récentes, comme les méthodes Agiles, considèrent le test comme étant même prioritaire sur le code. Le paradigme du test basé sur les modèles [5] est un des paradigmes les plus efficaces pour l'automatisation de génération de données de tests et de tests fonctionnels.

1.1 Tests unitaires dirigés par les contrats

Dans l'optique de faciliter la description de modèles, les langages d'annotation, introduits par Meyer [40], ont été conçus à travers la création du paradigme *Design-by-Contract*. Ces langages permettent l'expression de contraintes formelles (comme les pré-conditions, les post-conditions et les invariants) qui annotent directement les entités du programme (comme les attributs de classes, les arguments des méthodes et des fonctions etc.) dans le code source. Plusieurs langages d'annotations existent, tels que le *Java Modeling Language* (JML) [37] pour Java, *ANSI-C Specification Language* (ACSL) [4] pour C, ou encore Spec# [3] pour C#. Le paradigme *Design-by-Contract* considère que le système doit être utilisé de manière contractuelle : pour invoquer une méthode, l'appelant doit satisfaire sa pré-condition ; en

retour, la méthode satisfait sa post-condition.

Les annotations peuvent être vérifiées à l'exécution afin de s'assurer que le système se comporte comme spécifié et ne rompt aucun contrat. L'idée des tests dirigés par les contrats (ou *Contract-Driven Testing* [1]) est de se fier aux contrats pour d'une part produire des tests en calculant les données de tests satisfaisant la pré-condition du contrat, et d'autre part pour obtenir un verdict de test en vérifiant que la post-condition du contrat est vérifiée après l'exécution. D'un certain point de vue, les pré-conditions des méthodes peuvent être utilisées pour générer des données de tests car elles caractérisent l'état et la forme des arguments pour lesquels un appel de méthode est licite. Par exemple, l'outil Jartege [45] génère des données de tests aléatoires en accord avec les domaines des entrées pour une méthode Java donnée, et rejette les valeurs qui ne satisfont pas la pré-condition. D'une autre point de vue, les post-conditions sont employées similairement dans la plupart des approches existantes [6, 11], *i.e.* en vérifiant les post-conditions après chaque exécution de méthodes pour fournir un verdict de test. Les contrats sont donc adaptés pour du test et plus particulièrement pour du test unitaire [36].

1.2 Contributions

Dans ce mémoire, nous présentons un nouveau langage appelé Praspel pour *PHP Realistic Annotation and SPEcification Language* [19]. Praspel est un langage de spécification pour PHP [46] qui illustre le concept des domaines réalistes (notions introduites dans [19]). Praspel introduit le paradigme *Design-by-Contract* dans PHP en spécifiant des domaines réalistes sur les attributs de classes et les arguments de méthodes. Par conséquent, Praspel est adapté à la génération de tests : les contrats sont utilisés pour de la génération de données de tests unitaires et fournissent un oracle de tests pour vérifier les assertions à l'exécution.

Une autre contribution est un framework de test supportant ce langage. Ce générateur et cet exécuteur de tests fonctionnent en trois étapes : (*i*) l'outil génère des données de tests par rapport à un contrat (il peut utiliser plusieurs générateurs de données différents) ; (*ii*) il exécute le programme PHP avec les valeurs fraîchement générées et (*iii*) l'outil vérifie la post-condition du contrat pour définir le verdict du test.

Enfin, dernière contribution, nous avons rapproché la notion de *Grammar-based Testing* [31] du test unitaire en l'introduisant dans les domaines réalistes. Cette contribution bénéficie gratuitement de tous les efforts effectués autour des domaines réalistes, des générateurs de données et du framework de test utilisant Praspel. Cela nous permet de générer et de valider des données par

rapport à une grammaire exprimée dans un nouveau langage : le langage PP.

1.3 Plan du mémoire

Le mémoire est organisé comme suit :

— le chapitre 2 présente l'état de l'art sur les tests unitaires et la génération de données de tests unitaires ;
— le chapitre 3 rappelle les travaux effectués au cours de l'année de Master 1 par les auteurs de ce document en ajoutant certaines précisions ou en modifiant certains mécanismes ;
— le chapitre 4 présente l'introduction du paradigme *Grammar-based Testing* dans les domaines réalistes ;
— le chapitre 5 présente une expérimentation de Praspel sur des projets d'étudiants de Licence 2 ;
— enfin, le chapitre 6 conclut ce mémoire et présente quelques perspectives techniques et scientifiques.

Chapitre 2

État de l'art

Sommaire

Notre étude porte sur la différence entre les grandes familles de tests unitaires en mettant en exergue leurs avantages et inconvénients respectifs dans différents contextes d'utilisations avec différents objectifs.

Nous allons présenter trois grandes familles de tests unitaires, qui seront suivies par la présentation de techniques d'automatisation de génération de données de tests. Ce plan se veut incrémental dans le sens où chaque nouvelle partie approfondit la précédente.

2.1 Familles de tests unitaires

Le test unitaire est un procédé qui consiste à tester une unité d'un programme. L'unité trouve sa définition à l'intersection entre la portion la plus petite d'un programme et la plus pertinente. Généralement, nous considérons des fonctions (ou des méthodes) comme étant des unités raisonnables.

Nous appelons alors un système sous test (ou SUT, pour *System Under Test*) l'ensemble des unités.

2.1.1 Test structurel, ou *White-box*

Le test structurel, ou le test boîte-blanche, est un procédé qui influence le test en se basant sur la structure définie par le système sous test. La structure est définie par les propriétés suivantes.

Graphe de flot de contrôle La plupart des techniques relatives au test structurel font référence à un *graphe de flot de contrôle* [41] (ou CFG, pour *Control Flow Graph*) d'une unité. Un graphe de flot de contrôle G d'une unité U est un graphe dirigé :

$$G(U) = (N, E, n_s, n_e)$$

où N est l'ensemble des nœuds, $E \subseteq (N \times N)$ est l'ensemble des transitions et n_s et n_e sont respectivement les nœuds uniques d'entrée et de sortie du graphe. Chaque nœud $n \in N$ est un bloc d'instructions, où chaque transition $n_i \xrightarrow{e} n_j \in E$ est un possible transfert du contrôle d'un nœud n_i à un nœud n_j (avec $i \neq j$) déterminé par un prédicat e. Nous noterons en particulier qu'un graphe de flot de contrôle est un 1-graphe planaire.

Exemple 2.1. *Graphe de flot de contrôle*
La figure 2.1 présente un code et son graphe de flot de contrôle. Nous remarquons que les branches représentent des conditions et les nœuds représentent des blocs d'instructions.

Vecteur et domaine Un *vecteur d'entrées* V pour une unité U est un vecteur :

$$V(U) = (x_1, x_2, \ldots, x_i, \ldots, x_p)$$

où x_i, avec $i \in [1 \ldots p]$, sont des variables d'entrées pour l'unité U. Le domaine d'une entrée x_i est l'ensemble $D(x_i)$ de toutes les valeurs possibles pour x_i. Le *domaine* d'une unité U est le produit des domaines des éléments du vecteur $V(U)$:

$$D(U) = D(x_1) \times D(x_2) \times \cdots \times D(x_i) \times \cdots \times D(x_p)$$

Nous en déduisons qu'un vecteur d'entrées $V(U)$ est un point dans un espace k-dimensionnel décrit par $D(U)$.

```
if x <= 0 then
    x := -x
else
    x := 1 - x
end if

if x = -1 then
    x := 1
else
    x := x + 1
end if

writeln(x)
```

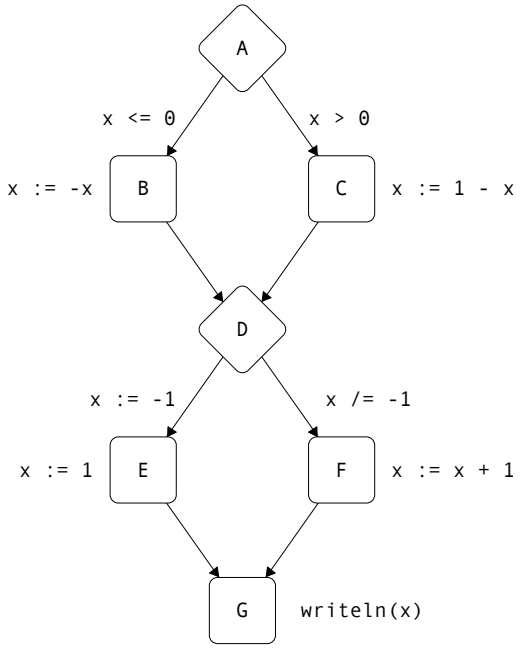

FIGURE 2.1 – Code et graphe de flot de contrôle associé

Chemin Un *chemin* c de longueur p dans un graphe de flot de contrôle est une suite finie de nœuds :

$$c = (n_0, n_1, \ldots, n_i, \ldots, n_p)$$

où $(n_i, n_{i+1}) \in E$, $n_0 = n_s$ et $n_p = n_e$. Ce chemin n'est pas nécessairement élémentaire. Un chemin est dit élémentaire quand :

$$\forall i \,.\, i \in [1 \ldots p] \Rightarrow \exists! j \,.\, j \in [1 \ldots p] \wedge n_i = n_j \wedge i = j$$

Le calcul de l'expression d'un chemin d'un graphe de flot de contrôle utilise l'opérateur d'addition (symbole $+$) pour exprimer une décision, l'opérateur de multiplication (symbole $*$) pour exprimer une itération ou aucun opérateur pour une exprimer une séquence. Les parenthèses (symboles (et)) peuvent être utilisées pour lever des ambiguïtés. Nous obtiendrons alors une expression algébrique avec les propriétés d'associativité, ce qui permet entre autre de simplifier les expressions. Par exemple :

$$c_0 = n_0 n_1 n_2 n_3 \qquad c_1 = n_0(n_1 + n_2)n_3 \qquad c_2 = n_0(n_1 n_2) * n_3$$

représentent respectivement une séquence, une décision et une itération. Nous pouvons faire un parallèle évident avec les expressions régulières.

Le calcul de l'expression de tous les chemins d'un graphe de flot de contrôle est la concaténation de tous ces chemins par l'opérateur d'addition (symbole +) où + est l'équivalent logique de $\vee$. Là encore, des simplifications sont possibles :

$$\begin{array}{ll} & c_0 + c_1 + c_2 \\ = & n_0 n_1 n_2 n_3 + n_0(n_1 + n_2)n_3 + n_0(n_1 n_2) * n_3 \\ = & n_0(n_1 n_2 + (n_1 + n_2) + (n_1 n_2)*)n_3 \end{array}$$

Nous noterons la forme simplifiée :

$$n_i + n_i n_j = n_i(1 + n_j)$$

Un chemin est dit *faisable* s'il existe un vecteur d'entrées qui permet de le *couvrir* (*i.e.* de le traverser), sinon il est dit *infaisable*.

Exemple 2.2. *Quelques exemples de chemins*
La figure 2.1 comporte 4 chemins : c_0 = `ABDEG`, c_1 = `ABDFG`, c_2 = `ACDEG`, c_3 = `ACDFG`. En revanche, `BDFG` n'est pas un chemin car nous devons systématiquement démarrer depuis le nœud d'entrée et terminer sur le nœud de sortie (rappelons-nous que $n_0 = n_s$ et $n_p = n_e$).
Nous pouvons exprimer et simplifier l'ensemble des chemins du graphe de la figure 2.1 assez facilement :

$$\begin{array}{ll} & c_0 + c_1 + c_2 + c_3 \\ = & \mathtt{A(BDE + BDF + CDE + CDF)G} \\ = & \mathtt{A(B + C)D(E + F)G} \end{array}$$

Couverture du graphe de flot de contrôle Lors de l'exécution de l'unité U, nous pouvons observer différentes couvertures de le graphe de flot de contrôle $G(U)$:

- *tous-les-nœuds* : chaque nœud est atteint par au moins un des chemins ;
- *tous-les-arcs* : chaque arc est couvert par au moins un des chemins ;
- *tous-les-chemins-indépendants* : tous les arcs sont couverts dans chaque configuration possible (et non pas au moins une fois comme dans le critère précédent) ;
- *tous-les-PLCS* : toutes les Portions Linéaires du Code suivie d'un Saut (ou LCSAJ, pour *Linear Code Sequence And Jump*), soit une séquence d'instructions entre deux branchements est couverte ;
- *chemins limites* : tous les chemins qui traversent une boucle mais ne l'itèrent pas ;

— *chemins intérieurs* : tous les chemins qui itèrent une boucle une seule fois ;
— *tous-les-i-chemins* : tous les chemins possibles en passant de 0 à i fois dans les boucles ;
— *tous-les-chemins* : si n est le nombre maximum d'itérations possibles pour le critère *tous-les-i-chemins*, alors cela revient à faire appliquer le critère *tous-les-*n*-chemins*.

Chaque niveau de couverture inclut le niveau précédent.

Flot de données Nous annotons le graphe de flot de contrôle afin de fournir des informations supplémentaires pour affiner le flot de données. Nous parlons de *définition* d'un symbole v quand celui-ci est modifié, et nous parlons d'*utilisation* d'un symbole v quand nous y accèdons. Toutefois, nous distinguons deux cas d'utilisations : les *p-utilisations* lorsque le symbole v est utilisé dans un prédicat et les *c-utilisations* pour un calcul.
Ces nouvelles informations permettent d'affiner les critères sur la couverture du flot de données.

Couverture du flot de données Nous appelons un chemin *DR-strict* tout chemin reliant une définition d'un symbole v à l'utilisation de ce même symbole v. Nous définissons alors de nouveaux critères de couverture :
— *toutes-les-définitions* : pour chaque définition, il y a au moins un chemin DR-strict, *i.e.* un chemin reliant l'instruction de la définition d'un symbole à une instruction utilisatrice ;
— *toutes-les-utilisations* : les tests produits couvrent toutes les utilisations ;
— *tous-les-DU-chemins* : ajoute au critère *toutes-les-utilisations* le fait de devoir couvrir tous les chemins possibles entre la définition et la référence, en se limitant aux chemins sans cycles.

La hiérarchie des critères est présentée dans la figure 2.2.

Le test structurel offre une couverture du code complète et facile à obtenir car nous avons connaissance de la structure du système sous test. En revanche, nous n'avons pas de spécification à laquelle nous référer et de ce fait il est difficile d'établir un verdict de conformité. Le test structurel seul, dans sa forme la plus simple, se limite *grosso modo* à de la détection d'erreurs à l'exécution.

Nous allons maintenant étudier une autre forme de test.

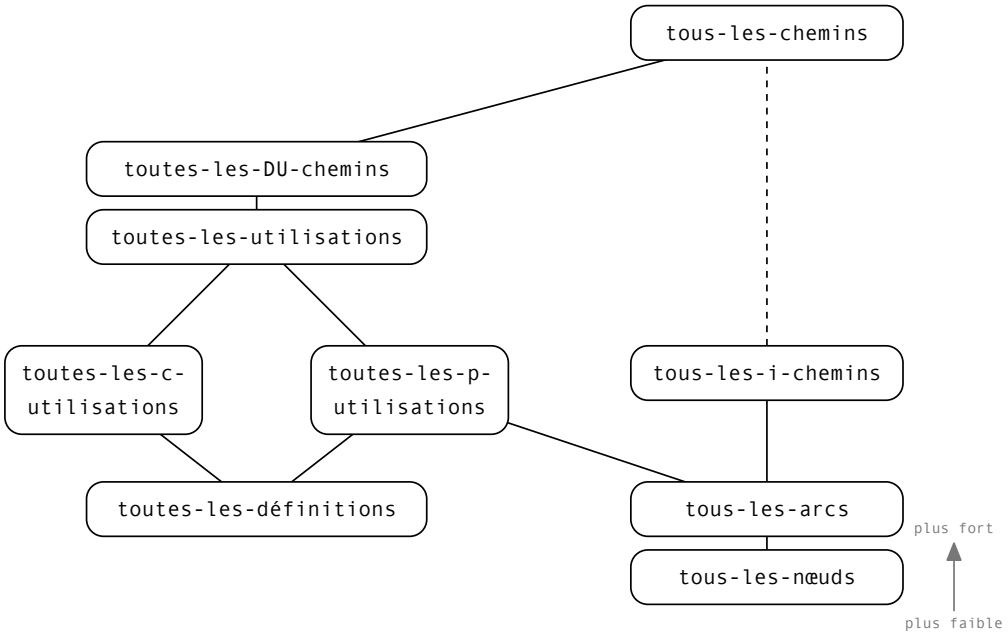

FIGURE 2.2 – Hiérarchie des critères de couverture

2.1.2 Test fonctionnel, ou *Black-box*

Le test fonctionnel, ou le test boîte-noire, est un procédé qui examine le comportement fonctionnel et sa conformité par rapport à une spécification d'un système sous test.

Oracle Un *oracle* aide à résoudre un *problème de décision* de n'importe quelle complexité. Nous utilisons souvent un oracle avec le test fonctionnel car ce sera à lui de décider si le test de la fonctionnalité est un succès ou un échec. En effet, l'oracle sait quel résultat est attendu. Le but recherché sera toujours de trouver un oracle le plus précis possible (en le lisant s'il est écrit, en le complétant, en le calculant etc.). L'oracle peut être plus ou moins fin et être attentif à plus ou moins de paramètres.
Nous parlons d'*oracle gratuit* quand il est explicitement donné, par exemple à travers une spécification.

Modèle et spécification Le *test à partir de modèles* [48, 54] (ou MbT, pour *Model-based Testing*) est une approche qui considère un modèle formel décrivant l'évolution du système à travers une vision abstraite de son fonc-

tionnement. Un modèle est souvent caractérisé par une *spécification*, la plus formelle possible. Cette spécification peut fournir un oracle gratuit si elle est complète. De part la nature abstraite des spécifications, les analyses sur les données de tests peuvent être réutilisées même si le système sous test est modifié. Nous sommes alors capables d'introduire la notation de *conformité* entre la spécification et l'implémentation du système sous test en comparant leurs exécutions [53].
Les modèles permettent d'établir une vision abstraite du système qui peut être parfois animé [8], comparé au système sous test ou fournir un oracle pour le système sous test. Les modèles formels donnent en général de bons résultats dans le cadre d'applications industrielles de grande taille [20].

2.1.3 Test combiné, ou *Grey-box*

Le test combiné, ou le test boîte-grise, est une approche hybride entre le test structurel et le test fonctionnel.

Contrat Un *contrat* établit une base formelle pour une spécification en proposant une pré et une post-condition. La pré-condition établit l'état du système sous test avant exécution, soit les conditions requises pour appeler l'unité. La post-condition établit l'état du système sous test après exécution. Si la pré-condition n'est pas respectée, le contrat ne peut en aucun cas garantir ou statuer sur la valeur de la post-condition. Un contrat est un engagement vis à vis d'un comportement souhaité : l'appelant de l'unité s'engage à l'appeler dans des conditions respectants la pré-condition, et en contre-partie l'unité s'engage à respecter sa post-condition. Un contrat peut également porter des invariants qui doivent être vrais à tout moment, *i.e.* avant, pendant et après l'exécution.
L'approche *Design by Contract*, introduite par Meyer [40], considère les contrats avec les modèles. Les contrats, grâce notamment aux post-conditions, fournissent un oracle gratuit, *i.e.* nous sommes capables de décider immédiatement si le test a été un succès ou un échec.

Langages de contrats Plusieurs langages de contrats existent, comme JML [37] pour Java, ACSL [4] pour C, Spec# [3] pour C# etc. Chacun apporte ses propres spécificités et abordent les problèmes différemments. Certains langages arrivent à exprimer plus de contraintes.
La figure 2.3 propose un exemple d'utilisation de JML. Tout d'abord, JML est placé dans les commentaires pour être servie en annotation. Chaque méthode et attribut sont annotés de clauses qui composent des contrats. Dans chaque

```
public class BankingExample {

    public static final
      int MAX_BALANCE = 1000;
    private /*@ spec_public @*/
      int balance;
    private /*@ spec_public @*/
      boolean isLocked = false;
    //@ public invariant balance >= 0
    //      && balance <= MAX_BALANCE;

    /**
     @ assignable balance;
     @ ensures balance == 0;
     */
    public BankingExample() {
        balance = 0;
    }

    /**
     @ requires 0 < amount
     @       && amount + balance < MAX_BALANCE;
     @ assignable balance;
     @ ensures balance == \old(balance) +
     @                    amount;
     */
    public void credit(int amount) {
        balance += amount;
    }

    /**
     @ requires 0 < amount
     @       && amount <= balance;
     @ assignable balance;
     @ ensures balance == \old(balance) -
     @                    amount;
     */
    public void debit(int amount) {
        balance -= amount;
    }

    /**
     @ ensures isLocked == true;
     */
    public void lockAccount() {
        isLocked = true;
    }

    /**
     @   requires !isLocked;
     @   ensures \result == balance;
     @ also
     @   requires isLocked;
     @   signals_only BankingException;
     */
    public /*@ pure @*/ int getBalance()
            throws BankingException {
        if(!isLocked)
            return balance;
        else
            throw new BankingException();
    }
}
```

FIGURE 2.3 – Exemple d'utilisation de JML

contrat, nous remarquons la présence de mots-clés comme `requires` ou `ensures` qui expriment respectivement une pré et une post-condition, ainsi que le mot-clé `invariant` qui exprime un invariant. Les mots-clés sont suivis par une expression logique avec quelques opérateurs et constructeurs supplémentaires (comme `old(`e`)`).

2.2 Automatisation de la génération des données de tests

Que ce soit avec le test structurel ou avec le test fonctionnel, nous avons suffisamment d'informations pour être capables de générer automatiquement des données de tests. Cela permet d'automatiser le processus de test, de génération de données de tests, l'orientation de la couverture des tests etc. Même les processus les plus triviaux sont capables d'éliminer les erreurs les

branches	probabilité
`if(x == y)`	
`// ...`	$\frac{1}{2^{128}}$
`else`	
`// ...`	$1 - \frac{1}{2^{128}}$

FIGURE 2.4 – Problème de couverture avec de la génération aléatoire pour `x` et `y` deux entiers

plus évidentes et ainsi faire gagner un temps certain quant à l'évaluation de la qualité du système.

2.2.1 Génération aléatoire

Le test par *génération aléatoire* consiste à générer ou exécuter une unité avec des vecteurs générés aléatoirement en considérant les domaines de l'unité.

Ce mécanisme fonctionne bien pour des unités simples. Cependant, les chemins du graphe de flot de contrôle atteignables avec une faible probabilité sont très souvent peu couverts. Une simple égalité entre deux entiers, comme le montre la figure 2.4, donne une probabilité de 1 sur 2^{128} (environ $3.4e^{+38}$) pour une machine en 64bits. Imaginons alors pour des tableaux, des structures ou des objets : les probabilités deviennent très rapidement faibles. Néanmoins, le test par génération aléatoire reste efficace pour éliminer les cas évidents, comme le montre la figure 2.5 où nous sommes rapidement satisfait mais que cette satisfaction se réduit par la suite.

Grâce à la satisfaction rapide qu'il offre, le test aléatoire est souvent utilisé pour faire apparaître la *surface* de notre graphe de flot de contrôle, *i.e.* sa couverture, ses objectifs difficiles etc.

2.2.2 Génération statique

Le test par *génération statique* est basé sur l'analyse de la structure du système sous test sans que l'exécution de cette dernière ne soit nécessaire.

Exécution symbolique Une *exécution symbolique* [33, 34] n'est pas une réelle exécution d'un système sous test, mais plutôt un processus utilisant des valeurs symboliques au lieu des valeurs concrètes. Ce processus peut être utilisé pour obtenir un système de contraintes sur les variables d'entrées qui décrit les conditions nécessaires pour traverser un chemin donné [13, 7, 50].

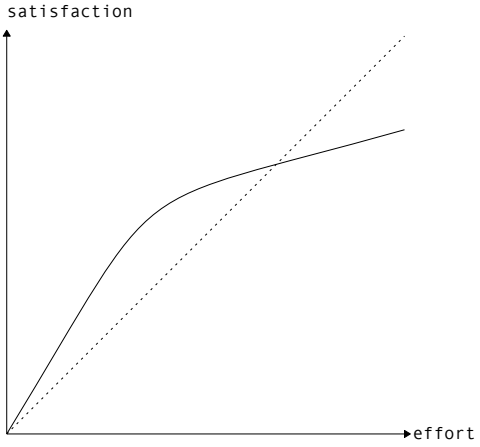

FIGURE 2.5 – Efficacité du test aléatoire

Exemple 2.3. *Parcours descendant symbolique*
Nous prenons l'exemple d'un parcours descendant d'un chemin de l'algorithme de détection du type du triangle et de son graphe de flot de contrôle présentés dans la figure 2.6. Soit le chemin `AEIJKLMNT` que nous voulons exécuter. Les arguments `x`, `y` et `z` de la fonction sont associés à des symboles, respectivement α, β et γ. Aux nœuds `A` et `E`, les branches négatives, *i.e.* les branches qui représentent la négation du prédicat, doivent être choisies. Ainsi, les deux premières contraintes de notre système de contraintes pour ce chemin sont :

$$\alpha \leq \beta \tag{1}$$

$$\alpha \leq \gamma \tag{2}$$

Pour que le chemin soit exécuté, il est également nécessaire que la branche positive du nœud `i` soit empruntée ; ce qui nous fournit une nouvelle contrainte à ajouter au système :

$$\beta > \gamma \tag{3}$$

Dans les nœuds `J`, `K` et `L`, nous trouvons les assignations suivantes :

$$\texttt{t} = \beta$$
$$\texttt{y} = \gamma$$
$$\texttt{z} = \texttt{t}$$

Une quatrième et dernière contrainte est ajoutée par le nœud `M`. Maintenant, avec `x` $= \alpha$, `y` $= \gamma$ et `z` `=` `t` $= \beta$, cette contrainte devient :

$$\alpha + \gamma \leq \beta \tag{4}$$

```
     int tri_type ( int x, int y, int z ) {

         int type

  A      if(x > y) {
B-D          int t = x; x = y; y = t;
         }

  E      if(x > z ) {
F-H          int t = x; x = z; z = t;
         }

  I      if(y > z) {
J-L          int t = y; y = z; z = t;
         }

  M      if(x + y <= z)
  N          type = NOT_A_TRIANGLE;
         else {
  O          type = SCALENE;

  P          if(x == y && y == z)
  Q              type = EQUILATERAL;
  R          elseif(x == y || y == z)
  S              type = ISOSCELES;
         }

  T      return type;
     }
```

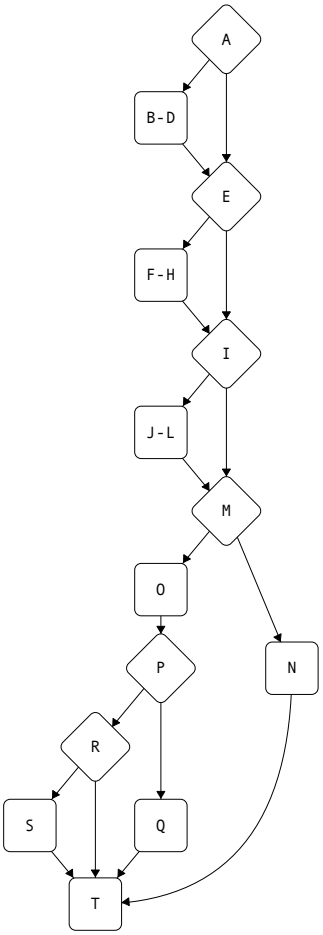

FIGURE 2.6 – Code et graphe de flot de contrôle de l'algorithme du triangle

Exemple 2.4. *Parcours montant symbolique*
Toujours avec le même exemple de la figure 2.6 et le chemin `AEIJKLMN` mais avec un parcours montant, *i.e.* depuis le nœud final vers le nœud de départ. Le système de contraintes résultant est strictement le même qu'avec un parcours descendant mais ne nécessite pas de stocker les expressions symboliques des variables (ici `x`, `y` et `z`). Cependant, un parcours descendant permet de détecter plus facilement les chemins dits infaisables si les contraintes générées sont inconsistentes. Si nous considérons le chemin `ABCDEFGHIM•T` (où `M•T` représente n'importe quel sous-chemin possible entre les nœuds `M` et `T`) qui nécessite que les branches positives des nœuds `A` et `E` soient choisies et que la

branche négative du nœud `I` soit choisie, le système de contrainte est alors :

$$\alpha > \beta \tag{5}$$

$$\beta > \gamma \tag{6}$$

$$\alpha \leq \beta \tag{7}$$

Il apparaît clairement que les équations 5 et 7 sont contradictoires, ce qui indique que le chemin est infaisable. Un parcours montant nécessiterait de parcourir plus de nœuds pour détecter cette infaisabilité.

Les solutions du système de contraintes sont les données des entrées qui permettent d'exécuter le chemin. Les problèmes de satisfaction de contraintes sont en général des problèmes NP-complet [24]. Notons que si les contraintes sont linéaires, un programme linéaire peut être utilisé pour trouver des solutions [13]. Quelques procédés génériques existent pour des cas moins triviaux comme nous allons le présenter.

Réduction de domaine La *réduction de domaine* est une technique de génération de données qui était à l'origine employée par DeMillo et Offutt pour le *test sous contraintes* [15]. Le test sous contraintes élabore un système de contraintes, dont sa satisfaction permet d'atteindre un objectif de test préalablement défini. L'intention initiale du test sous contraintes était de générer des données pour du test par mutation. Le test par mutation [32] consiste à modifier le système sous test, et tous les cas de tests qui ne détectent pas ou ne rejettent pas le système mutant seront considérés comme défectueux, sinon le mutant sera tué. Les opérations de mutations appliquées sont empiriquement connues pour être la cause d'erreurs (remplacer une conjonction par une disjonction, une comparaison stricte par une comparaison ou une égalité, initialiser des variables à 1 au lieu de 0 etc.).
Nous distinguons deux contraintes particulières : la *contrainte d'atteignabilité* qui décrit les conditions sous lesquelles un état particulier peut être atteint, et la *contrainte de nécessité* qui décrit les conditions sous lesquelles un mutant peut être tué.
La réduction de domaine est utilisée pour trouver une solution au système de contraintes construit par une exécution symbolique. Ce procédé commence par définir un domaine $D(U)$, *i.e.* définir le domaine de chaque variable. Si le langage analysé est typé statiquement, nous pouvons déduire sommairement le domaine des variables. Si une spécification existe, nous pouvons déduire plus finement les domaines. Le domaine de chaque variable est alors réduit en utilisant les informations fournies par le système de contraintes, en commençant par celles impliquant un opérateur relationnel, une variable et une

constante, et les contraintes impliquant une opération relationnelle et deux variables. Les contraintes restantes sont alors simplifiées par substitutions de valeurs. Quand plus aucune simplification n'est envisageable, le plus petit domaine est choisi et une valeur est tirée aléatoirement dans ce domaine pour être affectée à la variable correspondante. La valeur de la variable est alors réutilisée pour une nouvelle substitution parmi les contraintes restantes dans l'optique de réduire les domaines des variables restantes. Si toutes les variables peuvent être assignées d'une valeur, le système de contraintes sera satisfait, sinon le processus recommence dans l'espoir que les tirages aléatoires donneront de meilleurs résultats.

Réduction dynamique de domaine Avec le test sous contraintes, le système de contraintes doit être déduit avant d'être analysé. Comme ce système est déduit avec une exécution symbolique, il souffre de problèmes impliquant des boucles, des appels ou encore des accès au système de fichiers. C'est pourquoi, Offutt et al. ont introduit la *réduction dynamique de domaine* [43] pour essayer de combler certains de ces problèmes.
Bien qu'appelée *dynamique*, cette technique reste basée sur une exécution symbolique. Le début du processus est identique, à savoir que nous définissons des domaines pour chacune des variables. Cependant, en contraste avec de la réduction de domaine standard, ces domaines sont réduits « dynamiquement » durant l'exécution symbolique en utilisant les contraintes composées des prédicats des branches. Si le prédicat d'une branche implique une comparaison de variables, les domaines des variables concernées (*i.e.* qui peuvent changer la décision) sont scindés à un point arbitraire plutôt que d'effectuer un tirage aléatoire. Par exemple, soit `x` et `y` telles que $D(\texttt{x}) = D(\texttt{y}) = [-42 \dots 42]$ et un prédicat `x < y` qui doit être vrai pour exécuter la branche sous-jacente, alors les domaines sont scindés en $D(\texttt{x}) = [-42 \dots 0]$ et $D(\texttt{y}) = [1 \dots 42]$. Un processus de retour en arrière (ou *back-tracking*) peut être utilisé pour corriger le point de scission si celui s'avère mauvais pour la suite de l'exécution.
Malgré tout, la réduction dynamique de domaines souffre encore de certains problèmes, comme les boucles ou l'accès au système de fichiers. Notons que nous pouvons déplier les boucles manuellement, mais ce n'est pas toujours possible si nous ne pouvons pas connaître les bornes. Enfin, il n'a jamais été clarifié comment la réduction doit être appliquée sur des domaines non-ordinaux, telles que des énumérations.

code	valeur de `x`	contraintes de `x`
`unsigned int x, a;`		
`int array[5];`		
`scanf("%d", &x);`	2	$0 \leq \texttt{x} \leq \infty$
`if(x > 4) fatal("Index`	2	$0 \leq \texttt{x} \leq \infty$
`  out of bounds");`		
`x++;`	2	$0 \leq \texttt{x} \leq 4$
`a = array[x];`	3	$1 \leq \texttt{x} \leq 5$
		erreur

FIGURE 2.7 – Détection d'une erreur sur les bornes d'un tableau avec le concolic

2.2.3 Génération dynamique

Le test par *génération dynamique* est basé sur l'analyse de l'exécution réelle du système sous test. Toutefois, il n'est pas exclu d'utiliser de l'exécution symbolique de manière complémentaire.

Concolic La génération statique a des limites évidentes comme les boucles, les fonctions de hashages, des données structurelles dynamiques (comme des pointeurs) etc., qui ne peuvent être résolues symboliquement. Toutefois, une exécution concrète pourrait aider à résoudre la majorité de ces problèmes. C'est pourquoi, Larson et Austin ont proposé une combinaison astucieuse d'une exécution concrète *et* symbolique [35] (*concolic* signifie *conc(rete + symb)olic*). Leur approche consiste à exécuter le système sous test à partir de données concrètes fournies par l'utilisateur. Ils sont capables de détecter des erreurs qui n'apparaissent pas à l'exécution par simple vérification des contraintes.

Exemple 2.5. *Le concolic pour détecter des erreurs sur les tableaux*
Comme le montre la figure 2.7, les bornes de la variable `x` sont déduites au début grâce à son domaine : entier non-signé. Puis, si le prédicat `x > 4` est vrai, alors cela conduira à une erreur. Dans ce cas, nous savons qu'il faut affiner la borne gauche du domaine. Enfin, nous incrémentons `x` et c'est là qu'une erreur apparaît. En effet, le domaine de `x` a évolué de $[0 \ldots \infty]$ à $[1 \ldots 5]$ et cette dernière contrainte ne satisfait pas celle de la taille du tableau qui doit appartenir au domaine $[0 \ldots 4]$.

Sur cet exemple simple, une exécution symbolique pourrait rivaliser. Toutefois, Larson et Austin proposent d'autres exemples avec des copies de chaînes

```
   int f ( int x, int y ) {

A      if(x == hash(y))
B         error();

C      return 0;
   }
```

variables	1ère exécution	2nd exécution
x	7	153
y	42	42
hash(y)	153	153
chemin	ac	abc

FIGURE 2.8 – Diriger les données de tests avec une exécution concrète et symbolique

de caractères (donc des pointeurs, des tailles de chaînes etc.). Le même raisonnement est appliqué en faisant évoluer les contraintes et en évaluant leurs satisfactions, ce qui est possible dans ce cas car nous connaissons les valeurs concrètes des pointeurs.

Concolic et couverture Plus tard, Godefroid et al. se sont basés sur la combinaison d'une exécution concrète et symbolique pour générer des données de tests de manière incrémentale. Avec leur approche, durant une exécution concrète est générée une conjonction de contraintes. Ces contraintes sont modifiées et résolues pour générer de nouvelles données de tests afin d'améliorer la couverture. Plus précisément, les contraintes sont modifiées en prenant leur négation pour un parcours en profondeur des chemins. Si la résolution des contraintes modifiées n'est pas possible, une substitution des symboles par un tirage aléatoire est appliquée.

Exemple 2.6. *Analyse d'une fonction de hashage*
À la figure 2.8, nous pouvons voir comment résoudre le problème de l'analyse d'une fonction de hashage. En effet, l'objectif est d'avoir la couverture la plus complète possible. Malheureusement, il est impossible d'analyser statiquement une fonction de hashage à cause de son caractère injectif. C'est pourquoi nous commençons par tirer aléatoirement des valeurs pour `x` et `y`, ici 7 et 42, puis nous exécutons la fonction `f`. Nous voyons que `hash(y)`, soit `hash(42)` donne 153. Notre première exécution avec des valeurs aléatoires nous permet de couvrir le chemin `ac` car le prédicat `x == hash(y)` est faux. Si nous voulons obtenir une couverture de tous les chemins, il est nécessaire que ce prédicat soit vrai. Pour que `x == hash(y)`, comme nous connaissons le précédent résultat de `hash(y)`, nous allons définir `x = hash(y)`, soit `x = 153` et réutiliser la même valeur pour `y`. Ainsi, à la seconde exécution, le prédicat sera `x == hash(y)`, soit `153 == hash(42)`, soit `153 == 153`. Le prédicat sera vrai et nous couvrirons le chemin `abc`.

Ce concept a été implémenté dans l'outil DART [26]. Ce dernier a été précédé

par PathCrawler [55], le premier du genre, et par KLEE [10] (anciennement EXE [9]), tout aussi remarquable.
Une difficulté rencontrée avec le précédent concept est d'obtenir des méthodes capables d'extraire et de résoudre les contraintes générées lors de l'exécution du système sous test. Ce problème devient particulièrement complexe quand l'exécution fait intervenir des données structurelles dynamiques utilisant des opérations sur les pointeurs. Par exemple, les pointeurs peuvent avoir des alias. Parce que l'analyse des alias peut faire intervenir des opérations arithmétiques sur les pointeurs, utiliser des valeurs symboliques pour détecter de tels pointeurs peut aboutir sur des contraintes dont la satisfaction est indécidable. C'est pourquoi générer des données de tests par résolution de telles contraintes est infaisable. Alors, Sen et al. ont proposé une méthode pour représenter et résoudre *des contraintes approchées de pointeurs* [51] ; concept proposé à travers l'outil CUTE. Il semblerait que ce soit l'outil le plus avancé dans la résolution de contraintes pour une combinaison d'exécution concrète et symbolique. CUTE a été le premier a introduire le terme *concolic*.

Fuzzing Le *fuzzing* est une technique de test qui consiste à modifier des données de tests valides et à les réinjecter dans le système sous test. Nous parlons de mutation de données de tests et cette opération est aléatoire. Les nouvelles données mutées peuvent être valides ou invalides par rapport à une spécification. Si elles sont valides et que des erreurs sont détectées, alors le test est une réussite car il nous a permi de détecter une véritable erreur. Si elles sont invalides, nous sommes sur la défensive et nous attendons du système sous test qu'il sache réagir en conséquence, *i.e.* ne pas produire d'erreurs ou de comportements involontaires. Cette technique peut également être utilisée pour tester la conformité de la spécification avec l'implémentation.
Dans le cas d'un test structurel, nous pouvons améliorer l'opération de mutation. C'est ce que propose Godefroid et al. en mélangeant du concolic (en accord avec ses précédents travaux [26]) et du *fuzz testing*, pour le transposer dans l'outil SAGE [27], qui peut être vu comme une surcouche à DART. L'idée est de muter les contraintes du système de contraintes issu du concolic, et de s'en servir pour générer de nouvelles données (comme c'était fait avant). Ainsi, nous pouvons considérer que la mutation sera plus efficace et saura détecter plus d'erreurs.

2.3 *Grammar-based Testing*

La génération de données comme nous l'avons vue précédemment se cantonne à résoudre un système de contraintes. Nous avons pressenti que si une spécification existe, elle nous guiderait dans la génération des données. En effet, nous pouvons nous baser sur la spécification pour générer des données que nous pourrons affiner par la suite. Notons que selon le langage utilisé pour exprimer la contrainte, cela peut également revenir à résoudre un système de contraintes. Toutefois, nous aimerions exprimer des contraintes de manière plus synthétiques et plus précises pour représenter des arbres, des expressions arithmétiques, des programmes etc. Le paradigme du *test à partir d'une grammaire* (ou GbT, pour *Grammar-based Testing*) peut parfaitement s'intégrer dans une spécification pour caractériser des données non-numériques à travers un formalisme qui, ici, est une grammaire. Les données que nous souhaitons générer sont des données souvent structurelles, difficiles à exprimer de manière simple.

À partir d'une grammaire, une analyse lexicale et syntaxique permet d'extraire en général un arbre de syntaxe abstrait (ou AST, pour *Abstract Syntax Tree*). Soit cet AST est construit par analyse d'une donnée (ce qui est le principe des compilateurs), soit nous le générons. Ce qui est intéressant est ce qu'il advient de cet arbre. Plusieurs travaux ont été menés. Nous relèverons des outils intéressants, comme GenRGenS par Ponty et al. [47] qui permet de générer des séquences génomiques. Nous pouvons aussi relever yagg par Coppit et Lian [14] qui permet de générer des compilateurs. Ou encore, nous pouvons utiliser les AST pour appliquer de la mutation de données grâce à la représentation structurelle des données qu'offrent les grammaires [44].

Dans le cas de la génération aléatoire de données, nous pouvons nous intéresser à l'uniformité des données générées, *i.e.* que toutes les données ont autant de chances d'être produites. Pour arriver à satisfaire ce critère d'uniformité, plusieurs algorithmes peuvent être utilisés comme la méthode récursive [22], les générateurs de Bolzmann [16] et les techniques utilisant des parcours aléatoires dans des chaînes de Markov [49]. GenRGenS se base sur la méthode récursive, tout comme Seed par Héam et Nicaud [31]. Ce dernier outil est intéressant car il permet de générer des données pour une taille fixée.

Deuxième partie

Réalisations

Chapitre 3

Domaines réalistes et Praspel

Sommaire

Dans ce chapitre, nous traitons le travail qui a été effectué en tant que projet durant l'année de Master 1 (amélioré et clarifié lors de la rédaction de ce document) et qui a fait l'objet d'un rapport de recherche [18], pour être par la suite amélioré durant l'année de Master 2, et qui a fait l'objet d'une publication à la conférence ICTSS'11 [19]. Il concerne les domaines réalistes, présentés dans la section 3.1, le langage Praspel, présenté dans

la section 3.2, les générateurs de données, présentés dans la section 3.3, un aperçu du fonctionnement interne, dans la section 3.4 et enfin l'outil Praspel, présenté dans la section 3.5.

3.1 Domaines réalistes

Nous proposons les domaines réalistes pour exprimer des contraintes sur lesquelles nous pouvons raisonner facilement, automatiquement et rapidement. En effet, si nous considérons des contrats, introduits par Meyer [40], en utilisant des contraintes exprimées avec la logique du premier ordre, lorsqu'une fonction porte une pré-condition, il n'est pas toujours aisé de la satisfaire tout particulièrement lorsque l'on doit générer des données automatiquement. Nous pourrons toujours argumenter qu'il existe des cas simples que nous savons résoudre facilement, mais définir ce qu'est un cas simple n'est pas trivial.

Les domaines réalistes sont prévus pour être utilisés dans le cadre de la génération de données de tests. Ils spécifient un ensemble de valeurs qui peuvent être affectées à des données d'un programme. Nous avons décidé de réaliser une implémentation des domaines réalistes dans PHP [46] car ils s'y intègrent parfaitement. En effet, ce dernier n'a pas de type et nous avons toutes les libertés. PHP est un langage dynamiquement typé, *i.e.* une variable n'a pas de type déclaré et peut appartenir à plusieurs types en même temps. Toutefois, les domaines réalistes peuvent très bien s'adapter aux langages fortement typés.

Nous avons également décidé de développer les domaines réalistes dans Hoa [17], un ensemble de bibliothèques puissantes, rapides, hautement modulaires, sûres, sécurisées, innovantes et respectueuses des standards, sous *New BSD License*. Hoa est également un pont entre le monde de la recherche et le monde de l'entreprise. En développant dans Hoa, nous nous assurons une certaine pérennité pour le projet, ainsi qu'une documentation et une communauté.

Nous introduisons les caractéristiques fonctionnelles et structurelles des domaines réalistes, ainsi que leur implémentation dans PHP.

3.1.1 Caractéristiques fonctionnelles

Les domaines réalistes sont des structures qui proposent des propriétés nécéssaires à la validation et à la génération de données. Les domaines réalis-

tes peuvent représenter toutes sortes de données ; ils sont prévus pour spécifier des domaines de données pertinents pour un contexte particulier. Les domaines réalistes sont plus subtiles que les types usuels (qui sont les entiers, les chaînes de caractères, les tableaux etc.) et proposent un raffinement de ces derniers. Par exemple, si un domaine réaliste spécifie une adresse email, nous pourrons alors valider et générer des chaînes de caractères représentant des adresses emails syntaxiquement correctes.

Les domaines réalistes sont définis par deux caractéristiques fonctionnelles.

Prédicabilité La première caractéristique d'un domaine réaliste est de porter un *prédicat*, qui permet de vérifier si une valeur appartient à l'ensemble des valeurs possibles décrit par le domaine réaliste.

Générabilité La seconde caractéristique d'un domaine réaliste est de *générer* une valeur qui appartient au domaine réaliste. Pour cela, les domaines réalistes se basent sur des *générateurs de données* qui peuvent être de plusieurs natures : aléatoire, énumérateur, incrémental, par recherche etc. Un générateur de données se limite à la génération d'entiers et de nombres flottants. C'est alors au domaine réaliste d'utiliser le résultat de cette génération pour générer ses propres valeurs le définissant.

Dans PHP, qui propose entre autre le paradigme objet, nous avons implémenté les domaines réalistes comme des classes proposant au moins deux méthodes, correspondant aux deux caractéristiques fonctionnelles des domaines réalistes. La première méthode est appelée `predicate($q)` et prend une valeur appelée `$q` : elle retourne un booléen qui indique l'appartenance de cette valeur au domaine réaliste. La seconde méthode est appelée `sample()` et génère des valeurs qui appartiennent au domaine réaliste en fonction du générateur de données attribué.

Exemple 3.1. *Squelette naïf pour une adresse email*
Nous présentons un squelette simple d'un domaine réaliste pour une adresse email dans la figure 3.1. En réalité, un tel domaine réaliste est plus compliqué mais nous présentons ici une version simplifiée. Nous voyons comment utiliser les deux caractéristiques à travers les méthodes `predicate` et `sample`. Nous voyons également que nous utilisons une générateur de données à travers l'argument `$sampler` de la méthode `sample`. Le fonctionnement des générateurs de données est approfondi à la section 3.3.

```
class Email extends ... {

    // ...

    public function predicate ( $q ) {

        // Voir la RFC5322, section 3.4 Address Specification
        $specRegex = '...';

        return    parent::predicate($q)
               && 0 !== preg_match($specRegex, $q);
    }

    public function sample ( Sampler $sampler ) {

        // Caractères autorisés.
        $chars   = 'ABCDEF...';
        $cLength = strlen($chars) - 1;
        // TLD acceptés.
        $tld     = array('net', 'org', 'edu', 'com');
        $data    = null;
        $parts   = $sampler->getInteger(2, 4);

        for($i = 0; $i < $parts; ++$i) {

            if(0 < $i)
                $data .= $i == $parts - 1 ? '@' : '.';

            $length = $sampler->getInteger(1, 10);

            for($j = 0; $j < $length; ++$j)
                $data .= $chars[$sampler->getInteger(0, $cLength)];
        }

        return $data . '.' .
               $tld[$sampler->getInteger(0, count($tld) - 1)];
    }
}
```

FIGURE 3.1 – Ébauche naïve du domaine réaliste `email`

Nous présentons maintenant les caractères structurels des domaines réalistes.

3.1.2 Caractéristiques structurelles

L'implémentation des caractéristiques structurelles profite du paradigme objet que propose PHP mais il est tout à fait envisageable d'en faire une différente si le langage n'adopte pas ce paradigme tant que les fonctionnalités proposées ne sont pas réduites, *i.e.* si les caractéristiques structurelles suivantes sont respectées.

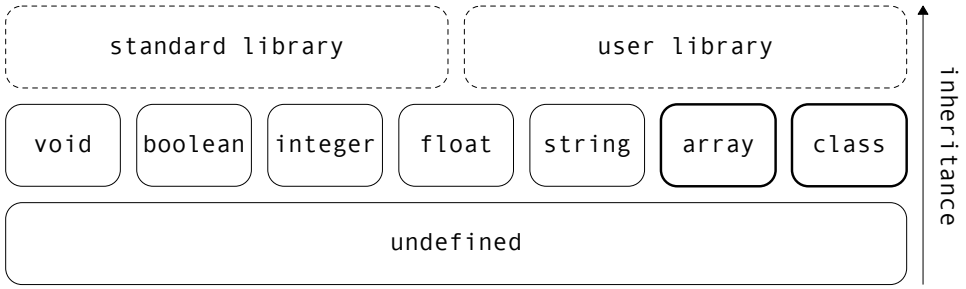

FIGURE 3.2 – Univers des domaines réalistes

Héritage hiérarchique Les domaines réalistes en PHP peuvent hériter entre eux. Un enfant hérite alors des deux caractéristiques de ses parents et est capable de les redéfinir. Conséquence immédiate, nous appelons l'ensemble des domaines réalistes un *univers* (résumé dans la figure 3.2) et est constitué de trois couches :

- la première couche, appelée *seed*, est un singleton : elle ne contient que le domaine réaliste `undefined` qui a un prédicat toujours vrai et un générateur qui ne retourne que des entiers ;
- la deuxième couche, appelée *sprout*, contient les domaines réalistes élémentaires qui sont cinq scalaires : `void`, `boolean`, `integer`, `float` et `string`, et deux structurels : `array` et `class` ;
- la troisième couche, appelée *fruit*, contient tous les autres domaines réalistes. Ces domaines réalistes héritent des domaines réalistes appartenant à la couche supérieure. Ils sont organisés en deux catégories : la bibliothèque standard et la bibliothèque utilisateur. La bibliothèque standard fournit entre autres les domaines réalistes suivants :
  - `boundinteger` et `boundfloat` qui représentent respectivement des intervalles d'entiers et de nombres flottants ;
  - `even` et `odd` qui représentent respectivement des nombres (des entiers ou des nombres flottants) pairs et impairs ;
  - `bag` qui représente un ensemble de constantes ou de domaines réalistes.

Domaines réalistes paramétrables Les domaines réalistes peuvent être paramétrés. Ils peuvent recevoir des arguments constants (comme des booléens — `true` ou `false` —, des entiers relatifs — `42`, `0x2a`, `052` — ou des nombres flottants — `4.2` —) ou des domaines réalistes. Cette dernière possibilité est très importante pour la génération récursive de structures, comme

les tableaux, objets, graphes, automates etc.

Exemple 3.2. *Domaine réaliste avec de simples arguments*
Le domaine réaliste `boundinteger(`X`, `Y`)` contient tous les entiers entre X et Y. Le domaine réaliste `string(`L`, `X`, `Y`)` est destiné à contenir toutes les chaînes de taille L constituées des caractères de X à Y en code-points Unicode. Dans le domaine réaliste `string(boundinteger(7, 16), 0x20, 0x7e)`, la taille de la chaîne est définie par un autre domaine réaliste.

Nous présentons maintenant Praspel, un langage de spécification basé sur les domaines réaslistes.

3.2 Praspel

Les domaines réalistes sont implémentés pour PHP dans Praspel, qui signifie *PHP Realistic Annotation and SPEcification Language*. Praspel est un langage d'annotation et de spécification basé sur le paradigme de la Conception par Contrat (*Design-by-Contract* [40]). Dans cette section, nous présentons la syntaxe et la sémantique du langage.

Praspel est écrit dans les commentaires API de PHP (lexème `T_DOC_COMMENT`), *i.e.* commençant par `/**` et terminant par `*/`, par exemple : `/** Documentation API */`.
La grammaire de Praspel est présentée dans la figure 3.3.

Praspel permet de mélanger la documentation informelle et les contraintes formelles, appelées clauses, au sein du même bloc de documentation API. Les clauses de Praspel sont ignorées par l'interpréteur PHP et par les IDE (*Integrated Development Environment*). De plus, comme les clauses commencent par le symbole `@`, qui se trouve être le formalisme majoritairement adopté par les documentations API, Praspel ne rentre pas en conflit avec les formatteurs et générateurs de documentations API.

Nous allons expliquer les notations utilisées dans la grammaire de Praspel présentée dans la figure 3.3. La notation $(\sigma)?$ signifie que σ est optionel. La notation $(\sigma)^r_s$ représente une séquence finie d'éléments reconnaissant σ, où r est soit $+$ pour reconnaître une ou plusieurs fois, soit $*$ pour zéro ou plusieurs fois, et s est le séparateur des éléments de la séquence.
Les autres entités syntaxiques sont expliquées dans le manuel de PHP [46].

Nous allons maintenant présenter l'assignation d'un domaine réaliste à une donnée dans la section 3.2.1 ainsi que l'écriture des contrats avec Praspel

annotation : := *(clause)**
clause : := *requires-clause* ;
| *ensures-clause* ;
| *throwable-clause* ;
| *invariant-clause* ;
| *behavior-clause*
requires-clause : := `@requires` *expressions*
ensures-clause : := `@ensures` *expressions*
throwable-clause : := `@throwable` *(identifier)*$^{+}_{,}$
invariant-clause : := `@invariant` *expressions*
behavior-clause : := `@behavior` *identifier* {
(*requires-clause* ;
| *ensures-clause* ;
| *throwable-clause* ;)$^{+}$ }
expressions : := *(expression)*$^{+}_{\texttt{and}}$
expression : := *real-dom-spec*
| `\pred(`*predicate*`)`
real-dom-spec : := *variable* (`:` *real-doms*
| `domainof` *variable)*
variable : := *constructors* | *identifier*
constructors : := `\old(`*identifier*`)` | `\result`
real-doms : := *real-dom*$^{+}_{\texttt{or}}$
real-dom : := *identifier* `(`*arguments*`)`
| *built-in*
built-in : := `void()`
| `integer()`
| `float()`
| `boolean()`
| `string(`*arguments*`)`
| `array(`*arguments*`)`
| `class(`*arguments*`)`
arguments : := *(argument)*$^{*}_{,}$
argument : := *number* | *string*
| *real-dom* | *array*
array : := `[`*pairs*`]`
pairs : := *(pair)*$^{*}_{,}$
pair : := (`from` *real-doms*)$^{?}$ `to` *real-doms*

FIGURE 3.3 – Grammaire de la syntaxe concrète

dans la section 3.2.2.

3.2.1 Assigner un domaine réaliste à une donnée

Un domaine réaliste est porté par un argument de méthode ou par un attribut de classe. Nous expliquons la syntaxe et la sémantique d'une telle

déclaration.

Assignation L'opérateur : représente une assignation de un ou plusieurs domaines réalistes pour une même donnée. Ainsi, la construction syntaxique :

$$i\texttt{: } t_1\texttt{(...) or ... or } t_n\texttt{(...)}$$

associe au moins un domaine réaliste (parmi t_1`(...)`, ..., t_n`(...)`) à un identifiant i. Nous utiliserons l'expression *disjonction de domaines* pour parler de la syntaxe et l'expression *union de domaines* pour parler de la sémantique. La partie gauche représente le nom de l'argument de la méthode, alors que la partie droite représente une liste de domaines réalistes séparés par le symbole `or`.
La sémantique d'une telle déclaration est que le domaine réaliste de l'identifiant i peut être t_1`(...)` ou ... ou t_n`(...)`, *i.e.* l'union des domaines réalistes t_1`(...)` à t_n`(...)`. Ces domaines réalistes sont (préférablement) mutuellement exclusifs. S'ils ne le sont pas, nous pourrions voir les mêmes données être générées involontairement et perdre en efficacité à cause de la redondance.

Exemple 3.3. *Un identifiant avec plusieurs domaines réalistes*
La déclaration `x: integer() or float() or boolean()` signifie que la variable `x` peut être un entier, un nombre flottant ou un booléen.

Dépendance L'opérateur `domainof` décrit une dépendance entre les domaines réalistes de deux identifiants. La construction syntaxique :

$$i\texttt{ domainof } j$$

crée cette relation. La sémantique d'une telle construction est une unification des domaines réalistes, *i.e.* le domaine réaliste choisi pour j est le même que pour i.

Exemple 3.4. *Unification entre deux données*
Nous présentons une unification entre `x: integer() or float()` et `y domainof x`, deux variables. Ces déclarations signifient que si `x` est un entier, alors `y` sera également un entier, sinon `x` et `y` seront des nombres flottants.

Domaines réalistes scalaires Nous donnons quelques exemples d'utilisation des domaines réalistes scalaires.

Exemple 3.5. *Utilisation des scalaires*
Nous présentons une suite de déclaration de domaines réalistes scalaires avec tous leurs paramètres : `void()`, `boolean()`, `integer()`, `float()` et

`string`(*length*, *from*, *to*).
Le domaine réaliste `void` représente l'ensemble vide, *i.e.* ne contenant que `null`. Le domaine réaliste `boolean` ne contient que deux valeurs : `true` et `false`. Le domaine réaliste `integer` contient tous les entiers relatifs que la machine peut supporter. Le domaine réaliste `float` contient tous les nombres flottants relatifs que la machine peut supporter. Et enfin, le domaine réaliste `string`, comme vu précédemment, contient des chaînes de caractères de la taille *length* constituées des caractères *from* à *to* en code-points Unicode.

Description de tableaux Un domaine réaliste peut être paramétré par une description de tableau, qui a la forme suivante :

$$\begin{array}{llllll} \texttt{[} & \texttt{from} & T_1^1\texttt{(...)} & \texttt{or ... or} & T_i^1\texttt{(...)} & \\ & \texttt{to} & T_{i+1}^1\texttt{(...)} & \texttt{or ... or} & T_n^1\texttt{(...),} & \\ & \texttt{...} & & & & \\ & \texttt{from} & T_1^k\texttt{(...)} & \texttt{or ... or} & T_j^k\texttt{(...)} & \\ & \texttt{to} & T_{j+1}^k\texttt{(...)} & \texttt{or ... or} & T_m^k\texttt{(...)} & \texttt{]} \end{array}$$

C'est une séquence entre les symboles `[` et `]` de paires séparées par le symbole `,`. Chaque paire est composée d'un domaine introduit par le symbole `from` et d'un co-domaine introduit par le symbole `to`. Chaque domaine et co-domaine sont une disjonction de domaines réalistes séparés par le symbole `or`. Le domaine est optionel et un entier auto-incrémenté sera utilisé s'il est manquant : nous prendrons le plus grand index numérique déjà existant auquel nous ajouterons 1, sinon nous utiliserons 0.
Nous détaillons la sémantique d'une description de tableau quand elle est utilisée en paramètre du domaine réaliste `array`. Ce dernier a deux arguments : une description de tableau et une taille.

Exemple 3.6. *Spécification d'un tableau*
Considérons les déclarations de la figure 3.4. `a1` décrit un tableau homogène

```
a1: array([from integer() to boolean()], boundinteger(7, 42))
a2: array([to boolean(), to float()], 7)
a3: array([from integer() to boolean() or float()], 7)
a4: array([from string(11) to boolean(),
           to float() or integer()], 7)
```

FIGURE 3.4 – Quelques déclarations de tableaux

d'entiers vers booléens. La taille du tableau produit est un entier entre 7 et 42. Dans l'optique de produire un tableau avec une taille fixe, nous utiliserons explicitement une constante, comme dans les exemples suivants. `a2` décrit deux

tableaux homogènes : soit un tableau de booléens, soit un tableau de nombres flottants, mais pas les deux. Dans tous les cas, le tableau produit aura une taille de 7. `a2` est strictement équivalent à `array([to boolean()], 7) or array([to float()], 7)`. `a3` décrit un tableau hétérogène de booléens et de nombres flottants ensemble. Finalement, `a4` décrit soit un tableau homogène de chaîne de caractères vers booléens, soit un tableau hétérogène de nombres flottant et d'entiers.

Domaines réalistes structurels Nous donnons quelques exemples d'utilisation des domaines réalistes structurels `array` et `class`.

Exemple 3.7. *Utilisation des structurels*
Nous présentons le prototype des domaines réalistes structurels avec tous leurs paramètres : `array([`*declaration*`],` *length*`)` et `class('`*classname*`')`. Le domaine réaliste `array` vient d'être abordé. Le domaine réaliste `class` représente une instance de la classe *classname* ou un de ses enfants. Par exemple : `class('Hoa\Realdom\Integer')` représente une instance du domaine réaliste `integer`.

3.2.2 Écrire des contrats avec Praspel

Cette section décrit la syntaxe et la sémantique de la partie de Praspel dédiée à la description de contrats pour les classes et les méthodes écrites en PHP. En programmation orientée objet, nous considérons que les contrats s'expriment sous trois formes : invariant, pré-condition et post-condition. Si en plus, nous avons des exceptions, alors nous pourrons considérer une quatrième forme de contrats : exceptionnels. Nous allons expliquer toutes les formes que les contrats peuvent adopter.

Invariants Dans un contexte de classe, ce sont sur les attributs que sont portés les invariants. Ils ont la syntaxe suivante :

`@invariant` I_1 `and ... and` I_p`;`

Classiquement, les invariants doivent être satisfaits après la création de l'objet et doivent être préservés à travers les exécutions successives des méthodes de l'objet.

Exemple 3.8. *Invariant simple*
L'invariant de la figure 3.5 spécifie que l'attribut `a` de la classe `C` est un booléen avant et après n'importe quel appel d'une méthode de l'objet.

```
class C {

    /**
     * @invariant a: boolean();
     */
    protected $a;
}
```

FIGURE 3.5 – Déclaration d'un invariant

Contrats sur les méthodes Praspel permet d'exprimer des contrats sur les méthodes d'une classe. Le contrat spécifie une pré-condition à travers la clause `@requires` qui doit être satisfaite pour que la méthode puisse être appelée. En retour, la méthode établit une post-condition à travers la clause `@ensures`. Le contrat spécifie également un ensemble possible d'exceptions qui peuvent être levées lors de l'exécution de la méthode à travers la clause `@throwable`. La syntaxe d'un contrat basique est la suivante :

```
/**
 * @requires  R_1 and ... and R_n;
 * @ensures   E_1 and ... and E_m;
 * @throwable T_1, ..., T_p;
 */
public function f ( ) { }
```

Nous expliquons la sémantique d'un tel contrat :

- chaque R_x ($1 \leq x \leq n$) représente l'affectation d'un domaine réaliste à une donnée qui établit une pré-condition. L'appelant de la méthode doit garantir que la méthode est appelée dans un état où la conjonction des propriétés $R_1 \wedge \cdots \wedge R_n$ est vérifiée (*i.e.* que les valeurs des données doivent satisfaire les prédicats des domaines réalistes qui leurs sont affectés). Par défaut, si un argument n'a pas de domaine réaliste assigné, alors il sera implicitement déclaré avec le domaine réaliste `undefined` ;
- chaque E_y ($1 \leq y \leq m$) est l'affectation d'un domaine réaliste à une donnée après l'exécution d'une méthode. La méthode, si elle termine sans erreur, devrait retourner une valeur telle que la conjonction $E_1 \wedge \cdots \wedge E_m$ soit satisfaite. Usuellement, les post-conditions peuvent faire référence aux variables avant l'exécution de la méthode et au résultat de la méthode. Ainsi, Praspel est étendu de deux nouvelles constructions : `\old(`e`)` réfère à la valeur de e avant l'exécution de la méthode, `\result` réfère à la valeur retournée par la méthode ;

```
/**
 * @requires  x:       boundinteger(0, 42) and
 *            y:       float();
 * @ensures   \result: float();
 * @throwable FoobarException;
 */
public function f ( $x, $y ) {

    if(42 == $x)
        throw new FoobarException('Bazqux');

    return $x * 2 + $y;
}
```

FIGURE 3.6 – Une fonction simple avec son contrat associé

— chaque T_z ($1 \leq z \leq p$) est le nom d'une exception. La méthode peut lever une des exceptions $T_1, \ldots, T_p$ ou un enfant d'une d'elle. Par défaut, la spécification n'autorise aucune exception ne pouvant être levée par la méthode.

Exemple 3.9. *Contrat simple*
Le contrat de la figure 3.6 spécifie une pré-condition, une post-condition et une clause exceptionnelle. La pré-condition dit que l'argument `x` doit être un entier entre 0 et 42 et que l'argument `y` doit être un nombre flottant. La post-condition dit que le résultat (`\result`) doit être un nombre flottant. Et la clause exceptionnelle autorise la levée de l'exception `FoobarException`.

Un contrat peut également porter une clause `@invariant` pour raffiner un invariant, *i.e.* lui ajouter des contraintes supplémentaires.

Clauses comportementales En complément, Praspel permet de décrire explicitement des comportements à l'intérieur des contrats. Par exemple, lorsqu'une méthode calcule plusieurs résultats en fonction du type de ses arguments (car un argument peut avoir plusieurs domaines réalistes) ou de la valeur de ses arguments, elle a différents comportements.
Un comportement est définie par un nom et des clauses `@requires`, `@ensures` et `@throwable` locales. La syntaxe d'une telle clause est la suivante :

```
/**
 * @requires R_1 and ... and R_n;
 * @behavior α {
 *     @requires  A_1 and ... and A_k;
 *     @ensures   E_1 and ... and E_j;
 *     @throwable T_1, ..., T_t;
 * }
 * @ensures   E_{j+1} and ... and E_m;
 * @throwable T_{t+1}, ..., T_l;
 */
```

La sémantique d'une clause comportementale α est la suivante :

- l'appelant de la méthode doit garantir que l'appel est effectué dans un état où la propriété $R_1 \wedge \cdots \wedge R_n$ est vérifiée. Néanmoins, la propriété $A_1 \wedge \cdots \wedge A_k$ doit également être vérifiée ;
- la méthode appelée établit l'état où la propriété $(A_1 \wedge \cdots \wedge A_k \implies E_1 \wedge \cdots \wedge E_j) \wedge E_{j+1} \wedge \cdots \wedge E_m$ est vérifiée, *i.e.* qu'uniquement la post-condition du comportement choisi doit être vérifiée si la pré-condition du comportement est satisfaite ;
- les exceptions T_i $(1 \leq i \leq t)$ peuvent seulement être levées si les pré-conditions $R_1 \wedge \cdots \wedge R_n$ et $A_1 \wedge \cdots \wedge A_k$ sont vérifiées. Enfin, les exceptions T_j $(t+1 \leq j \leq l)$ peuvent être levées si la pré-condition $R_1 \wedge \cdots \wedge R_n$ est vérifiée.

Seule la clause `@behavior` peut contenir d'autres clauses sauf `@behavior` elle-même. Si une de ces clauses est déclarée en dehors du comportement, elle sera automatiquement associée à un comportement global par défaut. Si une clause est manquante dans un comportement, alors la clause définie dans l'espace global sera utilisée.

Exemple 3.10. *Contrat à comportements*
Nous proposons un exemple d'un contrat avec des comportements dans la figure 3.7. Le contrat précise ceci : nous avons une méthode qui a deux comportements en fonction de son dernier argument. Dans tous les cas, elle va chercher et retourner si `needle` existe dans `haystack`. Elle va donc retourner un booléen. Nous remarquons que l'argument `matches` est passé en référence (grâce au symbole `&`). Si sa valeur est `null`, *i.e.* la seule valeur que contient le domaine réaliste `void`, alors nous allons la remplir d'un tableau qui contiendra tous les indexes de `needle` dans `haystack`.
Nous donnons un exemple d'utilisation (où `$obj` est l'objet portant la méthode `exists`) dans la figure 3.8. Nous remarquons que la première exécution de la méthode `exists` retourne `true` car 3 existe dans le tableau `$foo`. La seconde exécution utilise l'argument `$matches` avec la variable `$index`. Cette

```
/**
 * @requires needle:   integer() and
 *           haystack: array(
 *                         [to integer()],
 *                         boundinteger(1, 256)
 *                     );
 * @behavior matches {
 *     @requires matches: void();
 *     @ensures  matches: array(
 *                           [to boundinteger(0, 255)],
 *                           boundinteger(1, 256)
 *                        );
 * }
 * @ensures \result: boolean();
 */
public function exists ( $needle,
                         $haystack,
                         &$matches = false ) {

    $intersect = array_intersect($haystack, array($needle));

    if(null === $matches)
        $matches = array_keys($intersect);

    return 0 < count($intersect);
}
```

FIGURE 3.7 – Une méthode avec deux comportements

exécution retourne toujours `true` car 3 existe toujours dans le tableau `$foo`, mais en plus `$index` est remplie des indexes où la valeur 3 est présente dans le tableau. Nous observons bien plusieurs comportements.

Prédicats Il est possible d'écrire des prédicats en PHP directement dans un contrat à travers la construction `\pred(...)`. Cette construction permet d'ajouter des contraintes sur une ou plusieurs variables.

Exemple 3.11. *Contraintes sur plusieurs variables*
Nous voyons dans la figure 3.9 comment spécifier que `x` $\leq$ `y`. Notons que les `$` qui introduisent normalement les variables en PHP disparaissent à l'intérieur de la construction `\pred` pour simplifier la lecture.

Cette construction, si elle est située en pré-condition, peut introduire du rejet lors de la génération de données de tests. Cette construction peut faire l'objet de nouvelles recherches pour éviter certains rejets en faisant remonter

```
//           0  1  2  3  4  5  6  7  8  9   sont les indexes
$foo = array(0, 9, 3, 2, 4, 2, 7, 3, 5, 3);

var_dump($obj->exists(3, $foo));
// Première exécution : bool(true)

var_dump($obj->exists(3, $foo, $index));
// Seconde exécution : bool(true)

print_r($index);
// Array
// (
//     [0] => 2
//     [1] => 7
//     [2] => 9
// )
```

FIGURE 3.8 – Mise en évidence des comportements de la figure 3.7

```
/**
 * @requires x: integer() and
 *           y: integer() and
 *           \pred('x <= y');
 * @ensures  \result: integer();
 */
```

FIGURE 3.9 – `\pred` pour vérifier que $x \leq y$

les contraintes sur les générateurs des domaines réalistes. En revanche, en post-condition, elle peut être très utile pour écrire des contraintes plus fines.

Couverture de contrats et sélecteurs de domaines réalistes Avec la disjonction de domaine et la clause `@behavior`, nous voyons que nous pouvons avoir une couverture partielle du contrat. Par exemple, avec : `@requires i: integer() or float()`, si `i` a le domaine réaliste `integer`, la couverture pour cette clause n'est pas complète. Peut-être que si l'autre domaine réaliste avait été choisi, ça aurait eu des répercutions sur le reste du contrat et sur sa couverture.
Praspel supporte nativement, de part sa construction, la détection de couverture du contrat. Par la suite, les outils que nous allons utiliser nous afficherons la couverture du code.
Pour influencer la couverture, nous avons introduit la notion de sélecteurs de domaines réalistes. Pour l'instant, seul le sélecteur aléatoire est implémenté

et ce n'est encore qu'à un stade expérimental. Ces sélecteurs permettent de déterminer quel domaine réaliste choisir lorsque nous sommes devant une disjonction de domaines. C'est une façon possible d'influencer la couverture.

3.3 Générateurs de données

Les domaines réalistes sont conçus pour que les générateurs de données en soient découplés, afin que l'on ait deux entités bien différentes. Nous allons expliquer comment sont reliés les deux mécanismes après avoir expliqué comment fonctionnent les générateurs de données.

3.3.1 Entiers et nombres flottants

Les générateurs de données ne savent générer volontairement que des entiers et des nombres flottants. C'est alors aux domaines réalistes de se servir de ces données « élémentaires » pour construire leurs ensembles de valeurs.

Un générateur de données a donc deux méthodes : `getInteger` et `getFloat`. Ces deux méthodes ont deux paramètres, `$lower` et `$upper`, qui sont respectivement la borne minimum et maximum de l'intervalle dans lequel nous allons générer les données. Ces bornes doivent pouvoir être optionnelles, ce qui implique qu'elles doivent avoir des valeurs par défaut.

Exemple 3.12. *Squelette d'un générateur de données*
Nous donnons le squelette simplifié d'un générateur de données dans la figure 3.10. Nous remarquons que les arguments des méthodes `getInteger` et `getFloat` sont optionels, et valent `null` par défaut et non pas un nombre quelconque.

Ces deux méthodes sont les seules que les domaines réalistes doivent utiliser. Nous pourrons alors substituer un générateur de données par un autre sans que cela ne perturbe le fonctionnement de la génération de données dans sa globalité. En réalité, chaque générateur de données hérite d'un générateur de données abstrait qui propose des mécanismes pour faciliter la mise en place et l'utilisation de ces générateurs de données.

3.3.2 Lien entre les domaines réalistes et les générateurs de données

Nous insistons sur le fait que les générateurs de données sont totalement découplés des domaines réalistes, comme le montre la figure 3.11. Le

```
class Sampler {

    public function getInteger ( $lower = null,
                                 $upper = null ) {

        // calcul d'un entier.
        $integer = ...;

        return $integer;
    }

    public function getFloat ( $lower = null,
                               $upper = null ) {

        // calcul d'un nombre flottant.
        $float = ...;

        return $float;
    }
}
```

FIGURE 3.10 – Squelette épuré d'un générateur de données

lien entre les deux s'effectue à travers un mécanisme très simple. En effet, chaque domaine réaliste hérite d'un grand-parent qui porte tous les mécanismes nécessaires au bon fonctionnement de ces derniers ; par exemple : la construction du domaine réaliste, la gestion des arguments ou le lien entre les domaines réalistes et les générateurs de données etc.

Nous avons vu que les caractéristiques fonctionnelles des domaines réalistes s'expriment à travers les méthodes `predicate` et `sample`. L'implémentation de la méthode `predicate` est toujours laissée au soin du développeur. En revanche, la méthode `sample` est pré-implémentée et appelle la méthode `_sample` qui joue le vrai rôle de générateur.

C'est donc la méthode `sample` qui se charge d'utiliser le générateur de données courant et d'appeler `_sample`. En même temps, elle va toujours tester la valeur du générateur par rapport à `predicate` pour s'assurer qu'elle est correcte, ce qui doit toujours être le cas. Toutefois, si la valeur générée est fausse, `sample` va recommencer un certain de nombre de fois. Nous appelons cette notion *le nombre d'essais maximum* (ou *max try* [28]). Si cette limite est atteinte, une erreur sera levée en précisant que nous n'avons pas été capables de générer une valeur.

Nous présentons dans la figure 3.12 l'algorithme simplifié qui est utilisé. Ainsi, la méthode `_sample` reçoit le générateur de données et n'a qu'un seul

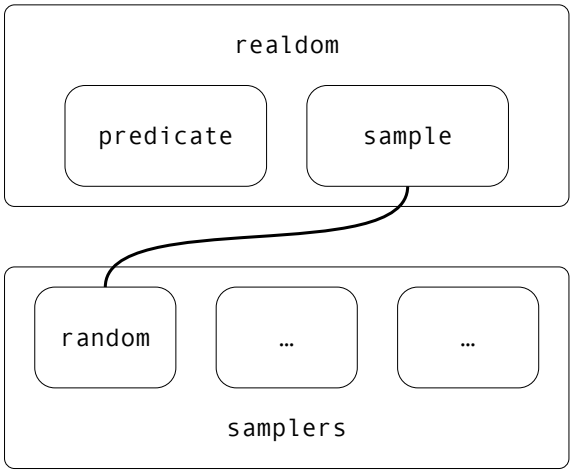

FIGURE 3.11 – Découplage entre les domaines réalistes et les générateurs de données

rôle, comme le montre cet exemple du domaine réaliste `boundinteger`.

Exemple 3.13. *Implémentation de la propriété de générabilité dans le domaine réaliste* ***`boundinteger`***
Nous donnons l'implémentation simplifiée de la propriété de générabilité pour le domaine réaliste `boundinteger` connaissant le fonctionnement de la méthode `sample`, dans la figure 3.13.

La méthode `sample` utilise la méthode `getSampler` pour obtenir le générateur de données. Ce dernier est affecté de manière statique grâce à la méthode `setSampler`.

Exemple 3.14. *Affectation d'un générateur de données aux domaines réalistes*
L'instruction suivante montre comment affecter un générateur de données aléatoires à tous les domaines réalistes courants : `Hoa\Realdom::setSampler(new Hoa\Test\Sampler\Random());`

Cela implique que l'on peut changer le générateur de données pour tous les domaines réalistes à la volée sans devoir réinitialiser l'ensemble des domaines réalistes.

La vérification de la construction `\pred` se fait à l'aide de la méthode `otherPredicates` qui va lier toutes les données des variables entre elles et évaluer le prédicat. La génération par rejet est plus efficace si la probabilité de rejet est faible. Alors que les méthodes `_sample` vont toujours générer des

```
public function sample ( ) {

    $maxtry  = $this->getMaxTry();
    $sampler = $this->getSampler();

    do {

        $sampled = $this->_sample($sampler);

    } while(   false === $this->predicate($sampled)
            && false === $this->otherPredicates($sampled)
            && 0 < --$maxtry);

    if(0 >= $maxtry)
        // error.

    return $sampled;
}
```

FIGURE 3.12 – Sur-algorithme des générateurs des domaines réalistes

```
protected function _sample ( \Hoa\Test\Sampler $sampler ) {

    // nous récupérons $lower et $upper, respectivement
    // pour la borne minimum et maximum.
    return $sampler->getInteger($lower, $upper);
}
```

FIGURE 3.13 – Squelette d'un véritable générateur d'un domaine réaliste

données valides, les prédicats seront les seuls à introduire du rejet et seront la cause de l'erreur *max-try*.

3.4 Fonctionnement interne

Nous expliquons certains fonctionnements internes de Praspel et des domaines réalistes.

3.4.1 Modèle objet de Praspel

Praspel est représenté en mémoire par un modèle objet, *i.e.* un ensemble de classes représente le langage. Ainsi, nous trouvons une classe pour représenter une clause, une variable, une disjonction de domaines, une description

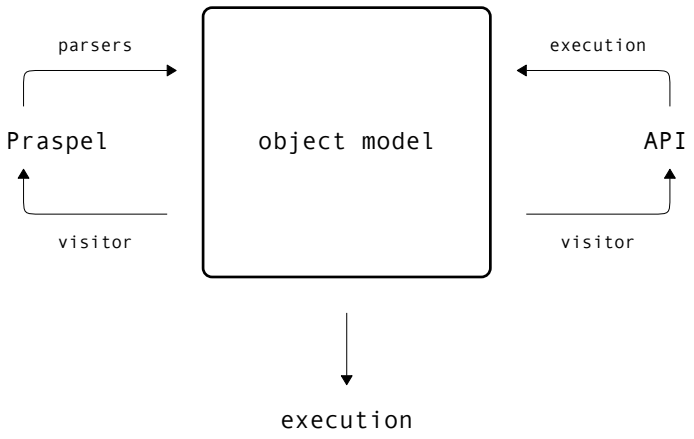

FIGURE 3.14 – Le modèle objet, point central de Praspel

```
$parser      = new Hoa\Test\Praspel\Compiler();
$parser->compile($praspel);
$objectModel = $parser->getRoot();
```

FIGURE 3.15 – Interprétation d'un code Praspel en modèle objet

de tableau etc. Ce modèle objet est le point central de Praspel. Nous expliquons comment instancier ce modèle objet. La figure 3.14 résume ce qui va suivre.

Interprétation Praspel est un langage qui peut être interprété. Si nous utilisons le langage Praspel tel que décrit précédemment, nous utiliserons les analyseurs lexicaux et syntaxiques pour vérifier les contraintes syntaxiques et construire le modèle objet directement en mémoire. Lors de sa construction, le modèle objet vérifiera les contraintes sémantiques de Praspel. Ce modèle objet, une fois construit sans erreur, est exécutable. Par exemple, nous sommes capables de sélectionner une variable, puis un de ses domaines et enfin de générer une valeur.

Exemple 3.15. *Interprétation d'un code Praspel*
Dans cet exemple ainsi que dans les exemples suivants, nous considérerons le code Praspel suivant : `$praspel = '@requires i: boundinteger(7, 42) or boolean();';`. Dans la figure 3.15, nous voyons comment construire un modèle objet à travers l'interprétation. Nous instancions l'interpréteur qui va lire le code Praspel. Puis nous allons récupérer la racine du résultat, souvent sous la forme d'un arbre. Ici, ce sera le modèle objet.

```
$contract  = new Hoa\Test\Praspel\Contract(...);
$contract
    ->clause('requires')
    ->variable('i')
        ->belongsTo('boundinteger')
            ->with(7)
            ->_comma
            ->with(42)
            ->_ok()
        ->_or
        ->belongsTo('boolean')
            ->_ok();
```

FIGURE 3.16 – Construction d'un modèle objet manuellement avec l'API

API Si nous ne voulons pas utiliser les analyseurs lexicaux et syntaxiques, nous pouvons utiliser directement l'API du modèle objet pour le construire. Nous ne pourrons plus parler de contraintes syntaxiques mais uniquement de contraintes sémantiques. L'API proposée se veut facile à lire, rapide à écrire et très intuitive.

Exemple 3.16. *Construction manuelle d'un modèle objet*
Dans la figure 3.16, nous construisons le modèle objet équivalent au code Praspel utilisé dans l'exemple 3.15. Comme nous pouvons l'observer, la lecture du code PHP qui construit le modèle objet à de fortes ressemblances avec la lecture du code Praspel : « nous avons une *clause* `@requires` avec une *variable* `i` qui *appartient au* domaine réaliste `boundinteger` *avec* deux arguments qui sont `7` et `42`, *ou* qui *appartient au* domaine réaliste `boolean` ».

Compilation Praspel est également un langage qui peut être compilé. En effet, une fois le modèle objet construit, nous pouvons lui appliquer des visiteurs [23] (patron de conception bien connu qui consiste à séparer la structure des traitements). Un premier visiteur intéressant permet de générer le code PHP qui permet de construire un tel modèle objet ; cela permet d'effectuer une « sauvegarde » du modèle pour l'exécuter plus tard. Un second visiteur intéressant permet de générer le code Praspel qui permettrait de construit un tel modèle objet ; cela permet par exemple de visualiser plus facilement ce que le modèle objet exprime.

Exemple 3.17. *Application des visiteurs sur le modèle objet*
Dans cet exemple, nous allons utiliser le modèle objet construit dans l'exemple 3.15 pour lui appliquer deux visiteurs : le premier sera le visiteur Praspel

```
$vPraspel = new Hoa\Test\Praspel\Visitor\Praspel();
$vPHP     = new Hoa\Test\Praspel\Visitor\Php();

echo 'Praspel:' . "\n" .
     $vPraspel->visit($objectModel) . "\n" .
     'PHP:' . "\n" .
     $vPhp->visit($objectModel) . "\n";
```

FIGURE 3.17 – Des visiteurs sur le modèle objet

et donnera le code Praspel utilisé dans l'exemple 3.15 et le second sera le visiteur PHP et donnera le code de l'exemple 3.16. La figure 3.17 nous montre comment appliquer les visiteurs.

Le processus qui sera majoritairement utilisé sera l'interprétation de Praspel pour ensuite le compiler en PHP afin de pouvoir exécuter le contrat plusieurs fois quand nous le souhaitons.

Collection de modèles objets Comme un modèle objet représente Praspel, soit un contrat, nous y avons inséré une légère couche afin de pouvoir manipuler une collection de contrats. À chaque contrat est associé un identifiant unique constitué du nom de la classe et de la méthode séparée par le symbole `::` (qui est l'opérateur de résolution statique de PHP). Cela nous évite de devoir re-instancier les modèles objets à chaque fois que nous en aurons besoin, surtout que nous sommes capables de réinitialiser les contrats.

Exemple 3.18. *Manipuler une collection de contrats*
La figure 3.18 nous montre comment manipuler plusieurs contrats à travers une collection. Si le contrat existe, alors nous le récupérons et nous le réinitialisons (*i.e.* toutes les valeurs générées disparaissent et les domaines réalistes sélectionnés sont désélectionnés). Sinon, nous le créons et l'ajoutons à la collection. Nous comprenons que le modèle objet ne sera construit qu'une seule fois et restera en mémoire.

3.4.2 Gestions des arguments des domaines réalistes

Une des promesses des domaines réalistes est la facilité de développement. Nous savons que les domaines réalistes peuvent recevoir des valeurs constantes ou d'autres domaines réalistes à travers des arguments. Dans cette optique de facilité, nous ne laissons plus le langage PHP gérer lui-même les arguments que le domaine réaliste va recevoir à travers un constructeur car nous pouvions rencontrer des problèmes. En effet, PHP utilise le principe de

```
$collection = Hoa\Test\Praspel::getInstance();

if(true === $collection->contractExists($id)) {

    $contract = $collection->getContract($id);
    $contract->reset();
}
else {

    $contract = new Hoa\Test\Praspel\Contract(...);
    $contract->clause(...)->...;
    $collection->addContract($contract);
}
```

FIGURE 3.18 – Utiliser la collection de contrats comme un multiton

substitution de Liskov [38], formulé par Liskov et Wing, pour son modèle d'héritage objet. Ce principe sied bien à PHP mais pas nécessairement à la philosophie d'héritage des domaines réalistes : les arguments d'un domaine réaliste ne font pas partie de l'héritage, *i.e.* la signature des domaines réalistes ne comprend pas les arguments. Ainsi, un enfant peut avoir des arguments totalement différents de son parent. C'est pourquoi nous devons les définir manuellement. Ce sera aux développeurs de réutiliser les arguments du parent s'il le désire mais ce comportement n'est pas supporté par défaut.

Nous étudions l'API pour manipuler des arguments. Nous sommes au niveau des domaines réalistes.

Définir des arguments Nous définissons des arguments pour un domaine réaliste à travers un simple attribut :

`protected $_arguments = array('`*name*`', ...);`

Et nous les utilisons dans nos méthodes à travers un accès de tableau, comme si l'objet avait des entrées de tableau :

`$this['`*name*`']`

Exemple 3.19. *Définir des arguments par défaut*
Pour savoir si un paramètre existe ou si valeur est `null`, nous pouvons utiliser l'instruction de PHP `isset`, comme le montre la figure 3.19 qui expose le code du domaine réaliste `boundinteger`. Nous remarquons sur cette figure que le constructeur des domaines réalistes est `construct` alors que le constructeur des objets en PHP est `__construct`. Cela est volontaire pour

```
class Boundinteger extends Integer {

    protected $_name      = 'boundinteger';
    protected $_arguments = array('lower', 'upper');

    public function construct ( ) {

        if(!isset($this['lower']))
            $this['lower'] = new Constinteger(~PHP_INT_MAX);

        if(!isset($this['upper']))
            $this['upper'] = new Constinteger( PHP_INT_MAX);

        return;
    }

    public function predicate ( $q ) {

        return    parent::predicate($q)
               && $q >= $this['lower']->getValue()
               && $q <= $this['upper']->getValue();
    }

    protected function _sample ( \Hoa\Test\Sampler $sampler ) {

        return $sampler->getInteger(
            $this['lower']->sample($sampler),
            $this['upper']->sample($sampler)
        );
    }
}
```

FIGURE 3.19 – Extrait du domaine réaliste `boundinteger`

ne pas confondre les deux constructions.
Le constructeur dit : si l'argument `lower` n'existe pas, alors nous le définissons. L'opération est répétée pour l'argument `upper`. Au final, nous l'utilisons de cette manière : `boundinteger(`*lower*`,` *upper*`)`, avec *lower* et *upper* optionnels.

Définir un nombre indéfini d'arguments Nous définissons un nombre indéfini d'arguments pour notre domaine réaliste avec le caractère « ... »[1] :

```
protected $_arguments = ...;
```

1. Unicode : 2026, UTF-8 : E2 80 A6

Pour accéder aux arguments, nous utiliserons un index numérique au lieu d'un index nommé.

Itérer les arguments L'API comprend d'autres méthodes, comme `getArguments()` pour avoir un tableau de tous les arguments et `count()` pour compter les arguments. Il existe aussi des facilitées pour itérer les arguments, comme la construction :

```
foreach($this as $name => $value)
```

ou avec un appel de méthode explicite :

```
foreach($this->getArguments() as $name => $value)
```

3.4.3 Paramétrage des générateurs de données

Dans certaines situations, il est préférable d'être capable de paramétrer les générateurs de données (qui, rappelons-le, ne génèrent que des entiers et des nombres flottants). Nous sommes désormais capables de paramétrer la borne minimum et maximum des entiers et nombres flottants à travers les paramètres `integer.min`, `integer.max`, `float.min` et `float.max`. Les paramètres sont une caractéristique native de Hoa (*i.e.* elle appartient au noyau) et permettent de paramétrer des classes ou des paquetages entiers à travers un mécanisme uniforme, soit au niveau du code, soit avec des fichiers de configuration externes.

Le besoin d'un tel paramétrage se fait ressentir pour certains domaines réalistes, par exemple générer une chaîne de caractères dont la taille maximum n'est pas précisée. Dans ce cas, c'est le générateur de données qui va décider de la taille maximum. Il n'est pas toujours souhaitable d'utiliser la borne minimum et maximum des entiers ou des nombres flottants telle que définie par la machine (2^{32} ou 2^{64} pour les entiers). Cela est facilement compréhensible si nous considérons les problèmes de mémoire que cela peut engendrer.

Exemple 3.20. *Paramétrer le générateur de données et l'assigner aux domaines réalistes*
La déclaration `boundinteger(2)` pourrait produire 153 car aucune borne maximum n'est précisée pour `boundinteger` et ce sera celle du générateur de données qui va être utilisée. Nous la redéfinissons lors de son instanciation dans la figure 3.20. Cela peut s'avérer très utile si nous voulons majorer la génération d'une donnée.

```
Hoa\Realdom::setSampler(new Hoa\Test\Sampler\Random(array(
    'integer.max' => 7
)));
```

FIGURE 3.20 – Modifier la borne maximum d'un générateur de données

3.5 Praspel et ses outils

Dans cette section, nous expliquons comment fonctionne Praspel et l'outil de gestion et génération automatique de tests dont il est accompagné.

3.5.1 Génération automatique de tests unitaires

La compilation des contrats écrits en Praspel couplé à l'outil Praspel nous permettent d'exécuter des contrats au choix. L'exécution d'un contrat signifie utiliser sa pré-condition pour générer des données de tests et les utiliser pour exécuter la méthode associée au contrat. De cette manière, nous avons bien un *générateur automatique de tests unitaires*.

Nous pouvons avoir une autre utilisation des contrats compilés. En effet, les contrats peuvent ne pas être utilisés pour de la génération de données, mais simplement pour de la vérification, *i.e.* que l'on peut écrire nos propres ensembles d'instructions en utilisant le code qui contient les contrats compilés et ne faire que de la vérification des pré et post-conditions.

Pour exécuter les tests, nous instrumentons le code, comme le montre la figure 3.21. Pour commencer, nous allons lire tout un dossier et le copier dans un dépôt de tests associé à une nouvelle révision. Le dossier et son contenu vont être copiés dans leur intégralité dans un dossier nommé `incubator` : il contient le code que nous testons. Dans un même temps, nous scannons tous les fichiers que nous recopions dans le dossier nommé `instrumented`. Si un fichier contient une classe PHP avec des méthodes annotées par du Praspel, nous allons compiler Praspel en PHP et modifier le code de la classe de manière à pouvoir exécuter les contrats finement (soit la pré-condition, soit la post-condition, soit les invariants etc.).

Dans le code instrumenté, une méthode `foo` annotée par du Praspel sera renommée `foo_body` et nous créerons les méthodes :

- `foo` qui va exécuter le contrat dans son ensemble ;
- `foo_pre` qui va exécuter la pré-condition ;
- `foo_post` qui va exécuter la post-condition ;
- `foo_exception` qui va exécuter la clause exceptionnelle ;

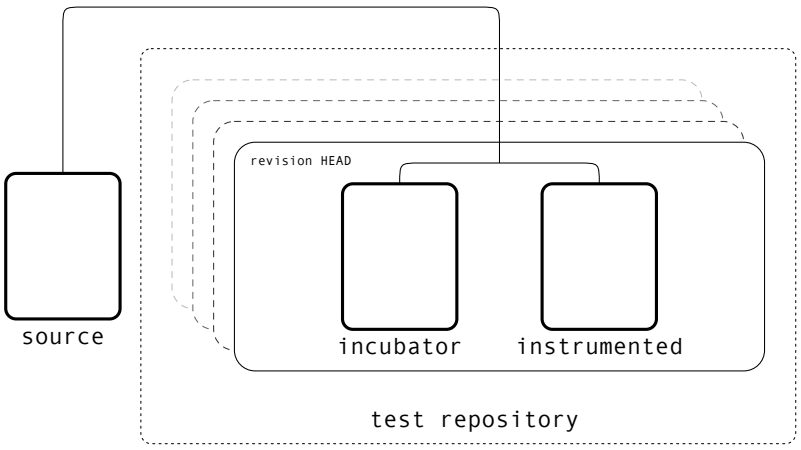

FIGURE 3.21 – Fonctionnement du générateur automatique de tests unitaires

— `foo_contract` qui va instancier le contrat si ce n'est pas déjà fait, grâce à la collection de contrats.

Nous exposons les squelettes de ces méthodes dans la figure 3.22. Nous voyons que le code instrumenté est strictement non-instrusif : il ne modifie en rien le comportement du code initial. Nous remarquons également que les méthodes introduites `*_pre`, `*_post` et `*_exception` retournent un booléen. Si une erreur est détectée, elle sera générée par une des méthodes `verify*` et sera propagée à travers des événements. Il existe deux catégories d'événements dans Hoa ; nous avons utilisé celle qui a le découplage le plus fort. Nous pouvons capturer les erreurs et établir des rapports de tests sans modifier le code instrumenté ni le code testé, mais en écrivant une capture d'événements dans un autre fichier à part.

Exemple 3.21. *Créer un rapport de tests*
Hoa propose également un système de flux unifiés, qui en plus est compris par les événements (c'est le principe d'appel dynamique étendu introduit par Hoa). La figure 3.23 nous montre comment créer un rapport de tests écrit au format XML que nous enverrons sur le réseau (par exemple sur une machine chargée de stocker les rapports de tests). Sur le canal d'événements `hoa://Event/Log/Test/Praspel`, nous attachons un flux dit composite, *i.e.* un flux qui contient un flux. Le flux contenu est un flux primitif et est un client qui se connecte sur la machine `192.168.0.2` en TCP. Les flux sont construits

```
public function foo ( ... ) {

    $this->foo_contract();
    $this->foo_pre(...);

    try {

        $result =
            $this->foo_body(...);
    }
    catch ( \Exception $e ) {

        $this->foo_exception($e);
        throw $e;
    }

    $this->foo_post($result, ...);

    return $result;
}
```

```
public function foo_pre ( ... ) {

    // nous retrouvons le contrat.
    return    $contract->verifyInvariants(...)
           && $contract->verifyPreCondition(...);
}

public function foo_post ( $result, ... ) {

    // nous retrouvons le contrat.
    return    $contract->verifyPostCondition(
                  $result, ...
              )
           && $contract->verifyInvariants(...);
}

public function foo_exception ( $exception ) {

    // nous retrouvons le contrat.
    return    $contract->verifyException($exception)
           && $contract->verifyInvariants(...);
}
```

FIGURE 3.22 – Squelette des méthodes instrumentées

```
event('hoa://Event/Log/Test/Praspel')->attach(
    new Hoa\Xml\Write(
        new Hoa\Socket\Client('tcp://192.168.0.2')->connect()
    )
);
```

FIGURE 3.23 – Créer un rapport de tests simplement grâce aux événements

de telles manières que lorsqu'ils vont recevoir une donnée, ils sauront, dans ce cas, la formater et l'envoyer.
Nous noterons que ce code peut-être placé n'importe où avant l'exécution des tests et qu'il est non-intrusif. Nous pouvons également attacher d'autres flux ou d'autres choses sur ce canal.

Nous allons maintenant expliquer comment fonctionne le verdict des tests.

3.5.2 Verdict basé sur le *Runtime Assertion Checking*

Le verdict du test est effectué à l'exécution, c'est pour cela que l'on parle de RAC, pour *Runtime Assertion Checking*. Quand la vérification d'une assertion échoue, une erreur spécifique est émise. Les erreurs du RAC (ou les échecs Praspel), peuvent être de cinq sortes :

1. échec d'une pré-condition, quand une pré-condition n'est pas satisfaite à l'appel d'une méthode ;
2. échec d'une post-condition, quand une post-condition n'est pas satisfaite à la fin d'une exécution d'une méthode ;
3. échec d'une clause exceptionnelle, quand la méthode lève une exception qui n'est pas attendue ;
4. échec d'un invariant, quand un invariant de classe est cassé ;
5. ou échec interne d'une pré-condition, qui correspond à la propagation d'un échec d'une pré-condition au niveau d'appel supérieur.

Les cas de tests sont générés et exécutés en ligne : les générateurs de données produisent des données de tests et la version instrumentée du code vérifie la conformance du code par rapport aux spécifications pour les entrées données. Le test réussit si aucun échec Praspel n'est détecté. Sinon, le test échoue et les erreurs émises indiquent où les échecs ont été détectés.

Chapitre 4

Grammar-based Testing avec les domaines réalistes

Sommaire

L'objectif que nous nous sommes fixés pour réussir une expérimentation est d'être capable de faire du *Grammar-based Testing* avec les domaines réalistes. Pour cela, il nous manquait de quoi exprimer une grammaire et travailler dessus. Nous avons donc construit un compilateur de compilateurs LL(k) [12] ainsi qu'un langage associé afin d'exprimer des grammaires plus facilement. Ensuite, nous nous sommes intéressés à la propriété de générabilité des domaines réalistes en voulant utiliser les récents travaux de Héam et Nicaud [31] pour les étendre à toutes les grammaires. Pour commencer, nous avons seulement considéré les feuilles des AST (pour *Abstract Syntax Tree*) produits par le compilateur de compilateurs qui sont exprimées

```
pp                ::= ( token | rule )+
token             ::= % ( skip | token ) ( identifier : )? identifier
                      ( -> identifier )?
rule              ::= identifier : ↩ rule-description
rule-description  ::= blank+ alternation
alternation       ::= concatenation+ |
concatenation     ::= quantification+
quantification    ::= simple quantifier? ( # identifier )?
simple            ::= ( ( alternation )
                        | :: identifier ( [ digit ] )? ::
                        | < identifier ( [ digit ] )? >
                        | identifier ()
                      ) ( # identifier )?
quantifier        ::= ? | + | * | { digit , digit }
```

FIGURE 4.1 – Grammaire du langage PP

avec le langage d'expressions régulières PCRE [30] (pour *Perl Compatible Regular Expressions*). Nous avons donc produit un compilateur de PCRE avec la propriété de générabilité intégrée. Une fois le mécanisme des feuilles suffisamment avancé, nous nous sommes plus particulièrement intéressés à l'implémentation des algorithmes de Héam et Nicaud pour les intégrer dans le compilateur de compilateurs. Cela consiste à parcourir l'AST afin d'atteindre les feuilles de manière uniforme, tout en se basant sur les générateurs de données qui peuvent influencer le parcours.

4.1 Compilateur de compilateurs LL(k)

Malgré la présence d'un compilateur de compilateurs LL(1) dans Hoa, nous n'en avions pas de LL(k) [12], tout comme dans le monde PHP. Nous en avons donc construit un en restant sur la même philosophie que PHP : simple, rapide et interprété.

4.1.1 Langage PP

Pour faciliter l'utilisation de ce compilateur de compilateurs LL(k), nous avons créé un langage reconnu par ce dernier qui est capable d'exprimer des grammaires : le langage PP. La figure 4.1 décrit la grammaire du langage PP. Les entités syntaxiques *identifier*, *digit* et *blank* représentent respectivement un identifiant, un nombre entier absolu et un blanc (des espaces ou des tabulations). Le symbole ↩ représente un retour à la ligne.

Les instructions `%skip` et `%token` servent respectivement à décrire des caractères à ne pas considérer et des lexèmes. Ensuite, nous nommons les règles suivies du symbole `:`, puis sur une nouvelle ligne indentée nous décrivons la règle :

$$\begin{aligned} &rule - name\texttt{:} \\ &\quad rule \\ &\quad \vdots \\ &\quad rule \end{aligned}$$

L'indentation peut être constituée d'au moins un espace ou tabulation, cela n'a pas d'importance.

Les règles supportent des expressions simples mais suffisantes comme :

- la disjonction, avec le symbole `|` ;
- le parenthésage, avec les symboles `(` et `)` ;
- les répétitions, avec le symbole `?` pour zéro ou une fois, le symbole `+` pour une ou plusieurs fois, le symbole `*` pour zéro ou plusieurs fois, et les symboles $\{n,m\}$ pour définir un intervalle de répétition entre n et m ;
- les captures :
  - silencieuses, avec le symbole `::`*token*`::`, pour déclarer un lexème *token* qui n'apparaîtra pas dans l'AST ;
  - persistantes, avec le symbole `<`*token*`>`, pour déclarer un lexème *token* qui apparaîtra dans l'AST.
- les appels de règle, avec le symbole *rule*`()` ;
- le nommage des nœuds dans l'AST, avec le symbole `#`*node* ; si plusieurs nommages sont rencontrés pour un même nœud, le dernier sera utilisé ; un nœud peut être une règle ;
- l'unification des lexèmes, avec le symbole `token[`n`]` qui va unifier la valeur de tous les lexèmes du même nom et avec le même $n \geq 0$ au sein d'une règle donnée.

Exemple 4.1. *Reconnaître des opérations arithmétiques*
La figure 4.2 nous montre la grammaire qui permet de reconnaître les opérations arithmétiques avec l'ordre de précédence sur les opérateurs ainsi que la forme de l'AST produit. Nous déclarons plusieurs lexèmes pour chaque opérateur et pour reconnaître les nombres (1, 2, 3 etc.) ainsi que les variables (x, y, z etc.). Nous avons deux niveaux de précédence pour les opérateurs : haute pour la multiplication et la division, basse pour l'addition et la soustraction. Nous définissons une équation, donc le symbole d'égalité doit apparaître entre les opérandes. Nous voyons dans l'AST que les nœuds `#addition`

```
%skip   s    \s+
%token  n    \d+
%token  l    [a-z]+
%token  add  \+
%token  sub  \-
%token  mul  \*
%token  div  \/
%token  eq   =

#equation:
  high() ::eq:: high()

high:
  low()
  (
    ( ::mul:: #mutiplication | ::div:: #division )
    high()
  )*

low:
  ( <n> | <l> )
  (
    ( ::add:: #addition | ::sub:: #substraction )
    high()
  )*
```

FIGURE 4.2 – Grammaire pour un langage arithmétique

et `#substraction`, par exemple, sont présents, alors que `high` et `low` ne le sont pas, car ils ne sont pas précédés par le symbole `#`.

Le langage PP propose d'autres fonctionnalités, comme les espaces de noms pour les lexèmes ou l'unification des lexèmes. Les espaces de noms sont utiles pour grouper les lexèmes entre eux et restreindre l'ensemble des lexèmes à considérer pendant une analyse. L'espace de nom est optionnel et sa valeur par défault est `default`. Ainsi, écrire : `%token name value` est strictement équivalent à `%token default:name value`. Les espaces de noms ne doivent être précisés que dans les déclarations des lexèmes, pas des règles. Pour changer d'espace de nom, nous utilisons l'opérateur `->`. Une règle complète de déclaration de lexème serait :

`%token` *namespace*`:`*name* *value* `->` *namespace*`:`*name*

L'unification des lexèmes permet d'unifier leur valeur au sein d'une même règle. Prenons l'exemple simple d'une analyse d'un langage qui propose deux catégories de guillemets : simple, avec `'`, et double, avec `"`. Le lexème sera `'|"`. Nous devons nous assurer que les formes : `"..."` et `'...'` seront reconnues comme valides, alors que `"...'` et `'..."` seront reconnues comme fausses.

Exemple 4.2. *Espace de nom et unification de lexèmes*
Si nous voulons analyser un document XML, le formalisme impose que le

```
%skip   space          \s
%token  lt             <      ->  in_tag       xml:
%token  inner          [^<]+                       tag()+

%skip   in_tag:space   \s                      #tag:
%token  in_tag:slash   /                           ::lt:: <tagname[0]> ::gt::
%token  in_tag:tagname [^>]+                       ( <inner> | tag() )*
%token  in_tag:gt      >      ->  default          ::lt:: ::slash:: ::tagname[0]:: ::gt::
```

FIGURE 4.3 – Les espaces de noms et l'unification des lexèmes

nom de la balise fermante soit le même que le nom de la balise ouvrante, autrement dit :

$$\texttt{<}\alpha\texttt{>}\ldots\texttt{</}\beta\texttt{>} \text{ avec } \alpha = \beta$$

Pour résoudre ce problème, nous allons utiliser l'unification. La figure 4.3 propose une grammaire pour reconnaitre une forme simplifiée du langage XML. Cette grammaire utilise aussi bien les espaces de noms que l'unification. Les espaces de noms sont obligatoires ici pour créer une forme de contexte avec les lexèmes. Nous avons deux contextes : hors-balise (`default`) et balise (`in_tag`). Le lexème `lt` fait passer de l'espace de nom `default` à `in_tag`, pour y revenir avec le lexème `gt`. Nous remarquons aussi l'unification sur le lexème `tagname` avec `tagname[0]` qui assure la contrainte $\alpha = \beta$. Ce serait impossible d'assurer cette contrainte sans l'unification car la valeur du lexème est totalement inconnue (`[^>]+` signifie que nous capturons tout jusqu'à rencontrer le symbole `>` exclu).

4.1.2 Fonctionnement

Le compilateur de compilateurs génère un compilateur LL(k) à la volée et nous sommes capables de l'utiliser directement. Pour cela, nous utilisons le mécanisme de création de fonctions à la volée interne à PHP pour analyser les règles, les générer, les imbriquer, gérer les erreurs, les retours en arrière etc. La création de fonction à la volée s'effectue à partir d'une chaîne de caractères représentant du code PHP. Ce ne sont pas des fonctions anonymes (qui ne pourraient pas répondre à cette problématique aussi facilement). L'objectif ici n'est pas d'avoir de hautes performances mais d'être honorablement rapide, ce qui est le cas.

À chaque opérateur proposé dans les règles de grammaire est associée une fonction qui sera générée à la volée. Par exemple, si nous avons une répétition, *i.e.* la présence des opérateurs `?`, `+`, `*` ou $\{n,m\}$, nous allons construire une fonction qui va essayer de consommer respectivement zéro ou un, un ou

```
$compiler = Hoa\Compiler\Llk::load(new Hoa\File\Read('Grammar.pp'));
$ast      = $compiler->parse('1 + x * 3  / y = 5 - z * 7');
$dump     = new Hoa\Compiler\Visitor\Dump();
echo $dump->visit($ast);
```

FIGURE 4.4 – Utilisation du compilateur de compilateurs et application d'un visiteur

plusieurs, zéro ou plusieurs, ou n à m lexèmes. Lors de l'analyse des règles, les fonctions vont se créer à la volée en suivant les opérations. Ces fonctions sont stockées dans un tableau ce qui nous aide pour imbriquer les fonctions entre elles. Nous nous servons de la nature même des fonctions pour gérer les retours en arrière causés par des erreurs. Si nous prenons le cas de nos opérateurs de répétition, ils vont essayer de consommer des lexèmes. S'ils y arrivent, l'analyse continuera, sinon la fonction s'arrêtera là et redonnera la main à la fonction parente (qui correspond à un autre opérateur).

La compilation des règles, ou la transformation en fonctions, ne s'effectue qu'une seule fois. Par la suite, le compilateur LL(k) vit en mémoire et peut être utilisé autant que nous le souhaitons.

L'analyse finale fonctionne comme ceci. Le compilateur commence par découper la donnée analysée (à savoir un langage représenté sous la forme d'une chaîne de caractères) en lexèmes en utilisant les instructions `%token`. À chaque lexème est associé sa valeur, la règle qui l'a extraite etc. Ensuite, les règles sont appliquées sur la liste des lexèmes qui sont consommés un par un. En cas d'erreurs, la lecture de cette liste se fait en sens inverse. Quand les bons lexèmes sont consommés, l'AST se construit.

4.1.3 *Abstract Syntax Tree*

Le compilateur de compilateurs LL(k) produit un arbre de syntaxe abstrait. Cet arbre contient tous les lexèmes ainsi que leurs valeurs. Nous pouvons y appliquer aisément des visiteurs pour le parcourir et y appliquer des traitements. La figure 4.4 montre comment analyser une chaîne et obtenir un AST pour lui appliquer un visiteur qui va afficher l'arbre comme nous l'avons vu dans les figures précédentes. Le résultat de cette figure est donné dans la figure 4.5.

Nous pouvons aussi seulement analyser une donnée afin de savoir si elle est reconnue par une grammaire, sans pour autant construire un AST grâce au second argument de la méthode `parse` (en donnant `false`).

La construction de l'AST se fait de manière incrémentale, *i.e.* au fur et à

```
>  #equation
>  >  #addition
>  >  >  token(n, 1)
>  >  >  #mutiplication
>  >  >  >  token(l, x)
>  >  >  >  #division
>  >  >  >  >  token(n, 3)
>  >  >  >  >  token(l, y)
>  >  #substraction
>  >  >  token(n, 5)
>  >  >  #mutiplication
>  >  >  >  token(l, z)
>  >  >  >  token(n, 7)
```

FIGURE 4.5 – AST construit pour l'expression `1+x*3/y = 5-z*7`

mesure de l'analyse de la donnée. Quand nous remontons d'une règle analysée correctement, que plus aucune ambiguïté n'existe et que le nommage d'un nœud est rencontré (opérateur `#`), nous récupérons la valeur du lexème et créons un nouveau nœud dans l'AST que nous ajoutons à son parent. Le nœud parent est toujours connu tout au long de l'analyse.

Un nœud est une classe portant trois données : identifiant (unique), valeur (soit le nom et la valeur du lexème) et la liste des enfants dans un tableau indexé. Nous trouvons les méthodes traditionnelles, comme `getId`, `getChildren`, `getChild` etc.

4.2 Lexèmes réguliers

Pour être capable d'assurer la propriété de générabilité sur toutes les grammaires, nous avons dû réaliser un compilateur de PCRE dans un premier temps, puis exploiter l'AST afin d'obtenir la propriété de générabilité dans un second temps.

Les lexèmes d'une grammaire, déclarés avec l'instruction `%token` en PP, sont exprimés à l'aide du langage d'expressions régulières PCRE [30], pour *Perl Compatible Regular Expressions*, proposé et maintenu par Hazel. Ce langage est plus riche que les expressions régulières traditionnelles, avec des assertions, des conditions, de l'internationalisation, de la localisation etc. Ce langage n'est pas le plus simple mais il est bien le plus puissant et surtout le plus répandu. En effet, il est autant utilisé par PHP que par Javascript, Perl, Python, Apache, KDE etc.

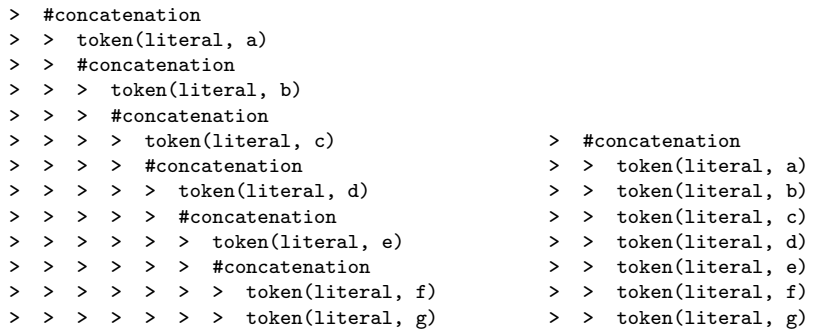

FIGURE 4.6 – Équilibre des arbres

4.2.1 Grammaire des PCRE

La grammaire des PCRE est disponible sur le site [30], disséminée dans la documentation. Elle est présente dans l'annexe A ce document et est écrite avec le langage PP. La grammaire a été optimisée pour produire un AST compact et au maximum équilibré (*i.e.* avec une profondeur quasi-équivalente pour toutes les branches).

Exemple 4.3. *Équilibre d'un arbre*
Deux arbres sont présentés dans la figure 4.6. Le premier est déséquilibré alors que le second est équilibré. Un arbre équilibré est plus facile à visualiser d'une part, mais est surtout plus rapide à construire car il nécessite moins d'allocations mémoire d'autre part. Le second arbre est également plus compact car il contient moins de nœuds, ce qui résultera sur une exécution plus rapide des traitements que nous lui appliquerons à travers des visiteurs.

4.2.2 Visiteur assurant la caractéristique de générabilité

Quand nous analysons une expression régulière avec cette grammaire, nous obtenons un AST. Pour lui faire adopter les caractéristiques fonctionnelles d'un domaine réaliste, notamment pour assurer la propriété de prédicabilité, nous utilisons les fonctions natives de PHP, comme `preg_match` qui regarde la correspondance entre une chaîne de caractères et une expression régulière. Notons que le compilateur de compilateurs LL(k) pourrait tout à fait jouer se rôle (c'est son premier objectif) mais il est plus lent que les fonctions natives de PHP ; pour la propriété de générabilité, nous utilisons un visiteur.

Le visiteur en question se base sur les générateurs de données pour, par exemple, guider son parcours ou choisir des éléments d'une classe, d'un rang

etc. Nous n'assurons pas pour l'instant l'uniformité sur le parcours, *i.e.* nous l'assurons uniquement sur les données « atomiques » et non pas sur la donnée totale générée (soit la concaténation des données atomiques). Les générateurs de données sont utilisés car tout peut se rapporter à un entier : un intervalle entre deux code-points Unicode, une quantification, une disjonction etc. Une fois que nous pouvons raisonner sur des entiers, nous utilisons le générateur de données et sa méthode `getInteger` (introduite à la section 3.3) pour générer un entier. La sémantique de cet entier est déterminée en fonction du contexte.

À chaque nœud est associé un opération ; nous énumérons les principales. Quand nous visitons un nœud `#concatenation`, nous concaténons les données. Quand nous visitons un nœud `#quantification`, nous regardons le type de la quantification : `zero_or_one`, `one_or_more` etc. Nous tirons une valeur dans l'intervalle décrit par cette quantification. Si une des bornes de cet intervalle est inconnue (par exemple avec `zero_or_more`), ce sera la valeur par défaut du générateur de données qui sera utilisée (voir la section 3.4.3). Quand nous visitons un nœud `#alternation`, nous générons un entier entre 0 et le nombre d'enfants du nœud. Cet entier sera l'index de la branche vers laquelle nous poursuivons le parcours.

Exemple 4.4. *Compiler le langage PCRE*
La figure 4.7 montre comment analyser une expression régulière en utilisant la grammaire des PCRE et comment appliquer un visiteur qui va nous permettre de générer une valeur qui correspond à l'expression régulière étudiée, ainsi qu'un des résultats possibles. Nous voyons que le résultat `abxyrssssst` correspond bien à l'expression régulière `ab(cd)?xy(z{2,4}|rs*t?)`. Lors d'une nouvelle exécution, nous pourrions avoir `abxy` ou encore `abcdxyrsssssssst`, toutes des données correspondant notre expression régulière.
Nous remarquons également que nous utilisons un générateur de données aléatoire qui permet de déterminer par exemple quelle branche choisir lorsque nous rencontrons un nœud `#alternation` ou qui nous permet de déterminer la quantification pour l'expression `z{2,4}`. Dans le premier cas, nous demandons au générateur de données de nous fournir un entier entre 0 et le nombre d'enfants que contient le nœud `#alternation`. Cet entier va être l'index de la branche à choisir pour le parcours. Le parcours est donc très dépendant du générateur que nous utilisons. Enfin, pour le second cas, avec `z{2,4}`, nous allons générer un entier i entre 2 et 4 et visiter le motif i-fois (ici, c'est simplement le litéral `z`).

```
$compiler = Hoa\Compiler\Llk::load(
    new Hoa\File\Read(
        'hoa://Library/Regex/Grammar.pp'
    )
);
$regex   = 'ab(cd)?xy(z{2,4}|rs*t?)';
$ast     = $compiler->parse($regex);

$dump    = new Hoa\Compiler\Visitor\Dump();
$realdom = new Hoa\Regex\Visitor\Realdom(
    new Hoa\Test\Sampler\Random(
        array('integer.max' => 7)
    )
);

echo 'Regex: ' . $regex . "\n" .
     $dump->visit($ast) . "\n" .
     'Example: ' . $realdom->visit($ast) .
     "\n";
```

```
Regex: ab(cd)?xy(z{2,4}|rs*t?)
>  #expression
>  >  #concatenation
>  >  >  token(literal, a)
>  >  >  token(literal, b)
>  >  >  #quantification
>  >  >  >  #capturing
>  >  >  >  >  #concatenation
>  >  >  >  >  >  token(literal, c)
>  >  >  >  >  >  token(literal, d)
>  >  >  >  token(zero_or_one, ?)
>  >  >  token(literal, x)
>  >  >  token(literal, y)
>  >  >  #capturing
>  >  >  >  #alternation
>  >  >  >  >  #quantification
>  >  >  >  >  >  token(literal, z)
>  >  >  >  >  >  token(n_to_m, {2,4})
>  >  >  >  >  #concatenation
>  >  >  >  >  >  token(literal, r)
>  >  >  >  >  >  #quantification
>  >  >  >  >  >  >  token(literal, s)
>  >  >  >  >  >  >  token(zero_or_more, *)
>  >  >  >  >  >  #quantification
>  >  >  >  >  >  >  token(literal, t)
>  >  >  >  >  >  >  token(zero_or_one, ?)
Example: abxyrsssst
```

FIGURE 4.7 – Utilisation du visiteur domaine réaliste des PCRE

4.2.3 Domaine réaliste `regex`

Nous avons à présent tout ce qui nécessaire pour réaliser un domaine réaliste `regex`. En effet, nous avons les fonctions `preg_*()` de PHP, comme `preg_match` qui permet de vérifier si une chaîne de caractères correspond bien à une expression régulière, soit la propriété de prédicabilité. Et nous avons également un visiteur qui, appliqué sur l'AST résultant de l'analyse d'une expression régulière grâce à notre compilateur de compilateurs LL(k), permet d'assurer la propriété de générabilité en produisant une chaîne de caractères qui correspond bien à l'expression régulière considérée. Alors, le domaine réaliste `regex` se résume à un simple assemblage. Ainsi, le domaine réaliste `regex` se présente de la façon suivante : `regex('`*regex*`')` où *regex* est une expression régulière exprimée en PCRE.

À l'issue de la période de rédaction du mémoire, tout le langage PCRE est compris par le compilateur mais tout n'est pas encore générable. Nous avons couvert ce qui était nécessaire pour le projet. Le reste sera soit fait par nous dans un futur proche, soit par Hoa et sa communauté.

4.3 Génération d'AST contraints par la taille

Dans les sections précédentes, nous avons vu comment produire un AST à partir d'une donnée et comment faire l'opération inverse. Quand nous faisons l'inverse, l'AST généré n'est soumis à aucune contrainte sur les données qu'il est capable de générer. Le seul élément constant avec les deux approches est l'utilisation de la grammaire. En effet, dans le premier cas nous avons une donnée qui est analysée en fonction d'une grammaire et qui produit un AST. Dans l'autre cas, nous aurions la grammaire qui va nous permettre de déduire un AST, qui pourra nous servir à générer une donnée susceptible de produire un tel AST.

L'objectif ici est de contraindre la forme de l'AST afin de pouvoir générer des données d'une taille connue. Par exemple, « nous voulons générer une équation avec 8 opérateurs » est une requête tout à fait acceptable. Comme nous l'avons vu avec la figure 4.2, nous avons déjà la grammaire pour caractériser une équation et nous avons déjà étudié la forme de l'AST.

Nous allons présenter l'approche choisie et comment générer de tels AST.

4.3.1 Approche par le dénombrement

Il existe plusieurs approches pour générer un arbre (ou un automate) à partir d'une grammaire, nous trouverons beaucoup de références dans le livre de Flajolet et Sedgewick [21]. Plus spécifiquement, il existe deux grandes approches pour assurer une certaine *uniformité* dans la génération des AST : les algorithmes de Glushkov [42] ou les algorithmes par dénombrement [22].

Nous avons choisi d'utiliser le dénombrement car l'implémentation de ces algorithmes est plus facile dans notre contexte et permet de gérer des équivalences [31] (comme l'idempotence, la commutativité et l'associativité) dans le cas où nous introduirions de nouveaux opérateurs.

Vulgairement, le principe est le suivant : nous allons dénombrer le nombre de nœuds qui vont constituer l'arbre en fonction de la taille de la donnée souhaitée. Puis, nous allons parcourir cet arbre de manière uniforme grâce à des probabilités qui tiendront compte du nombre de nœuds déjà parcourus et des nœuds restant à parcourir afin de générer une donnée de la taille souhaitée.

Méta-AST Il devient évident que nous devons alors parcourir la grammaire pour être capable de générer un AST. Pour parcourir la grammaire, nous allons générer son propre AST. Dans ce cas, pour éviter toute ambiguïté,

```
$compiler = Hoa\Compiler\Llk::load(
    new Hoa\File\Read(
        'hoa://Library/Compiler/Llk.pp'
    )
);
$rule = 'low()' .
        '(' .
        '  ( ::mul:: #multiplication' .
        '  | ::div:: #division' .
        '  )' .
        '  high()' .
        ')*';
$dump = new Hoa\Compiler\Visitor\Dump();
$metaAst = $compiler->parse($rule);

echo 'Rule: ' . $rule . "\n" .
     $dump->visit($metaAst) . "\n" .
```

```
> #rule
> > #concatenation
> > > #named
> > > > token(token, low)
> > > #quantification
> > > > #capturing
> > > > > #concatenation
> > > > > > #capturing
> > > > > > > #alternation
> > > > > > > > #skipped
> > > > > > > > > token(token, mul)
> > > > > > > > > token(node,
> > > > > > > > >     #multiplication)
> > > > > > > > #skipped
> > > > > > > > > token(token, div)
> > > > > > > > > token(node, #division)
> > > > > > #named
> > > > > > > token(token, high)
> > > > token(zero_or_more, *)
```

FIGURE 4.8 – Utilisation du compilateur de compilateurs LL(k) au niveau méta

nous parlerons de niveau méta, *i.e.* la grammaire de la grammaire est appelée la *méta-grammaire* et l'AST de la grammaire est appelé un *méta-AST*. Nous saurons que si nous parlons d'un AST, ce sera pour désigner le résultat de l'analyse d'une donnée par rapport à une grammaire, et que si nous parlons de méta-AST, nous parlons de l'AST d'une grammaire (qui est alors la donnée analysée).
Pour être capable d'appliquer les algorithmes de dénombrement, nous devons parcourir le méta-AST. La figure 4.8 nous montre comment obtenir un méta-AST en utilisant les constructions déjà bien connues. Mais nous utilisons la méta-grammaire `hoa://Library/Compiler/Llk.pp` au lieu d'une grammaire et nous analysons la règle `high` de la grammaire présentée dans la figure 4.2. Le méta-AST produit est présenté à côté. Nous remarquons qu'il est facile de passer au niveau méta car nous avons la méta-grammaire exprimée en langage PP prête à servir.

Parcours uniforme Maintenant que nous avons notre méta-AST, il faut décider comment le parcourir. La figure 4.9 montre le problème avec un parcours isotrope : l'uniformité concerne les enfants du nœud parcouru mais elle ne concerne pas les feuilles, ce que nous désirons. En effet, la probabilité de parcourir un fils plutôt qu'un autre est de $\frac{1}{n}$ si nous avons n fils. Typiquement, le parcours de l'AST des expressions régulières présentés dans la section 4.2.2 est isotrope. Nous devons considérer l'ensemble des nœuds et des chemins pour obtenir l'uniformité. C'est là que des récents travaux, comme GenRGenS par Ponty et al. [47] ou SEED par Héam et Nicaud [31], nous ont

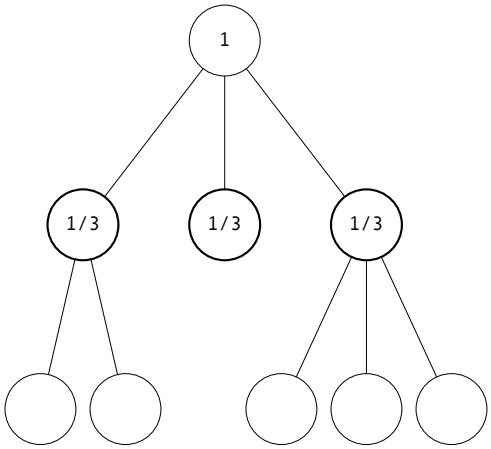

FIGURE 4.9 – Probabilité pour un tirage isotrope

été utiles. Nous nous sommes basés sur les travaux de Héam et Nicaud.
La figure 4.10 reprend la figure 4.9 mais avec une probabilité uniforme pour chaque nœud. Pour illustrer, nous sommes positionnés sur la racine comme nœud courant. Chacun de ses enfants est capable de produire respectivement 2, 0 et 3 sous-enfants (la descendance est indicée à gauche par rapport au nœud). La racine peut donc produire 5 sous-enfants auxquels nous ajoutons 3 enfants, soit une descendance totale de 8 enfants.
Pour calculer la probabilité $P(i)$ qu'un enfant soit choisi plutôt qu'un autre, nous pouvons vulgariser que :

$$P(i) = \frac{C(i) + 1}{n + \sum_{e=1}^{n} C(e)}$$

où n représente le nombre d'enfants directs du nœud courant, $i \in [1 \dots n]$ un enfant direct du nœud courant et $C(i)$ la fonction qui calcule le nombre de descendants pour l'enfant direct i. Ainsi, si nous revenons à la figure 4.10, la probabilité que le premier enfant soit choisi est de $\frac{3}{8}$, pour le deuxième enfant $\frac{1}{8}$ et pour le troisième enfant $\frac{4}{8}$. En utilisant ces probabilités, nous sommes assurés d'avoir un parcours qui soit uniforme, *i.e.* que les probabilités d'atteindre les feuilles soient identiques.
Deux phases de calcul se dégagent alors : une phase de calcul de descendance et une phase de calcul de probabilités pour chaque nœud. Nous remarquons que ces deux phases peuvent s'effectuer avec un seul parcours en largeur, res-

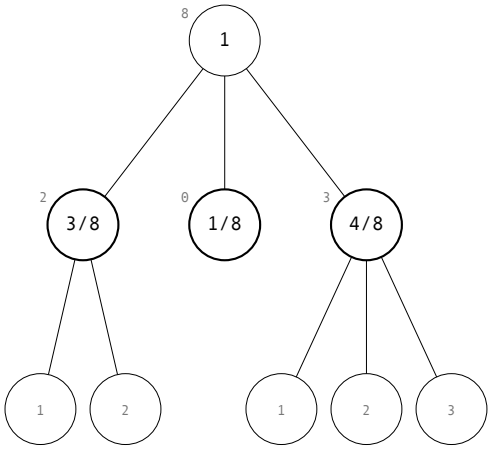

FIGURE 4.10 – Probabilité pour un tirage uniforme

pectivement en descendant et en remontant. La complexité de cette opération est de $\mathcal{O}(n)$ et elle n'est à réaliser qu'une seule fois.

Toutefois, avoir un parcours uniforme ne nous aide pas pour générer un AST capable de produire une donnée d'une taille connue. Ça ne nous est utile que pour pressentir le type de parcours.

4.3.2 Algorithme de génération d'AST

Nous savons faire un parcours uniforme dans un arbre. À présent, nous voulons générer un arbre qui produira une donnée d'une taille prédéfinie. Pour recalculer nos probabilités de parcours, nous devons déterminer la forme de notre arbre, *i.e.* quelles seront les règles à utiliser pour construire l'AST.

Dénombrer les lexèmes Pour calculer la taille de notre AST, nous devons dénombrer les lexèmes qui peuvent être produits. Pour chaque règle de la grammaire, nous comptons le nombre de lexèmes qu'elle peut produire sans oublier de propager le dénombrement aux règles imbriquées. L'idée principale est de savoir combien une règle peut produire de lexèmes, il faut donc compter toutes les combinaisons possibles.

Exemple 4.5. *Dénombrement sur des opérateurs unaires et binaires*
Nous allons considérer deux opérateurs, l'un est unaire, l'autre binaire, par exemple $-$, comme pour -1, et $\times$ comme pour 2×3. Une grammaire pourrait être celle proposée dans la figure 4.11. Quelques exemples intuitifs de formules

```
%skip  space           \s+
%token a               a
%token b               b
%token minus           -
%token multiplication  \*

multiply:
    number() ( ::multiplication:: #multiplication multiply() )?

number:
    ( ::minus:: #minus )? ( <a> | <b> )
```

FIGURE 4.11 – Grammaire $G : a \mid b \mid -G \mid G \times G$ traduite en langage PP

	$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = \ldots$
`number`(n)	2	2	2	2	...
`multiply`(n)	0	0	4	8	...

FIGURE 4.12 – Nombre de données de taille n que chaque règle de la figure 4.11 peut produire

que nous pourrions obtenir pour différentes tailles :

- pour une taille de 1, nous pourrions avoir a et b, soit 2 ;
- pour une taille de 2, nous pourrions avoir $-a$ et $-b$, soit 2 ;
- pour une taille de 3, nous pourrions avoir $--a$ et $--b$ mais aussi $a \times a$, $a \times b$, $b \times a$ et $b \times b$, soit 6 ;
- pour une taille de 4, nous pourrions avoir $---a$, $---b$, $-a \times a$, $a \times -a$, $-a \times b$, $a \times -b$, $-b \times a$, $b \times -a$, $-b \times b$ et $b \times -b$, soit 10.

Nous comprenons qu'il faut dénombrer le nombre de lexèmes pour chaque règle, ici pour `multiply` et `number`. La figure 4.12 résume les possibilités pour différentes tailles n. Nous utilisons la notation $R(n)$ pour dire que nous dénombrons la règle R pour une taille n. Ainsi, nous pouvons lire : pour produire une donnée de taille $n = 4$, la règle `number(4)` pourra produire 2 données et `multiply(4)` pourra produire 8 données.
Le reste du calcul se fait de manière récursive.

Pour chaque règle et chaque opérateur, il faut être capable de trouver une règle de récursivité pour dénombrer les lexèmes. Héam et Nicaud en proposent pour des opérateurs unaires et binaires à travers 11 équations (qui permettent de gérer des équivalences).
Dans le langage PP, nous introduisons des opérateurs de répétitions (`?`, `+` et `*`). Dénombrer les lexèmes appliqués à de tels opérateurs n'est pas précisé. Notons que trouver une solution pour l'opérateur `*` est suffisante car les autres opérateurs `?` et `+` sont des sous-opérateurs de `*`. Nous expliquons comment

dénombrer de tels opérateurs à travers un exemple.

Exemple 4.6. *Dénombrer des quantifications*
Nous expliquons comment appliquer l'algorithme de dénombrement lorsque nous rencontrons des opérateurs de quantification, comme $*$. En effet, cette quantification signifie 0 ou plus. Dans le pire des cas, si nous voulons générer une donnée de taille n, alors la quantification signifirait entre 0 et n.
Si nous prenons l'exemple de la grammaire suivante :

$$\begin{array}{rcl} plus & : & mult\ (\ \texttt{x}\ mult)^* \\ mult & : & nb\ (\ *\ nb\ )^* \\ nb & : & a \mid b \end{array}$$

Si nous dénombrons $mult(1)$, nous obtenons 2 données, tout comme pour $nb(1)$. Nous avons aussi $mult(2) = 0$, $nb(2) = nb(3) = 0$. D'une manière plus générale, nous avons :

$$\begin{array}{rl} plus(4) = & mult(4) + \displaystyle\sum_{i_1+i_2=4-1} mult(i_1) \times mult(i_2) + \\ & \displaystyle\sum_{i_1+i_2+i_3=4-1} mult(i_1) \times mult(i_2) \times mult(i_3) \\ \text{mult}(4) & nb(4) + \displaystyle\sum_{i_1+i_2=4-1} nb(i_1) \times nb(i_2) + \\ & \displaystyle\sum_{i_1+i_2+i_3=4-1} nb(i_1) \times nb(i_2) \times nb(i_3) \end{array}$$

Dans les règles *plus* et *mult*, nous remarquons que nous faisons $4-1$ car elles peuvent produire chacune 1 lexème. Nous faisons la taille moins le nombre de lexèmes qui peut être produit.
Si nous déplions $plus(4)$, nous obtenons :

$$plus(4) = 0 + \underbrace{mult(1) \times mult(2)}_{0} + \underbrace{mult(2) \times mult(1)}_{0} + \underbrace{mult(1)^3}_{8}$$

Nous comprenons que cela revient à dénombrer toutes les combinaisons possibles.

Générer les AST La génération de l'AST se résume au choix du parcours. Dans la section 4.3.1, nous voyons comment parcourir un arbre de manière uniforme. Le même principe est appliqué ici sauf qu'il faut considérer le nombre de lexèmes qu'une règle (donc qu'un nœud) peut produire au lieu de considérer sa réelle descendance. Pour calculer le nombre de lexème, si nous

avons nos règles récursives, alors nous pouvons calculer le $i + 1$ facilement.

À l'issue de la période de rédaction du mémoire, la génération aléatoire uniforme d'AST n'est pas encore implémentée mais ce travail sera implémenté prochainement.

Chapitre 5

Expérimentation

Sommaire

Dans ce chapitre, nous allons présenter une expérimentation que nous avons réalisé sur des projets réalisés par sept étudiants en deuxième année de Licence. Nous allons tout d'abord installer Hoa, puis voir comment utiliser Praspel et enfin réaliser l'expérience avec les étudiants.

5.1 Installation

Dans cette section, nous décrivons comment installer Hoa ainsi que l'expérimentation.

5.1.1 Installer Hoa

Pour une installation sur tous les systèmes à l'aide de plusieurs formats, il faudra regarder le manuel d'apprentissage de Hoa, au chapitre « Installa-

tion ». Nous allons installer Hoa en utilisant le gestionnaire de dépôt choisi, à savoir Mercurial. L'installation se résume à une seule commande :

```
$ hg clone http://hg.hoa-project.net/Central /usr/local/lib/hoa
```

Hoa nécessite PHP 5.3 au minimum. Pour tester que Hoa est bien installé et que PHP fonctionne bien, une autre ligne de commande est nécessaire :

```
$ php -r "require '/usr/local/lib/hoa/Core/Core.php';
          var_dump(HOA);"
```

Si le résultat est `bool(true)` alors c'est que tout est correctement installé et prêt à l'emploi.

Nous rappelons que Praspel et les domaines réalistes sont développés dans Hoa. Ils y sont présents depuis la version 0.5.5b. Nous conseillons toutefois la branche 1.0.0b pour bénéficier des évolutions introduites durant l'année de Master 2.

5.1.2 Installer l'expérimentation

L'expérimentation est une archive à télécharger à l'adresse `http://download.hoa-project.net/Laboratory/MasterThesis/` et à extraire n'importe où. Dans cette archive, nous trouvons un dossier `Data` et un dossier `SUT`. Pour comprendre à quoi correspond le dossier `Data`, il faudra regarder le manuel d'apprentissage de Hoa, au chapitre « Aspect framework ». Le dossier `SUT` quant à lui contient les projets des étudiants.

Hoa est un ensemble de bibliothèques mais propose un aspect framework. Les deux sont fortement découplés. Précédemment, nous avons installé les bibliothèques avec quelques outils. Mais le dossier `Data` surcharge ces outils. Une seule opération est nécessaire pour pouvoir les utiliser : leur indiquer où se trouve Hoa. Pour cela, une seule ligne est nécessaire :

```
$ Data/Bin/whereishoa
```

Pour tester que tout fonctionne bien, il suffit de lancer l'écran d'accueil des commandes :

```
$ Data/Bin/myapp
// Homescreen.
```

Dans la suite de l'expérimentation, nous considérons que `Data/Bin/myapp` est placé dans le `$PATH` afin qu'il soit accessible partout. Ce n'est pas une obligation mais cela facilite la lecture des exemples.

5.2 Prise en main de l'outil Praspel

L'écran d'accueil affiché par `myapp` nous propose des groupes de commandes. Nous trouvons le groupe `test:*` avec les sous-commandes :

- `test:praspel` pour utiliser Praspel en mode interactif ;
- `test:initialize` pour initialiser une nouvelle révision dans le dépôt de tests ;
- `test:remove` pour supprimer une révision dans le dépôt de tests ;
- `test:run` pour exécuter des tests, *i.e.* générer des données et vérifier le contrat pour un fichier, une classe et une méthode donnée, le tout pour une révision du dépôt donnée.

5.2.1 Praspel en mode interactif

Le mode interactif est très pratique pour tester des domaines réalistes, tester des générateurs, mieux comprendre Praspel etc. Pour le démarrer, nous ferons :

```
$ myapp test:praspel
Usage:
    h[elp]         to print this help;
    <praspel code> to interprete a Praspel code;
    c[ode]         to print current interpreted Praspel code;
    [r[esample]]   to get a new value;
    d[ebug]        to print informations about debug;
    q[uit]         to quit.


> _
```

Une fois le mode interactif démarré, nous pouvons exécuter du Praspel :

```
> integer()
int(7179564725080766464)

> boundinteger(7, 42)
int(15)

> resample
int(31)

> array([to string(boundinteger(3, 6))], 3)
array(3) {
  [0]=>
  string(4) "|m~!"
  [1]=>
  string(3) "s[S"
```

```
  [2]=>
  string(5) "8fD8^"
}
```

Il est possible d'utiliser le mode `debug` et d'utiliser la commande `php` pour afficher le code PHP associé au code Praspel. Le code PHP affiché représente la création du modèle objet de Praspel :

```
> debug
Usage:
    h[elp]    to print this help;
    ph[p]     to print PHP code;
    pr[aspel] to print Praspel code;
    q[uit]    to quit.


debug> php
$contract  = new Hoa\Test\Praspel\Contract(...);
$contract
    ->clause('requires')
    ->variable('i')
        ->belongsTo('array')
            ->withArray()
                ->from()
                ->to()
                    ->belongsTo('string')
                        ->withDomain('boundinteger')
                            ->with(3)
                            ->_comma
                            ->with(6)
                            ->_ok()
                        ->_ok()
                    ->end()
            ->_comma
            ->with(3)
            ->_ok()
;
```

Cet interpréteur interactif est très utile pour apprendre à utiliser Praspel et à développer dans Praspel.

5.2.2 Compiler et exécuter les tests

Pour instrumenter le code, *i.e.* le compiler, nous utiliserons la commande `test:initialize` :

```
$ myapp test:initialize SUT/
Initializing a new test revision in the repository:
* incubator from SUT/.                                                [ok]
```

```
* instrumented code.                                                    [ok]

Repository root:
hoa://Data/Variable/Test/Repository/20110717220751/
```

Pour exécuter un test, nous utiliserons la commande `test:run` :

```
$ myapp test:run --help
Usage   : test:run <options>
Options :
 -r, --revision=  : Revision of the repository tests:
                  :     [revision name] for a specified
                  :                     revision;
                  :     HEAD            for the latest
                  :                     revision.
 -f, --file=      : File to test in the repository.
 -c, --class=     : Class to test in the file.
 -m, --method=    : Method to test in the class.
 -i, --iteration= : Number of iterations.
 -s, --sampler=   : Which sampler to use.
 -h, --help       : This help.
 -?, --help       : This help.
```

Sur toutes les commandes, nous pourrons trouver de l'aide en utilisant les options `-h`, `--help` ou `-?`. Si en plus nous utilisons Zsh, nous trouverons une auto-complémentation avancée des commandes et de leurs options. Pour plus d'informations, voir le manuel d'apprentissage de Hoa, chapitre « Outils autour de Hoa ».

5.3 Expérience

Nous avons décidé d'utiliser Praspel pour valider une application Web. Nous avons ciblé des étudiants qui étaient en cours de « Languages Web ». Le sujet des projets proposé aux étudiants était de générer du code HTML. Plus particulièrement, à partir d'une date donnée, ils devaient générer une balise `<select>` pour les jours, une pour les mois et une pour les années. La spécification était la suivante :

— le code produit est de l'HTML ;
— le code produit contient trois balises `select` ;
— les caractères spéciaux sont encodés en entités XML (`&`*ent*`;`) ;
— le jour doit appartenir à l'intervalle 1 à 31, le mois de 1 à 12 et l'année de 1911 à 2011 ;
— exactement une seule option doit être sélectionnée par défaut.

```
/**
 * @requires n: string(boundinteger(1, 10)) and
 *           d: boundinteger(1, 31) and
 *           m: boundinteger(1, 12) and
 *           y: boundinteger(1911, 2011);
 * @ensures  \result;
 */
public function test ( $n, $d, $m, $y ) { ... }
```

FIGURE 5.1 – Spécification utilisée pour l'expérimentation

5.3.1 Une première approche simple

Nous avons spécifié les entrées avec les domaines réalistes `string` et `boundinteger`, et le résultat avec le domaine réaliste `xmlcode`, spécialement écrit pour l'occasion. La figure 5.1 représente le contrat utilisé.

Chaque fichier dans `Data`, *i.e.* `Student*.php`, peut être testé indépendamment. Nous devons d'abord initialiser une révision dans le dépôt, qui rappelons-le va compiler Praspel :

```
$ myapp test:initialize SUT/
```

Par la suite, chaque fichier peut être testé en utilisant cette commande :

```
$ myapp test:run -f Student1.php -c Student1 -m test
```

Si nous regardons les options de plus près, nous remarquons que l'option `-f` spécifie le fichier à tester, l'option `-c` la classe et `-m` la méthode.

Nous pouvons ajouter l'option `-i` pour préciser un certain nombre d'itérations, *i.e.* un nombre de fois où le mécanisme de test sera répété. Nous pouvons exécuter les tests sur tous les fichiers et toutes les classes pour comprendre le fonctionnement. L'exécution ne montrera aucune erreur par rapport à la spécification exprimée en Praspel, *i.e.* le code HTML résultant est bien formé.

5.3.2 Une seconde approche plus sophistiquée

Pour l'instant, nous avons seulement vérifié la conformité du résultat par rapport à la syntaxe XML, mais nous pouvons avoir une vérification plus sophistiquée en vérifiant que le code généré contient bien trois balises `select` avec une option sélectionnée à l'intérieur.

Pour atteindre cet objectif, nous avons deux choix possibles. Le premier consiste à créer un domaine réaliste dédié à cette tâche. Mais comme ce

```
/**
 * @requires nom:   string(boundinteger(1, 10)) and
 *           jour:  boundinteger(1, 31) and
 *           mois:  boundinteger(1, 12) and
 *           annee: boundinteger(1911, 2011);
 */
public function test1 ( $nom, $jour, $mois, $annee ) {

    $this->testOne(new Student1(), $nom, $jour, $mois, $annee);
}
```

FIGURE 5.2 – Méthode « adapteur » de test

domaine réaliste serait trop spécifique et que nous préférons avoir des domaines réalistes réutilisables (donc abstraits), nous allons utiliser Praspel pour spécifier une méthode « adapteur » de test qui aura la charge d'appeler la méthode sous test et de faire des vérifications particulières (que nous aurions fait dans notre domaine réaliste spécifique).

Le fichier `StudentsCode.php` contient les adapteurs en question pour tester nos méthodes. Une première méthode `test`x est annotée avec les préconditions présentées dans la figure 5.2 pour générer les données de tests.

La méthode `test`x appelle une seconde méthode chargée d'analyser le code HTML et retourne 1 si tout est correct, 0 sinon. Pour que le test soit un succès, la post-condition spécifie que le résultat doit être égal à 1, comme le montre la figure 5.3. Maintenant, la vérification assure qu'il y a trois balises `select` bien formées avec une option sélectionnée, *i.e.* avec l'attribut booléen `selected` présent.

Pour exécuter ces tests, nous appelons la méthode « adapteur » :

```
$ myapp test:run -f StudentsCode.php -c StudentsCode -m test1
```

Pour les sept projets, nous avons détecté les erreurs suivantes :
- élèves 1 et 2 : aucune erreur ;
- élèves 3 et 6 : une erreur, il écrivait `selected="yes"` au lieu de `selected` (ou `selected="selected"`) ;
- élève 4 : une erreur, le code HTML contient des caractères non échappés (`é` au lieu de `&eacute;`) ;
- élève 5 : aucune erreur.

Enfin, nous avons modifié la dernière ligne de la figure 5.3 comme le montre la figure 5.4, afin de vérifier que seulement une option parmi les balises `select` a bien été sélectionnée.

```
/**
 * @ensures \result: 1;
 */
public function testOne ( $s, $nom, $jour, $mois, $annee ) {

    try {
        $r = $this->htmlParser->parse(
            $s->test($nom, $jour, $mois, $annee),
            '3select',
            true
        );
    }
    catch ( \Exception $e ) { return 0; }

    return !isset($r) ? 0 : 1;
}
```

FIGURE 5.3 – Méthode « adapteur » de test plus spécifique

```
return null === $r ? 0 : $this->checkOneOptionSelected($r);
```

FIGURE 5.4 – Méthode « adapteur » de test plus spécifique modifiée

Avec de nouvelles exécutions sur les projets n'ayant montré aucune erreur, voici les résultats :

— élèves 1 et 2 : toujours aucune erreur ;
— élève 5 : une erreur, aucun mois par défaut n'était sélectionné.

Cette petite expérimentation a montré qu'il était possible d'utiliser Praspel aussi bien pour du test unitaire que pour du test par adapteurs, *i.e.* de piloter une couche intermédiaire entre notre code et une spécification très particulière, voire un scénario !

Troisième partie

Bilan

Chapitre 6

Conclusion et perspectives

Sommaire

Nous nous sommes intéressés à proposer un mécanisme permettant de représenter une donnée de la manière la plus réaliste possible et d'inscrire ce mécanisme dans la programmation par contrat afin d'automatiser la génération de tests unitaires. Les différents sujets abordés dans ce mémoire se sont inscrits dans le cadre d'une méthodologie originale pour la génération de données de tests. Cette méthodologie considère le test comme le moyen principal pour la validation d'un programme et doit donc être accessible à tout le monde facilement à travers l'utilisation d'un langage simple. Les sujets présentés dans ce mémoire se sont articulés autour de deux points principaux, présentés en tant que contributions majeures.

6.1 Contributions majeures

Les deux contributions majeures de ce mémoire s'articulent autour de la programmation par contrat. Nous les résumons dans les sections suivantes.

6.1.1 Domaines réalistes

Les domaines réalistes permettent de représenter une donnée la plus réaliste possible pour un contexte précis. Les domaines réalistes sont accompagnés de mécanismes qui permettent la génération d'une nouvelle donnée valide et la validation d'une donnée par rapport à sa définition. Les domaines réalistes peuvent se coupler à plusieurs générateurs de données afin d'influencer la forme des données qui seront générées. Actuellement, seul un générateur aléatoire uniforme est présenté. Toutefois, le mécanisme des générateurs de données a été conçu pour être extensible.

Nous avons proposé d'introduire le principe du *Grammar-based Testing* dans un domaine réaliste à titre de *proof-of-concept*. Ce travail n'est pas terminé mais semble prometteur quant aux capacités de génération et de validation de données qu'il propose. Ce travail nous a conduit à la création d'un compilateur de compilateurs LL(k) accompagné d'un langage associé : le langage PP. Il nous a également conduit à l'élaboration d'algorithmes de parcours uniformes d'arbres afin de générer des données d'une taille connue, mais aussi à un compilateur d'expressions régulières. Un grand nombre d'outils ont été développés tout en restant découplés les uns des autres, ils peuvent donc être réutilisés individuellement pour d'autres contextes d'applications.

6.1.2 Praspel

Praspel est un langage d'annotations et de spécifications respectant le paradigme de la programmation par contrat. En effet, Praspel supporte les pré-conditions, les post-conditions, les invariants, les exceptions et les comportements. Le langage est simple, intuitif et est basé sur une syntaxe épurée. Praspel propose un mécanisme d'expression de contrats et se repose entièrement sur les domaines réalistes pour l'interprétation des contrats. En effet, c'est à travers Praspel que nous pourrons utiliser les domaines réalistes : les construire, les paramétrer, les combiner etc. De ce fait, les domaines réalistes doivent être suffisamment abstraits pour être combinés avec n'importe quel autre domaine réaliste.

Nous avons amélioré la manière dont nous pouvions développer les domaines réalistes afin que de nouveaux développeurs puissent en écrire facilement et ainsi expérimenter de nouveaux générateurs ou de nouvelles données comme pour le *Grammar-based Testing*, notamment en reconsidérant le mécanisme de gestions des arguments des domaines réalistes.

Nous avons utilisé Praspel pour mener une expérimentation sur des travaux d'étudiants en deuxième année de Licence. L'utilisation des outils autour de Praspel et de Praspel même nous ont permis de déceler des er-

reurs à différents niveaux : syntaxique et sémantique. L'expérimentation consistait à valider une génération XML selon certaines contraintes. Cette expérimentation nous a amené à considérer Praspel dans un autre contexte que du test unitaire.

6.2 Perspectives techniques

Plusieurs aspects peuvent être améliorés dans les travaux qui ont été menés.

La gestion des comportements, avec la clause `@behavior`, n'est pas encore bien supportée dans Praspel et peut donc être largement améliorée. Le domaine réaliste relatif au *Grammar-based Testing* n'est pas terminé et il sera intéressant de pouvoir élaborer une expérimentation. Le support du domaine réaliste `class` reste aussi basique. Nous pouvons peut-être étendre Praspel de clauses pour faciliter la détection de patrons de conceptions afin de générer plus facilement tout type de classes.

Concernant l'outil Praspel, nous pouvons améliorer les rapports de tests en proposant des formats répandus dans l'industrie. Nous pouvons également déployer une base de données pour mémoriser les tests et ainsi rejouer des tests plus facilement. Enfin, nous pouvons ajouter de nouvelles informations à celles collectées lors de l'exécution des tests pour ajouter un aspect *profilage* à Praspel. En effet, si l'outil propose plus de fonctionnalités, il sera plus facilement adopté par l'industrie et nous aurons de ce fait plus de retours.

6.3 Perspectives scientifiques

Les perspectives scientifiques sont nombreuses. Praspel et les domaines réalistes avaient la vocation initiale de proposer une base solide pour effectuer de la génération de tests unitaires ainsi que des données de tests. Cette base a été conçue dans l'optique d'implémenter et d'expérimenter plusieurs concepts scientifiques relatifs à la sécurité et à la sûreté du logiciel. Nous avons à présent le choix.

6.3.1 Étendre les prédicats

Les prédicats, à travers la construction `\pred`, ne sont actuellement disponibles qu'en post-condition. Si nous les étendons aux pré-conditions et invariants, il serait confortable de trouver un moyen de réduire le rejet qu'ils peuvent engendrer. Cela nécessite des travaux approfondies qui peuvent faire l'objet d'une thèse à eux seuls.

6.3.2 Efficacité de la génération de données de tests

Les domaines réalistes sont découplés des générateurs de données. Actuellement, un seul générateur est répertorié : le générateur aléatoire.

Search-based Testing Avant que le sujet du mémoire ne s'oriente vers le *Grammar-based Testing* appliqué aux domaines réalistes, nous envisagions le *Search-based Testing* [39] en tant que nouveau générateur de données. Cette technique utilise des méta-heuristiques de recherche basées sur la programmation génétique pour résoudre les contraintes de générations de données. Plus précisémment, elle se base sur des fonctions de *fitness* (ou fonctions objectifs) positionnées sur les points de choix (conditions) de notre unité sous test et évalue la distance de la donnée exécutée par rapport à notre objectif.

Exemple 6.1. *Le* Search-based Testing *par l'exemple*
Si nous rencontrons un prédicat `x == y`, comme nous l'avions vu avec la figure 2.4 page 18, il est improbable que ce prédicat soit vrai. Avec le *Search-based Testing*, nous définissons une fonction objectif, ici : $abs(\mathtt{x} - \mathtt{y})$; plus la valeur tend vers 0, plus nos données évoluent dans la bonne direction et nos chances d'atteindre l'objectif sont plus importantes.

Autant les fonctions de *fitness* que les métaheuristiques utilisées pour faire évoluer nos données sont importantes dans ce contexte. L'idée principale serait d'équiper les domaines réalistes de caractéristiques fonctionnelles suffisantes pour faire du *Search-based Testing*. Cette approche est généraliste et apporte des résultats très satisfaisants mais les fonctions de *fitness* se bornent actuellement aux nombres. Il n'est pas précisé comment réagir face à d'autres types de données [2], en particulier face à des chaînes de caractères, des tableaux, des graphes, des objets etc.

Fuzzing Le *fuzz testing* [27] peut également être intégré dans Praspel en tant que nouveau générateur de données. Le *fuzz testing* repose sur de la mutation aléatoire de données valides et reste une technique répandue pour trouver des erreurs à faibles coûts. Si nous ajoutons une mémoire à l'outil Praspel, le générateur de données *fuzz* se réduirait à sélectionner une donnée valide pour un test ou d'en générer une aléatoirement s'il n'en existe pas, et de la muter aléatoirement (*i.e.* modifier un bit à un endroit). Il serait notamment intéressant d'étudier la pertinence des codes de Gray [29] pour appliquer des mutations.

Générateurs de générateurs D'autres approches consistent à générer des générateurs de tests pour un contexte particulier. Gligoric et al. ont

proposé un outil qui s'appelle UDITA [25] et qui, à partir d'une sur-couche de Java, permet d'exprimer une génération de tests. C'est alors qu'ils utilisent cette description pour générer des générateurs de tests avec leurs propres algorithmes.

6.3.3 Tests unitaires paramétrés

Dans l'expérimentation que nous avons mené, nous avons appliqué Praspel non pas directement sur les méthodes à tester mais sur une méthode qui va tester nos méthodes à tester. Cette technique s'appelle du *test unitaire paramétré* (ou PUT, pour *Parameterized Unit Testing*). Pex [52] est un outil répandu basé sur le test unitaire paramétré. Cette technique est pratique pour introduire une couche « adaptatrice » entre l'implémentation et le code. Elle permet d'écrire un test que nous pouvons piloter grâce à une spécification. Cela revient à créer plusieurs tests à partir d'un test concret manuel.

Notons qu'aucun effort d'implémentation n'est nécessaire. Au lieu d'instrumenter et de piloter l'unité, nous nous concentrons sur le test. Il n'y a aucune différence au niveau de l'implémentation.

Annexe A

Grammaire des PCRE en langage PP

Nous donnons la grammaire du langage PCRE exprimée avec le langage PP. Ceci est une synthèse de la grammaire disponible dans le manuel du langage PCRE [30], qui a été par la suite optimisée. L'ensemble du langage PCRE est reconnu.

```
// Character classes.
%token  negative_class_          \[\^
%token  class_                   \[
%token _class                    \]
%token  range                    -

// Lookahead and lookbehind assertions.
%token  lookahead_               \(\?=
%token  negative_lookahead_      \(\?!
%token  lookbehind_              \(\?<=
%token  negative_lookbehind_     \(\?<!

// Conditions.
%token  named_reference_         \(\?\(<            -> nc
%token  absolute_reference_      \(\?\((?=\d)       -> c
%token  relative_reference_      \(\?\((?=[\+\-])   -> c
%token  c:index                  [\+\-]?\d+         -> default
%token  assertion_reference_     \(\?\(

// Comments.
%token  comment_                 \(\?#              -> co
%token  co:_comment              \)                 -> default
%token  co:comment               .*?(?=(?<!\\)\))

// Capturing group.
%token  named_capturing_         \(\?<              -> nc
%token  nc:_named_capturing      >                  -> default
%token  nc:capturing_name        .+?(?=(?<!\\)>)
%token  non_capturing_           \(\?:
%token  non_capturing_reset_     \(\?\|
%token  atomic_group_            \(\?>
%token  capturing_               \(
```

```
%token _capturing               \)

// Quantifiers (by default, greedy).
%token  zero_or_one_possessive   \?\+
%token  zero_or_one_lazy         \?\?
%token  zero_or_one              \?
%token  zero_or_more_possessive  \*\+
%token  zero_or_more_lazy        \*\?
%token  zero_or_more             \*
%token  one_or_more_possessive   \+\+
%token  one_or_more_lazy         \+\?
%token  one_or_more              \+
%token  exactly_n                \{[0-9]+\}
%token  n_to_m_possessive        \{[0-9]+,[0-9]+\}\+
%token  n_to_m_lazy              \{[0-9]+,[0-9]+\}\?
%token  n_to_m                   \{[0-9]+,[0-9]+\}
%token  n_or_more_possessive     \{[0-9]+,\}\+
%token  n_or_more_lazy           \{[0-9]+,\}\?
%token  n_or_more                \{[0-9]+,\}

// Alternation.
%token alternation               \|

// Literal.
%token character                 \\([aefnrt]|c[\x00-\x7f])
%token dynamic_character         \\([0-7]{1,3}|x[0-9a-zA-Z]{1,2}|x{[0-9a-zA-Z]+})
// Please, see PCRESYNTAX(3), General Category properties, PCRE special category
// properties and script names for \p{} and \P{}.
%token character_type            \\([CdDhHNRsSvVwWX]|[pP]{[^}]+})
%token anchor                    \\(bBAZzG)|\^|\$
%token match_point_reset         \\K
%token literal                   \\.|.


// Rules.

#expression:
    alternation()

alternation:
    concatenation() ( ::alternation:: concatenation() #alternation )*

concatenation:
    (   assertion() | quantification() | condition() )
    ( ( assertion() | quantification() | condition() ) #concatenation )*

#condition:
    (
        ::named_reference_:: <capturing_name> ::_named_capturing:: #namedcondition
      | (
            ::relative_reference_:: #relativecondition
          | ::absolute_reference_:: #absolutecondition
        )
        <index>
      | ::assertion_reference_:: alternation() #assertioncondition
    )
    ::_capturing:: concatenation()?
    ( ::alternation:: concatenation()? )?
    ::_capturing::

assertion:
    (
```

```
        ::lookahead_::           #lookahead
      | ::negative_lookahead_::  #negativelookahead
      | ::lookbehind_::          #lookbehind
      | ::negative_lookbehind_:: #negativelookbehind
    )
    alternation() ::_capturing::

quantification:
    ( class() | simple() ) ( quantifier() #quantification )?

quantifier:
    <zero_or_one_possessive>  | <zero_or_one_lazy>  | <zero_or_one>
  | <zero_or_more_possessive> | <zero_or_more_lazy> | <zero_or_more>
  | <one_or_more_possessive>  | <one_or_more_lazy>  | <one_or_more>
  | <exactly_n>
  | <n_to_m_possessive>       | <n_to_m_lazy>       | <n_to_m>
  | <n_or_more_possessive>    | <n_or_more_lazy>    | <n_or_more>

#class:
    (
        ::negative_class_:: #negativeclass
      | ::class_::
    )
    ( range() | literal() )+
    ::_class::

#range:
    literal() ::range:: literal()

simple:
    capturing()
  | literal()

#capturing:
    ( ::comment_:: <comment>? ::_comment:: #comment )
  | (
        ::named_capturing_:: <capturing_name> ::_named_capturing:: #namedcapturing
      | ::non_capturing_:: #noncapturing
      | ::non_capturing_reset_:: #noncapturingreset
      | ::atomic_group_:: #atomicgroup
      | ::capturing_::
    )
    alternation() ::_capturing::

literal:
    <character>
  | <dynamic_character>
  | <character_type>
  | <anchor>
  | <match_point_reset>
  | <literal>
```

Bibliographie

[1] B. K. Aichernig. Contract-based testing. In *Formal Methods at the Crossroads : From Panacea to Foundational Support*, volume 2757 of *Lecture Notes in Computer Science*, pages 34–48. Springer, 2003.

[2] André Baresel, Harmen Sthamer, and Michael Schmidt. Fitness function design to improve evolutionary structural testing. In William B. Langdon, Erick Cantú-Paz, Keith E. Mathias, Rajkumar Roy, David Davis, Riccardo Poli, Karthik Balakrishnan, Vasant Honavar, Günter Rudolph, Joachim Wegener, Larry Bull, Mitchell A. Potter, Alan C. Schultz, Julian F. Miller, Edmund K. Burke, and Natasa Jonoska, editors, *GECCO*, pages 1329–1336. Morgan Kaufmann, 2002. ISBN 1-55860-878-8.

[3] M. Barnett, K.R.M. Leino, and W. Schulte. The spec# programming system : An overview. *Construction and analysis of safe, secure, and interoperable smart devices*, pages 49–69, 2005.

[4] Patrick Baudin, Pascal Cuoq, Jean-Christope Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL : ANSI/ISO C Specification Language*, 2005. URL `http://frama-c.com/download/acsl_1.5.pdf`.

[5] Boris Beizer. *Black-box testing - techniques for functional testing of software and systems*. Wiley, 1995. ISBN 978-0-471-12094-0.

[6] F. Bouquet, F. Dadeau, and B. Legeard. Automated Boundary Test Generation from JML Specifications. In *FM'06, 14th Int. Conf. on Formal Methods*, volume 4085 of *LNCS*, pages 428–443, Hamilton, Canada, August 2006. Springer.

[7] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. Select – a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the international conference on Reliable software*, pages 234–245, New York, NY, USA, 1975. ACM. doi : http://doi.acm.org/

10.1145/800027.808445. URL http://doi.acm.org/10.1145/800027.808445.

[8] Pierre-Christophe Bué, Frédéric Dadeau, and Pierre-Cyrille Héam. Model-based testing using symbolic animation and machine learning. In *ICST Workshops*, pages 355–360. IEEE Computer Society, 2010.

[9] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe : automatically generating inputs of death. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM Conference on Computer and Communications Security*, pages 322–335. ACM, 2006.

[10] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. Klee : Unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *OSDI*, pages 209–224. USENIX Association, 2008. ISBN 978-1-931971-65-2.

[11] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Testing concurrent object-oriented systems with spec explorer. In J. Fitzgerald, I. Hayes, and A. Tarlecki, editors, *International Symposium of Formal Methods (FM'2005)*, volume 3582 of *LNCS*, pages 542–547. Springer, 2005. ISBN 3-540-27882-6.

[12] Noam Chomsky. Three models for the description of language. *IEEE Trans. Information Theory*, 2(3) :113– 124, 1956. ISSN 0018-9448. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1056813.

[13] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2(3) :215–222, 1976.

[14] David Coppit and Jiexin Lian. yagg : An Easy-To-Use Generator for Structured Test Inputs. In David F. Redmiles, Thomas Ellman, and Andrea Zisman, editors, *ASE*, pages 356–359. ACM, 2005.

[15] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Trans. Software Eng.*, 17(9) :900–910, 1991.

[16] Philippe Duchon, Philippe Flajolet, Guy Louchard, and Gilles Schaeffer. Boltzmann samplers for the random generation of combinatorial structures. *Combinatorics, Probability & Computing*, 13(4-5) :577–625, 2004.

[17] Ivan Enderlin. Hoa, 2011. URL `http://hoa-project.net`.

[18] Ivan Enderlin, Abdallah Ben Othman, Frédéric Dadeau, and Alain Giorgetti. Realistic Domains for Unit Tests Generation. Research Report RR2010-01, LIFC - Laboratoire d'Informatique de l'Université de Franche Comté, September 2010.

[19] Ivan Enderlin, Frédéric Dadeau, Alain Giorgetti, and Abdallah Ben Othman. Praspel : A specification language for contract-based testing in php. In B. Wolff and F. Zaidi, editors, *23th IFIP International Conference on Testing Software and Systems (ICTSS'11)*, pages ***–***, Paris, France, November 2011. Springer. To appear.

[20] Eitan Farchi, Alan Hartman, and Shlomit S. Pinter. Using a model-based test generator to test for standard conformance. *IBM Systems Journal*, 41(1) :89–110, 2002.

[21] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009. ISBN 978-0-521-89806-5.

[22] Philippe Flajolet, Paul Zimmermann, and Bernard Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theor. Comput. Sci.*, 132(2) :1–35, 1994.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, 1995. ISBN 0201633612. URL `http://www.amazon.co.uk/exec/obidos/ASIN/0201633612/citeulike-21`.

[24] M. R. Garey and David S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. ISBN 0-7167-1044-7.

[25] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in udita. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 225–234. ACM, 2010.

[26] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart : directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, *PLDI*, pages 213–223. ACM, 2005. ISBN 1-59593-056-6.

[27] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *NDSS*. The Internet Society, 2008.

[28] Sandrine-Dominique Gouraud. *Utilisation des Structures Combinatoires pour le Test Statistique*. PhD thesis, Université Paris XI, Orsay, June 2004.

[29] Frank Gray. Pulse code communication, March 17 1953. US Patent 2,632,058.

[30] Philip Hazel. Perl compatible regular expressions, 2011. URL `http://pcre.org`.

[31] Pierre-Cyrille Héam and Cyril Nicaud. Seed : An easy-to-use random generator of recursive data structures for testing. In *ICST*, pages 60–69. IEEE Computer Society, 2011.

[32] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 2010.

[33] James C. King. A new approach to program testing. In Clemens Hackl, editor, *Programming Methodology*, volume 23 of *Lecture Notes in Computer Science*, pages 278–290. Springer, 1974. ISBN 3-540-07131-8.

[34] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7) :385–394, 1976.

[35] Eric Larson and Todd Austin. High coverage detection of input-related security faults. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, pages 9–9, Berkeley, CA, USA, 2003. USENIX Association. URL `http://portal.acm.org/citation.cfm?id=1251353.1251362`.

[36] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. In F. S. de Boer, Bonsangue M. M., S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects : First International Symposium, FMCO 2002, Lieden, The Netherlands, November 2002, Revised Lectures*, volume 2852 of *LNCS*, pages 262–284. Springer, Berlin, 2003.

[37] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML : a behavioral interface specification language for java. *ACM SIGSOFT Software Engineering Notes*, 31(3) :1–38, 2006.

[38] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6) :1811–1841, 1994.

[39] Phill McMinn. Search-based software test data generation : a survey. *Software Testing, Verification and Reliability*, 14(2) :105–156, 2004.

[40] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25 (10) :40–51, 1992.

[41] Glenford J. Myers. *The art of software testing*. Business data processing - A Wiley-Interscience publication. Wiley, New York, 1979. ISBN 0-471-04328-1.

[42] Cyril Nicaud. On the average size of glushkov's automata. In Adrian Horia Dediu, Armand-Mihai Ionescu, and Carlos Martín-Vide, editors, *LATA*, volume 5457 of *Lecture Notes in Computer Science*, pages 626–637. Springer, 2009. ISBN 978-3-642-00981-5.

[43] A. Jefferson Offutt, Zhenyi Jin, and Jie Pan. The dynamic domain reduction procedure for test data generation. *Softw. Pract. Exper.*, 29 :167–193, February 1999. ISSN 0038-0644. doi : 10.1002/(SICI)1097-024X(199902)29:2⟨167::AID-SPE225⟩3.3.CO;2-M. URL `http://portal.acm.org/citation.cfm?id=309087.309101`.

[44] Jeff Offutt, Paul Ammann, and Lisa Liu. Mutation testing implements grammar-based testing. In *Mutation Analysis, 2006. Second Workshop on*, pages 12–12. IEEE, 2006.

[45] C. Oriat. Jartege : A Tool for Random Generation of Unit Tests for Java Classes. In R. Reussner, J. Mayer, J.A. Stafford, S. Overhage, S. Becker, and P.J. Schroeder, editors, *First Int. Conf. on the Quality of Software Architectures, QoSA 2005 and Second Int. Workshop on Software Quality, SOQUA 2005*, volume 3712 of *LNCS*, pages 242–256, Erfurt, Germany, September 2005. Springer. ISBN 3-540-29033-8.

[46] PHP Group. The PHP website, 2011. URL `http://php.net`.

[47] Yann Ponty, Michel Termier, and Alain Denise. Genrgens : software for generating random genomic sequences and structures. *Bioinformatics*, 22(12) :1534–1535, 2006.

[48] Alexander Pretschner. Model-based testing. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *ICSE*, pages 722–723. ACM, 2005.

[49] James Gary Propp and David Bruce Wilson. Exact sampling with coupled markov chains and applications to statistical mechanics. *Random Struct. Algorithms*, 9(1-2) :223–252, 1996.

[50] C. V. Ramamoorthy, Siu-Bun F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Trans. Software Eng.*, 2(4) :293–300, 1976.

[51] Koushik Sen, Darko Marinov, and Gul Agha. Cute : a concolic unit testing engine for c. In Michel Wermelinger and Harald Gall, editors, *ESEC/SIGSOFT FSE*, pages 263–272. ACM, 2005. ISBN 1-59593-014-0.

[52] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .net. In Bernhard Beckert and Reiner Hähnle, editors, *TAP*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008. ISBN 978-3-540-79123-2.

[53] Jan Tretmans. Conformance testing with labelled transition systems : Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29(1) :49–79, 1996.

[54] Mark Utting and Bruno Legeard. *Practical Model-Based Testing - A tools approach*. Elsevier, 2006. ISBN 0-12-372501-1. 550 pages, ISBN 0-12-372501-1.

[55] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. Pathcrawler : Automatic generation of path tests by combining static and dynamic analysis. In Mario Dal Cin, Mohamed Kaâniche, and András Pataricza, editors, *EDCC*, volume 3463 of *Lecture Notes in Computer Science*, pages 281–292. Springer, 2005. ISBN 3-540-25723-3.

Résumé

Les travaux présentés dans ce mémoire proposent un langage d'annotation et de spécification, nommé Praspel, basé sur le paradigme *Design-by-Contract*. Ce langage se base sur les domaines réalistes afin d'obtenir un générateur automatique de données de tests pour réaliser du test unitaire en PHP. Les domaines réalistes sont des structures portant deux caractéristiques fonctionnelles essentielles qui sont la prédicabilité (pour tester si une valeur appartient au domaine réaliste) et la générabilité (pour générer une valeur parmi les valeurs du domaine réaliste). Nous sommes capables d'assigner différents générateurs de données aux domaines réalistes afin d'influencer la génération de données de tests, et donc, la couverture des tests. Praspel permet ainsi de spécifier des invariants et des pré- et post-conditions, dans lesquels s'expriment des comportements, des exceptions, des prédicats etc. La génération des données de test est réalisée par un générateur aléatoire.

Nous proposons dans ce stage de Master 2 d'appliquer le concept de *Grammar-based Testing* à l'intérieur d'un domaine réaliste. Ainsi une grammaire sera utilisée comme référence à la fois pour vérifier la syntaxe d'un texte (caractéristique de prédicabilité) et pour générer uniformément des textes, de taille spécifiée, respectant les règles de cette grammaire (caractéristique de générabilité). Cela nous a mené à concevoir un générateur d'analyseurs syntaxiques en PHP, ainsi qu'un algorithme de génération uniforme d'arbres de syntaxe abstraite de taille donnée. Enfin, une expérimentation a été conduite sur des projets d'étudiants en Licence 2 afin de tester de la génération de code HTML réalisée dans le module de Langages du Web.

Abstract

Works presented in this Master thesis propose an annotation and a specification language, named Praspel, based on the Design-by-Contract paradigm. This language is based upon realistic domains in order to produce an automatic test data generator to make unit testing in PHP. Realistic domains are structures that come with two essential functional features which are the predicability (to test if a value belongs to the realistic domain) and the samplability (to sample a value among values from the realistic domain). We are able to assign different data samplers to realistic domains so as to influence the test data generation, and thus, the test coverage. Then, Praspel allows to specify invariants, pre- and post-conditions, which include behaviors, exceptions, predicates etc. The test data generation is performed by a random generator.

In this Master thesis, we propose to apply the Grammar-based Testing concepts into a realistic domain. Thus, a grammar will be used as a reference both to verify the syntax of a text (the predicability feature) and to sample text uniformly (the samplability feature). This work leads us to design a lexer and parser generator in PHP, along with an algorithm that uniformly generates abstract syntax trees of a given size. Finally, an experimentation has been conducted using student projects from 2nd year in order to test HTML code generation in the Web Languages module.