Génération automatique de tests unitaires avec Praspel, un langage de spécification pour PHP

Ivan Enderlin

▶ **To cite this version:**

Ivan Enderlin. Génération automatique de tests unitaires avec Praspel, un langage de spécification pour PHP. Génie logiciel [cs.SE]. Université de Franche-Comté, 2014. Français. NNT : . tel-01093355v1

HAL Id: tel-01093355

https://inria.hal.science/tel-01093355v1

Submitted on 10 Dec 2014 (v1), last revised 19 Oct 2016 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

SPIM

Thèse de Doctorat

école doctorale sciences pour l'ingénieur et microtechniques

UFC UNIVERSITÉ DE FRANCHE-COMTÉ

N° | X | X | X

THÈSE présentée par

Ivan ENDERLIN

pour obtenir le

Grade de Docteur de
l'Université de Franche-Comté

Génération automatique de tests unitaires avec Praspel, un langage de spécification pour PHP

The Art of Contract-based Testing in PHP with Praspel

Spécialité Informatique

Instituts Femto-ST (département DISC) et INRIA (laboratoire LORIA)

Soutenue publiquement le 16 juillet 2014 devant le Jury composé de :

Rapporteurs	Lydie DU BOUSQUET	Professeur à l'Université Joseph Fourier
	Arnaud GOTLIEB	Chargé de recherche habilité à l'IRISA
Examinateur	Michel RUEHER	Professeur à l'Université de Nice Sophia Antipolis
Directeurs	Fabrice BOUQUET	Professeur à l'Université de Franche-Comté
	Frédéric DADEAU	Maître de conférences à l'Université de Franche-Comté
	Alain GIORGETTI	Maître de conférences à l'Université de Franche-Comté

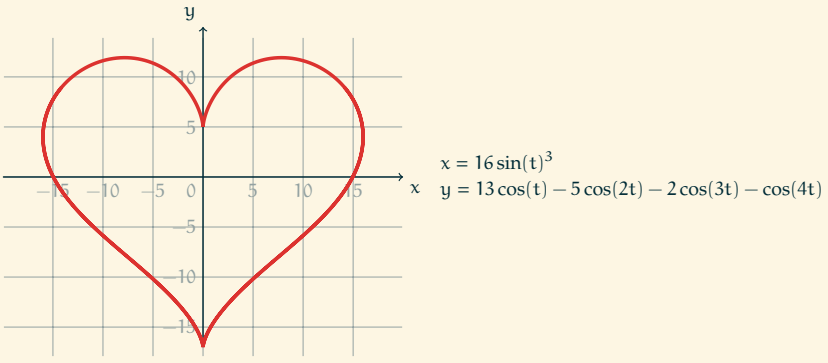
Remerciements Je tiens à profiter de l'occasion qui m'est offerte pour remercier toutes les personnes qui ont été soutenantes. Je tiens à remercier avant tout mon Dieu, *the Everlasting*, pour cette épreuve, ses promesses et son soutien en tout temps. Merci.

Je souhaite remercier ma petite femme, Hend, *lady caramel*, pour son éternel amour, sa patience, son écoute et son infini tendresse. Je t'aime.
اريد ان اشكر زوجتِي العزيزة و الحلوة هند علَا صبرهَا و اِنصَاتهَا و علَا حنَانهَا الَادمي. احبك.

Je souhaite particulièrement remercier mes parents et beaux-parents, Nadine et Christophe, حيَاة و سمير, mes sœurs et belles-sœurs, Naomi et Joanna, فيروز و يسرِي, ainsi que toute ma famille, pour leur présence et leur soutien.

Merci à tous les amis de France, de Suisse, de Tunisie, du Maroc, d'Italie, de Grèce, du Brésil, du Kenya, de Belgique, du Mexique, de Macédoine et de partout, qui m'ont aidé dans ma thèse et qui m'aident dans Hoa. Par ordre alphabétique, un immense et sincère merci à Abdallah Ben O., Alexandre V., Alexis von G., Baptiste et Anne-Laure F., Елизабета F.-C., Emmanuel T., Frédéric H., Guislain D., Gérard E., Isabelle et Yves D., محمد جلَال, Jean-Marie G., Julien B., Julien C., Julien L., Kalou C. C., Laura et Raphaël E., Lucie et آمين M., María Aydeé S. S. et Cédric J., Marta P., Mikaël R., Naomy W., Nawo M., Ophélie et Matthieu C., Raphaëlle M., Σοφία και Κώστας T., Stéphane P., Sylvie et Kiko F., Sébastien H., Willy M., Wilma et Fabio S. (et toute la troupe !), Yohann D., toute la communauté de Hoa, d'atoum et de PHP, et tout ceux que j'aurais oublié. . .

Et bien sûr, merci à mes encadrants : Fabrice B., Frédéric D. et Alain G. pour leur savoir, leur patience et être restés à mes côtés durant cette aventure !



$$x = 16\sin(t)^3$$
$$y = 13\cos(t) - 5\cos(2t) - 2\cos(3t) - \cos(4t)$$

Table des matières

Table des définitions

Table des figures

Table des exemples

Chapitre 1.
Introduction

Table des matières

Aujourd'hui, nous sommes dans l'ère du tout numérique. Le nombre de logiciels produits et utilisés chaque jour est énorme. Nous utilisons de plus en plus de machines dans notre quotidien : téléphones, tablettes, ordinateurs, télévisions, appareils électroménagers... Une très grande majorité d'entre eux sont connectés à un réseau, le plus souvent Internet. Ce dernier nous permet d'accéder à des milliers de millions de sites Web. Plus de 80% de ces sites Web sont développés en PHP [W3Techs, 2014]. Il est nécessaire que ces programmes soient vérifiés et validés.

Les travaux présentés dans ce mémoire portent sur la validation de programmes PHP à travers un nouveau langage de spécification, accompagné de ses outils. Ces travaux s'articulent selon trois axes majeurs : langage de spécification, génération automatique de données de test et génération automatique de tests unitaires.

Ces travaux ont été réalisés au Département d'Informatique des Systèmes Complexes (abrégé DISC [1]) de l'institut CNRS Femto-ST [2], mais également au Laboratoire lorrain de Recherche en Informatique et ses Applications (abrégé LORIA [3]) de l'Institut National de Recherche en Informatique et en Automatique (abrégé INRIA [4]). J'ai été accueilli en septembre 2011 au sein de l'équipe VErification, SpécificatiON, Test et Ingénierie des mOdèles (abrégé VESONTIO) pour le DISC, et de l'équipe *Combining ApproacheS for the Security of Infinite state Systems* (abrégé CASSIS) pour le LORIA. Ce mémoire a été financé à travers Squash [5], un projet *open-source* visant à structurer et industrialiser les activités du test fonctionnel. Dans le cadre de ce projet, plusieurs partenaires contribuent pour : développer un outillage *open-source* mature et innovant, proposer une méthodologie *open-source* documentant l'ensemble des processus de la qualification fonctionnelle et définir des normes et modèles permettant de mieux mesurer l'activité du test fonctionnel. Les travaux présentés dans ce mémoire ne s'inscrivent pas dans la même thématique que le projet Squash. Ces travaux se situent en amont de la chaîne de production. Le contexte scientifique de ce mémoire et les problématiques qui y sont développés sont liés aux savoir-faire et thématiques des équipes VESONTIO et CASSIS. Les deux grandes thématiques sont la modélisation pour vérifier et valider, et la vérification et la validation symbolique et à contraintes.

Informellement, un test est exécuté sur un Système Sous Test (*System Under Test*, abrégé **SUT**) et est composé de deux parties :

1. des **données de test** pour exécuter ce SUT ;
2. un **oracle**, permettant d'établir le verdict du test : est-ce que le produit de l'exécution et l'état du SUT après son exécution sont ceux attendus ou non ? Les valeurs du verdict sont : succès, échec ou inconclusif.

Aujourd'hui, dans le monde industriel, la majorité des tests sont écrits **manuellement**, par des ingénieurs de test. Un SUT leur est fourni, ils l'exécutent avec des données de test et établissent eux-mêmes le verdict du test. Dans le cas des tests **automatisés**, l'exécution du SUT et de l'oracle sont faites par la machine. C'est le rôle qui incombe aux *frameworks* « xUnit », comme JUnit [Beck and Gamma, 1997] pour Java, atoum [Hardy, 2010] ou PHPUnit [Bergmann, 2001] pour PHP, CUnit [Kumar, 2001] pour C etc. Le travail des ingénieurs de test peut se résumer à transformer le cahier des charges en tests afin de valider le programme. Cela reste toutefois informel, et surtout, une tâche laborieuse. Pourquoi alors ne pas écrire une spécification

1. Voir http://disc.univ-fcomte.fr/.
2. Voir http://femto-st.fr/.
3. Voir http://www.loria.fr/.
4. Voir http://inria.fr/.
5. Voir http://squashtest.org/.

formelle et la rendre exécutable ? C'est à dire utiliser un langage formel pour décrire le fonctionnement du programme, puis dériver, à partir de cette spécification, des tests qui soient exécutables et qui vont vérifier ou valider plusieurs aspects de cette spécification.

1.1 *Behavioral Interface Specification Language*

Le terme **spécification** signifie généralement une description précise des comportements et propriétés d'un artéfact [Hatcliff, Leavens, Leino, Müller, and Parkinson, 2012]. **Vérification** signifie prouver qu'une implémentation (un programme) satisfait une spécification particulière dans toutes les exécutions et configurations possibles. Généralement, une telle preuve est accomplie à l'aide de raisonnements statiques, c'est à dire uniquement par analyse du programme. Lorsqu'un raisonnement statique n'est pas possible ou suffisant, nous pourrons utiliser une **validation** dynamique, pour tester l'implémentation, c'est à dire que nous exécuterons le programme dans des configurations différentes (qui sont des sous-ensembles de toutes les configurations possibles). Une validation dynamique est un test. Elle peut être accompagnée d'un *Runtime Assertion Checker*, abrégé RAC [Geller, 1976], qui permet de valider une spécification lors d'une exécution du programme.
Un **langage de spécification** formel est une notation précise pour décrire les comportements et propriétés d'un programme. Les **notations formelles** aident à rendre les spécifications non-ambiguës, moins dépendantes des normes culturelles [6], et ainsi moins sujettes à une mauvaise interprétation. Les langages de spécification sont conçus pour répondre à des problèmes généraux ou spécialisés. Le plus souvent ils sont spécialisés, en fonction de la catégorie de problèmes ou langages qu'ils visent : ensembliste, logique, typage statique et fort, ou dynamique et faible, orienté objet ou fonctionnel etc.
Par ailleurs, quand une notation est formelle, elle peut être exploitée par une machine pour y appliquer un traitement automatique. Les spécifications formelles peuvent alors aider à **automatiser le test**, autant pour générer les données de test [Bernot, Gaudel, and Marre, 1991; Korel and Al-Yami, 1998; Sankar and Hayes, 1994; Jalote, 1992] que pour calculer le verdict du test [Cheon and Leavens, 2002b]. Nous allons explorer ces deux directions.
Les spécifications peuvent documenter les **interfaces** d'un programme. Par interfaces, il faut comprendre les parties du programme permettant de manipuler les données. Une catégorie de ces spécifications est appelée **langage d'annotations**. De tels langages peuvent jouer un rôle central dans la programmation. Par exemple, ils facilitent la maintenance du code, qui n'est généralement pas suffisament ex-

6. Par exemple, une pinte représente 473ml aux États-Unis d'Amérique, mais 586ml en Angleterre.

pressif pour faire comprendre au développeur ses tenants et ses aboutissants. Par ailleurs, lors d'une phase de *debugging*, les annotations permettent de localiser plus facilement l'erreur. Par exemple, si une propriété est violée, il sera plus facile à l'utilisateur de situer l'erreur dans un comportement spécifique. Dans le cas où nous travaillons en **boîte blanche** (*white-box*), c'est à dire quand le code source du programme est accessible, les annotations peuvent faire partie de la grammaire du langage, ou alors, elles sont écrites dans des commentaires. Dans le cas où nous travaillons en **boîte noire** (*black-box*), c'est à dire quand il n'y a pas ou partiellement d'accès au code source, les annotations peuvent parfois être externes au programme, par exemple sur un langage de définition d'interface (*Interface Definition Language*, abrégé IDL) ou sur un modèle, ce qui nous rapprocherait du *Model-based Testing*, abrégé MbT [Utting and Legeard, 2007]. La différence entre boîte noire et boîte blanche peut parfois être plus subtile [Gaudel, 2011] : nous considérons que nous sommes en boîte noire quand nous n'exploitons pas la structure du code, basiquement le contenu des fonctions et méthodes, sinon, nous sommes en boîte blanche.
Les langages d'annotations sont des *Behavioral Interface Specification Language*, abrégé BISL. C'est dans cette catégorie de langages que se situent nos contributions.

1.2 Contexte et problématique

Ce mémoire traite de la conformité entre un programme et sa spécification, à travers la création d'un nouveau langage de spécification simple, pragmatique pour le Web et les développeurs, et assemblant plusieurs méthodes du test. Chaque méthode introduite se verra enrichie pour la rendre compatible avec les besoins liés au Web. Les parties suivantes expliquent notre motivation, le contexte et les enjeux.

1.2.1 *PHP: Hypertext Preprocessor*

PHP [The PHP Group, 2001] se définit comme étant un « langage de script universel populaire qui est particulièrement adapté pour le développement Web ». PHP se définit également avec les adjectifs suivants : « rapide, flexible et pragmatique ». C'est aussi un langage multiparadigmes. Il est par exemple procédural avec des fonctions nommées, orienté objet avec des classes (héritage vertical), des interfaces (typage) et des traits (héritage horizontal), ou encore fonctionnel avec des fonctions anonymes (λ) et des fermetures ($\overline{\lambda}$). Son système de typage est dynamique et faible, c'est à dire que les types sont non-déclarés et qu'il est possible de transtyper les données. PHP est un langage de script interprété. Il existe plusieurs interpréteurs, appelés machines virtuelles. Nous pouvons voir PHP comme un langage « glue » (comme la majorité des langages de script), c'est à dire qu'il communique avec à peu près tout et est au centre de beaucoup de logiciels. La syntaxe du langage est

inspirée de C, C++, Java ou encore Perl. Depuis plusieurs années, le langage connaît des améliorations importantes afin de faire oublier les erreurs de conception passées qui lui sont trop souvent associées [7]. Nous nous intéressons à des programmes écrits en PHP pour plusieurs raisons.
Comparés à des langages plus traditionnels, comme C ou Java, les langages de scripts accélèrent le processus de développement grâce à la flexibilité qu'ils offrent avec un typage dynamique, faible et un mélange de paradigmes. Cependant, cette flexibilité rend plus difficile la compréhension du comportement de certains programmes, tout comme il est plus difficile de s'assurer que le programme n'est pas affecté par une modification (il est alors question de régression).
Par ailleurs, plus de 80% des programmes Web fonctionnent avec PHP. Cela implique qu'il y a un marché important avec d'immenses besoins ; que ce soient des sites de commerces (comme Etsy [8]), de banques, d'assurances, de réseaux sociaux (comme Facebook [9]), de moteurs de recherche (comme Yahoo ! [10]), d'encyclopédies (comme Wikipedia [11]), gouvernementaux (France, Belgique, Suisse, USA, Canada etc.) et autres. Tous ces domaines ont besoin de qualité logicielle.

1.2.2 Test unitaire

Plusieurs travaux cherchent à améliorer la sécurité des programmes écrits en PHP à l'aide d'analyses statiques [Yu, Alkhalaf, and Bultan, 2010; Balzarotti, Cova, Felmetsger, Jovanovic, Kirda, Kruegel, and Vigna, 2008; Wassermann and Su, 2008, 2007; Xie and Aiken, 2006] du code source, des flots des données pour trouver des chaînes de caractères malicieuses, des injections SQL etc. Toutefois, un grand nombre des solutions proposées souffrent de limitations majeures [Hauzar and Kofron, 2012] comme une détection faible des erreurs, un taux de faux-positifs important ou même un support faible des constructions du langage ; l'aspect dynamique et multiparadigmes du langage n'aidant pas. D'autres travaux testent PHP à un niveau plus élevé, par exemple en testant le site Web écrit en HTML directement [Artzi, Kiezun, Dolby, Tip, Dig, Paradkar, and Ernst, 2010; Kiezun, Guo, Jayaraman, and Ernst, 2009; McAllister, Kirda, and Kruegel, 2008; Benedikt, Freire, and Godefroid, 2002]. Ces travaux se situent du côté client et observent les résultats produits par PHP côté serveur. Ces travaux sont intéressants pour, d'une part, tester la sécurité, et d'autre part, tester des programmes existants.
Toutefois les méthodes de développement ont énormément évolué ces dernières années avec la qualité logicielle [Lépine, Seguy, and Jean-Marc, 2009]. Les « horribles

7. Voir https://wiki.php.net/rfc.
8. Voir https://etsy.com/.
9. Voir https://facebook.com/.
10. Voir https://yahoo.com/.
11. Voir https://wikipedia.org/.

programmes » tels ceux que nous avons lus ou écrits en PHP tendent à disparaître. De nouveaux *frameworks* ou de nouvelles bibliothèques PHP ont fait leur apparition et mettent en avant l'usage intensif de bonnes pratiques pour le développement de programmes maintenables et testables. Des *frameworks* de test de plusieurs natures ont également fait leur apparition et un programme non testé ne sera même plus installé.
En outre, aucun travail de recherche n'a fait de propositions pour spécifier des programmes écrits en PHP. Enfin, aucun travail ne considère PHP pour la génération automatique de données de test ou encore la génération automatique de tests unitaires.
Nous profitons de cette tendance d'amélioration de la qualité logicielle et de l'absence de travaux dans le domaine des langages de spécification pour introduire un langage de spécification dans PHP. Nos travaux s'inscrivent au niveau des tests unitaires et au niveau du programme PHP lui-même, c'est à dire que nous ne tenons pas compte du contexte d'utilisation du programme (client-serveur, client seul, démon etc.).

1.2.3 Langage de spécification

Comme PHP est un langage « glue », il traite des données de toutes sortes. C'est une première contrainte. De plus, les utilisateurs de PHP sont aussi bien des débutants que des experts. La palette des utilisateurs est donc très large. C'est un avantage et un inconvénient du langage : il est simple à utiliser et permet d'obtenir des résultats rapidement mais il permet aussi de développer des programmes plus complexes. Toutefois, tous ces utilisateurs suivent la tendance d'amélioration de la qualité logicielle. C'est une deuxième contrainte. Enfin, le langage PHP est très pragmatique, c'est ce qui le rend si facile d'accès. Il est possible d'écrire un programme PHP en très peu de lignes de code sans connaître et installer des dizaines de bibliothèques ou d'autres outils. Les utilisateurs du langage sont habitués à ce pragmatisme. C'est une troisième contrainte.
Si nous proposons un langage de spécification pour PHP, il doit impérativement tenir compte de ces trois contraintes. Il doit être simple pour être utilisé et compris par la majorité des utilisateurs. Il doit être pragmatique pour que le travail de spécification et de génération automatique de tests unitaires puisse s'intégré dans la chaîne industrielle. L'objectif n'est pas de traiter tous les problèmes mais les problèmes les plus courants et les plus préoccupants. Et enfin, puisque culturellement PHP est un langage « glue » et que nous voulons traiter plusieurs problèmes, le langage de spécification que nous proposons doit assembler plusieurs méthodes du test.
Nous pensons que la programmation par contrat est pertinente pour répondre à toutes ces contraintes et pour générer automatiquement des tests sunitaires.

1.3 Contributions

Dans ce mémoire, nous proposons un nouveau langage de contrat pour PHP. À partir de ce langage, nous voulons être capables de générer automatique des tests unitaires. Ainsi, nous avons trois axes de réflexion : le langage lui-même (sa syntaxe et sa sémantique), les algorithmes de génération de données, et enfin les algorithmes de génération de tests unitaires. Nous voulons que les générations soient automatiques, c'est à dire que l'humain intervienne le moins possible.

1.3.1 Praspel et les domaines réalistes

Le langage de contrat que nous proposons s'appelle **Praspel** : *PHP Realistic Annotation and SPEcification Language*. Cette première contribution, présentée dans le chapitre 3, traite du langage en lui-même, à savoir sa syntaxe et sa sémantique. Nous avons également étudié comment évaluer un contrat pour qu'il vérifie des données à l'exécution. Ce langage est inspiré de JML [Leavens, Baker, and Ruby, 1999] (s'applique à Java) mais se différencie sur la façon de spécifier les données : Java a un typage déclaré et fort, PHP a un typage dynamique et faible. Pour spécifier les données, Praspel s'appuie en grande partie sur les **domaines réalistes**, des structures permettant de valider et générer des données et pouvant être emboîtées afin de représenter des données plus complexes. Cette contribution, à savoir Praspel et les domaines réalistes, a été publiée dans Enderlin, Dadeau, Giorgetti, and Ben Othman [2011].

1.3.2 Générateurs de données

Un contrat peut être utilisé pour générer automatiquement des tests unitaires. Un test est constitué de deux parties : des données de test et un oracle. Cette contribution, présentée dans le chapitre 4, traite de la génération des données de test. Ces données sont spécifiées au mieux par le contrat. Nous devons proposer des solutions capables de générer toutes sortes de données : des booléens, des entiers, des réels, des chaînes de caractères, des tableaux, des objets etc. Les premières générations étaient aléatoires uniformes pour des booléens, des entiers et des réels. Toutefois, cette approche a rapidement montré ses limites avec des chaînes de caractères, des tableaux ou des objets. Nous avons proposé des algorithmes de génération de chaînes de caractères s'appuyant sur des grammaires. Ensuite, nous nous sommes concentrés sur les tableaux avec un solveur et des contraintes dédiées. Ces contributions ont donné lieu à deux publications, respectivement Enderlin, Dadeau, Giorgetti, and Bouquet [2012] et Enderlin, Giorgetti, and Bouquet [2013]. Enfin, nous avons amélioré la génération des objets en combinant toutes ces approches.

1.3.3 Critères de couverture des contrats

Cette dernière contribution, présentée dans le chapitre 5, « boucle la boucle ». D'une part, nous avons un langage de contrat qui peut être évalué pour valider et vérifier les données manipulées par le programme. D'autre part, nous avons des algorithmes capables de générer automatiquement des données de test à partir d'un morceau d'un contrat. La dernière étape explique le fait qu'un contrat peut exprimer plusieurs comportements et que nous devons nous assurer que tous ces comportements sont testés. Nous avons alors défini plusieurs critères de couverture sur les contrats afin d'avoir des objectifs de test à atteindre. Par conséquent, nous avons développé des algorithmes générant autant de tests unitaires que nécessaire pour satisfaire certains critères. Les données nécessaires pour exécuter ces tests seront générées automatiquement avec la contribution précédente. Cette contribution, accompagnée des contributions précédentes, ont été soumises au journal *Software and Systems Modeling*, abrégé SoSyM, pour une édition spéciale sur les méthodes formelles intégrées.

1.4 Plan du mémoire

Ce mémoire s'articule en 8 chapitres plus annexes et bibliographie. Le chapitre 1, ce chapitre, contient l'introduction, le contexte, la problématique et les contributions. Le chapitre 2 contient l'état de l'art nécessaire pour comprendre cette thèse et situer nos contributions. Les chapitres 3 à 5 présentent nos contributions : les domaines réalistes ainsi que Praspel, au niveau syntaxique et sémantiquen dans le chapitre 3, les différents algorithmes et stratégies de génération de données de test dans le chapitre 4 et les critères de couverture pour obtenir des objectifs de test à satisfaire afin de produire des tests unitaires pertinents dans le chapitre 5. Le chapitre 6 décrit les outils développés durant ce mémoire ou les outils incluant nos travaux. Le chapitre 7 valide nos contributions à travers des expérimentations. Et enfin, le chapitre 8 présente les conclusions de nos travaux et décrits plusieurs perspectives nous semblant pertinentes.

Chapitre 2.
État de l'art

Table des matières

Ce mémoire présente Praspel, un nouveau langage de spécification pour PHP qui utilise la programmation par contrat. À partir de ce langage nous générons des données de test et des suites de tests unitaires grâce à des critères de couverture sur les contrats. Ce chapitre est un état de l'art sur ces questions. La partie 2.1 présente les différents langages de contrat existants, pour d'autres langages de programmation, car aucun langage de contrat n'existe en PHP. Ensuite, la partie 2.2 présente des outils de génération de suites de tests à partir de contrats. Enfin, la partie 2.3 présente des techniques et outils de génération de données de test.

2.1 Langages de contrat

Des spécifications existent sous la forme de **contrats** [Liskov and Guttag, 1986; Meyer, 1992]. Nous parlons de programmation par contrat ou encore de *Design by Contract* (abrégé DbC). Le terme *contrat* est une métaphore conceptuelle des conditions et des obligations d'un contrat d'entreprise. Ce paradigme demande au développeur d'écrire une spécification formelle qui étend la définition classique des types abstraits [Liskov and Zilles, 1974] avec les contraintes formelles sui-

vantes : des **préconditions**, des **postconditions** et des **invariants**. Une précondition spécifie l'état du SUT (*System Under Test*) avant son exécution. Une postcondition spécifie l'état du SUT après son exécution (et peu comparer cet état avec l'état précédent). Un invariant, quant à lui, spécifie l'état du SUT avant et après son exécution. Ces contraintes formelles sont aussi appelées des **clauses**. Il est admis de dire qu'une clause contient des **assertions**. Les contrats sont apparus dans le langage Eiffel [Meyer, Nerson, and Matsuo, 1987], basé sur le paradigme de la programmation orienté objet. Dans un tel contexte, les préconditions et postconditions portent sur les méthodes (aussi appelées procédures ou routines) et les invariants sur les classes (aussi appelés modules). D'autres usages des contrats ont été faits hors du paradigme de la programmation orientée objet, par exemple avec des fonctions d'ordre supérieur [Findler and Felleisen, 2002]. Le *Design by Contract* définit les critères de conformité suivants :

– si les invariants et la précondition de la méthode sont satisfaits avant son appel,
– alors la méthode s'engage à satisfaire les invariants et sa postcondition après son exécution.

Avant l'exécution d'une méthode, le SUT se trouve dans un **pré-état**, et après l'exécution d'une méthode, il se trouve dans un **post-état**. Pour les langages supportant les exceptions, il existe un post-état normal et un post-état exceptionnel, dans le cas où une exception est levée. Par ailleurs, quand le contrat est évalué, nous disons que c'est un **succès** si aucune clause n'est violée, sinon c'est un **échec**. Nous verrons par la suite comment est évalué un contrat.

Les langages décrivant une spécification formelle à travers des contrats pour le comportement des programmes, et utilisés comme annotations, sont appelés des **langages de contrat**. Il en existe plusieurs [Hatcliff et al., 2012]. La suite de cette partie décrit les plus influents et connus : Eiffel, JML pour Java, ACSL pour C et Spec# pour C#.

Eiffel. Le langage Eiffel [Meyer et al., 1987] est un langage de programmation orienté objet. Sa contribution majeure au domaine a été l'utilisation importante du *Design by Contract* au sein même du langage (c'est à dire sans être placé en annotation), dont les assertions, préconditions, postconditions et invariants étaient utilisés pour assurer la conformité du programme. Les contributions originales comprenaient l'application au paradigme orienté objet et le support des exceptions. L'aspect objet apporte quelques spécificités, comme les invariants de classe, la gestion de l'héritage multiple ou la résolution de conflits avec l'héritage multiple comme le renommage de certains attributs ou méthodes. De même, la gestion du *dynamic binding* (aussi appelé *late binding*, représenté en général par la variable `this` pour l'instance de l'objet), demande à ce que la résolution des appels se fasse à l'exécution et non pas à la compilation. Les exceptions nécessitent également de définir dans

quels contextes elles peuvent être levées et dans quel état se trouve l'objet après la levée d'une exception. Cette information est d'autant plus importante que le langage Eiffel permet de ré-exécuter la routine (méthode) qui a déclenché l'exception. Les préconditions et postconditions sont exprimées avec les mots-clés `require` et `ensure`. Le contenu de ces clauses est exprimé dans le langage Eiffel lui-même. Le langage introduit également le mot-clé `assert` pour exprimer des assertions ou des invariants de boucle.
Les contrats sont compilés avec le reste du programme mais sont placés dans des fichiers à part. Lors de la construction finale de l'exécutable du programme, il est possible d'y inclure ou d'en exclure un ensemble de contrats. Le programme peut être intensivement testé avec tous les contrats, puis être distribué avec seulement un sous-ensemble de ses contrats pour des parties plus critiques. Moins il y a de contrats présents dans le programme, plus son exécution sera rapide, mais il est nécessaire de s'assurer de la fiabilité du programme.

JML. Le *Java Modeling Language* [Leavens et al., 1999] est un langage de contrat pour Java [Gosling and Joy, 1990]. À l'inverse du langage Eiffel, JML est un langage d'annotations pour Java. Il utilise les mots-clés `requires` et `ensures` pour introduire les préconditions et les postconditions. Une postcondition exceptionnelle est introduite par le mot-clé `signals`. Les invariants de classe sont introduits par le mot-clé `invariant`, et les invariants de boucle avec le mot-clé `loop_invariant`. JML propose la clause `also` qui permet de raffiner, par un nouveau contrat, des contrats hérités d'une classe parente. Les expressions JML sont des expressions Java complétées par des identifiants réservés, comme `\result`, présent uniquement dans une postcondition normale, et qui fait référence au résultat de la méthode. De même, nous trouvons la construction `\old(i)`, présente uniquement dans une postcondition, qui fait référence à la valeur de la donnée `i` dans le pré-état. JML propose également des constructions logiques, comme la relation d'implication (==>), d'équivalence (<==>) ou des quantificateurs comme `\forall` (∀) et `\exists` (∃).
La distribution officielle de JML fournit principalement deux outils. Le premier, le *Runtime Assertion Checker*, abrégé RAC, est utilisé pour du *monitoring* de programme Java, et le second, JMLUnit, permet de transformer un contrat en test (nous le détaillons dans la partie 2.2). Le RAC de JML [Cheon and Leavens, 2002a] est un outil permettant de valider les contrats JML lors de l'exécution du programme. Les contrats JML sont traduits en Java [Raghavan and Leavens, 2005]. La proximité entre ces deux langages est la clé de la démarche. En effet, la syntaxe des contrats JML est similaire à celle de Java, et les mots-clés JML spécifiques (par exemple pour les clauses, les quantificateurs etc.) sont traduits par des structures Java adéquates. Néanmoins, cette similitude a des limites : certaines constructions du langage, comme les quantifications sur les objets, ne sont pas traduisibles et donc ne peuvent

pas être validées.
Le programme Java d'origine est ainsi enrichi par la validation des contrats JML. Un inconvénient est que le binaire résultant est plus important, ce qui engendre des ralentissements à l'exécution. En revanche, si une assertion n'est pas validée, une exception spécifique est déclenchée signalant le type de clause qui n'a pas été validée (invariant, précondition etc.) ainsi que l'état visible du système au moment de l'exécution. L'utilisation du RAC confère un moyen direct et efficace de s'assurer que les contrats JML ne sont pas violés durant une exécution.

ACSL. Le *ANSI/ISO C Specification Language* [Baudin, Cuoq, Filliâtre, Marché, Monate, Moy, and Prevosto, 2013] est un langage de contrat pour C [Ritchie, 1972], inspiré principalement du langage de spécification de l'outil Caduceus [Filliâtre and Marché, 2007], dédié à la vérification déductive de propriétés comportementales de programmes C. Caduceus est lui-même inspiré de JML. Les langages se ressemblent : les mot-clés `requires` et `ensures` introduisent respectivement une précondition et une postcondition. Les invariants de boucle sont introduits avec le mot-clé `invariant`. Plusieurs comportements peuvent être exprimés pour une même méthode, avec le mot-clé `behavior`. ACSL propose d'autres clauses pour exprimer des lemmes, des axiomes ou des fonctions logiques.
Contrairement à JML qui utilise Java pour exprimer ces assertions, ACSL n'utilise pas C mais un langage plus simple, qui s'inspire toutefois de C (pour les opérateurs arithmétiques, logiques, les tableaux etc.) afin de ne pas perturber l'utilisateur. D'autres contraintes sont présentes car intrinsèques au langage, comme la logique des pointeurs, et d'autres ne sont pas présentes, comme les exceptions.
Les outils d'ACSL sont intégrés dans le *Framework for Modular Analyses of C*, abrégé Frama-C [CEA and INRIA, 2008; Cuoq, Kirchner, Kosmatov, Prevosto, Signoles, and Yakobowski, 2012]. Ce dernier propose plusieurs analyses de programmes C. Les outils d'ACSL l'utilisent pour de la vérification statique. C'est la grande différence avec JML, qui lui repose sur un RAC, donc une validation à l'exécution, ce que ne fait pas ACSL. Des extensions à Frama-C peuvent être installées pour permettre à des outils externes, comme des assistants de preuve interactifs ou des prouveurs automatiques de théorèmes, d'aider à la vérification de certaines spécifications.

Spec#. Le langage Spec# [Barnett, Leino, and Schulte, 2004] permet de faire de la programmation par contrat pour C# [Microsoft, 2001]. Ce langage est très similaire à JML ou ACSL, avec toutefois moins de clauses. Les préconditions sont introduites avec le mot-clé `requires`, les postconditions avec le mot-clé `ensures`. De même, les invariants sont introduits avec le mot-clé `invariant`. Spec# propose les quantificateurs `forall`, `exists` et `exists unique`, `sum`, `product`, `min` etc. Les assertions sont exprimées avec un langage proche de C#, toujours pour ne pas perturber

l'utilisateur, mais toutes les constructions de C# ne sont pas présentes, seulement un sous-ensemble. Par ailleurs, Spec# enrichit le langage C# en lui ajoutant des constructions permettant de vérifier l'absence ou la possible présence de valeurs nulles (avec `!` ou `?`).
Tout comme ACSL, la vérification des contrats se fait statiquement, et non pas dynamiquement via un RAC. Contrairement à JML où les contrats sont transformés en Java, ils sont ici transformés en Boogie [Barnett, Chang, DeLine, Jacobs, and Leino, 2005], un langage intermédiaire pour encoder des conditions de vérification pour les langages impératifs et orientés objets. Boogie s'appuie sur des solveurs externes, notamment Z3 [De Moura and Bjørner, 2008], pour vérifier les contrats.

Dans cette étude des langages de contrat, deux grandes familles de vérification de contrats se distinguent : statique, à l'aide de prouveurs dédiés et externes, ou dynamique, à l'aide d'un RAC. Notre contribution, Praspel, appartient à cette seconde famille et utilise un RAC. Le vocabulaire pour introduire les préconditions et postconditions est souvent identique d'un langage de contrat à un autre, à savoir `requires` et `ensures`. PHP supporte les exceptions, donc il est important de faire la distinction entre les postconditions normales et exceptionnelles. ACSL introduit le mot-clé `behavior` pour clairement spécifier les comportements du SUT, Praspel s'en est inspiré. Les constructions `\result` et `\old()` sont également répandues, Praspel ne change pas de vocabulaire. En revanche, la façon d'exprimer les assertions change beaucoup d'un langage à l'autre : soit avec un sous-ensemble des constructions du langage cible (comme c'est le cas pour JML), ou avec un langage différent (comme c'est le cas pour ACSL ou Spec#). Utiliser un sous-ensemble du langage cible a pour avantage principal de ne pas demander un effort trop important à l'utilisateur. Toutefois, l'utilisateur peut écrire des assertions qui seront difficiles à analyser pour la génération automatique de données (par exemple avec l'utilisation de quantificateurs). En effet, nous utilisons Praspel pour faire du *Contract-based Testing*, présenté dans la partie suivante.

2.2 *Contract-based Testing*

Les contrats sont utilisés pour **filtrer** les données manipulées par un programme ou pour vérifier que le programme ne produira pas d'erreur. Cependant, l'un des intérêts et usages principaux des contrats est pour le test. Le **test à partir de contrats** (*Contract-based Testing*, abrégé CbT) a été introduit pour exploiter les langages de contrat afin de tester des programmes [Aichernig, 2003] :
– les invariants et les préconditions sont utilisés pour **générer des données** de test ;
– les postconditions **fournissent un oracle** permettant de calculer le verdict du test à l'exécution.

À partir d'une spécification formelle, sous la forme d'un contrat, il est possible de générer des suites de tests automatiquement. Comme nous sommes au niveau du code source du programme, et, plus précisément, sur les méthodes des classes (dans un contexte orienté objet), les tests seront **unitaires**.
Différents outils exploitant des contrats pour générer des suites de tests sont présentés ci-dessous.

JMLUnit. L'outil JMLUnit [Cheon and Leavens, 2002b; Zimmerman and Nagmoti, 2010] permet de transformer un contrat JML en une suite de tests exécutable avec JUnit. L'outil assure un moyen simple et efficace d'automatiser la génération d'oracles de test. L'objectif est de décharger l'utilisateur du calcul du verdict de test, à savoir si le test unitaire est un succès ou un échec. L'outil génère automatiquement deux classes pour chaque interface ou classe Java annotée en JML : un squelette où déclarer les données de test, et les oracles, permettant de calculer le verdict des tests. Les données de test ne sont pas générées automatiquement : un dictionnaire de données de test pré-définies selon certaines stratégies est utilisé à la place. Pour des types de données non-primitifs au langage (donc des données de test qui ne sont pas des booléens, des nombres, des chaînes de caractères etc.), le dictionnaire ne contient qu'une valeur nulle. C'est alors à l'utilisateur de fournir lui-même les données de test.
Comme l'outil se concentre sur le test unitaire, chaque test ne concerne qu'une seule méthode. Les postconditions et invariants JML sont transformés en prédicats Java et constituent les oracles de test. Si l'exécution d'une méthode avec un certain ensemble de données de test ne lève aucune erreur, alors le test est considéré comme un succès, sinon il est considéré comme un échec. Il est toutefois possible d'obtenir un troisième résultat qui est une violation d'une précondition d'une autre méthode appelée par la méthode testée. Ce cas est considéré comme inconclusif par l'outil car aucune précondition ou postcondition n'a été violée.
Enfin, le rapport de test est le même que celui produit habituellement par JUnit.

Jartege. L'outil Jartege [Oriat, 2005] est un générateur de tests aléatoires pour des programmes écrits en Java. Jartege produit des séquences de test, dont le nombre et la taille sont fixés par l'utilisateur.
Jartege sélectionne aléatoirement des méthodes à exécuter. Chaque méthode est annotée par un contrat JML. Les données de test, soient les valeurs des paramètres de ces méthodes, sont également générées aléatoirement à partir de la précondition. Cette dernière sert aussi de filtre pour sélectionner les méthodes à exécuter. L'exécution des méthodes et les postconditions et invariants sont utilisés pour connaître la conformité du programme avec sa spécification. La vérification des contraintes est ainsi conjointe à la création des séquences de test.

Pour les types primitifs de Java, Jartege demande à l'utilisateur de définir des bornes pour la génération aléatoire, ceci afin de générer des données de test plus pertinentes. L'outil permet à l'utilisateur de fournir une fonction qui détermine la probabilité de créer un nouvel objet ou d'en réutiliser un déjà existant. Selon les structures de données manipulées, cette fonction permettra de détecter plus ou moins d'erreurs.

JML-Testing-Tools. L'outil JML-Testing-Tools [Bouquet, Dadeau, Legeard, and Utting, 2005; Bouquet, Dadeau, and Legeard, 2006] est développé comme une extension aux outils BZ-Testing-Tools [Legeard, Peureux, and Utting, 2002], un ensemble d'outils d'animation et de génération automatique de programmes B [Abrial, 2005], Z [Spivey, 1989] ou de spécifications Statechart [Harel, 1987]. L'outil JML-Testing-Tools comprend un animateur symbolique et un générateur de tests.
Le premier permet d'animer symboliquement des méthodes Java en utilisant des spécifications écrites en JML. Le code des méthodes n'est jamais utilisé car l'outil manipule des modèles. JML-Testing-Tools permet d'appeler des méthodes avec des paramètres symboliques. Les valeurs de ces paramètres sont des contraintes définies sur le domaine lié au type des paramètres. Ces contraintes sont supportées par un solveur spécifique. Par conséquent, JML-Testing-Tools est capable de représenter un grand nombre d'états d'un programme. Une valuation pour obtenir l'ensemble de tous les états concrets est possible. L'outil est également capable de vérifier les propriétés JML à la volée et d'afficher un contre-exemple si l'une d'entre elles est violée.
Le générateur de test se base sur les spécifications JML. L'outil va tout d'abord extraire un ensemble de comportements (normaux et exceptionnels) à partir de la spécification JML. Un comportement est représenté par un prédicat en forme disjonctive. Cet ensemble de comportements va être réduit en fonction du critère de couverture du modèle choisi par l'utilisateur. L'outil propose quatre critères de couverture qui correspondent à des règles de réécriture de ces prédicats. Les données de test sont générées automatiquement en utilisant les valeurs aux limites des paramètres des méthodes. Cette approche fonctionne pour les types primitifs de Java avec un domaine ordonné. L'outil considère également qu'une valeur aux limites d'un objet est une valeur aux limites de l'un de ses attributs. Quand toutes les valeurs sont spécifiées, elles sont instanciées grâce à l'animateur symbolique et les tests peuvent être exécutés.

TestEra. L'outil TestEra [Marinov and Khurshid, 2001] est un langage d'annotations de code Java. Ses préconditions, postconditions et invariants expriment des contraintes avec le langage Alloy [Jackson, 2002]. Alloy est un langage de spécification déclaratif pour exprimer des contraintes structurelles complexes. Son objectif

est d'instancier des micro-modèles, à l'aide de son propre analyseur Alcoa [Jackson, Schechter, and Shlyakhter, 2000]. TestEra passe des invariants accompagnés de préconditions à l'analyseur d'Alloy, qui va lui générer plusieurs instances de modèles satisfaisant cette spécification. Chaque instance générée par Alcoa représente un test. TestEra va concrétiser ces instances une par une, afin de produire des données valides en Java. Ces dernières sont utilisées en tant que données de test, c'est à dire qu'elles vont servir à exécuter le SUT. Ensuite, TestEra regarde les sorties produites par le SUT ainsi que son post-état, qui sont abstraits pour se ramener au formalisme d'Alloy afin d'y être confrontés à la postcondition et aux invariants. Si cette dernière vérification échoue, un contre-exemple est produit. Sinon, le processus itère à l'instance suivante.

Korat. L'outil Korat [Boyapati, Khurshid, and Marinov, 2002] est la succession de TestEra. L'objectif de Korat est de couvrir des structures de données complexes non-isomorphiques de taille finie. Pour cela, il demande à l'utilisateur d'écrire un prédicat en Java (appelé `repOK`), qui va valider ou invalider les structures générées automatiquement. Il y a plusieurs avantages à cette démarche : l'utilisateur est familier avec le langage puisqu'il l'utilise pour développer, il y a plusieurs outils et environnements de développement qui peuvent aider à la rédaction ou l'analyse de ce prédicat, et enfin, le prédicat peut potentiellement déjà exister dans le programme. Korat va ensuite générer des structures à partir des entrées de ce prédicat. Pour cela, Korat va analyser le prédicat et indexer quels sont les accès aux méthodes et attributs sur la structure qui influencent la décision du prédicat. Korat va ensuite jouer sur ces méthodes et attributs afin de générer plusieurs structures candidates. Korat demande également à l'utilisateur d'écrire une méthode de *finitization* qui précise la taille et la nature des données portées par la structure. L'idée est de préciser à Korat dans quelle mesure il peut modifier la structure pour générer de nouvelles structures candidates. Cette technique assure qu'il y a un ensemble fini de structures candidates qui vont être sélectionnées.
Comme TestEra avant lui, Korat est utilisé depuis des annotations d'un programme Java. Plus précisément, les prédicats de Korat sont appelés depuis JML et la postcondition fournit l'oracle du test. Les tests sont alors exécutés sur un programme Java enrichi.
Korat se démarque de TestEra en proposant ses propres algorithmes de génération de données candidates à partir de prédicats Java. Cette approche demande un effort supplémentaire à l'utilisateur, car en plus d'écrire des contrats en JML, il doit écrire une méthode pour un prédicat et une autre pour une *finitization*. Notons que cet effort est partiellement compensé par le fait que, d'une part, écrire des spécifications est une bonne pratique, et que d'autre part, l'utilisateur connaît déjà le langage dans lequel écrire ces méthodes et que les prédicats lui seront très probablement utiles

dans son programme.

UDITA. Le langage UDITA [Gligoric, Gvero, Jagannath, Khurshid, Kuncak, and Marinov, 2010] répond à l'objectif suivant : s'assurer que des structures de données complexes sont bien supportées par un programme Java, c'est à dire qu'un programme sait manipuler sans erreur toutes les formes d'une certaine structure de données non-scalaire, récursive, imbriquée etc. Pour atteindre cet objectif, UDITA se présente comme un langage étendu du langage Java permettant de décrire des structures de données complexes. Ces descriptions vont être utilisées pour générer des ensembles finis de ces structures. UDITA est la succession de Korat. Nous retrouvons ainsi la démarche de demander à l'utilisateur d'écrire lui-même des méthodes de génération de données (description des structures). En revanche, nous ne retrouvons plus les contrats. UDITA ajoute à Java deux extensions : des primitives de choix non-déterministe et une primitive pour restreindre ces choix. Ces constructions sont familières aux utilisateurs du vérificateur de modèle Java PathFinder [Visser, Havelund, Brat, Park, and Lerda, 2003], sur lequel UDITA s'appuie. UDITA est d'ailleurs intégré au *framework* Java PathFinder en tant qu'extension. Grâce aux points de choix non-déterministes, l'utilisateur ne décrit qu'une seule fois sa structure, et UDITA est capable d'en générer plusieurs. L'outil propose aussi à l'utilisateur des générateurs basiques, comme un générateur de booléens, d'entiers, de nouveaux objets ou d'objets déjà créés. Ces deux derniers permettent à l'utilisateur de décrire des structures de données récursives ou liées. En plus d'une description, UDITA demande à l'utilisateur d'écrire un filtre, notion identique aux prédicats de Korat, si ce n'est qu'UDITA ne va pas s'en servir pour la génération de données, mais uniquement pour filtrer les données générées : chaque donnée qui franchit le filtre sera mise de côté pour servir de donnée de test.
Contrairement à Korat, UDITA ne demande pas de méthode de *finitization*, mais s'appuie sur ces points de choix non-déterministes pour explorer l'espace de la structure tout en restant borné [Marinov, Andoni, Daniliuc, Khurshid, and Rinard, 2003; Sullivan, Yang, Coppit, Khurshid, and Jackson, 2004]. L'efficacité de la méthode s'appuie sur les choix retardés [Noll and Schlich, 2007], c'est à dire sur des évaluations non-déterministes tardives [Fischer, Kiselyov, and chieh Shan, 2009].

JSConTest. Le langage JSConTest [Heidegger and Thiemann, 2012] est un langage de contrat annotant des programmes Javascript. Ce dernier est un langage à typage non déclaré. C'est pourquoi JSConTest propose que le contrat ne spécifie que la signature de la fonction annotée, soient ses entrées et sa sortie, respectivement une précondition et une postcondition. JSConTest ne permet pas d'exprimer d'invariants ni de comportements. Le fait de préciser les types permet à JSConTest de réduire l'effort d'inférence en se limitant à un minimum de fonctions à analyser. Les résultats

de l'inférence servent à guider la génération aléatoire de données de test. Le langage permettant d'exprimer la signature est enrichi de quelques constructions, toujours pour guider la génération aléatoire. Un outil permet d'exécuter les tests et d'en calculer le verdict grâce à un RAC.

JSConTest, contrairement à JML ou UDITA, demande très peu d'efforts à l'utilisateur. Les auteurs pensent qu'écrire la signature ne peut pas être négatif pour le développeur, surtout dans un langage où les types ne sont pas déclarés. Cependant, le niveau d'expressivité sur les préconditions et postconditions reste très faible. En revanche, comme démontré dans le travail sur les types graduels [Siek and Taha, 2007], pour un langage à typage dynamique et faible comme Javascript ou PHP, même un contrat très simple qui spécifie le typage des données peut être très utile et offrir de bons résultats, notamment dans la détection de régressions.

Tous les outils de *Contract-based Testing* ne génèrent pas de données automatiquement. Quand ils génèrent des données, ce sont souvent une petite catégorie de données, comme les entiers, les réels et parfois les objets. La génération des données se fait à partir d'un solveur ou aléatoirement. La capacité des outils à générer des données automatiquement est liée à la façon dont les assertions sont exprimées. En JML, il est nécessaire d'utiliser un solveur, alors que JSConTest utilise de l'aléatoire mais sur tous les types natifs du langage. Des outils comme Korat ou UDITA permettent de générer des données plus finement mais demandent un effort plus important à l'utilisateur.

Praspel a sa propre manière de définir les assertions car nous voulons que chaque construction du langage puisse être utilisée pour de la validation **et** de la génération : tout ce que l'utilisateur peut écrire doit fonctionner dans ces deux contextes. JSConTest montre que l'utilisation de contrats simples spécifiant le typage des données offrent déjà de bons résultats, mais l'expressivité qu'il offre est très faible comparée à JML. Ainsi, Praspel doit permettre de spécifier plus que du typage simple et doit permettre de valider et générer des données de tout type, ou en tout cas, des types les plus manipulés par les développeurs au quotidien.

2.3 Génération de données

Dans Praspel, nous exprimons beaucoup plus que les types usuels mais nous n'exprimons pas des contraintes sous forme logique. Nous sommes alors intéressés par la génération (et la validation) de données au sens large. Cette partie présente tout d'abord des techniques de génération de données puis, orthogonalement, des outils qui génèrent des types de données spécifiques.

2.3.1 Technique de génération

Beaucoup de travaux et d'outils considèrent la génération de données aléatoires comme efficace pour trouver des erreurs. Elle permet de très rapidement explorer un modèle, une spécification ou une structure de code à moindre coût. Le *fuzzing* [Godefroid, Levin, and Molnar, 2012], par exemple, est une technique s'appuyant entre autre sur ces deux aspects : beaucoup de données différentes générées rapidement, qui permettent d'éprouver le programme. Malgré ses limitations évidentes (un test d'égalité entre deux entiers générés aléatoirement a une probabilité très faible d'être validé), la génération aléatoire est souvent un moyen de « déblayer le terrain » et de repérer les points non triviaux à tester du programme. C'est pourquoi plusieurs outils considèrent une approche hybride : utiliser de l'aléatoire pour éliminer les cas triviaux et éviter l'utilisation systématique de techniques de génération plus gourmandes. D'autres techniques vont guider l'aléatoire afin de conserver un certain degré de non-déterminisme tout en respectant par exemple des critères de couverture.

D'autres techniques travaillent de manière symbolique. Elles explorent et analysent le code de manière symbolique afin de constituer un système de contraintes. Ce dernier sera résolu à l'aide de solveurs de contraintes pour vérifier la consistence du code. Il est possible d'alterner une exécution symbolique avec une exécution concrète : le système de contraintes est résolu pour produire des données, qui sont utilisées pour exécuter le programme. Cette nouvelle exécution va permettre de constituer un nouveau système de contraintes. Le terme *concolic* est la contraction de *concrete* et *symbolic* et désigne ce genre de technique qui effectue des allers-retours entre une exécution symbolique et concrète.

Nous présentons quelques outils qui adoptent des approches symboliques et *concolic*.

EXE. *EXecution generated Executions*, abrégé EXE [Cadar, Ganesh, Pawlowski, Dill, and Engler, 2006, 2008b], est un outil d'exécution symbolique pour programmes C. Au lieu d'exécuter le programme avec des valeurs construites manuellement ou aléatoirement, EXE va exécuter le programme, préalablement instrumenté, de manière symbolique. Durant cette exécution symbolique, EXE va analyser et indexer toutes les contraintes sur les entrées symboliques (comme les paramètres des fonctions) et ainsi calculer tous les chemins possibles dans le programme. Si EXE rencontre une instruction contenant une valeur symbolique, alors il va analyser son environnement et générer de nouvelles contraintes. Quand des embranchements sont rencontrés (comme une condition par exemple), EXE va dupliquer les exécutions et les contraindre à emprunter chaque chemin de l'embranchement (pour une condition, ce sera un chemin à vrai et un à faux). Quand EXE arrive à la fin d'un che-

min ou qu'il détecte une erreur, alors il va automatiquement générer des données de test à partir des contraintes récoltées. EXE est accompagné de STP [Ganesh and Dill, 2007], un solveur de contraintes dédié conçu à l'origine pour EXE. C'est grâce à ce solveur que les valeurs vont être concrétisées et utilisées comme données de test sur le programme non-instrumenté. Les mêmes chemins seront empruntés et les mêmes erreurs seront trouvées, mais avec des valeurs concrètes, donc exploitables par le développeur.

KLEE. L'outil KLEE [Cadar, Dunbar, and Engler, 2008a] est la suite d'EXE. Ses auteurs font le constat suivant : malgré le fait que les travaux sur l'exécution symbolique soient encourageants, les outils ne sont toujours pas capables de passer à l'échelle. En effet, le nombre de chemins dans les programmes est exponentiel, et les techniques d'exécution symbolique ne permettent plus à résoudre toutes les contraintes générées. De même, l'exécution symbolique ne permet pas de prendre en compte l'environnement du programme, comme l'OS, le réseau ou l'utilisateur. Ce sont ces deux problèmes qu'adresse KLEE grâce à de la réduction et simplification de contraintes, des améliorations sur l'exploration des chemins et des optimisations de la représentation de l'environnement. KLEE utilise toujours STP comme solveur de contraintes.

DART. *Directed Automated Random Testing*, abrégé DART [Godefroid, Klarlund, and Sen, 2005], est également un outil de génération de données de test à partir de l'analyse du programme C. DART opère en trois étapes. Il commence par extraire les interfaces du programme pour connaître ses entrées. Ensuite il génère aléatoirement des données pour ses entrées, et exécute le programme. Enfin, il analyse le comportement du programme pendant son exécution pour détecter quels sont les chemins ou zones qui n'ont pas été atteints. Cette analyse va permettre de guider la génération des nouvelles entrées pour la prochaine exécution.
DART et EXE ont une approche similaire pour l'analyse des chemins. À chaque embranchement du programme, DART va extraire des contraintes symboliques. Pour générer les données de test suivantes, il prendra la négation des contraintes une par une, générera de nouvelles valeurs et le processus continuera. Toutefois, DART ne sait gérer que les contraintes sur les entiers alors qu'EXE sait gérer des contraintes sur les entiers, les tableaux, les vecteurs de bits, le transtypage, les opérations arithmétiques etc.

PathCrawler. L'outil PathCrawler [Williams, Marre, Mouy, and Roger, 2005; Botella, Delahaye, Ha, Kosmatov, Mouy, Roger, and Williams, 2009] est un générateur de tests structurels pour des programmes écrits en C, combinant, lui aussi, des exécutions symboliques et concrètes. Le test structurel vise à satisfaire certains cri-

tères de couverture de code [Miller and Maloney, 1963; Myers, 2004; Gargantini, Guarnieri, and Magri, 2012]. PathCrawler utilise son propre solveur de contraintes, appelé Colibri. Ce dernier est utilisé par d'autres outils de test et implémente des fonctionnalités avancées comme le support des réels ou de l'arithmétique modulaire sur les entiers. PathCrawler propose plusieurs stratégies de couverture de code, comme *all-*k*-paths* (tous les chemins atteignables avec au plus k itérations dans les boucles) et *all-paths* (tous les chemins atteignables sans limitation d'itérations dans les boucles). PathCrawler est correct, c'est à dire que chaque cas de test active un objectif de test (un chemin, une branche, une instruction etc.) pour lequel il a été généré. Cette propriété est validée par une exécution concrète. Quand toutes les fonctionnalités du programme testé sont supportées par PathCrawler et quand la génération de test, impliquant le solveur de contraintes, termine pour tous les chemins, l'outil est aussi complet, c'est à dire que l'absence de cas de test pour certains objectifs de test signifie que ces objectifs sont inatteignables. Cette propriété est possible car PathCrawler ne fait pas d'approximation pour les contraintes sur les chemins, comme c'est le cas dans d'autres outils de test *concolic*. PathCrawler a toutefois des limitations. Par exemple, il ne supporte pas les structures récursives, les fonctions récursives et les pointeurs sur les fonctions.

Nous pouvons citer CUTE [Sen, Marinov, and Agha, 2005], Pex [Tillmann and de Halleux, 2008], SAGE [Godefroid et al., 2012] ou ARTOO [Ciupa, Leitner, Oriol, and Meyer, 2008] qui sont aussi des outils de test *concolic*. Nous voyons qu'il existe plusieurs techniques avancées pour la génération de données de test à partir d'exécution symbolique ou concrète. Toutefois, si nous repensons aux contrats et que nous prenons du recul, nous remarquons que l'exécution symbolique ou concrète ne permet pas de valider qu'un programme fait bien ce que nous souhaitons. Les erreurs à l'exécution (comme des pointeurs nuls, des dépassements de tableaux, de tampons etc.) seront détectées, mais rien ne nous assure que le programme est correct par rapport à une spécification. Cependant, des travaux mélangent les exécutions symboliques ou concrètes de code avec les contrats, comme c'est le cas de SANTE ou Euclide.

SANTE. Les outils de test *concolic* sont efficaces pour valider des programmes mais passent difficilement à l'échelle sur des programmes de taille importante. Le prototype *Static ANalysis and TEsting* [Chebaro, Kosmatov, Giorgetti, and Julliand, 2012] répond à ce constat en utilisant la technique du *slicing*. L'idée est de supprimer du code inutile pour l'objectif de test fixé.
Nous mentionnons ces travaux car ils font un usage intéressant des contrats. SANTE propose de combiner PathCrawler avec des extensions Frama-C pour détecter des erreurs à l'exécution. SANTE est une extension à PathCrawler et travaille donc sur

des programmes écrits en C. Ces programmes sont annotés par des contrats écrits en ACSL. Il n'est pas tout à fait correct de parler de contrats car SANTE ne supporte qu'un petit sous-ensemble du langage ACSL : les assertions prévenant les divisions par zéro, les dépassements de tableaux et des débordements arithmétiques. Il est alors plus réaliste de parler d'assertions ACSL. Le code des programmes annotés a été simplifié par *slicing* pour faciliter le travail de PathCrawler. Avant de simplifier le code, les assertions ACSL sont compilées en code C (la démarche est similaire à JML qui est compilé en Java avec son RAC [Raghavan and Leavens, 2005]), et le programme est instrumenté pour les exécuter. Ainsi, PathCrawler prendra en considération des assertions ACSL lors de la génération de tests.

Euclide. L'outil Euclide [Gotlieb, 2009] permet de faire du test à partir de contraintes en mélangeant trois aspects : analyse structurelle du code source du programme, production de contre-exemples et preuve partielle du programme. L'objectif est de mélanger les techniques pour dépasser la limite de chacune et ainsi, de générer des données permettant de satisfaire plus de critères de couverture de code. Optionellement, Euclide peut utiliser des contrats écrits en ACSL pour exprimer des préconditions et des postconditions. Dans ce cas, Euclide est capable de prouver que le programme est valide et conforme par rapport aux contrats.

2.3.2 Données générées

Cette partie s'intéresse à des travaux sur la génération de données (parfois hors contexte, c'est à dire sans considérer un programme). Nos contributions se situent dans PHP, un langage essentiellement utilisé pour des programmes Web, c'est à dire manipulant majoritairement deux types de données : des chaînes de caractères et des collections (aussi appelées tableaux). Cette partie est découpée en fonction de ces deux types de données.

Génération de chaînes de caractères. Les chaînes de caractères sont intensivement utilisées dans les programmes Web. Leurs formes sont très différentes, allant d'une adresse email à des langages comme XML [W3C, 2006] ou JSON [ECMA, 2013], en passant par des requêtes SQL etc. La façon la plus répandue dans l'industrie pour représenter une donnée textuelle est d'utiliser une expression régulière ou une grammaire algébrique [Chomsky, 1956]. Une grammaire est exprimée à partir d'un langage de description qui déclare des lexèmes, unités lexicales atomiques d'un langage, et des entités syntaxiques (aussi appelées règles) exprimant l'enchaînement possible des lexèmes les uns par rapport aux autres.
L'idée commune que nous retrouvons dans la majorité des travaux est d'explorer ces règles afin de générer une séquence de lexèmes qui représente une donnée

valide. Quand les grammaires sont utilisées pour représenter des données de test, nous parlons de *Grammar-based Testing*, abrégé GbT.
yet another generator-generator, abrégé yagg [Coppit and Lian, 2005] est un outil de génération de données textuelles complexes à partir d'une grammaire décrite avec une syntaxe très similaire à LEX [Lesk and Schmidt, 1975] et YACC [Johnson, 1975], des outils d'analyse syntaxique et lexicale très largement utilisés. Ce choix de syntaxe est montré comme plus attractif et plus efficace pour les utilisateurs comparé à TestEra à besoin équivalent. Quand est traité le problème de génération de données structurelles complexes, notamment avec des grammaires, la question de la combinatoire se pose forcément. Certains opérateurs de répétition de séquences de lexèmes, comme + (une ou plusieurs fois) ou * (zéro ou plusieurs fois) n'ont pas de bornes supérieures. Afin d'éviter la génération d'énormes données ou de données explosives, yagg propose de borner ces opérateurs. yagg a une approche exhaustive, c'est à dire qu'il va énumérer toutes les séquences possibles de lexèmes. Les lexèmes sont exprimés avec un sous-langage des expressions régulières qui ajoute des constructions comme l'alternance équivalente ou des générateurs d'équivalence. Ce choix sert également à borner les valeurs possibles des lexèmes.
L'outil Geno [Lämmel and Schulte, 2006] propose une solution au problème de l'explosion combinatoire. Certains travaux proposent des critères de couverture sur les grammaires ou d'annoter des grammaires avec par exemple des poids probabilistes afin de guider l'exploration de la grammaire [Lämmel, 2001; Maurer, 1990; Sirer and Bershad, 1999; McKeeman, 1998]. Geno propose à l'utilisateur d'annoter une grammaire avec des mots-clés afin de guider la génération. Ces mots-clés sont une abstraction de choix probabilistes, *a priori* plus simples et concrets. Ces mots-clés sont traités de façon à conserver une certaine exhaustivité dans les résultats. Similairement, l'outil YouGen [Hoffman, Ly-Gagnon, Strooper, and Wang, 2011] propose à l'utilisateur d'ajouter des *tags* pour borner l'exploration de plusieurs règles en appliquant des réductions *pair-wises*.
La plupart des techniques ayant une approche aléatoire ou exhaustive bornée se basent respectivement sur les travaux de Howden [1986] et Flajolet, Zimmermann, and Cutsem [1994]. La démarche est très souvent similaire : à partir d'une grammaire, il est possible de générer une donnée. Mais des annotations sont ajoutées pour guider cette génération afin d'obtenir des données plus réalistes selon le contexte d'utilisation. Ces outils demandent toujours une intervention du développeur mais ces interventions tendent à devenir de plus en plus simples.

Génération de tableaux. Les tableaux sont aussi très utilisés dans le Web pour représenter toutes sortes de collections ou organiser de manière structurelles des informations. Plusieurs solveurs supportent les tableaux en plus d'autres données, comme c'est le cas de STP (pour EXE et KLEE), Colibri (pour PathCrawler), Alcoa

(pour Alloy), le solveur de l'outil Euclide etc. Toutefois, nous pouvons citer FDCC qui est un solveur dédié uniquement au tableau.
FDCC [Bardin and Gotlieb, 2012] est un solveur pour les tableaux avec contraintes sur des domaines finis. Les auteurs font le constat que les tableaux sont omniprésents dans les outils de vérification, mais que des approches efficaces pour les traiter sont assez rares dans la programmation par contraintes. Les auteurs proposent alors une approche combinant deux solveurs, l'un raisonnant de manière symbolique (pour connaître les accès, les écritures et la taille des tableaux), l'autre manipulant des contraintes sur les domaines finis. La partie originale et difficile est la communication bi-directionnelle entre ces deux outils.
Mentionnons aussi l'outil Minion [Gent, Jefferson, and Miguel, 2006] qui fait un constat et une approche similaire à FDCC : les auteurs préfèrent coupler des solutions ensemble et les faire communiquer. Les résultats sont très encourageants : comme pour FDCC, le solveur converge beaucoup plus rapidement et est capable de répondre à davantage de problèmes.
La partie délicate est toujours de faire communiquer les solveurs entre eux. En effet, chaque solveur a son propre formalisme et son propre langage de description de contrainte. Ainsi, le travail de l'outil Minion consiste à proposer un langage commun entre tous ces solveurs, et de faire des traducteurs pour chacun d'eux. Cette phase de transformation fait intervenir des filtres, et donc engendre, potentiellement, des pertes d'informations. Par conséquent, même si les solveurs sont capables de trouver des solutions à nos problèmes, ces solutions sont difficilement compréhensibles par le programme d'origine car beaucoup d'informations ont été transformées ou perdues (car non nécessaires aux solveurs mais nécessaires pour la tracabilité des données).

2.4 Synthèse

Dans ce chapitre, nous avons décrit des langages de contrat de plusieurs langages de programmation. Nous avons remarqué que les langages majeurs ont leur propre langage de contrat. Ce n'est pas le cas de PHP. Nous avons vu par la suite comment ces langages de contrat étaient utilisés pour générer des suites de tests. Cette technique s'appelle le *Contract-based Testing* : les invariants et les préconditions sont utilisés pour générer des données de test et les postconditions fournissent un oracle permettant de calculer le verdict du test. Un RAC (*Runtime Assertion Checker*) permet de valider ce verdict à l'exécution. Enfin, nous avons vu d'autres techniques de génération de données notamment pour des chaînes de caractères et des tableaux.
Certains des outils ne respectent pas vraiment le *Contract-based Testing* car ils demandent à l'utilisateur de définir des données manuellement. Toutefois, ils offrent un gain de temps non négligeable par la production d'une suite de tests exécu-

table. D'autres outils génèrent eux-mêmes les données mais à l'aide de solveurs de contraintes car les assertions à l'intérieur des contrats ou dans le code du programme sont exprimées avec des prédicats ou des expressions similaires. Malgré le niveau d'expressivité important que cela offre, il est difficile pour les solveurs de générer des données satisfaisant toutes ces contraintes ou de passer à l'échelle, c'est à dire d'appliquer ses techniques sur de très grands programmes. Ainsi, les données de test générées vont permettre de détecter les erreurs à l'exécution mais rien n'assure que le programme fait bien ce que l'utilisateur en attend.
Nous remarquons qu'il existe plusieurs techniques et plusieurs approches. Malheureusement, malgré leur intérêt, leur « éparpillement » les rend inutilisables pour un ingénieur de test car la chaîne de développement deviendrait trop complexe. Une solution est alors de les assembler de manière cohérente.
Nous proposons Praspel, un langage de spécification pour PHP à partir de contrats. Ce langage trouve son inspiration dans JML ou ACSL. Les outils de Praspel sont capables de générer automatiquement des suites de tests exécutables à partir de contrats. Les données de test seront générées automatiquement. Un juste milieu a été trouvé pour exprimer les assertions à l'intérieur des contrats, supportant la validation et la génération pour toutes les constructions. C'est un langage simple (pour l'utilisateur), pragmatique et suffisament souple pour assembler plusieurs méthodes du domaine du test, comme nous le verrons dans les chapitres suivants.

Chapitre 3.
Langage de spécification

Table des matières

Les domaines réalistes sont des structures permettant de valider et générer des données. Présentés dans la partie 3.1, ils sont un composant essentiel de Praspel, un nouveau langage de spécification pour PHP, présenté dans la partie 3.2.

3.1 Domaines réalistes

Un des moyens les plus répandus pour caractériser les données manipulées par un programme est d'utiliser un système de **typage**. Il en existe grossièrement deux grandes familles : **statique** et **dynamique**, que nous pouvons comprendre comme

types déclarés syntaxiquement ou calculés lors de la compilation ou de l'exécution. Dans tous les cas, le type d'une donnée est connu au moment de l'exécution du programme. Le typage permet de **vérifier** la cohérence entre les données manipulées. Ainsi, la plupart des systèmes de typage ne permettent pas la soustraction d'une chaîne de caractères avec un entier, sauf si ce système est dit faible. Prenons l'exemple de PHP qui a un système de typage dynamique (déduit à l'exécution) et faible : la soustraction de l'entier `4` avec la chaîne de caractères `'1.2'` produit le réel `2.8`. Dans ce cas, le langage opère implicitement un transtypage, c'est à dire qu'à partir de règles internes il transforme les données vers un autre type.
Notons que dans un langage comme PHP où les types ne sont pas déclarés syntaxiquement, il est culturellement admis d'écrire une documentation API qui décrit la nature des données manipulées par le programme [PHP Documentor], comme leurs types, par exemple `/** @param int i An integer */`.
Nous profitons du fait que les développeurs connaissent bien les systèmes de typage grâce à leur présence dans de nombreux langages de programmation pour introduire la notion de **domaine réaliste**. Naïvement, nous pouvons dire qu'ils raffinent les types usuels comme les entiers, les chaînes de caractères, les tableaux etc., et qu'ils permettent d'exprimer des données plus complexes, comme des grammaires, des graphes etc. Le mot **réaliste** signifie qu'ils sont conçus pour spécifier des domaines de données pertinents pour un contexte spécifique. Par exemple, une adresse email peut être un domaine réaliste. En effet, plusieurs logiciels identifient leurs utilisateurs avec leur adresse email. C'est plus qu'une chaîne de caractères : il y a certaines règles, notamment syntaxiques, à respecter.

Définition 3.1 (Caractéristiques d'un domaine réaliste)**.** Les domaines réalistes ont été conçus afin de répondre aux problématiques du test. C'est pourquoi ils ont les deux caractéristiques suivantes :
– la **prédicabilité** qui permet de valider qu'une donnée appartient à l'ensemble des valeurs décrites par le domaine réaliste ;
– la **générabilité** qui permet de générer une valeur décrite par le domaine réaliste.

3.1.1 Implémentation

Nous proposons une implémentation des domaines réalistes en PHP. Dans cette implémentation, un domaine réaliste est représenté par une classe. Ses propriétés de prédicabilité et de générabilité sont respectivement représentées par les méthodes suivantes :
– `predicate($q)`, où `$q` est la valeur à valider ;
– `sample($sampler)`, où `$sampler` est un **générateur numérique** : il génère uniquement des entiers et des réels, et permet de guider la génération d'une donnée ;

par exemple nous allons utiliser ce générateur numérique pour choisir un caractère parmi une plage de caractères donnée. Pour l'instant, le seul générateur numérique disponible est pseudo-aléatoire.

3.1.2 Hiérarchie et univers

Les domaines réalistes étant implémentés à travers des classes et PHP étant orienté objet, nous pouvons appliquer de l'**héritage**. Par conséquent, un domaine réaliste enfant peut hériter et raffiner les propriétés de son parent. Par exemple, le domaine réaliste `Color` (qui représente des couleurs) hérite de `String` (qui représente des chaînes de caractères), et le domaine réaliste `Boundinteger` (qui représente des intervalles d'entiers) hérite de `Integer` (qui représente l'ensemble des entiers de la machine). Tous les domaines réalistes doivent hériter de la classe `Realdom`, qui assure le fonctionnement des domaines réalistes (en proposant plusieurs fonctionnalités usuelles et partagées, comme la gestion des paramètres, des arguments, du générateur numérique etc.).
La figure 3.1 montre un début d'implémentation du domaine réaliste `Boundinteger`. Il hérite du domaine réaliste `Integer` en (1). Nous remarquons aussi de nombreuses interfaces pour caractériser la « forme » ou les propriétés du domaine réaliste (détaillé dans la partie 3.1.4). La propriété de prédicabilité raffine celle du parent en (2), et ajoute de nouvelles contraintes en (3) qui vérifient que `$q` appartient à l'intervalle décrit par le domaine réaliste.
Les domaines réalistes forment ensemble un univers, appelé $\mathcal{U}_{\text{realdom}}$. Les couches de cet univers sont présentées dans la figure 3.2. La couche 0, la plus basse, ne contient que le domaine réaliste `undefined`, qui reconnaît toutes les données et produit seulement des entiers. La couche supérieure 1 contient les domaines réalistes scalaires (les booléens, les entiers, les réels et les chaînes de caractères) et non-scalaires (les tableaux et les classes). La couche supérieure 2 est la bibliothèque standard, présentée dans la partie 6.1.1 page 105. Enfin, la couche 3, la dernière, contient les domaines réalistes définis par les utilisateurs.

3.1.3 Paramètres

A l'instar des fonctions, les domaines réalistes sont paramétrables. Les données reçues par le domaine réaliste sont appelées ses **arguments**. Les paramètres sont très utiles pour représenter des structures de données complexes, telles que des tableaux imbriqués, des graphes, des automates, des arbres etc. Tous sont des imbrications de types scalaires et structurels.

Exemple 3.1 (Paramétrage d'un domaine réaliste)**.** Par exemple, le domaine réaliste `string(0x61, 0x7a, boundinteger(4, 12))` représente une chaîne de caractères

```
class          Boundinteger
    extends    Integer                                        /* (1) */
    implements IRealdom\Interval,  IRealdom\Finite            /* (5) */
               IRealdom\Nonconvex, IRealdom\Enumerable {

    protected $_arguments = [
        'Constinteger lower' => PHP_INT_MIN,                  /* (4) */
        'Constinteger upper' => PHP_INT_MAX
    ];

    public function predicate ( $q ) {

        return    parent::predicate($q)                       /* (2) */
               && $q >= $this['lower']->getConstantValue()    /* (3) */
               && $q <= $this['upper']->getConstantValue();
    }

    public function sample ( $sampler ) {

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

    // ...
}
```

FIGURE 3.1 – Début d'implémentation du domaine réaliste `Boundinteger`.

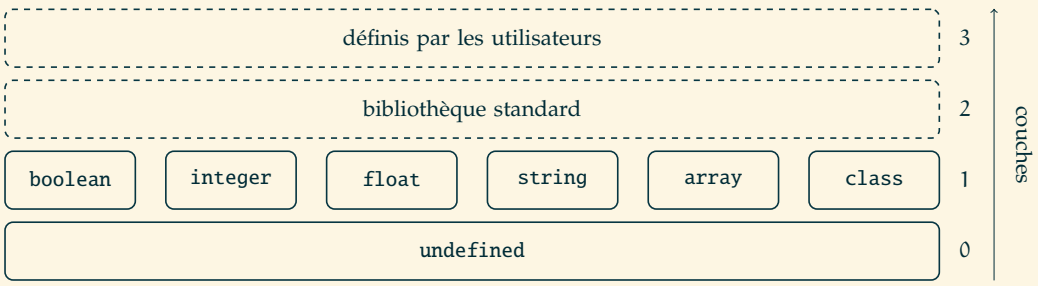


FIGURE 3.2 – Univers des domaines réalistes.

dont la taille est comprise entre 4 et 12 et dont les code-points (Unicode) des caractères sont entre `0x61` et `0x7a`.

La figure 3.1 montre la déclaration de deux paramètres : `lower` et `upper` en (4), qui ont pour domaine réaliste `Constinteger`, avec respectivement en valeur par défaut `PHP_INT_MIN` et `PHP_INT_MAX` qui sont des constantes de PHP. Si aucune valeur par défaut n'est précisée, alors le paramètre est obligatoire, sinon il est optionnel.
Quand nous décrivons les paramètres d'un domaine réaliste, nous pouvons les filtrer. Le filtre est exprimé avec un ou des noms de domaines réalistes, séparés par le symbole |. En effet, un domaine réaliste accepte plusieurs sortes d'arguments et est capable de faire la traduction d'un domaine réaliste vers un autre. Ainsi, nous restons proches de la philosophie de PHP et de son typage dynamique.

Exemple 3.2 (Multi-typage des paramètres)**.** Nous présentons un exemple de multi-typage de paramètres avec le domaine réaliste `String`, dont les deux premiers paramètres peuvent être des entiers ou des caractères, déclarés comme suit :

```
protected $_arguments = [
    'Constinteger|Conststring codepointMin' => 0x20,
    'Constinteger|Conststring codepointMax' => 0x7e,
    'Integer                  length'
];
```

où `Constinteger` et `Conststring` représentent respectivement un entier et une chaîne de caractères Praspel comme `7` ou `'foo'`. Ainsi, nous pouvons écrire `string('a', 'z', 4..12)` sans produire d'erreur, mais `string(true, 'z', 4..12)` produira une erreur car `true` est une valeur du domaine réaliste `Constboolean` et le paramètre `codepointMin` n'accepte que les domaines `Constinteger` ou `Conststring`.
Nous voyons que les deux premiers paramètres sont optionnels et que seul le dernier est obligatoire. Pour utiliser la valeur par défaut d'un paramètre, nous pouvons utiliser le mot-clé `default` (ou `...`). Ainsi, nous pouvons écrire : `string(default, default, 7)` (ou `string(..., ..., 7)`).
Notons également que les chaînes de caractères de Praspel (représentées par le domaine réaliste `Conststring`) sont au format Unicode (UTF-8), ce qui nous permet d'écrire : `string('α', 'ω', 4..12)` pour représenter un mot en minuscule écrit en grec.

3.1.4 Classification

L'héritage que nous venons de voir en 3.1.2 est une classification. Une autre classification transversale est possible grâce aux interfaces. Elles aident à caractériser les domaines réalistes. Les interfaces les plus intéressantes sont :

- `Constant`, pour représenter un domaine réaliste immuable avec seulement une valeur, comme `42` (du domaine réaliste `Constinteger`), `true` (du domaine réaliste `Constboolean`) etc. ;
- `Interval`, pour représenter un intervalle à travers une borne inférieure et supérieure, qui peuvent être réduites dynamiquement (à l'exécution) ;
- `Finite`, pour représenter un domaine réaliste avec un ensemble de valeurs fini ;
- `Nonconvex`, pour représenter un domaine réaliste avec des trous, *i.e.* où des valeurs ont été exclues ;
- `Enumerable`, pour représenter un domaine réaliste dont les valeurs peuvent être énumérées.

Ainsi, par exemple, dans la figure 3.1 en (5), nous voyons que le domaine réaliste `Boundinteger` implémente les interfaces `Interval`, `Finite`, `Nonconvex` et `Enumerable`.

3.2 Praspel

Praspel est un acronyme anglais signifiant *PHP Realistic Annotation and SPEcification Language*. C'est un langage et un *framework* de test à partir de contrats en PHP. Ce langage repose sur les domaines réalistes.

Praspel est un **langage d'annotation** car il s'écrit dans les commentaires du code source du programme. PHP comprend trois catégories de commentaires : en ligne (avec `//` ou `#`), multi-lignes (entre `/*` et `*/`) et en bloc (entre `/**` et `*/`). Cette dernière catégorie est culturellement dédiée à l'écriture des annotations, documentations API etc. [PHP Documentor], et est par conséquent celle où nous écrirons Praspel.

Le langage Praspel est un *Behavioral Interface Specification Language* (**BISL**) basé sur les contrats. Un contrat est un modèle du comportement du code, décrit à travers des contraintes formelles, appelées clauses, comme les préconditions, les postconditions et les invariants. Ces contraintes sont généralement localisées dans le code source autour des données. Dans le cas de Praspel, les invariants référencent les attributs des classes et les préconditions et postconditions référencent les paramètres des méthodes et les attributs des classes. La sémantique d'un contrat est la suivante :

- l'appelant de la méthode s'engage à satisfaire la précondition ;
- seulement dans ce cas, la méthode appelée s'engage à satisfaire sa postcondition ;
- les invariants, quant à eux, doivent être satisfaits avant et après l'exécution de la méthode.

Nous décrivons ci-après les parties de la grammaire de Praspel en forme normale. La grammaire complète se trouve dans l'annexe 1 page 156.

specification ::= *attribute-clauses* | *method-clauses*
attribute-clauses ::= *invariant-clause*$^?$
method-clauses ::= (*description-clause* `;`)$^?$
rbdet-clauses
rbdet-clauses ::= (*requires-clause* `;`)$^?$
(
(*behavior-clause*$^+$ *default-clause*$^?$)$^?$
| (*ensures-clause* `;`)$^?$ (*throwable-clause* `;`)$^?$
)

FIGURE 3.3 – Entités syntaxiques Praspel de plus haut niveau.

invariant-clause ::= `@invariant` *expression*
requires-clause ::= `@requires` *expression*
behavior-clause ::= `@behavior` `php-identifier` `{`
(*description-clause* `;`)$^?$
rbdet-clauses
`}`
default-clause ::= `@default` `{`
(*description-clause* `;`)$^?$
(*ensures-clause* `;`)$^?$
(*throwable-clause* `;`)$^?$
`}`
ensures-clause ::= `@ensures` *expression*
throwable-clause ::= `@throwable` *exceptional-expression*
description-clause ::= `@description` `php-string`

FIGURE 3.4 – Grammaire de Praspel : entités syntaxiques des clauses.

3.2.1 Clauses

Un contrat est composé de clauses, dont la syntaxe est décrite dans les figures 3.3 et 3.4. Dans ces figures, le style `token` représente un lexème de la grammaire de Praspel, *rule* représente une entité syntaxique (règle) de la grammaire, et `php-token` représente un lexème de la grammaire de PHP (pour être le plus proche possible du langage manipulé par le développeur). La notation e_s^r signifie que le motif e est répété r fois, et séparé par s. r peut être $?$, $+$ ou $*$, respectivement pour 0 ou 1 fois, 1 ou plusieurs fois et 0 ou plusieurs fois. Si s n'est pas vide, alors s doit être un lexème. Ainsi, la construction $\texttt{a}_{\texttt{.}}^{+}$ reconnaîtra `a`, `a.a`, `a.a.a` etc., mais pas `a.a.` car ici le lexème `.` n'est pas utilisé comme séparateur.

La figure 3.3 montre les familles de clauses, dont le contenu est décrit par la figure 3.4. Les entités syntaxiques *attribute-clauses* et *method-clauses* définissent respectivement les clauses annotant les attributs d'une classe et ses méthodes. L'entité syntaxique *method-clauses* montre l'ordre des clauses en forme normale. La grammaire réelle de Praspel est bien plus souple : elle accepte des clauses dans n'importe quel ordre. Mais, pour rester simple, nous ne présenterons et ne travaillerons que sur une forme normale.

La figure 3.5 montre la localisation typique des clauses sur une classe `C` avec un attribut `a` et une méthode `f`. La clause `@invariant` est localisée avant l'attribut `a`. Les autres clauses sont des clauses de méthode. Elles sont localisées avant l'entête des méthodes. La plupart de ces clauses introduisent des expressions Praspel, définies par les entités syntaxiques *expression* et *exceptional-expression*, détaillées dans la partie 3.2.2.

La précondition d'une méthode est exprimée à l'aide de la clause `@requires`, la postcondition à l'aide de la clause `@ensures`. La précondition exprime des contraintes sur le **pré-état** du système, alors que la postcondition exprime des contraintes sur le pré-état et le **post-état** du système. Afin de modifier l'état du système, il doit être exécuté en appelant une méthode. La clause `@ensures` représente une **postcondition normale**, mais le système peut aussi lever une exception et être alors placé dans un **post-état exceptionnel**.La clause `@throwable` exprime des contraintes sur cet état particulier : elle exprime les conditions sous lesquelles des exceptions peuvent être levées par la méthode et aussi l'état du système associé. La syntaxe de T ($T_{\alpha.\beta}$ et $T_{\mathcal{D}}$ dans la figure 3.5) est définie par l'entité syntaxique *exceptional-expression* dans la figure 3.7. T est de la forme T_C `with` T_E où T_C est une liste de noms de classes asssociée à un identifiant et T_E est une expression, appelée **postcondition exceptionnelle**, dont la syntaxe est détaillée dans la partie 3.2.2. Tous les identifiants définis dans T_C peuvent apparaître dans T_E. La sémantique est la suivante : si l'exception levée est une instance d'une classe représentant une exception listée dans T_C, alors elle est affectée à l'identifiant associé et la postcondition exceptionnelle T_E doit être

```
class C {

    /**
     * @invariant I;
     */
    protected $a;

    /**
     * @requires P;
     * @behavior α {
     *     @description 'Apply the α process.';
     *     @requires P_α;
     *     @behavior β {
     *         @requires  P_α.β;
     *         @ensures   Q_α.β;
     *         @throwable T_α.β;
     *     }
     * }
     * @behavior γ {
     *     @requires P_γ;
     *     ...
     * }
     * @default {
     *     @description 'Apply the fallback process.';
     *     @ensures   Q_D;
     *     @throwable T_D;
     * }
     */
    public function f ( ) { }
}
```

FIGURE 3.5 – Un contrat Praspel typique avec toutes les clauses.

satisfaite.
Une méthode peut avoir différents **comportements** relatifs à ses arguments et à l'état du système. Praspel propose la clause `@behavior` pour représenter un comportement. Un comportement est identifié par un nom, et peut contenir une description informelle, grâce à la clause `@description` ; ces deux informations peuvent s'avérer très utiles pour offrir un retour à l'utilisateur. Les clauses `@requires`, `@ensures`, `@throwable` et `@behavior` elle-même peuvent apparaître à l'intérieur d'une clause `@behavior`. Cette structure décrit des comportements imbriqués, comme illustré dans la figure 3.5, dans laquelle le comportement β est imbriqué dans le comportement α. Nous pouvons également décrire des comportements alternatifs en juxtaposant des clauses `@behavior`. Ceci est illustré dans la figure 3.5 avec le comportement γ, frère du comportement α. Les comportements sont, dans la pratique, mutuellement exclusifs. Cependant, la syntaxe autorise l'écriture de comportements non-exclusifs.
La clause `@default` est strictement équivalente à une clause `@behavior` avec une clause `@requires` implicite décrivant la conjonction de toutes les négations des clauses `@requires` des comportements frères précédents. Par exemple, dans la figure 3.5, la clause `@requires` de `@default` pourrait s'écrire $\neg P_\alpha \wedge \neg P_\gamma$.
Afin d'« activer » un comportement donné, sa clause `@requires` doit être satisfaite. Après l'exécution, le système est dans le post-état. Nous faisons alors face à deux situations. Si c'est un post-état normal, alors la clause `@ensures` du dernier comportement activé doit être satisfaite. Si c'est un post-état exceptionel, alors c'est la clause `@throwable` qui doit être satisfaite, *i.e.* les deux T_C et T_E doivent être satisfaits, comme décrit précédemment.
Quand une clause `@requires` est manquante et que le SUT a besoin de paramètres, une erreur sera levée car les paramètres seront non-spécifiés. Une clause `@ensures` (respectivement `@invariant`) manquante revient à accepter tous les post-états normaux (respectivement tous les invariants de classes). Une clause `@throwable` manquante revient à refuser tous les post-états exceptionels.

Exemple 3.3 (Contrat d'un système de fichier)**.** La figure 3.6 montre l'exemple concret d'une classe `Filesystem` annotée, avec deux méthodes `getUsage` et `store`. Cet exemple sera utilisé pour illustrer chaque notion introduite dans la suite de cette partie.
La méthode `getUsage` est déclarée comme `pure`, ce qui veut dire qu'elle ne modifie pas son environnement. Elle n'a pas de clause `@requires`, donc elle acceptera tous les arguments. Elle a une postcondition que nous décrirons dans les parties suivantes. La méthode `store` a deux comportements : un comportement `full` quand le système de fichier est plein, et un comportement par défaut. Le premier comportement décrit une précondition et une postcondition uniquement exceptionnelle (car

```
class Filesystem {

    const SIZE = 1024;

    /**
     * @invariant _usage: boundinteger(0, static::SIZE);
     */
    protected $_usage = 0;

    /**
     * @invariant _map: array([to class('File')], 0..0xffff);
     */
    protected $_map = array();

    /**
     * @ensures \result: this->_usage;
     */
    public function getUsage ( ) { /* ... */ }

    /**
     * @requires file: class('File') and
     *           index: 0..0xffff or void;
     * @behavior full {
     *     @description 'The filesystem is full.';
     *     @requires  \pred('$this->getUsage() + $file->getSize()
     *                          > static::SIZE');
     *     @ensures   \none;
     *     @throwable AllocationException e with
     *                    file->isAttached(): false and
     *                    e->getFilesystem(): this;
     * }
     * @default {
     *     @ensures file->isAttached(): true and
     *              \result: boolean();
     * }
     */
    public function store ( File $file, $index = null ) { /* ... */ }
}
```

FIGURE 3.6 – Exemple d'une classe `Filesystem` annotée par des contrats.

expression	::=	`\none`
		| (*declaration* `and`)*
		(*predicate* `and`)*
		qualification$^*_{\texttt{and}}$
exceptional-expression	::=	((*exception-identifier*)$^+_{\texttt{or}}$
		`with` *expression*)$^+_{\texttt{or}}$
exception-identifier	::=	`php-classname` `php-identifier`

FIGURE 3.7 – Grammaire de Praspel : les entités syntaxiques d'expressions.

`@ensures` refuse tous les post-états normaux à cause de `\none`, nous y reviendrons). Ici, seule l'exception `AllocationException` peut être levée. Elle sera associée à l'identifiant `e`, puis le reste de la clause devra être valide. Dans le second comportement (par défaut), aucune exception ne peut être levée car la clause `@throwable` est absente. Nous pouvons voir également que les attributs `_usage` et `_map` sont annotés d'invariants avec la clause `@invariant`.

3.2.2 Expressions

Les figures 3.7 et 3.8 décrivent les expressions Praspel. Nous avons principalement trois sortes d'expressions : déclarations, prédicats et qualifications, respectivement présentées dans les parties suivantes.

Déclarations

Une **déclaration** associe un ou plusieurs domaines réalistes à une variable, à travers l'opérateur `:`. Une variable est soit un attribut de classe, spécifié dans une clause `@invariant`, soit un paramètre de méthode, spécifié dans les autres clauses. La valeur d'une variable peut appartenir à plusieurs domaines réalistes si la variable est définie par une **disjonction** de domaines réalistes, représentée par le mot-clé `or`. Dans l'exemple de la figure 3.6, la variable `index` du contrat de la méthode `store` peut avoir deux valeurs : soit un entier entre 0 et 65535, soit `null`. Une disjonction de domaines réalistes peut contenir n'importe quelle sorte de domaine réaliste, conformément à l'aspect dynamique de PHP.
Par défaut, les variables utilisées dans le contrat appartiennent au système, c'est à dire qu'elles existent en tant qu'argument d'une méthode ou attribut de classe. Toutefois, il est possible de déclarer une variable qui appartient au modèle si elle est précédée par le mot-clé `let`. Nous pouvons voir cette variable comme étant locale à la spécification. Cette catégorie de variables est très utile pour manipuler des expressions apparaissant par exemple plusieurs fois.

declaration	::=	`let`$^{?}$ *extended-identifier* `:` *disjunction*
predicate	::=	`\pred(` `php-string` `)`
qualification	::=	*identifier* `is` `php-identifier`$^{+}_{,}$
disjunction	::=	(*constant* | *realdom* | *extended-identifier*)$^{+}_{\texttt{or}}$
realdom	::=	`php-identifier` `(` *argument*$^{?}_{,}$ `)`
argument	::=	`default` | *realdom* | *constant* | *array* | *extended-identifier*
constant	::=	*scalar* | *array*
scalar	::=	`null` | `php-boolean` | *number* | `php-string` | *range*
number	::=	`php-binary` | `php-octal` | `php-hexa` | `php-decimal`
range	::=	*number* `..` *number*
array	::=	`[` *pair*$^{?}_{,}$ `]`
pair	::=	`from`$^{?}$ *disjunction* `to` *disjunction* | `to`$^{?}$ *disjunction*
extended-identifier	::=	*array-access*
array-access	::=	*identifier* (`[` *scalar* `]`)$^{?}$
identifier	::=	`php-identifier` | `this` (`->` `php-identifier`)* | (`self` | `static` | `parent`) (`::` `php-identifier`)$^{+}$ | `\old(` *extended-identifier* `)` | `\result`

FIGURE 3.8 – Grammaire de Praspel : entités syntaxiques de construction des expressions.

La variable spéciale `\result` représente la valeur **retournée** par la méthode. Elle ne peut apparaître que dans la clause `@ensures`. Les clauses `@ensures` et `@throwable` peuvent également faire référence à la valeur d'une variable dans le pré-état grâce à la construction `\old(i)`, où `i` est le nom d'une variable.

Prédicats

Toutes les constructions décrivant les données dans Praspel supportent deux aspects : la validation et la génération, à travers les caractéristiques de prédicabilité et de générabilité des domaines réalistes. Chaque concept introduit dans le langage doit supporter ces deux aspects. Mais il arrive qu'il soit impossible d'exprimer certaines contraintes avec les constructions actuelles. C'est pourquoi nous avons la construction « boîte noire » `\pred(p)`, où `p` est du code PHP, pouvant contenir les constructions `\result` et `\old(i)`. Cette construction `\pred(p)` permet à l'utilisateur d'exprimer des contraintes arbitraires en utilisant PHP lui-même au lieu de Praspel. Le code `p` doit être un prédicat en forme normale **disjonctive** (DNF). La construction `\pred(p)` supporte la validation mais pas la génération. Toutes les variables présentes dans `p` doivent avoir un domaine réaliste associée (en dehors de la construction `\pred(p)`). Le prédicat `\pred('$this->getUsage() + $file->getSize() > static::SIZE')` dans la figure 3.6 signifie que s'il n'y a plus de place pour ajouter un nouveau fichier, alors cette partie de la précondition est satisfaite.
Le mot-clé `\none` est strictement équivalent à `\pred('false')`. Ainsi, écrire `@ensures \none;` signifie que tous les post-états normaux seront refusés car la postcondition normale sera toujours fausse. Ce mot-clé est plus qu'une équivalence car, selon la grammaire dans la figure 3.7, il empêche l'utilisation de déclarations, d'autres prédicats ou de qualifications.

Qualifications

Praspel permet de qualifier une donnée en utilisant le mot-clé `is` suivi d'une liste d'adjectifs. Par exemple, dans le cas d'un tableau `a`, nous pouvons lui attribuer les adjectifs `a is unique, sorted` pour préciser que toutes ces valeurs sont uniques et triées. Un exemple de qualification est présenté dans la partie 4.2 page 48.

Identifiants

Dans la figure 3.6 ainsi que dans la règle *identifier* de la grammaire de la figure 3.8, pour manipuler un identifiant, nous avons les mots-clés spéciaux `this`, `self`, `static` et `parent`, avec deux opérateurs différents `->` et `::`. Tout d'abord, pour un accès sur un objet (accès dynamique), nous utilisons l'opérateur `->`, sinon, pour un accès sur

une classe (accès statique), nous utilisons l'opérateur `::`. Nous faisons référence à l'objet courant avec le mot-clé `this`, et nous faisons référence à la classe courante avec les mots-clés `self` et `static`. Enfin, nous faisons référence à la classe parente avec le mot-clé `parent`.
La différence entre `self` et `static` est que `self` fera toujours référence à la classe qui déclare, alors que `static` fera référence à la classe qui utilise. Le mot-clé `static` est un équivalent à un `this` statique. Ce mécanisme est appelé le *Late Static Binding* [1].

Exemple 3.4 (Différence entre `self` et `static`)**.** Soit une classe `BigFilesystem` qui hérite de `Filesystem` et redéfinit uniquement la constante `SIZE` :

```
class BigFilesystem extends Filesystem {

    const SIZE = 1048576; // 2^20
}
```

Supposons que la méthode `getUsage` de `Filesystem` utilise la constante avec `self::SIZE`. Alors, lorsque nous appelons la méthode `getUsage` depuis `Filesystem`, ce sera la constante `SIZE` de `Filesystem` qui sera utilisée. La même chose se produit lorsque nous appelons la méthode `getUsage` depuis `BigFilesystem` malgré le fait que `BigFilesystem` ait sa propre constante `SIZE`, car `self` fait référence à la classe qui déclare. Dans ce cas, ce n'est pas le comportement attendu. Si maintenant `getUsage` utilise la constante avec `static::SIZE`, alors lorsque nous appellerons la méthode `getUsage` depuis `Filesystem`, la constante `SIZE` de `Filesystem` sera utilisée. Et lorsque nous appellerons la méthode `getUsage` depuis `BigFilesystem`, la constante `SIZE` de `BigFilesytem` sera utilisée.

3.2.3 Description de tableaux

La description de tableaux est une partie très importante de Praspel.

Dans PHP, un tableau est toujours un **tableau associatif** (ou une *map*, un dictionnaire), *i.e.* une collection de paires clé-valeur, où chaque clé apparaît au maximum une fois. Les clés peuvent être de type nul, booléen, entier, réel ou chaîne de caractères. PHP accepte ces types de clés mais les booléens sont transtypés en entier et les réels sont réduits à leur partie entière. Les valeurs quant à elles peuvent être de n'importe quel type. Un tableau peut être **homogène** ou **hétérogène**. Dans un tableau homogène, toutes les clés ont le même type, ainsi que toutes les valeurs. Dans un tableau hétérogène, les clés peuvent avoir des types distincts, tout comme les valeurs. Les clés peuvent être auto-incrémentées, en ajoutant 1 à la dernière clé

1. Voir http://php.net/lsb.

entière, à partir de 0. La longueur (ou la taille) d'un tableau est son nombre de paires. Un tableau n'a pas de longueur prédéfinie, mais sa longueur (stockée en interne par le moteur PHP) peut être connue grâce à la fonction PHP `count()`. Un tableau n'a également pas de profondeur prédéfinie, *i.e.* il peut contenir un nombre arbitraire de sous-tableaux.

Dans Praspel, `array(D, L)` dénote le domaine réaliste des tableaux dont les domaines et codomaines sont décrits par D et dont la longueur appartient à la disjonction L de domaines réalistes d'entiers naturels. D est une liste séparée par une virgule, entre `[` et `]`, de **descriptions** de paires de la forme `from K to V`, où K et V sont des disjonctions de domaines réalistes, respectivement pour les clés et les valeurs. Quand le mot-clé `from` est manquant, nous introduisons le domaine réaliste représentant un entier auto-incrémenté démarrant de 0 avec un pas de 1.

Exemple 3.5 (Tableaux homogènes et hétérogènes)**.** La syntaxe des descriptions de tableaux est illustrée avec les déclarations de tableaux suivantes :

```
a1: array([to boolean()], 7..42)
a2: array([from 0..5 or 10 to integer()], 7)
a3: array([from 0..10 to boolean(),
           from 20..30 to float()], 7)
a4: array([from 0..10 or 20..30 to boolean() or float()], 7)
```

L'identifiant `a1` est déclaré comme un tableau homogène de booléens avec une longueur comprise entre 7 et 42. L'identifiant `a2` est déclaré comme un tableau homogène de longueur 7, dont les clés sont des entiers entre 0 et 5 ou simplement 10, et dont les valeurs sont des entiers. Les identifiants `a3` et `a4` sont déclarés comme des tableaux hétérogènes. Les deux tableaux peuvent contenir les paires `(5, true)` et `(25, 4.2)`, mais `a4` peut contenir la paire `(5, 4.2)`, alors que `a3` ne peut pas la contenir.

Dans la figure 3.6, l'attribut `_map` est spécifié comme un tableau d'objets `File` d'une longueur de 0 à 65535, avec :

```
@invariant _map: array([to class('File')], 0..0xffff);
```

Nous introduisons une **forme normale** pour supprimer les disjonctions dans les descriptions des paires, en appliquant itérativement la règle de réécriture suivante :

$$\texttt{from } F_1 \texttt{ or } F_2 \texttt{ to } T_1 \texttt{ or } T_2 \longrightarrow \begin{array}{l}\texttt{from } F_1 \texttt{ to } T_1\texttt{,}\\ \texttt{from } F_1 \texttt{ to } T_2\texttt{,}\\ \texttt{from } F_2 \texttt{ to } T_1\texttt{,}\\ \texttt{from } F_2 \texttt{ to } T_2\end{array}$$

Une description de tableau est en forme normale quand elle ne peut plus être réduite par cette règle. Cette forme normale n'est volontairement pas présente dans la grammaire de Praspel.

Exemple 3.6 (Description de tableau en forme normale)**.** La déclaration suivante de `a4` est en forme normale :

```
a4: array([from  0..10 to boolean(),
           from  0..10 to float(),
           from 20..30 to boolean(),
           from 20..30 to float()], 7)
```

3.3 Synthèse

Dans ce chapitre, nous avons tout d'abord présenté les domaines réalistes, des structures dédiées au test avec deux caractéristiques : la générabilité et la prédicabilité, permettant respectivement de générer et de valider une donnée par rapport au domaine réaliste. Les domaines réalistes sont paramétrables, ce qui permet de représenter des structures imbriquées. Les domaines réalistes sont classifiables selon deux axes : l'héritage et les interfaces.
Nous avons ensuite présenté au niveau syntaxique et sémantique un nouveau langage de spécification appelé Praspel. Ce dernier repose sur les domaines réalistes pour exprimer des contrats sur du code. Même s'il se veut être un langage agnostique, il est tourné vers PHP en essayant de répondre à tous ses aspects dynamiques. Praspel permet d'assembler plusieurs méthodes du domaine du test au sein d'un même langage, de manière pragmatique comme nous le verrons avec les chapitres suivants. La contribution de chapitre a été publiée dans l'article Enderlin et al. [2011].
Nous connaissons maintenant le langage. Nous savons qu'il permet de générer et valider des données (de test). Le chapitre suivant s'intéresse à la génération (et à la validation) de plusieurs types de données.

Chapitre 4.
Génération de données de test

Table des matières

Praspel a été présenté dans le chapitre précédent. À partir de ce langage, nous voulons faire du *Contract-based Testing*, c'est à dire que les préconditions sont utilisées pour générer des données de test. Ces données sont spécifiées avec des domaines réalistes. Nous allons nous appuyer sur les caractéristiques de prédicabiblité et de générabilité des domaines réalistes pour assembler différentes méthodes du test au sein du langage Praspel. Nous voulons traiter tous les types de données manipulés au quotidien par les utilisateurs de PHP.
Ce chapitre présente les différentes techniques utilisées par les domaines réalistes pour valider et générer des données de test de plusieurs natures : les booléens, les entiers et les réels dans la partie 4.1, les tableaux dans la partie 4.2, les chaînes de caractères dans la partie 4.3 et enfin les objets dans la partie 4.4.

4.1 Génération standard

Le processus de génération des données dans Praspel est illustré par la figure 4.1. Il se déroule basiquement en 3 étapes, détaillées ci-après :

1. sélection d'une variable ;
2. génération d'une donnée pour cette variable, validation et affectation de cette donnée à cette variable ;
3. sélection de la variable suivante.

Les variables qui nous intéressent sont écrites dans les préconditions (clause `@requires`) ainsi que dans les invariants (clause `@invariant`). Nous notons par V la liste de ces variables. Les invariants seront traités dans la partie 4.4. Rappelons qu'une précondition est constituée de déclarations, de prédicats et de qualifications (voir la figure 3.7 page 38). Le processus est contrôlé par une variable τ et une constante τ_{max}, détaillées ci-après. Dans la figure 4.1, v est une variable de V. Les annotations sur les transitions représentent des actions, les annotations entre crochets représentent des gardes.
Le premier état du processus est l'état `pick`. Lors d'une génération de données, nous commençons par traiter les déclarations. Nous choisissons alors une variable v dans la liste des variables V (représenté par l'opération pop dans la figure) tant que V est non vide, sinon le processus termine. Les variables sont choisies dans l'ordre décroissant du nombre de contraintes portées : la variable avec le plus de contraintes sera la première.
L'état suivant est `sample`. Supposons que la variable v est déclarée par :

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

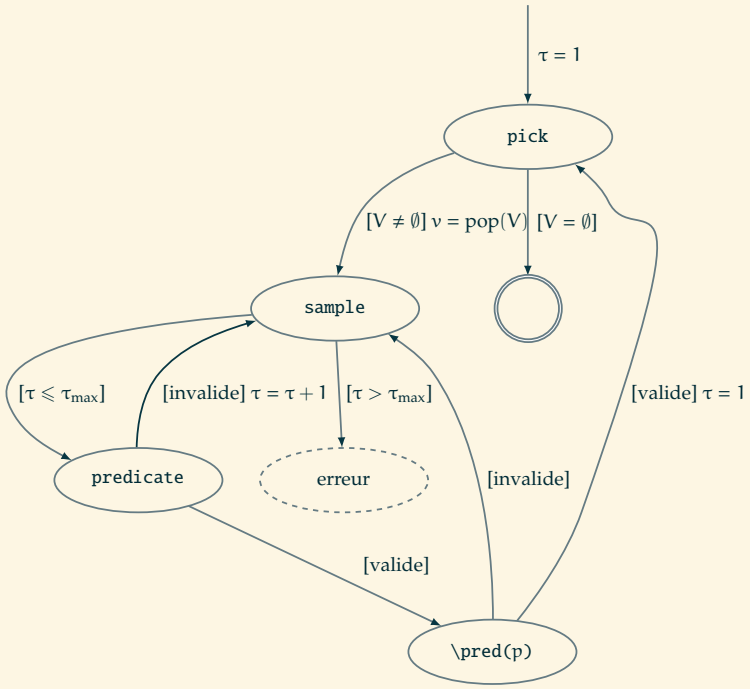


FIGURE 4.1 – Processus de génération des données pseudo-aléatoire.

où $t_1, \ldots, t_n$ sont n domaines réalistes. Nous allons choisir pseudo-aléatoirement le domaine réaliste t_c avec $1 \leqslant c \leqslant n$. Sur ce domaine réaliste t_c, nous allons utiliser sa caractéristique de générabilité, c'est à dire appeler sa méthode `sample` pour générer une donnée qui appartient à ce domaine.

Au sein d'un domaine réaliste, quand une donnée est générée, elle est aussitôt confrontée à sa caractéristique de prédicabilité, c'est à dire sa méthode `predicate`, afin de savoir si la donnée est valide ou non. C'est l'état `predicate` de la figure.

Si la précondition a des prédicats (avec la construction `\pred(p)`), nous avons deux cas. Si toutes les variables du prédicat ont une valeur, alors le prédicat sera évalué. Si des variables n'ont pas de valeur, l'évaluation du prédicat sera repoussée. Si malgré cela, certaines valeurs de variables sont toujours manquantes (par exemple si elles n'ont pas de domaines réalistes associés), le prédicat sera évalué à faux avec une erreur spécifique.

Si, pendant ces étapes, la propriété de prédicabilité d'un domaine réaliste ou un prédicat invalide une donnée, il y a un **rejet**. Suite à un rejet, une donnée est re-générée. Un nombre maximum τ_{max} de re-générations τ est fixé afin d'éviter des générations et des rejets en boucle trop importants et donc trop longs. À chaque génération, τ est incrémenté de 1. La valeur de τ_{max} est paramétrable.

Il n'y a pas que le choix d'un domaine réaliste parmi une disjonction qui est pseudo-aléatoire. Par défaut, un domaine réaliste génère une donnée de manière pseudo-aléatoire. Si nous écrivons `boundinteger(7, 42)`, alors un entier de l'intervalle [7;42] sera généré pseudo-aléatoirement. Nous avons précisé dans la partie 3.1.1 qu'une méthode `sample` d'un domaine réaliste reçoit un générateur numérique en argument. Le seul générateur actuel est pseudo-aléatoire, basé sur l'algorithme de Mersenne Twister [Matsumoto and Nishimura, 1998]. Il permet de générer des entiers et des réels. Un exemple d'utilisation a été montré dans la figure 3.1 page 30. C'est aussi cette méthodologie qu'utilise le domaine réaliste `String` pour générer des chaînes de caractères. Cette approche pseudo-aléatoire pose rapidement des problèmes. Le nombre de valeurs possibles pour `string('a', 'z', 10)`, c'est à dire pour une chaîne de 10 caractères entre `a` et `z` en minuscules, est de 26^{10}, soit environ $1.412e^{14}$ valeurs différentes. La probabilité qu'une de ces valeurs détecte une erreur dans notre programme est extrêmement faible. C'est un problème inhérent à l'approche (pseudo-)aléatoire. Cependant, l'ensemble des valeurs générables peut être considérablement réduit si la donnée est mieux spécifiée. Nous avons observer que les utilisateurs recourent à la construction `\pred(p)` pour mieux spécifier des données. Une tâche récurrente durant nos travaux de recherche est d'observer et analyser ce que les utilisateurs écrivent le plus dans cette construction, et enrichir Praspel avec des outils de générations pour les prédicats les plus fréquents. L'objectif est de faire évoluer le langage en supportant plus de constructions usuelles nativement au lieu de les avoir dans une boîte noire, et ainsi, entre autres, réduire le rejet lors de la génération de données.
Dans les parties suivantes, nous allons nous intéresser à mieux spécifier et mieux générer des tableaux, des chaînes de caractères et des objets.

4.2 Génération à partir de propriétés sur les tableaux

En PHP, les tableaux sont utilisés pour représenter toutes sortes de collections. Ce sont des structures incontournables. Nous parlons de tableaux associatifs, ou encore *hashmaps*, car ils sont constitués de paires clé-valeur. Utiliser une génération aléatoire pour le domaine réaliste `array` est efficace si aucune contrainte n'est exprimée. Toutefois, si l'utilisateur veut spécifier qu'une clé, qu'une valeur ou qu'une paire est présente dans ce tableau, il doit recourir à la construction `\pred()`. Cette construction introduit un nombre trop important de rejets dans le cas des tableaux. C'est pourquoi nous avons d'abord étudié, dans la partie 4.2.1, plusieurs projets PHP pour connaître les fonctions les plus utilisées sur les tableaux. Ensuite, nous avons étendu Praspel pour introduire des propriétés sur les tableaux, présentées dans la partie 4.2.2. Enfin, dans la partie 4.2.3, nous traduisons ces propriétés en contraintes

pour notre propre solveur, présenté dans la partie 4.2.4. Dans cette dernière partie, nous expliquons pourquoi nous avons choisi de développer notre propre solveur.

4.2.1 Étude des propriétés les plus utilisées

Nous avons sélectionné 61 projets PHP sur plusieurs plateformes, comme Github [1] ou Sourceforge [2], à partir de leur popularité, leur impact sur l'industrie et leur complexité. Tous ces projets représentent 28 066 fichiers, soit 5 220 547 lignes de code. Dans ces lignes de code, nous avons compté le nombre d'occurences de chaque fonction de tableaux disponible dans la bibliothèque standard de PHP [3]. La figure 4.2 présente partiellement ces résultats avec les fonctions en abscisse et leur nombre d'occurence en ordonnée. Les fonctions sont triées par leur nombre d'occurences n ; celles avec $n < 100$ n'apparaissent pas. Les trois fonctions les plus utilisées sont `count()`, `array_key_exists()` et `in_array()`. La fonction `count()` compte le nombre d'éléments contenus dans un tableau, la fonction `array_key_exists()` vérifie si une clé est présente dans un tableau (indépendamment de la valeur associée, c'est à dire que cette fonction retourne `true` même si la valeur est `null`), et enfin, la fonction `in_array()` vérifie si une valeur est présente dans le tableau. Toutes ces fonctions travaillent sur un seul tableau à la fois. Cette étude suggère que nous pouvons considérer ces fonctions sans effet de bord en tant que propriétés les plus fréquentes sur les tableaux.

4.2.2 Syntaxe des propriétés

Dans Praspel, nous étendons la syntaxe d'une déclaration de tableau :

```
a: array(D, L)
```

où D est une déclaration de tableau et L la taille du tableau, avec les propriétés suivantes. Une propriété sur une paire est de la forme :

```
a[K]: V
```

où K et V sont des disjonctions de domaines réalistes. La propriété signifie que toute valeur de K est la clé d'une paire dans le tableau a, et que toute paire dont la clé est dans K a sa valeur dans V. K accepte seulement les domaines réalistes qui implémentent au moins les interfaces `Constant`, `Interval` et `Enumerable`. Cette propriété est équivalente à utiliser les fonctions `array_key_exists()` et `in_array()` combinées.

1. Voir http://github.com/.
2. Voir http://sourceforge.net/.
3. Voir http://php.net/ref.array.

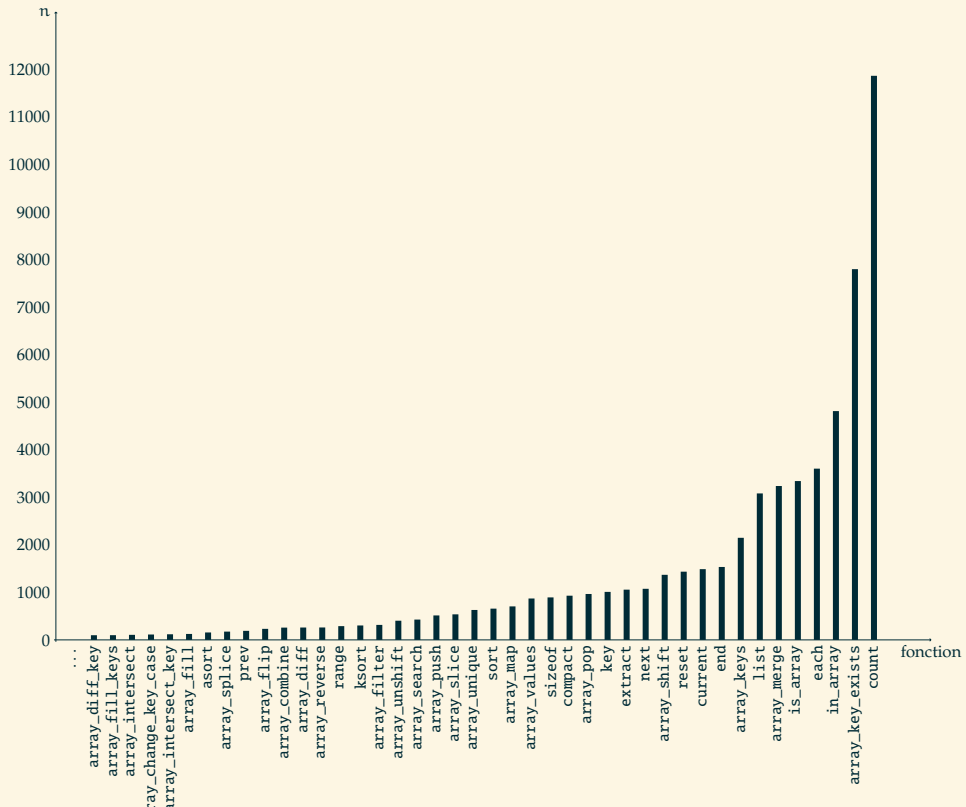


FIGURE 4.2 – Étude des fonctions sur les tableaux.

Exemple 4.1 (`a[K]: V` en PHP)**.** La propriété `a[7..8]: true or 42` signifie que les valeurs `true` ou `42` sont associées aux clés `7` et `8`, ce qui correspond en PHP au prédicat suivant :

```
   array_key_exists(7, $a)
&& array_key_exists(8, $a)
&& (in_array(true, $a) || in_array(42, $a))
&& ($a[7] === true     || $a[7] === 42)
&& ($a[8] === true     || $a[8] === 42)
```

Nous vérifions que les clés `7` et `8` existent dans le tableau `$a`, puis nous vérifions que les valeurs `true` ou `42` existent également, et enfin, nous vérifions qu'elles forment une paire.

Si nous voulons exprimer une contrainte seulement sur `K`, nous pouvons utiliser le symbole `_`. La propriété :

`a[K]: _`

signifie que toute valeur de `K` doit être une clé du tableau `a`. Cette propriété est équivalente à utiliser la fonction `array_key_exists` avec toutes les clés de `K` en conjonction. De même, la propriété :

`a[_]: V`

signifie que toute valeur de `V` doit être une valeur du tableau `a`. Cette propriété est équivalente à utiliser la fonction `in_array` avec toutes les valeurs de `V` en conjonction.

Exemple 4.2 (`a[K]: _` et `a[_]: V` en PHP)**.** Les propriétés `a[7..8]: _` et `a[_]: 'foo'` signifient que les clés `7` et `8` sont présentes dans le tableau `$a`, ainsi que la valeur `'foo'` ; ce qui correspond en PHP au prédicat suivant :

```
   array_key_exists(7, $a)
&& array_key_exists(8, $a)
&& in_array('foo', $a)
```

Nous pouvons également utiliser le symbole `!:`. La propriété `a[K]!: V` signifie que toute valeur de `K` est la clé d'une paire dans le tableau `a`, et que toute paire dont la clé est dans `K` n'a pas sa valeur dans `V`. La signification est similaire avec le symbole `_`. La propriété `a[K]!: _` signifie qu'aucune paire du tableau `a` n'a sa clé dans `K`.
Les clés des tableaux sont toujours uniques, mais pas les valeurs. Nous pouvons exprimer la propriété d'unicité sur les valeurs en écrivant :

`a is unique`

Dans ce cas, nous ne pouvons pas avoir deux fois la même valeur dans le tableau `a`.

Exemple 4.3 (Propriétés sur un tableau)**.** Pour illustrer toutes les sortes de propriétés, nous allons utiliser l'exemple suivant qui porte sur un tableau `a` :

```
length: 0..5 or 10
a     : array([to string('a', 'e', 1)], length)
a[0]  : 'b' or 'c'
a is unique
```

Deux variables sont présentes dans cet exemple : `length` qui représente la taille du tableau, ici par un intervalle $[0;5]$ ou 10, et `a` qui est un tableau de caractères de `a` à `e` et dont la taille est définie par la variable `length`. Il y a deux propriétés : `a[0]: 'b' or 'c'` qui signifie que les valeurs associées à la clé `0` sont le caractère `'b'` ou `'c'` et `a is unique` qui signifie que toutes les valeurs du tableau doivent être uniques.

4.2.3 Sémantique ensembliste des propriétés

Sur un ensemble de propriétés d'un tableau `a`, nous proposons d'invoquer un solveur de contraintes pour construire un tableau satisfaisant toutes ces propriétés. Cette partie explique comment les propriétés sur les tableaux sont transformées en contraintes pour le solveur, ou plus précisément en problème de satisfaction de contraintes (*Constraint Satisfaction Problem*, abrégé CSP [Tsang, 1993]).

Définition 4.1 (*Constraint Satisfaction Problem*)**.** Un CSP est un triplet (Σ, Δ, Π) où Σ est l'ensemble des variables, Δ est l'ensemble des domaines de valeurs pour ces variables et Π est l'ensemble des contraintes. Les variables $\sigma \in \Sigma$ sont déclarées en Praspel, les domaines $\delta \in \Delta$ sont donnés par des disjonctions de domaines réalistes, et les contraintes $\pi \in \Pi$ entre ces variables sont données par des propriétés sur les tableaux.

Une de ces propriétés est supposée être une déclaration de tableau de la forme `a: array(D, L)`, où `D` est en forme normale, c'est à dire que `D` est une liste de p constructions `from` F_i `to` T_i avec $1 \leqslant i \leqslant p$, et `L` est une disjonction de domaines réalistes $L_1, \ldots, L_m$ héritant du domaine réaliste `Integer`. La taille doit être supérieure ou égale à zéro.
Dans l'exemple 4.3, $p = 1$, $m = 2$, $L_1 = [\texttt{0..5}]$ et $L_2 = \{\texttt{10}\}$. Dans la description du tableau, aucun domaine n'a été déclaré. Dans ce cas, le domaine réaliste `natural(0, 1)` est utilisé par défaut : il représente un entier auto-incrémenté commençant à 0 et avec un pas de 1, soit `0`, `1`, `2`, `3` etc. Dans cet exemple, F_1 = `natural(0, 1)` et T_1 = `string('a', 'e', 1)`.
Nous précisons qu'une disjonction de domaines réalistes D_1 `or ... or` D_n sera souvent identifiée avec l'ensemble $D_1 \cup \cdots \cup D_n$.

Variables

Les variables du CSP sont :

1. la taille du tableau, notée S, qui est un entier positif ou nul ;
2. les ensembles X et Y qui sont respectivement le domaine du tableau (l'ensemble des clés) et le co-domaine (l'ensemble des valeurs) ;
3. le contenu du tableau, noté H, qui est une fonction totale de X vers Y (totale car les clés sont uniques) ;
4. les domaines réalistes $X_1, \ldots, X_p$ (respectivement $Y_1, \ldots, Y_p$) qui sont des sous-ensembles des domaines réalistes $F_1, \ldots, F_p$ (respectivement $T_1, \ldots, T_p$), compatibles avec toutes les propriétés sur le tableau.

Le solveur doit trouver le contenu de S, X et les valeurs de la fonction H. Les autres variables sont seulement introduites pour simplifier l'expression des contraintes. La contrainte $H(x) = y$, avec $x \in X$, traduit que la paire (x, y) est dans le tableau. Nous étendons H aux parties de X par la fonction $\hat{H}$ définie par $\hat{H}(E) = \{H(x) \mid x \in E\}$ pour toute partie E de X.

Contraintes de cardinalité

Nous désignons par $\mathrm{card}(E)$ la cardinalité de l'ensemble fini E. Les contraintes :

$$\mathrm{card}(X) = S \quad \text{et} \quad S \geqslant 0$$

expriment que la taille du tableau est son nombre de clés et que ce nombre est positif ou nul. Par défaut, il n'y a pas de contrainte d'unicité sur les co-domaines, donc nous avons seulement la contrainte :

$$\mathrm{card}(X) \geqslant \mathrm{card}(Y)$$

Cependant, en présence de la propriété `a is unique`, cette contrainte devient :

$$\mathrm{card}(X) = \mathrm{card}(Y)$$

Contraintes sur la taille du tableau

Pendant la propagation des contraintes, le solveur peut raffiner les domaines réalistes $L_1, \ldots, L_m$, représentant les valeurs possibles pour la taille S du tableau. Cette taille appartient à un de ces domaines réalistes, nous avons donc la contrainte suivante :

$$S \in L_1 \cup \cdots \cup L_m$$

Dans l'exemple 4.3, nous avons $L_1 \subseteq [\texttt{0..5}]$ et $L_2 \subseteq \{\texttt{10}\}$. La taille S est contrainte par $S \in L_1 \cup L_2$.

Contraintes sur les domaines et co-domaines

Le domaine X et le co-domaine Y de H sont reliés par la contrainte :

$$Y = \hat{H}(X)$$

Nous attendons du solveur qu'il nous propose une solution pour le domaine X (respectivement le co-domaine Y) sous la forme d'une disjonction de domaines réalistes X_1 `or ... or` X_p (respectivement Y_1 `or ... or` Y_p), compatibles avec toutes les propriétés sur le tableau. Nous devrions avoir les égalités $X = \bigcup_{1 \leqslant i \leqslant p} X_i$ et $Y = \bigcup_{1 \leqslant i \leqslant p} Y_i$, ainsi que les inclusions $X_i \subseteq F_i$ et $Y_i \subseteq T_i$ pour $1 \leqslant i \leqslant p$. La paire (X_i, Y_i) devrait aussi satisfaire la contrainte $\hat{H}(X_i) = Y_i$, c'est à dire que Y_i est le co-domaine de la restriction de H à X_i ($\subseteq X$).

Contraintes sur les paires

Pour chaque propriété `a[K]:` V, où K et V sont des disjonctions de domaines réalistes, nous introduisons les contraintes :

$$K \subseteq X \quad \text{et} \quad \hat{H}(K) \subseteq V$$

Une propriété négative sur une paire `a[K]!:` V est traduite en contraintes :

$$K \subseteq X \quad \text{et} \quad \hat{H}(K) \cap V = \emptyset$$

Pour la propriété `a[0]: 'b' or 'c'` dans l'exemple 4.3, nous avons $K = \{\texttt{0}\}$ et $V = \{\texttt{'b'}, \texttt{'c'}\}$. Les contraintes sont $\{\texttt{0}\} \subseteq X$ et $\hat{H}(\{\texttt{0}\}) \subseteq \{\texttt{'b'}, \texttt{'c'}\}$.

Contraintes sur les clés et les valeurs

La propriété `a[K]: _` est traduite en contrainte $K \subseteq X$ et la propriété `a[K]!: _` en contrainte $K \cap X = \emptyset$. La propriété `a[_]:` V est traduite en contrainte $V \subseteq Y$ et la propriété `a[_]!:` V en contrainte $V \cap Y = \emptyset$.

4.2.4 Solveur de contraintes

Notre solveur de contraintes fonctionne en deux étapes : propapagation et consistance dans un premier temps et *labelling* dans un second temps. La première étape consiste à réduire (simplifier, normaliser) au maximum les contraintes, à vérifier si elles sont consistantes et à assurer que le CSP a une solution. Si une solution existe, une valeur doit être générée pour chaque variable du CSP : c'est l'étape de *labelling* (de l'étiquetage).

Nous avons choisi de développer notre propre solveur pour plusieurs raisons. Traditionnellement, les solveurs font du *labelling* sur des ensembles connus et prédéfinis dans le solveur. Si nous voulons ajouter un ensemble de données, il est nécessaire de réimplémenter un type dans le solveur avec les méthodes de génération. Ce travail est déjà effectué dans les domaines réalistes, qui sont écrits par l'utilisateur (l'ingénieur de test) ou par d'autres développeurs d'une collection de domaines réalistes. Dans ce dernier cas, l'utilisateur n'a pas forcément connaissance du fonctionnement du domaine réaliste. Nous ne voulons pas demander à l'utilisateur de réécrire un domaine réaliste dans un solveur pour en profiter, surtout qu'il est souvent nécessaire d'apprendre un nouveau formalisme. De plus, puisque les solveurs ont leur propre langage, il est nécessaire d'effectuer une étape de traduction : transformer les contraintes exprimées dans notre formalisme dans le formalisme du solveur, et inversement avec le résultat du solveur. Cette étape peut être compliquée et peut faire perdre des informations, qui ne seraient pas toujours complétables. Enfin, il aurait été nécessaire d'embarquer le solveur avec Praspel. Un de nos objectifs est de proposer un langage et des outils simples, ce qui implique une installation sur toutes les plateformes et rapide. Par conséquent, pour toutes ces raisons, nous avons préféré développer notre propre solveur ensembliste, ce qui était moins coûteux et plus pertinent pour l'utilisateur final. Nous insistons sur le fait que certains domaines réalistes peuvent être éloignés des types usuels que nous trouvons dans les solveurs, comme des adresses emails. Ainsi, nous pouvons exprimer un tableau d'adresses emails avec des contraintes sur l'unicité des adresses emails par exemple.
Nous expliquons maintenant le fonctionnement du solveur de contraintes.

Propagation et consistance

La propagation des contraintes utilise un algorithme AC3 [Mackworth, 1977], pour *Arc Consistency Algorithm #3*, implémenté en PHP.
Cet algorithme travaille sur un CSP transformé en graphe orienté où les nœuds représentent les variables et les arcs représentent les contraintes entre deux variables. Un arc (X, Y) est considéré comme consistant si et seulement si pour chaque valeur x de la variable X, il existe une valeur y de la variable Y telle que la paire (x, y) satisfasse la contrainte entre X et Y. Si ce n'est pas le cas, l'algorithme supprime des valeurs du domaine de x pour obtenir la consistance de l'arc. Cet algorithme permet aussi de propager les contraintes d'une variable à l'autre en même temps qu'il vérifie la consistance. Seules les contraintes unaires et binaires sont gérées par cet algorithme, c'est à dire des contraintes portant sur une et deux variables. La figure 4.3 présente l'algorithme AC3, tandis que l'exemple 4.4 illustre le problème auquel répond cet algorithme.

Variables : set⟨Variable⟩
Domains : λ(Variable → set⟨Domain⟩)
Neighbours : λ(Variable → set⟨Variable⟩)

---

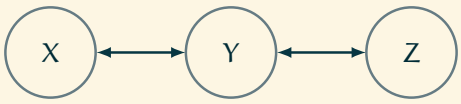
function AC3
  input *csp* : CSP
  queue : list⟨(Variable, Variable)⟩ ← all the arcs
  while *queue* ≠ ∅ **do**
    $(X, Y) \leftarrow$ *queue*.pop()
    if AC3_Arc_Reduce(X, Y) **then**
      for all $Z \in$ *Neighbours*(X) **do**
        queue.push$((Z, X))$
      end for
    end if
  end while
end function

---

function AC3_Arc_Reduce
  input X : Variable
  input Y : Variable
  output *removed* : boolean ← false
  for all $x \in$ *Domains*(X) **do**
    if no $y \in$ *Domains*(Y) such that (x, y) satisfies the selected constraint **then**
      delete x from *Domains*(X)
      removed ← true
    end if
  end for
  return *removed*
end function

FIGURE 4.3 – Algorithme AC3.

Exemple 4.4 (Illustration de l'algorithme AC3)**.** Supposons que nous avons le CSP suivant :
– variables : X, Y, Z ;
– domaines : $D(X) = \{4, 5, 6, 7\}$, $D(Y) = \{4, 5, 6, 8, 9\}$ et $D(Z) = \{3, 5, 6, 7, 9\}$;
– contraintes : X = Y et Y = Z.
Sans algorithme AC3, intuitivement, nous commençons par vérifier la contrainte X = Y, c'est à dire que toutes les valeurs du domaine D(X) qui ne sont pas dans le domaine D(Y) seront supprimées, et inversement. Alors nous obtenons $D(X) = \{4, 5, 6\}$, $D(Y) = \{4, 5, 6\}$ et D(Z) est inchangé. Ensuite, nous vérifions la contrainte Y = Z. Nous obtenons $D(Y) = \{5, 6\}$, $D(Z) = \{5, 6\}$ et D(X) est inchangé. Mais nous avons un problème : la contrainte X = Y n'est plus vraie, $4 \in D(X)$ alors que $4 \notin D(Y)$. Nous savons alors qu'il faut re-vérifier la contrainte X = Y pour obtenir $D(X) = \{5, 6\}$, mais une machine ne le saurait pas nécessairement.
Maintenant, voyons comment opère l'algorithme AC3. Nous avons les mêmes variables, domaines et contraintes qu'au début de cet exemple. Le graphe du CSP est le suivant, il représente les contraintes entre les variables par des arcs :



D'après la figure 4.3, nous avons :
– $Variables = \{X, Y, Z\}$;
– $Domains = \{X \to \{4, 5, 6, 7\}, Y \to \{4, 5, 6, 8, 9\}, Z \to \{3, 5, 6, 7, 9\}\}$;
– $Neighbours = \{X \to \{Y\}, Y \to \{X, Z\}, Z \to \{Y\}\}$.
Déroulons l'algorithme AC3 . La variable *queue* contient au départ tous les arcs du graphe, soit $queue = \{(X, Y), (Y, X), (Y, Z), (Z, Y)\}$. Nous commençons par traiter la première paire de *queue* après l'avoir supprimée. Nous traitons alors la contrainte impliquant X et Y, soit $X = Y$ mais la fonction AC3_Arc_Reduce ne va travailler que sur X. Ainsi, cette fonction va regarder si pour toute valeur x de $Domains(X)$ la contrainte avec les valeurs de $Domains(Y)$ est satisfaite. Si une valeur x ne satisfait pas la contrainte, elle est supprimée de $Domains(X)$. Nous obtenons alors $Domains(X) = \{4, 5, 6\}$, $Domains(Y)$ et $Domains(Z)$ restant inchangés. La variable X n'a pas d'arc vers d'autres variables, donc nous ne modifions pas *queue*. Ensuite, la paire suivante est (Y, X), donc nous traitons la contrainte $X = Y$ mais cette fois-ci, nous modifions Y. Ainsi $Domains(Y) = \{4, 5, 6\}$, $Domains(X)$ et $Domains(Z)$ restant inchangés. La variable Y a aussi un arc vers Z, donc nous devrions ajouter la paire (Z, Y) mais elle existe déjà dans *queue*, nous ne faisons rien. La paire suivante dans *queue* est (Y, Z), donc nous traitons la contrainte $Y = Z$ en modifiant uniquement Y. Ainsi : $Domains(Y) = \{5, 6\}$, $Domains(X)$ et $Domains(Z)$ restant inchangés. La variable Z n'a pas d'arc vers une autre variable, donc nous ne modifions pas *queue*. La

paire suivante est (Z, Y), donc nous vérifions la contrainte $Y = Z$ en modifiant Z. Ainsi (en rappelant tous les domaines) : $\mathit{Domains}(X) = \{4, 5, 6\}$, $\mathit{Domains}(Y) = \{5, 6\}$ et $\mathit{Domains}(Z) = \{5, 6\}$. La variable Y a un arc vers la variable X, donc nous ajoutons la paire (X, Y) dans *queue*. Comme *queue* était vide, cela devient notre paire suivante que nous traitons. Nous appliquons alors la contrainte $X = Y$ en modifiant X. Ainsi, nous obtenons : $\mathit{Domains}(X) = \mathit{Domains}(Y) = \mathit{Domains}(Z) = \{5, 6\}$. La variable X n'a pas d'arc vers d'autres variables que Y, donc nous ne modifions pas *queue*. Cette dernière est vide, donc l'algorithme se termine.

Notre implémentation est une variante de l'algorithme AC3. Les variables n'ont pas des domaines mais une disjonction de domaines réalistes. La complexité de cet algorithme est de $\mathcal{O}(mn^3)$ où m est le nombre de contraintes (d'arcs) et n le nombre de valeurs de la « plus grande » disjonction de domaines réalistes. Rappelons que nous manipulons cinq sortes de domaines réalistes classés par les interfaces `Constant`, `Interval`, `Nonconvex`, `Finite` et `Enumerable`. Pour chaque sorte de domaine réaliste, nous avons implémenté une ou des méthodes de raffinement pour permettre la réduction du domaine réaliste. Par exemple, `Nonconvex` attend une méthode `discredit` pour supprimer une valeur du domaine, `Interval` attend les méthodes `reduceRightTo` et `reduceLeftTo` respectivement pour réduire la borne droite et gauche etc.

Exemple 4.5 (Propagation et consistance avec un algorithme AC3)**.** Si nous reprenons notre exemple 4.3, nous avons les contraintes suivantes :

$$
\begin{aligned}
S &\geqslant 0 \\
\mathrm{card}(X) &= S \\
\mathrm{card}(X) &= \mathrm{card}(Y) \\
S &\in [\texttt{0..5}] \cup \{\texttt{10}\} \\
X &= X_1 \\
X_1 &\subseteq \texttt{natural(0, 1)} \\
Y &= Y_1 \\
Y_1 &\subseteq \texttt{string('a', 'e', 1)} \\
Y &= \hat{H}(X) \\
\{\texttt{0}\} &\subseteq X \\
\hat{H}(\{\texttt{0}\}) &\subseteq \{\texttt{'b','c'}\}
\end{aligned}
$$

Nous appliquons l'algorithme. Au départ, nous avons la variable *queue* initialisée à

$\{(X, X), (X, Y), (Y, X), (S, S), (S, X), (X, S)\}$.

Nous sélectionnons et retirons la paire (X, X), qui est la première de la variable *queue*, représentée à droite. Nous avons la contrainte unaire (ne portant que sur une seule variable) $\{\texttt{0}\} \subseteq X$. Nous vérifions $\texttt{0} \in \texttt{natural(0, 1)}$. Tout est consistant. Le tableau ne peut pas être vide car il contiendra au moins une paire.

(X, X)
(X, Y)
(Y, X)
(S, S)
(S, X)
(X, S)

La paire suivante que nous sélectionnons et retirons est (X, Y). Nous avons la première contrainte $\text{card}(X) = \text{card}(Y)$. Nous traitons uniquement la variable X et nous n'avons rien à faire. La seconde contrainte est $\hat{H}(\{\texttt{0}\}) \subseteq \{\texttt{'b'}, \texttt{'c'}\}$. Nous vérifions $\texttt{0} \in \texttt{natural(0, 1)}$, $\texttt{'b'} \in \texttt{string('a', 'e', 1)}$ et $\texttt{'c'} \in \texttt{string('a', 'e', 1)}$. Tout est consistant.

(X, Y)
(Y, X)
(S, S)
(S, X)
(X, S)

Nous sélectionnons et retirons la paire (Y, X). Nous avons toujours la contrainte $\text{card}(X) = \text{card}(Y)$ mais nous traitons maintenant la variable Y. La cardinalité de Y est égale la cardinalité de Y_1, qui est inférieure ou égale à la cardinalité de `string('a', 'e', 1)`, qui vaut 5 car ce domaine réaliste peut produire les données a jusqu'à `e`. Nous savons aussi que le tableau contiendra au moins une paire. Ainsi, $\text{card}(X) \in [1; 5]$. Nous devons ajouter une paire dans la variable *queue*, mais Y n'a pas d'arc vers d'autre variable, donc nous ne faisons rien.

(Y, X)
(S, S)
(S, X)
(X, S)

Nous sélectionnons et retirons la paire (S, S). Nous avons la contrainte unaire $S \geqslant 0$. Nous n'avons rien à réviser.

(S, S)
(S, X)
(X, S)

Nous sélectionnons et retirons la paire (S, X). Nous travaillons sur la variable S avec la contrainte $\text{card}(X) = S$. Puisque $\text{card}(X) \in [1; 5]$, nous devons réviser $S \in [\texttt{0..5}] \cup \{\texttt{10}\}$, pour enlever les valeurs `0` et `10`. `0..5` est un domaine réaliste représentant un intervalle d'entiers (domaine réaliste `Boundinteger`) implémentant entre autre les interfaces `Interval` et `Nonconvex`. Nous appelons la méthode `discredit` pour supprimer `0` de l'intervalle. `10` est un domaine réaliste représentant une constante pour l'entier 10. Ce dernier est supprimée de la disjonction de domaines réalistes. Ainsi, nous obtenons $S \in [\texttt{1..5}]$. Dans la variable *queue*, nous devrions ajouter la paire (X, S) mais elle est déjà présente.

(S, X)
(X, S)

Nous sélectionnons et retirons la paire (X, S). Nous travaillons sur la variable X avec la contrainte $\text{card}(X) = S$. Nous n'avons rien à faire. $\boxed{(X, S)}$

La variable *queue* est vide, l'algorithme se termine.

La consistance vérifie également qu'il n'y a pas de domaine réaliste vide pour les quatres variables S, H, X et Y. L'objectif est de détecter les inconsistances au plus tôt.

Labelling

Le *labelling* (étiquetage) est le processus qui trouve une valeur pour chaque variable. Il s'applique si la fonction AC3 termine. L'idée est de sélectionner une variable et de lui choisir une valeur, puis de recommencer avec une autre variable etc. Le choix de la variable à sélectionner dépend de la dépendance entre les variables (les arcs vus dans la partie précédente).
À chaque choix de valeur, la consistance entre les contraintes est vérifiée. Si une inconsistance est détectée, alors le processus va revenir en arrière, nous parlons de *backtracking*.
À chaque fois que l'algorithme sélectionne une nouvelle variable, sa disjonction de domaines réalistes est clonée (dupliquée) ; ainsi lors du *backtracking*, nous restaurons la disjonction précédente pour « restaurer l'état ». En même temps, la valeur qui a conduit à l'inconsistance est discréditée afin de ne pas être re-sélectionnée.
Dans notre implémentation, le choix d'une valeur utilise la caractéristique de générabilité des domaines réalistes, avec un générateur numérique pseudo-aléatoire. Afin de faire converger le solveur rapidement vers une solution, si le CSP est satisfaisable, vers une erreur sinon, nous utilisons une heuristique qui consiste à choisir une valeur pour la variable S en premier. Ensuite, le solveur essaie de calculer les ensembles X_i et Y_i.
Quand toutes les variables sont étiquetées, c'est à dire que chacune a une valeur valide, le solveur retourne la solution.

Exemple 4.6 (*Labelling*)**.** Une solution pour l'exemple 4.3 est :

```
[
    0 => 'c',
    1 => 'd',
    2 => 'a',
    3 => 'e'
]
```

La taille du tableau est de 4, la clé `0` a comme valeur `'c'` et toutes les valeurs sont uniques.

4.3 Génération à partir de grammaire pour les chaînes de caractères

Après les tableaux, les chaînes de caractères sont le type de données le plus utilisées dans le Web. Elles sont utilisées aussi bien à l'intérieur des programmes qu'à l'extérieur, pour des bases de données, des protocoles de communication, des descriptions d'objets etc. Il est donc nécessaire de proposer un moyen rapide et simple de spécifier des chaînes de caractères dans Praspel.
Une grammaire formelle permet de décrire avec précision des données textuelles complexes. Nous proposons des domaines réalistes de chaînes de caractères définis par une grammaire. Des analyseurs lexicaux et syntaxiques permettent de s'assurer qu'une chaîne de caractères est conforme à une grammaire. Ceci correspond à la caractéristique de prédicabilité d'un domaine réaliste : ce qui fait des grammaires de bonnes candidates pour être un domaine réaliste.
Nous allons, dans les parties 4.3.1 et 4.3.2, nous concentrer sur la première caractéristique des domaines réalistes, à savoir la prédicabilité, et présenter un langage simple de description de grammaire que nous avons créé, ainsi que son interprétation avec un compilateur de compilateurs dédié. Ensuite, dans la partie 4.3.3, nous allons voir comment exploiter une grammaire pour générer des données valides, et ainsi assurer la seconde caractéristique des domaines réalistes, à savoir la générabilité.

4.3.1 Langage de description de grammaire

Définition 4.2 (Grammaire)**.** Une grammaire est définie par un quadruplet $G = (\Sigma, N, P, S)$, où :
- Σ est l'ensemble fini de **symboles terminaux** disjoint de N ; aussi appelés **lexèmes**, ce sont les unités atomiques lexicales d'un language ;
- N est l'ensemble fini de **symboles non-terminaux** ;
- P est l'ensemble fini des **règles de production** de la forme $(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$, avec * représentant l'étoile de Kleene et $\cup$ représentant l'union d'ensembles ; les règles expriment l'enchaînement possible des lexèmes les uns par rapport aux autres ;
- $S \in N$ est l'axiome de la grammaire, le symbole indiquant le point de départ de lecture de la grammaire.

Selon la hiérarchie de Chomsky [Chomsky, 1956], les grammaires sont classées en

quatre niveaux :

1. grammaires **générales**, ou *unrestricted grammars*, reconnaissant les langages dits de Turing, aucune restriction n'est imposée sur les règles ;
2. grammaires **contextuelles**, ou *context-sensitive grammars*, reconnaissant les langages contextuels ;
3. grammaires **algébriques**, ou *context-free grammars*, reconnaissant les langages algébriques, basés sur les automates à pile ;
4. grammaires **régulières**, ou *regular grammars*, reconnaissant les langages réguliers.

Chaque niveau est plus général que le niveau suivant.
Nous proposons le langage *PHP Parser*, abrégé PP, pour exprimer des grammaires algébriques (et donc également régulières). Sa syntaxe est principalement inspirée de JavaCC [Viswanadha and Sankar, 1996] et un peu de YACC [Johnson, 1975], avec l'ajout de nouvelles constructions. La déclaration d'un lexème est de la forme suivante :

```
%token ns_source:name value -> ns_dest
```

où `name` représente son nom, `value` sa valeur exprimée avec une expression régulière, et `ns_source` et `ns_dest` sont des espaces de noms optionnels. Les expressions régulières sont écrites en utilisant la syntaxe standard des *Perl Compatible Regular Expressions*, abrégé PCRE [Hazel, 1997], proposant de nombreux raccourcis, largement utilisées et supportées (par exemple dans PHP, Javascript, Perl, Python, Apache, KDE etc.). Les espaces de noms servent à représenter des sous-ensembles disjoints de lexèmes pour la phase d'analyse lexicale (détaillée ci-après). Une déclaration `%skip` est similaire à une déclaration `%token` excepté qu'elle représente un lexème à ne pas retenir pour la suite. Typiquement les espaces dans une expression arithmétique n'ont pas d'importance, nous pouvons ne pas les retenir, ils n'apportent rien à l'expression. Si un espace de noms source n'est pas précisé, alors il prend la valeur `default`, qui est l'espace de noms par défaut. L'espace de noms de destination peut être de la forme spéciale `__shift__ *` j, avec $j \geqslant 1$, ou simplement `__shift__` (équivalent à avoir $j = 1$), pour revenir de j espaces de noms précédents. Il est parfois possible d'accéder à un lexème depuis plusieurs espaces de noms différents ; cette forme spéciale permet de revenir dans les espaces de noms d'origine.

La figure 4.4 montre l'exemple P des règles de production d'une grammaire (Σ, N, P, S) simplifiée d'un document XML [W3C, 2006]. Elle commence par la déclaration des lexèmes, en utilisant des espaces de noms pour identifier si l'analyse se déroule

```
%skip          space    \s
%token         lt       <            -> in_tag
%token         text     [^<]*
%skip   in_tag:space    \s
%token  in_tag:name     \w+
%token  in_tag:slash    /
%token  in_tag:gt       >            -> default
%token  in_tag:equal    =
%token  in_tag:value    ".*?(?<!\\)"

xml:
    tag()+ #root
tag:
    ::lt:: <name[0]> attributes()
    (
      ::slash:: ::gt:: #atomic
    | ::gt::
      (
          tag()+ #composite
        | <text> #textual
      )?
      ::lt:: ::slash:: ::name[0]:: ::gt::
    )
#attributes:
    ( <name> ::equal:: <value> )*
```

FIGURE 4.4 – Grammaire simplifiée d'un document XML.

à l'extérieur ou à l'intérieur d'une balise. Nous avons alors :

$$\Sigma = \{\texttt{space}, \texttt{lt}, \texttt{text}, \texttt{in\_tag:space}, \texttt{in\_tag:name}, \texttt{in\_tag:slash}, \texttt{in\_tag:gt}, \texttt{in\_tag:equal}, \texttt{in\_tag:value}\}$$

Nous avons aussi les non-terminaux :

$$N = \{\texttt{xml}, \texttt{tag}, \texttt{attributes}\}$$

La règle `xml` est l'axiome de la grammaire, S = `xml`, car c'est la première règle déclarée. Elle décrit un document XML comme une séquence de balises, chaque balise ayant potentiellement des attributs et étant atomique (`<aTag />`), composite (contenant d'autres balises), contenant du texte (`<aTag>with text</aTag>`) ou étant vide. Un nom de règle (un non-terminal) est suivi du symbole `:` puis d'une nouvelle ligne, immédiatement suivie par une règle de production, préfixée par des caractères blancs horizontaux (espaces ou tabulations). Les lexèmes peuvent être référencés en utilisant deux types de constructions. La construction `::t::` signifie que le lexème `t` ne sera pas conservé à la fin du processus de compilation (détaillé ci-après), contrairement à la construction `<t>`. La construction `r()` représente un appel à une règle `r`. Nous pouvons avoir des appels récursifs. Les opérations de répétition sont classiques : `{x, y}` pour répéter le motif de `x` à `y` fois, `?` est identique à `{0, 1}`, `+` à `{1, }`, `*` à `{0, }`. Quand `y` est vide, cela signifie l'infini. Les disjonctions sont représentées par le symbole `|` et le groupement par `(` et `)`. La construction `#n` (parfois utilisé comme nom de règle), avec `n` un identifiant, est utile pour la fin du processus de compilation, comme nous le verrons dans la partie suivante.
Une nouvelle construction présente dans le langage PP, par rapport aux langages de description de gramaires existants, est l'unification. Présente uniquement sur les lexèmes, elle impose que tous les `t[i]` avec le même `i` aient la même valeur localement à la règle, avec $i \geqslant 0$. Remarquons la présence d'une unification de lexèmes dans la figure 4.4 : `name[0]` indique que les balises d'ouverture et de fermeture doivent avoir le même nom. Ainsi, la donnée `<foo>...</foo>` sera considérée comme valide, alors que `<foo>...</bar>` sera considérée comme invalide. Un autre usage courant est la gestion des guillemets représentés par le lexème `%token quote '|"`, soit un guillemet simple, soit un double. Ainsi, `"..."` comme `'...'` doivent être valides, alors que `"...'` ou `'..."` devront être invalides. Pour cela, nous écrirons par exemple `quote[1]` dans une règle.
Nous trouverons la grammaire du langage PP en langage PP dans l'annexe 2 à la page 162.

4.3.2 Compilateur de compilateurs LL($\star$)

Le fonctionnement schématique d'un compilateur est présenté dans la figure 4.5. À partir d'un **mot** (d'une donnée textuelle), nous allons extraire une **séquence** de

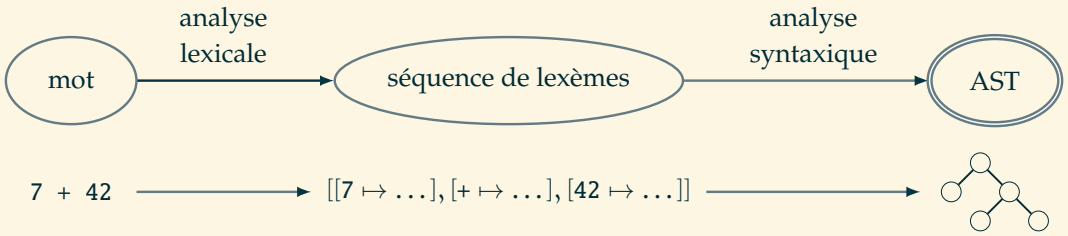


FIGURE 4.5 – Fonctionnement d'un compilateur.

lexèmes grâce à l'analyseur **lexical**. Ensuite, cette séquence sera **dérivée** grâce à l'analyseur **syntaxique** pour savoir si elle est valide ou non. Un *Abstract Syntax Tree*, abrégé AST, sera produit (détaillé ci-après). Un **compilateur de compilateurs** est un programme qui transforme une grammaire en compilateur, c'est à dire en un analyseur lexical et un analyseur syntaxique.
Nous expliquons le fonctionnement de l'analyseur lexical par l'algorithme de la figure 4.6. À partir d'une donnée textuelle (le mot *word*), nous allons calculer une séquence *sequence* de lexèmes. Nous allons appliquer (ligne 10) toutes les expressions régulières *regex* des lexèmes de l'espace de noms courant *tokens*[*namespace*] les uns après les autres sur le mot (ligne 11) jusqu'à trouver un lexème valide (reconnaissant un sous-mot à la position *offset*). Quand un lexème est trouvé, nous lui associons certaines informations, comme son nom *name*, sa valeur *value*, sa position *offset* et son espace de noms *namespace* (ligne 12). Ensuite, nous mettons à jour l'espace de noms courant à partir de l'espace de noms sortant *namespaceOut* du lexème : soit un nom d'espace, soit le mot-clé `__shift__ * j` (lignes 13-18). Si aucun lexème n'est trouvé, une erreur sera levée (ligne 23), sinon nous ajoutons le lexème *nextToken* à la séquence *sequence* (ligne 26) et nous continuons jusqu'à atteindre la fin de la donnée analysée (ligne 8).

Exemple 4.7 (Analyse lexicale de `<a x="y"><b /><c>foo</c></a>`)**.** Avec la grammaire de la figure 4.4, l'analyse lexicale de la donnée :

```
<a x="y"><b /><c>foo</c></a>
```

produit la séquence suivante :

	espace de noms	lexème	valeur	position
0	`default`	`lt`	<	0
1	`in_tag`	`name`	a	1
2	`in_tag`	`name`	x	3

3	in_tag	equal	=	4
4	in_tag	value	"y"	5
5	in_tag	gt	>	8
6	default	lt	<	9
7	in_tag	name	b	10
8	in_tag	slash	/	12
9	in_tag	gt	>	13
10	default	lt	<	14
11	in_tag	name	c	15
12	in_tag	gt	>	16
13	default	text	foo	17
14	default	lt	<	20
15	in_tag	slash	/	21
16	in_tag	name	c	22
17	in_tag	gt	>	23
18	default	lt	<	24
19	in_tag	slash	/	25
20	in_tag	name	a	26
21	in_tag	gt	>	27
22	default	EOF		29

L'analyseur syntaxique prend la suite en dérivant la séquence par rapport aux règles de la grammaire. Cela consiste à dépiler les lexèmes de la séquence un par un et à regarder si nous pouvons déplier les règles.

Exemple 4.8 (Analyse syntaxique de <b /><c>foo</c>)**.** Nous allons illustrer l'analyse syntaxique de la séquence obtenue dans l'exemple 4.7.

1. Le premier lexème est déplié : `lt`. La règle `xml` est composée de une ou plusieurs règles `tag`. La règle `tag` commence par un lexème `lt`. Il y a une correspondance, le processus continue.
2. Le deuxième lexème est `name`. La règle `tag` se poursuit avec le lexème `name`, il y a une correspondance. Le lexème `name` porte une unification. Le prochain lexème `name` pour cette règle devra avoir comme valeur `a`.
3. Le troisième lexème de la séquence est `name`. La règle `tag` poursuit avec un appel à la règle `attributes` qui commence avec le lexème `name`. Encore une fois, il y a une correspondance.

4-5. L'opération est identique pour le quatrième (`equal`) et cinquième lexème (`value`).

```
1:  struct Token{name : string, value : string, at : integer, namespace : string}
2:  function Lexer
        input   word : string
        input   tokens : map⟨string, map⟨string, (regex, string)⟩⟩
        output  sequence : list⟨Token⟩ ← []
3:      namespace : string ← 'default'
4:      namespaceStack : stack⟨string⟩ ← []
5:      offset : integer ← 0
6:      nextToken : Token
7:      length : integer ← word.length()
8:      while offset < length do
9:          nextToken ← null
10:         for all {name, regex, namespaceOut} ∈ tokens[namespace] do
11:             if word matches regex at offset → value then
12:                 nextToken ← Token{name, value, offset, namespace}
13:                 if namespaceOut matches '__shift__ * j' then
14:                     j.times(namespaceStack.pop)
15:                 else
16:                     namespaceStack.push(namespaceOut)
17:                 end if
18:                 namespace ← namespaceStack.last()
19:                 break
20:             end if
21:         end for
22:         if null = nextToken then
23:             error
24:         end if
25:         offset ← offset + nextToken.value.length()
26:         sequence.push(nextToken)
27:     end while
28: end function
```

FIGURE 4.6 – Algorithme de l'analyseur lexical.

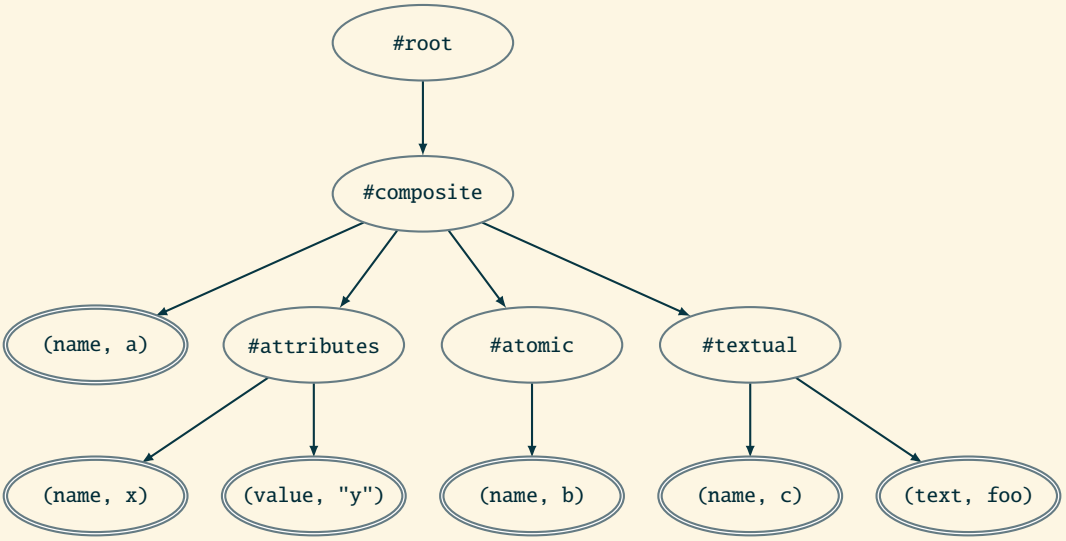


FIGURE 4.7 – Arbre de syntaxe abstrait de la donnée `<a x="y"><b /><c>foo</c></a>`.

6. Le sixième lexème est `gt`. La règle `attributes` demande soit d'avoir un nouvel attribut, soit que la fin de la règle soit atteinte, ce qui est le cas. Donc le processus revient dans la règle `tag`. Il y a une disjonction : le premier élément ne peut pas dériver `gt` (il attend `slash`), mais le second élément le permet. Il y a une correspondance, le processus continue.

... Le lexème suivant dans la séquence est utilisé. Et ainsi de suite, jusqu'à atteindre la fin de la séquence.

Si nous ne pouvons plus dériver la séquence, nous allons revenir en arrière de k lexèmes jusqu'à retomber sur un point de choix (une disjonction ou une répétition). Cette étape s'appelle le *backtracking*. Dans notre cas, k n'est pas fixé, nous disons que nous revenons en arrière de $\star$ lexèmes, soit d'autant de lexèmes que nécessaire.
La séquence est construite en analysant la donnée textuelle de gauche à droite (***L**eft to right*) et la séquence est dérivée en utilisant toujours le lexème le plus à gauche de la séquence (***L**eftmost derivation*), nous avons donc un analyseur LL($\star$).
Une fois que la donnée a été validée par l'analyseur syntaxique, le compilateur est capable de produire un AST, pour *Abstract Syntax Tree*, soit un arbre de syntaxe abstrait. Les constructions `#node` dans la grammaire permettent de forcer la création d'un nœud dont les enfants seront des nœuds ou des lexèmes déclarés avec la syntaxe `<token>` (l'autre construction `::token::` n'autorise pas les lexèmes à apparaître dans l'arbre). La figure 4.7 montre l'AST résultant des exemples 4.7 et 4.8. Un

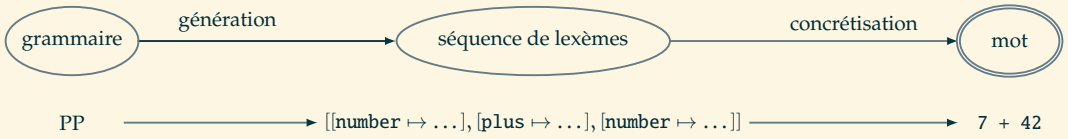


FIGURE 4.8 – Principe des algorithmes de génération de données.

AST est une structure récursive. Nous pouvons lui appliquer un **visiteur** [Gamma, Helm, Johnson, and Vlissides, 1995]. Ce patron de conception permet à l'utilisateur d'appliquer des traitements supplémentaires : des contraintes qui ne peuvent pas être représentées par la grammaire, comme par exemple une vérification de typage.

Ce compilateur de compilateurs LL($\star$) nous permet de valider une donnée textuelle complexe. Ainsi, nous respectons la première caractéristique d'un domaine réaliste : la prédicabilité. Les parties suivantes illustrent plusieurs algorithmes de génération de chaînes de caractères à partir d'une grammaire.

4.3.3 Algorithmes de génération à partir de grammaires

Nous décrivons maintenant l'utilisation de grammaires pour la génération de données textuelles complexes, afin d'assurer la caractéristique de générabilité des domaines réalistes.
Nous proposons trois algorithmes de génération différents : aléatoire uniforme, exhaustive bornée et enfin basée sur la couverture. Le principe de ces algorithmes est résumé dans la figure 4.8 : Ils vont générer différentes séquences de lexèmes à partir d'une grammaire. Ces lexèmes vont être concrétisés pour obtenir une donnée textuelle. L'algorithme de concrétisation est présenté à la fin de la partie.
Pour générer des séquences de lexèmes, les algorithmes vont explorer les règles de la grammaire selon différentes contraintes : la taille de la séquence, la fréquence d'apparition des lexèmes, la quantité de séquences etc. Ces contraintes sont extraites d'une étude menée de manière empirique auprès d'une demi-douzaine d'ingénieurs de test qualifiés, ajoutée à notre propre expérience. Certains souhaitaient des données de taille fixe ou bornée, d'autres insistaient sur la diversité des données, ou encore sur la quantité avec par exemple de l'exhaustivité.
Nous présentons trois algorithmes qui implémentent ces solutions.

Génération aléatoire uniforme

Cet algorithme permet de générer des séquences de lexèmes de taille n fixe. Chaque séquence est générée aléatoirement et avec une distribution uniforme parmi toutes

les séquences possibles de taille n. Afin d'assurer cette uniformité, nous nous basons sur les travaux de Flajolet et al. [1994]. Les auteurs proposent une méthode de calcul récursive pour compter le nombre de sous-structures de taille n. Ce dénombrement va nous permettre de calculer des fonctions de répartition pour guider l'algorithme dans l'exploration des règles à chaque point de choix rencontré.
Cet algorithme fonctionne donc en deux étapes. La première est le dénombrement et la seconde est la génération à proprement parler.
À chaque construction du langage PP est associée une fonction de dénombrement $\psi(n, e)$ qui compte le nombre de sous-structures de taille n qu'il est possible de générer pour la construction grammaticale e. Ainsi :

$$\psi(n, e) = \delta_n^1 \qquad \text{si } e \text{ est un lexème}$$

$$\psi(n, e_1 \cdot \ldots \cdot e_k) = \sum_{\gamma \in \Gamma_k^n} \prod_{\alpha=1}^{k} \psi(\gamma_\alpha, e_\alpha)$$

$$\psi(n, e_1 \mid \ldots \mid e_k) = \sum_{\alpha=1}^{k} \psi(n, e_\alpha)$$

$$\psi(n, e^{\{x,y\}}) = \sum_{\alpha=x}^{y} \sum_{\gamma \in \Gamma_\alpha^n} \prod_{\beta=1}^{\alpha} \psi(\gamma_\beta, e) \qquad \text{avec } 0 \leqslant x \leqslant y$$

Dans la première formule, δ_i^j est le symbole de Kronecker égal à 1 si $i = j$, à 0 sinon. Γ_k^n désigne l'ensemble des k-uplets d'entiers naturels dont la somme des éléments est égale à n. Pour chaque k-uplet γ et chaque $\alpha \in [1; k]$, γ_α désigne le $\alpha^{\text{ième}}$ élément de γ. Par exemple :

$$\Gamma_3^2 = \{(2,0,0), (1,1,0), (1,0,1), (0,2,0), (0,1,1), (0,0,2)\}$$

et pour le premier k-uplet, $\gamma_1 = 2$ et $\gamma_2 = \gamma_3 = 0$.
Pour la concaténation, représentée par l'opérateur $\cdot$, la fonction ψ somme la distribution de n parmi toutes les sous-structures. Pour la disjonction, représentée par l'opérateur $|$, la fonction ψ somme la taille des sous-structures de taille n. Une répétition, représentée par $\{x, y\}$, est une disjonction de concaténations (par exemple, $e^{\{2,4\}}$ est équivalent à $e \cdot e \mid e \cdot e \cdot e \mid e \cdot e \cdot e \cdot e$). Quand y est non défini (avec * et +), il est considéré égal à n.

Exemple 4.9 (Exploration aléatoire uniforme)**.** Soit la grammaire suivante :

```
f: <a> g()
g: ( <b> <c> | <d> | f() ){1,3}
```

n	f: ...	g: (<b> <c> | <d> | f()) {1, 3}						
					r =	1	2	3
5	24	73	0	0	-	(24,	28,	21)
4	8	24	0	0	-	(8,	10,	6)
3	3	8	0	0	-	(3,	4,	1)
2	1	3	1	0	-	(2,	1,	0)
1	0	1	0	1	-	(1,	0,	0)

FIGURE 4.9 – Exemple des résultats de la fonction ψ.

L'axiome est `f`. Nous voulons compter le nombre de sous-structures possibles de taille 5. Les résultats sont présentés dans la figure 4.9. La première colonne montre les valeurs possibles de n. La deuxième colonne montre les résultats pour $\texttt{f}(n)$, qui signifie : le nombre de sous-structures de taille n que peut produire la règle `f`. Ensuite, nous avons les résultats pour $\texttt{g}(n)$, avec le détail pour chaque branche de la disjonction et pour la répétition avec $r = 1, 2$ et 3. Le symbole « - » représente une donnée dupliquée sur la même ligne.
Avec $n = 5$, la règle `f` peut produire $\psi(5, \texttt{<a>} \cdot \texttt{g()})$ sous-structures. Cette règle commence par un lexème <a>, donc le reste de la séquence (donné par la règle `g`) devra faire une taille de 4. Nous calculons alors $\texttt{g}(4)$, soit $\psi(4, \texttt{(<b> <c> | <d> | f())}^{\{1,3\}})$. La règle `g` est une répétition $r \in [1;3]$ d'une disjonction. Avec $r = 1$, le seul moyen d'avoir une sous-structure de taille 4 et de passer par `f`, donc nous devons calculer $\texttt{f}(4)$. Avec $r = 2$, nous pouvons avoir <b> <c> <b> <c>, ou <b> <c> puis $\texttt{f}(2)$, ou <d> puis $\texttt{f}(3)$ etc. Tous les résultats sont présents dans la figure 4.9.
Maintenant, nous sommes outillés pour générer des chaînes de caractères à partir de la grammaire. Imaginons que nous sommes sur l'exploration de la règle `g` avec $n = 3$. Nous avons une répétition $r \in [1;3]$. Nous savons grâce à la fonction ψ que $\texttt{g}(3) = 8$, c'est à dire que `g` peut produire 8 sous-structures de taille 3. Il faut choisir aléatoirement et uniformément une valeur pour r, grâce aux probabilités données par la fonction $\psi(3, \texttt{(<b> <c> | <d> | f())}^{r})$. Avec $r = 1$, 3 sous-structures peuvent être produites, avec $r = 2$, 4 sous-structures peuvent être produites et avec $r = 3$, 1 sous-structure peut être produite. Ainsi :

$$p(r = 1) = \frac{3}{8} \qquad p(r = 2) = \frac{4}{8} \qquad p(r = 3) = \frac{1}{8}$$

Pour appliquer les probabilités, nous choisissons un entier $i \in [1;8]$ aléatoirement et uniformément. Avec $i = 5$, nous obtenons $r = 2$. Donc nous répétons la disjonction 2 fois. Rappelons que nous traitons la règle `g` avec une taille de 3, donc 3 lexèmes à répartir sur 2 répétitions. Nous utilisons la fonction Γ pour calculer toutes les

répétitions possibles : $\Gamma_2^3 = \{(2,1),(1,2)\}$ (tous les k-uplets contenant un 0 ont été supprimés car il est impossible de distribuer 0 lexème dans une répétition). Nous allons utiliser soit 2 lexèmes lors de la première répétition et 1 lexème pour la seconde, soit l'inverse. Pour choisir, nous générons un entier $j \in [1;2]$ aléatoirement et uniformément pour sélectionner le k-uplet. Nous obtenons $j = 1$, le k-uplet $(2,1)$ sera alors utilisé.
Pour la première répétition, il y a $n = 2$ lexèmes à distribuer. La disjonction peut produire 2 sous-structures. Nous allons utiliser une nouvelle fonction de répartition. La disjonction a trois branches : la première peut produire 1 sous-structure, la deuxième 0 et la dernière 1 sous-structure. La probabilité de parcourir la première branche ou la dernière branche est de $\frac{1}{2}$. Et ainsi de suite.

Cet algorithme est efficace pour calculer des petites séquences de taille fixe avec beaucoup de diversité dans les résultats (puisqu'aléatoire). Toutefois, la complexité de l'étape du dénombrement avec la fonction ψ est quadratique : $\mathcal{O}(n^2)$. Et malgré l'utilisation d'heuristiques inspirées de la programmation dynamique, pour une valeur de $n > 10$, au moins 2 heures de calcul sont nécessaires. Ce problème ne concerne que la première étape de l'algorithme, car l'étape de génération a une complexité linéaire $\mathcal{O}(n)$: la génération d'une séquence est la plus rapide de tous les algorithmes.

Génération exhaustive bornée

Cet algorithme permet de générer toutes les séquences de lexèmes dont la taille est au maximum n. C'est une génération exhaustive bornée. Des expériences [Marinov and Khurshid, 2001; Sullivan et al., 2004] ont montré que générer énormément de données de façon exhaustive peut être efficace pour détecter des erreurs car toutes les données sont testées.
À chaque construction du langage PP est associée une fonction $\beta(n, e)$ qui calcule le multi-ensemble contenant des séquences de lexèmes de taille n pour la construction grammaticale e. Un multi-ensemble, noté $\{A\}$, est un ensemble avec des répétitions : en effet, rien ne nous assure que la grammaire ne puisse pas produire au moins deux séquences identiques mais par des dérivations/chemins différents. Dans notre implémentation, cette fonction se comporte comme un itérateur où chaque pas calcule la séquence de lexèmes suivante. Pour simplifier l'écriture de la fonction β, nous considérons que la grammaire PP est en forme normale de Chomsky [Chomsky, 1956]. Ainsi :

$$\beta(1, e) = \{e\} \quad \text{si } e \text{ est un lexème}$$
$$\beta(n, e) = \{\} \quad \text{si } n \neq 1$$

$$
\begin{aligned}
\beta(n, e_1 \mid e_2) &= \beta(n, e_1) \cup \beta(n, e_2) \\
\beta(n, e_1 \cdot e_2) &= \bigcup_{p=1}^{n-1} \beta(p, e_1) \cdot \beta(n-p, e_2) \\
\beta(n, e^{\{x,y\}}) &= \bigcup_{p=x}^{y} \beta(n, e^p) \\
\beta(n, e^0) &= \{\} \\
\beta(n, e^1) &= \beta(n, e) \\
\beta(n, e^p) &= \beta(n, e \cdot e^{p-1}) \qquad \text{si } p \geqslant 2
\end{aligned}
$$

Les opérateurs $\cup$ et $\bigcup$ correspondent à l'union des multi-ensembles. Quand y est non défini (avec * et +), il est considéré égal à n.

Exemple 4.10 (Exploration exhaustive bornée)**.** Soit la même grammaire que dans l'exemple 4.9, à savoir :

```
f: <a> g()
g: ( <b> <c> | <d> | f() ){1,3}
```

Nous allons calculer les premières séquences générées pour $n = 10$. Nous commençons par explorer la règle `f`. Nous produisons `<a>`, puis nous explorons la règle `g` qui propose une répétition d'une disjonction. Nous commençons par la borne inférieure de la répétition puis la première branche de la disjonction, soit `<b> <c>`. La première séquence est donc : `<a> <b> <c>`. La séquence suivante revient au dernier point de choix pour prendre le chemin suivant, à savoir la deuxième branche de la disjonction. La deuxième séquence est donc : `<a> <d>`. Nous continuons avec le troisième point de choix de la disjonction qui est la règle `f`. Nous reprenons un chemin équivalent : `<a>` puis `g`. Nous explorons à nouveau notre répétition, nous choisissons la borne inférieure et le premier élément de la disjonction. La troisième séquence est donc : `<a> <a> <b> <c>`. Les séquences suivantes sont :

4. `<a> <a> <d>`
5. `<a> <a> <a> <b> <c>`
6. `<a> <a> <a> <d>`
7. `<a> <a> <a> <a> <b> <c>`

... ...

12150. `<a> <a> <a> <a> <a> <a> <a> <a> <a> <d>`

Cet algorithme est rapide pour calculer des petites séquences avec une complexité $\mathcal{O}(n)$ dans le meilleur des cas, et l'exhaustivité est très efficace pour détecter des

erreurs. Toutefois, le nombre de séquences générables peut être important. En effet, nous travaillons avec un multi-ensemble, soit un ensemble avec des répétitions car une même séquence peut être produite depuis plusieurs chemins différents si la grammaire n'est pas déterministe. Ainsi, dans le pire des cas, l'algorithme est exponentiel, dans le meilleur des cas, quand la grammaire est déterministe, il est linéaire.

Génération basée sur la couverture

Cet algorithme essaye de réduire l'explosion combinatoire de l'algorithme précédent. Il va générer des séquences en respectant deux critères de couverture sur la grammaire :

– une règle est dite couverte si et seulement si toutes ses sous-règles et tous ses lexèmes ont été couverts ;
– un lexème est dit couvert s'il a été utilisé dans une séquence.

Pour assurer une diversité dans les séquences générées, les points de choix sont résolus aléatoirement parmi les sous-règles non-couvertes. Une heuristique est utilisée pour borner les répétitions afin de réduire encore plus l'explosion combinatoire et garantir la terminaison de l'algorithme : l'opérateur * est déplié 0, 1 ou 2 fois, + est déplié 1 ou 2 fois, et $\{x, y\}$ est déplié x, $x+1$, $y-1$ et y fois. L'aspect aléatoire est représenté par la fonction $\text{rand}(i, j)$ dans la fonction ϕ.

À chaque construction du langage PP est associée une fonction $\phi(p, e)$ qui va calculer la prochaine séquence. Cette fonction prend en argument p, qui est le préfixe de la séquence déjà produite (vide au départ), et e est la construction grammaticale traitée. Dans notre implémentation, cet algorithme se comporte également comme un itérateur. Ainsi :

$$
\begin{aligned}
\phi(p, e) &= [e] && \text{si } e \text{ est un lexème} \\
\phi(p, e_1 \cdot e_2) &= \phi(\phi(p, e_1), e_2) \\
\phi(p, e_1 | \ldots | e_k) &= \phi(p, e_1) \oplus \ldots \oplus \phi(p, e_k) \\
\phi(p, e^?) &= [] \oplus \phi(p, e) \\
\phi(p, e^*) &= [] \oplus \bigoplus_{i=1}^{\text{rand}(0,2)} \phi(p, \underbrace{e \cdot \ldots \cdot e}_{i}) \\
\phi(p, e^+) &= \bigoplus_{i=1}^{\text{rand}(1,2)} \phi(p, \underbrace{e \cdot \ldots \cdot e}_{i}) \\
\phi(p, e^{\{x,y\}}) &= \bigoplus_{i=x}^{y} \phi(p, \underbrace{e \cdot \ldots \cdot e}_{i})
\end{aligned}
$$

Une séquence vide est représentée par []. Les opérateurs $\oplus$ et $\bigoplus$ désignent un choix entre plusieurs appels récursifs de la fonction.

Exemple 4.11 (Exploration basée sur la couverture)**.** Soit la même grammaire que dans les exemples précédents, à savoir :

```
f: <a> g()
g: ( <b> <c> | <d> | f() ){1,3}
```

Nous déroulons un exemple. Nous commençons par déplier la règle `f`. Nous produisons `<a>` avant d'explorer la règle `g`. Puis nous choisissons aléatoirement de répéter 3 fois la disjonction. La première fois, nous choisissons le lexème `<d>`. La seconde fois, nous avons le choix entre `<b> <c>` ou `f` (`<d>` ayant déjà été couvert, nous ne voulons pas le choisir). Nous choisissons `<b> <c>`. Enfin, à la troisième répétition, il n'y a plus qu'un seul choix n'ayant pas été couvert dans la disjonction : nous déplions `f` à nouveau. Nous produisons `<a>` et nous continuons dans `g`. Tout a été couvert, nous choisissons alors aléatoirement une répétition (disons 1 fois), puis `<d>` dans la disjonction. L'unique séquence `<a> <d> <b> <c> <a> <d>` couvre toutes les règles et tous les lexèmes de la grammaire.

Cet algorithme est rapide pour des séquences de taille moyenne ou grande avec une complexité logarithmique $\mathcal{O}(\log\ n)$. En pratique, l'algorithme de génération aléatoire uniforme est plus rapide pour calculer des séquences avec $n \leqslant 10$, mais il est rapidement dépassé par celui-ci au-delà. La diversité entre les séquences générées est dû au choix aléatoire sur les points de choix. Il génère peu de données mais il couvre toutes les règles et tous les lexèmes. Enfin, contrairement aux algorithmes précédents, nous ne pouvons pas contrôler la taille de la séquence.

Concrétisation aléatoire isotropique de lexèmes

Les trois algorithmes génèrent des séquences de lexèmes, mais nous n'avons pas de données concrètes. Pour cela, nous devons transformer les lexèmes en données afin de produire un mot.
Nous rappelons que les lexèmes sont exprimés avec des expressions régulières au format PCRE, qui est un langage comme un autre ! Afin d'être capable de transformer les lexèmes en données concrètes, nous devons les analyser. C'est pourquoi nous avons écrit la grammaire du langage PCRE avec le langage PP. Ainsi, nous analysons une expression régulière avec cette grammaire, le compilateur de compilateurs produit un AST que nous parcourons pour générer une donnée valide correspondant à l'expression régulière de départ. Ce processus est détaillé dans la figure 4.10. L'algorithme de génération est aléatoire isotropique [Sawyer, 1978] : c'est à dire qu'il est aléatoire uniforme mais l'uniformité est locale à un point de choix, et

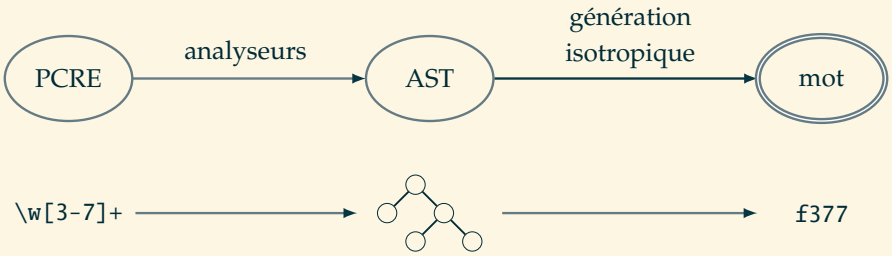


FIGURE 4.10 – Fonctionnement de la concrétisation.

non pas parmi toutes les sous-structures possibles. C'est une approche nettement plus naïve qu'avec les algorithmes précédents, mais la phase de concrétisation ne nécessite pas autant d'attention que la génération des séquences de lexèmes. Les lexèmes représentent l'unité atomique de nos langages, leur valeur n'est pas censée avoir d'influence lorsque nous utilisons la donnée concrétisée pour du test ou d'autres applications. De plus, nous sommes plus intéressés par la forme de la séquence (comme toutes les séquences de manière exhaustive ou couvrant toute la grammaire) car elle caractérise véritablement notre donnée.

Exemple 4.12 (Génération aléatoire isotropique)**.** Soit l'expression régulière suivante :

```
([ae]+|[x-y]!){1,3}
```

Nous allons parcourir cette expression comme si nous parcourions son AST, et à chaque point de choix, nous allons faire un tirage aléatoire uniforme pour continuer notre parcours :

`([ae]+|[w-z]!){1,3}`	nous choisissons le nombre de répétitions $r_1 \in [1;3]$, posons $r_1 = 2$;
`([ae]+|[w-z]!)([ae]+|[w-z]!)`	nous traitons la première répétition qui est une disjonction, nous traitons la branche $d_1 \in [1;2]$, posons $d_1 = 1$;
`([ae]+)([ae]+|[w-z]!)`	nous avons une répétition $r_2 \in [1;+\infty[$, posons $r_2 = 2$;
`[ae][ae]([ae]+|[w-z]!)`	nous avons une classe de caractères, c'est équivalent ici à une disjonction entre `a` et `e`, nous choisissons `e` ;
`e[ae]([ae]+|[w-z]!)`	nous traitons la seconde répétition de r_2 et nous choisissons `a` ;

`ea([ae]+|[w-z]!)`	nous traitons la seconde répétition de r_1 qui est une disjonction, nous traitons la branche $d_2 \in [1;2]$, posons $d_2 = 2$;
`ea([w-z]!)`	nous avons une classe de caractères qui est une plage entre `w` et `z`, nous choisissons `y` ;
`eay!`	nous finissons avec la concaténation, que nous avons déjà appliquée tout le long de cet exemple.

Chaque tirage a été fait aléatoirement et uniformément, mais localement.

Chaque opérateur non borné, comme * ou +, est borné à un entier arbitraire, par défaut 2.

Nous nous permettons une petite parenthèse afin d'être complet dans notre discours. Si nous souhaitions utiliser les algorithmes vus précédemment pour concrétiser les lexèmes (exprimés avec le langage PCRE), ceci demanderait un travail supplémentaire. En effet, il faudrait transformer le langage des PCRE vers le langage PP, et ainsi, nous nous retrouverions dans la même position qu'au début de cette partie où nous avions une grammaire et nos algorithmes de génération. Dans une telle grammaire, les nouveaux lexèmes seraient alors des littéraux (comme `a`, `b`, `0` et autre symbole « unitaire ») facilement générables avec un algorithme aléatoire isotropique. Pour résumer, cela reviendrait à écrire un compilateur PCRE vers PP. Ce travail a été commencé par un étudiant d'une autre école [Kühner, 2014] suite à un projet sur la visualisation graphique d'expressions régulières, réalisé avec notre compilateur et notre grammaire des PCRE.

Nous n'avons vu que très peu de constructions qu'offre la syntaxe PCRE, mais il est possible d'exprimer des choses bien plus complexes. Le lecteur intéressé pourra trouver plus de détails sur ce très riche langage dans le manuel des PCRE [Hazel, 1997].

Domaines réalistes `Grammar` et `Regex`

Les quatre algorithmes précédents complètent les caractéristiques d'un domaine réaliste en assurant la générabilité. Nous pouvons alors ajouter à la bibliothèque standard le domaine réaliste :

`grammar(F)`

où `F` est le chemin vers le fichier PP contenant la grammaire. Par défaut, ce domaine réaliste utilise la génération basée sur la couverture car elle produit peu de données

différentes, relativement vite et couvrant toute la grammaire. Bien sûr, il est possible de modifier l'algorithme à utiliser pour la génération (se reporter au chapitre 6). Dans ce cas, un second paramètre au domaine réaliste pourra être utilisé pour représenter la taille de la séquence souhaitée.

Exemple 4.13 (Utilisation du domaine réaliste `Grammar`)**.** La précondition du contrat suivant spécifie que l'argument `$payload` de la fonction `handleRequest` doit être au format XML et que la fonction doit retourner un booléen :

```
/**
 * @requires payload: grammar('Xml.pp');
 * @ensures  \result: boolean();
 */
function handleRequest ( $payload ) { ... }
```

Toutefois, l'expérience nous a montré que les développeurs utilisent plus souvent des expressions régulières que des grammaires. En effet, ils utilisent les PCRE quasi-quotidiennement. C'est pourquoi nous ajoutons à la bibliothèque standard le domaine réaliste :

`regex(R)`

où `R` est une expression régulière. Ce domaine réaliste utilise une génération aléatoire isotropique, ce qui correspond à l'usage qu'en font les développeurs. S'il y a besoin d'utiliser les autres algorithmes, il sera nécessaire de passer par une grammaire qui offre également plus de finesse dans les résultats. Ce domaine réaliste est tellement utilisé que Praspel propose une syntaxe simplifiée pour l'utiliser encore plus rapidement. Ainsi, une expression régulière doit être encadrée de délimiteurs, par exemple `/`, et peut être suivie d'options (un caractère active une option, comme `i` pour modifier la sensibilité à la casse). La syntaxe simplifiée de Praspel impose le délimiteur `/`.

Exemple 4.14 (Utilisation du domaine réaliste `Regex`)**.** La précondition du contrat suivant spécifie que l'argument `$ip` est une IPv4 valide, avec ou sans port :

```
/**
 * @requires ip: regex(
 *                   '/^(\*|\d{1,3}(\.\d{1,3}){3})(:\d+)?$/'
 *               );
 */
```

Cette expression régulière accepte des adresses IP de la forme : `*`, `*:80`, `127.0.0.1` ou `127.0.0.1:80`. Si nous devions l'écrire avec la syntaxe simplifiée, nous aurions :

```
/**
 * @requires ip: /^(\*|\d{1,3}(\.\d{1,3}){3})(:\d+)?$/;
 */
```

Les parties précédentes expliquent comment générer des données scalaires, telles que des booléens, des entiers, réels ou chaînes, et aussi des données plus complexes, telles que des tableaux. Nous allons à présent décrire comment générer des objets, puisque le paradigme le plus utilisé en PHP5 [4] est l'objet.

4.4 Génération d'objets

Un objet est une agrégation d'attributs et de méthodes. Les deux sont annotés par des contrats Praspel, respectivement par la clause `@invariant` et les autres clauses (voir les entités syntaxiques *attribute-clauses* et *method-clauses* de la figure 3.3 page 33, et de la figure 3.5 page 35 pour les positionner). Le domaine réaliste :

```
class(C)
```

représente une instance d'un objet de type C. L'algorithme de la méthode `sample` (représantant la caractéristique de générabilité) de ce domaine réaliste est présenté dans la figure 4.11 à travers la fonction Class_sample . L'entrée *class* représente la classe à instancier, les entrées *store*, *decision* et *pick* représentent respectivement une collection d'objects instanciés, une fonction de décision et une fonction pour obtenir un élément dans la collection. La stratégie pour générer un objet est de l'instancier en appelant son constructeur, et de définir une valeur à ses attributs. Les valeurs des arguments du constructeur sont générées grâce à sa précondition. Les valeurs des attributs sont générées grâce aux invariants. Si un argument du constructeur ou un attribut est lui-même un objet (par conséquent spécifié avec le domaine réaliste `Class`), ce processus est répété récursivement. Nous contournons l'encapsulation pour accéder aux attributs ou méthodes protégés et privés en instrumentant le code source.
Afin de gérer les références circulaires (par exemple quand un premier objet a un attribut qui pointe sur un deuxième objet, qui a lui-même un attribut qui pointe vers le premier objet), nous appliquons une stratégie inspirée de Jartege [Oriat, 2005] et UDITA [Gligoric et al., 2010]. Cette stratégie consiste à enregistrer les objets créés dans une collection. Quand un objet est demandé, nous avons le choix entre en générer un nouveau ou en choisir un dans la collection. Ce choix est dynamiquement réalisé en fonction de l'état de la collection : plus elle contient d'objets, moins il y aura de nouveaux objets générés. Ce choix est réalisé à l'aide de la fonction placée dans l'entrée *decision* par l'utilisateur.

4. Sorti en juillet 2004.

```
function Class_sample
    input  class : Class
    input  store : set⟨Object⟩
    input  decision : λ(class : Class × store : set⟨Object⟩ → boolean)
    input  pick : λ(store : set⟨Object⟩ → Object)
    output  object : Object
    if false  = decision(class, store) then
        object ← pick(store)
    else
        constructor : Method ← class.getConstructor()
        object ← constructor.call(Generate_data(constructor))
        for all attribute ∈ class.getAttributes(object) do
            specification : Praspel ← attribute.getSpecification()
            invariant : Clause ← specification.getClause('invariant')
            attribute.value ← invariant.sample()
        end for
    end if
    return object
end function
```

FIGURE 4.11 – Caractéristique de générabilité du domaine réaliste `Class`.

4.5 Synthèse

Dans ce chapitre, nous avons présenté les différentes techniques de génération de données utilisées dans Praspel. Nous avons d'abord présenté la technique utilisée par défaut, qui est pseudo-aléatoire. Nous avons observé un grand nombre de mauvaises données générées qui introduisaient du rejet lorsque nous manipulions des données structurelles ou complexes, comme les tableaux, les chaînes de caractères ou les objets. Nous avons abordé ces problèmes un par un.
Nous avons proposé un solveur de contraintes sur les tableaux. Le langage Praspel a été étendu pour gérer nativement plusieurs contraintes sur ces derniers. Nous avons au préalable effectué une étude sur des dizaines de projets PHP afin de connaître les contraintes les plus populaires sur les tableaux. Ces travaux ont été publiés dans l'article Enderlin et al. [2013].
Nous avons également introduit une approche *grammar-based testing* dans Praspel avec les domaines réalistes `grammar` et `regex`. Ces derniers permettent de valider et générer des données textuelles complexes. Pour y arriver, nous avons écrit notre propre compilateur de compilateurs LL($\star$) avec son langage de description de grammaire associé, appelé PP. Nous avons proposé trois algorithmes de génération de données à partir d'une grammaire pour différents usages et contextes : des données variées, aléatoires, de taille fixe, bornée ou encore libre etc. Ces travaux ont été publiés dans l'article Enderlin et al. [2012].

Nous connaissons le langage et nous savons comment générer des données satisfaisant des contraintes. Dans le chapitre suivant, nous allons aborder la génération des tests à partir d'un contrat.

Chapitre 5.
Critères de couverture de contrat

Table des matières

Le langage de contrat Praspel et la génération de données pour plusieurs types ont été présentés dans les chapitres précédents. L'étape suivante est la génération automatique de tests unitaires. Pour accomplir cette étape, il est nécessaire d'avoir des objectifs de test. Comme nous sommes en boîte noire, nous allons une fois de plus exploiter le contrat pour obtenir nos objectifs de test.
Dans ce chapitre, nous définissons des critères de couverture sur les contrats. Ces critères vont nous offrir des objectifs de test. Dans la partie 5.1, nous transformons

les contrats en graphe. Ce formalisme nous permet de plus facilement définir et exprimer les critères sous la forme d'un chemin dans un graphe. Il sera également plus facile de vérifier qu'un critère est satisfait. Cette démarche est similaire à la définition des critères de couverture de code [Miller and Maloney, 1963]. Sur la base de quelques définitions introduites dans la partie 5.2, nous établissons les critères de couverture sur un contrat transformé en graphe dans la partie 5.3.

5.1 Couverture de contrat

Afin de définir des critères de couverture sur les contrats, nous transformons ces derniers en graphes. Dans cette partie, nous expliquons comment construire le graphe d'un contrat Praspel en forme normale (défini dans le chapitre 3 page 27). Ce graphe reflète la structure, les déclarations de domaines réalistes et les prédicats d'un contrat.
Dans toute la suite, nous notons A l'ensemble composé de toutes les expressions Praspel générées par les entités syntaxiques *expression* et *exception-identifier* de la grammaire de la figure 3.7, et de toutes les expressions de la forme $\neg t_1 \wedge \cdots \wedge \neg t_n$ où t_i est généré par l'entité syntaxique *expression*, pour $1 \leqslant i \leqslant n$. Nous considérons deux familles de graphes : les **graphes d'expressions** et les **graphes de contrats**, définis comme suit.

Définition 5.1 (Graphe d'expression)**.** Un **graphe d'expression** est un tuple (V, D, i, n) où V est un ensemble fini de **sommets**, $D \subseteq V \times A \times V$ est un ensemble fini d'**arcs** de V dans V annotés par une **expression** dans A, $i \in V$ est le **sommet initial** et $n \in V$ est le **sommet final**.

Définition 5.2 (Graphe de contrat)**.** Un **graphe de contrat** est un tuple (V, D, i, N, E, U) où V est un ensemble fini de **sommets**, $D \subseteq V \times A \times V$ est un ensemble fini d'**arcs** de V dans V annotés par une expression dans A, $i \in V$ est le **sommet initial**, N est un ensemble de **sommets finaux normaux**, E est un ensemble de **sommets finaux exceptionnels** et U est un ensemble de **sommets erreurs**, avec $N \uplus E \uplus U \subseteq V$, où $\uplus$ représente l'union disjointe.

La partie 5.1.1 transforme n'importe quelle expression Praspel en un graphe d'expression. Les autres parties définissent la transformation de tout contrat Praspel en graphe de contrat. Le cas le plus simple est celui d'un contrat sans comportement, appelé **contrat atomique**. Il est considéré dans les parties 5.1.2 et 5.1.3. Ensuite, les graphes de contrat avec des comportements sont définis par induction. Le cas de base, avec seulement un comportement, est traité dans la partie 5.1.4. Le pas d'induction considérant un ou plusieurs comportements est traité dans la partie 5.1.5.

Le cas d'un contrat avec une clause `@default` (respectivement `@invariant`) est traité dans la partie 5.1.6 (respectivement 5.1.7).

5.1.1 Graphe d'une expression

Les expressions Praspel en forme normale sont décrites dans la figure 3.7 page 38 par l'entité syntaxique *expression*. Une expression est une conjonction de déclarations, de prédicats et de qualifications, respectivement décrits dans la figure 3.8 page 39 par les entités syntaxiques *declaration*, *predicate* et *qualification*. Nous transformons seulement les deux premières sortes d'expressions car les expressions de sorte *qualification* ne sont pas pertinentes pour la couverture : elles représentent un seul comportement (par exemple `a is unique` dans le cas d'un tableau `a`) et sont toujours évaluées. Les expressions que nous manipulons sont de la forme `d and p`, où `d` (respectivement `p`) est une conjonction d'expressions de sorte *declaration* (respectivement *predicate*). Son graphe d'expression est la concaténation par une transition ε du **graphe des domaines** obtenu à partir de `d` et du **graphe des prédicats** obtenu à partir de `p`. Ces deux constructions sont détaillées ci-dessous.

Graphe des domaines

Une conjonction de déclarations a la forme générale $\mathtt{i}_1\mathtt{:}\ \mathtt{D}_1\ \mathtt{and}\ \ldots\ \mathtt{and}\ \mathtt{i}_m\mathtt{:}\ \mathtt{D}_m$, où $\mathtt{D}_j = \mathtt{d}_{j,1}\ \mathtt{or}\ \ldots\ \mathtt{or}\ \mathtt{d}_{j,k_j}$ est une disjonction de domaines réalistes, pour $1 \leqslant j \leqslant m$. Nous notons aussi par $\mathtt{D}_j$ la liste $[\mathtt{d}_{j,1}, \ldots, \mathtt{d}_{j,k_j}]$ de domaines réalistes dans la disjonction. Soit $S = [\mathtt{i}_1, \ldots, \mathtt{i}_m]$ la liste des identifiants déclarés dans cette expression. La notation $s \in S$ signifie que s est un élément de la liste S. Pour construire le graphe des domaines, nous utilisons la définition suivante.

Définition 5.3 (Graphe des domaines)**.** Le **graphe des domaines** associé à l'expression $\mathtt{i}_1\mathtt{:}\ \mathtt{D}_1\ \mathtt{and}\ \ldots\ \mathtt{and}\ \mathtt{i}_m\mathtt{:}\ \mathtt{D}_m$ est :

$$G = (\{dom\} \cup \{v_s \mid s \in S\}, D, dom, v_{i_m})$$

avec :

$$D = \bigcup_{d \in D_1} \{(dom, \mathtt{i}_1\mathtt{:}\ d, v_{i_1})\} \cup \bigcup_{2 \leqslant j \leqslant m} \bigcup_{d \in D_j} \{(v_{i_{j-1}}, \mathtt{i}_j\mathtt{:}\ d, v_{i_j})\}$$

Les étiquettes *dom* et *pred* (dans la partie suivante) permettent de typer les graphes. Ainsi, il est plus facile de savoir si ce sont des domaines ou des prédicats qui sont traités lors du parcours du graphe.

Exemple 5.1 (Deux déclarations de domaines réalistes)**.** Dans l'expression suivante :

```
i: foo() or bar() and j: baz() or qux()
```

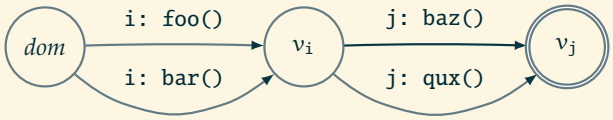


FIGURE 5.1 – Graphe des domaines.

nous avons $S = [\texttt{i}, \texttt{j}]$, $D_1 = [\texttt{foo()}, \texttt{bar()}]$ et
$D_2 = [\texttt{baz()}, \texttt{qux()}]$. La figure 5.1 présente graphiquement le graphe des domaines associé à cet exemple.

Graphe des prédicats

Un graphe des prédicats est similaire à un graphe des domaines car la forme de l'expression est très similaire. En effet, trivialement, nous pouvons dire que l'opérateur `and` est remplacé par `&&` et que `or` est remplacé par `||`. Plus formellement, un prédicat Praspel est de la forme générale `\pred(p)`, où `p` est une chaîne de caractères contenant une expression logique PHP en forme normale disjonctive (*Disjunctive Normal Form*, abrégé DNF), c'est à dire que `p` est de la forme P_1 `and` `...` `and` P_m, où $P_j = p_{j,1}$ `or` `...` `or` p_{j,k_j} est une disjonction de variables ou de prédicats PHP (fonctions booléennes ou opérations logiques), pour $1 \leqslant j \leqslant m$. Nous notons aussi par P_j la liste $[p_{j,1}, \ldots, p_{j,k_j}]$ de variables ou prédicats PHP déclarés dans la conjonction. Pour construire le graphe des prédicats, nous utilisons la définition suivante.

Définition 5.4 (Graphe des prédicats)**.** Le **graphe des prédicats** associé à l'expression `\pred('`P_1 `&&` `...` `&&` P_m`')` est :

$$G = (\{pred\} \cup \{v_j \mid j \in [1; m]\}, D, pred, v_m)$$

avec :

$$D = \bigcup_{p \in P_1} \{(pred, p, v_1)\} \cup \bigcup_{2 \leqslant j \leqslant m} \bigcup_{p \in P_j} \{(v_{j-1}, p, v_j)\}$$

Exemple 5.2 (Graphe d'un prédicat)**.** Dans l'expression suivante :

```
\pred('($a || $b) && ($c || $d)')
```

nous avons $m = 2$, $P_1 = [\texttt{\$a}, \texttt{\$b}]$ et $P_2 = [\texttt{\$c}, \texttt{\$d}]$. La figure 5.2 présente graphiquement le graphe associé à cet exemple de prédicat.

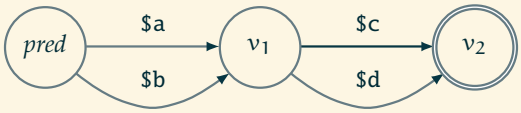


FIGURE 5.2 – Graphe des prédicats.

5.1.2 Graphe d'un contrat atomique

Un **contrat atomique** est un contrat composé uniquement des clauses `@requires`, `@ensures` et `@throwable`. Il ne contient ni clauses `@behavior`, ni `@default`, c'est à dire aucun sous-contrat, et n'est, par conséquent, pas décomposable en sous-contrats. Un contrat atomique en forme normale adopte la forme générale suivante :

```
@requires  P;
@ensures   Q;
@throwable T_C with T_E;
```

où P et Q sont des expressions de sorte *expression* respectivement associées aux clauses `@requires` et `@ensures`. L'expression de sorte *exceptional-expression* associée à la clause `@throwable` est composée de deux parties : T_C représente le nom de la classe de l'exception et l'identifiant qui porte l'exception, soit une construction de sorte *exception-identifier*, et T_E représente la postcondition exceptionnelle, qui est aussi une expression de sorte *expression*. Le graphe de contrat G associé à ce contrat atomique est :

$$G = (\{v_1, v_2, v_3, v_4, v_5, v_6\}, D, v_1, \{v_3\}, \{v_5\}, \{v_6\})$$

où

$$D = \{(v_1, P, v_2), (v_2, Q, v_3), (v_2, T_C, v_4), (v_4, T_E, v_5), (v_2, \neg T_C, v_6)\}$$

La figure 5.3 présente graphiquement ce graphe. Dans cette figure ainsi que dans toutes celles qui vont suivre, les sommets normaux finaux sont représentés par un double cercle, les sommets finaux exceptionnels par un double rectangle et les sommets d'erreurs par un cercle avec des tirets.

5.1.3 Contrat atomique avec plusieurs clauses `@throwable`

La partie 5.1.2 considérait un contrat avec un seule clause `@throwable`. Dans cette partie, nous voyons comment un contrat est modifié quand nous ajoutons une clause `@throwable` T_C `with` T_E à un contrat dont le graphe est $G = (V, D, i, N, E, U)$. Nous pouvons montrer par récurrence sur le nombre de clauses `@throwable` que l'ensemble des sommets V contient au moins les six sommets $v_1 = i$, v_2, v_3, v_4, v_5 et v_6 introduits dans la partie 5.1.2.

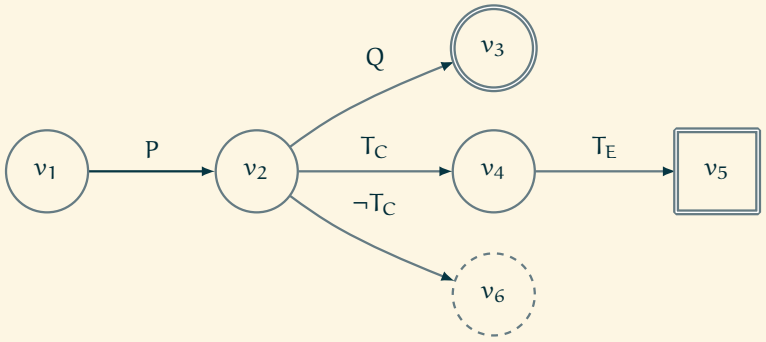


FIGURE 5.3 – Graphe correspondant à un contrat atomique.

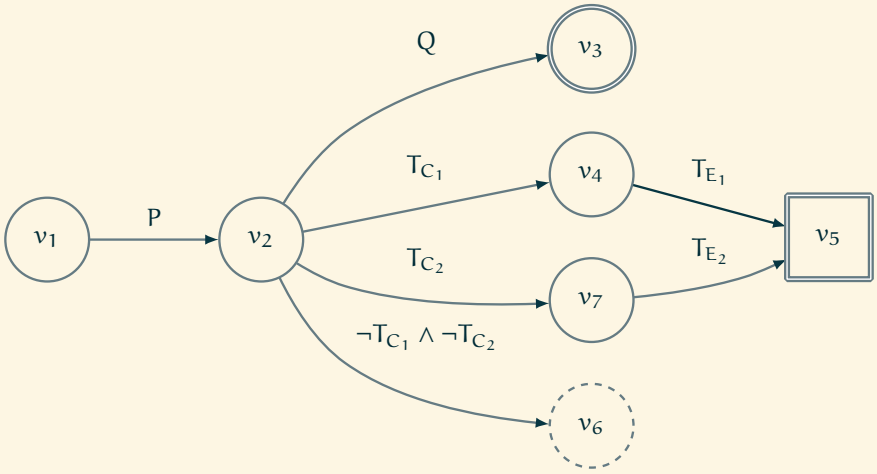


FIGURE 5.4 – Graphe correspondant à un contrat atomique avec deux clauses `@throwable`.

Le graphe résultant est $G' = (V \cup \{v'\}, D', i, N, E, U)$ avec un nouveau sommet $v' \notin V$ et un ensemble d'arcs :

$$D' = D \cup \{(v_2, T_C, v'), (v', T_E, v_5)\} \\ - \{(v_2, \varphi, v_6) \mid (v_2, \varphi, v_6) \in D\} \cup \{(v_2, \varphi \wedge \neg T_C, v_6) \mid (v_2, \varphi, v_6) \in D\}$$

avec une expression $\varphi \in A$ quelconque.

Exemple 5.3 (Graphe d'un contrat atomique avec deux clauses `@throwable`)**.** La figure 5.4 présente graphiquement ce graphe quand il y a deux clauses : `@throwable` T_{C_1} `with` T_{E_1} et `@throwable` T_{C_2} `with` T_{E_2} (dans la figure, v' est noté v_7).

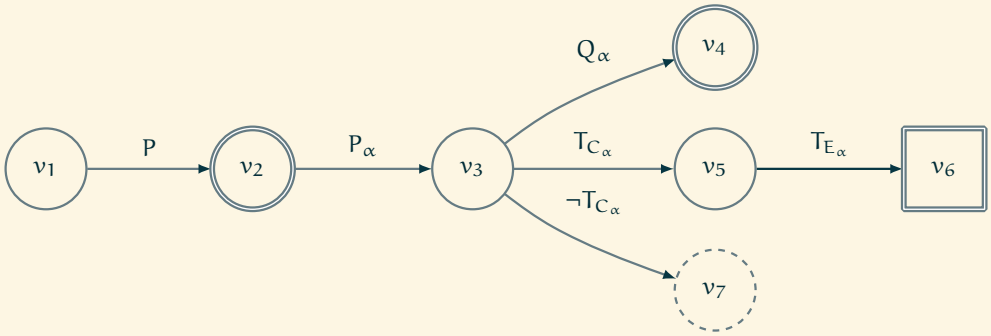


FIGURE 5.5 – Graphe correspondant à un contrat avec une seule clause `@behavior`.

5.1.4 Graphe d'un contrat avec une clause `@behavior`

Considérons à présent le cas d'un contrat contenant exactement une clause `@behavior`. Soit C le contrat :

```
@requires P;
@behavior α { B }
```

où B est le contrat du comportement α. Soit $G_\alpha = (V_\alpha, D_\alpha, i_\alpha, N_\alpha, E_\alpha, U_\alpha)$ le graphe de contrat associé à B. Alors, le graphe de contrat associé au contrat C est :

$$G = (\{v\} \cup V_\alpha, \{(v, P, i_\alpha)\} \cup D_\alpha, v, \{i_\alpha\} \cup N_\alpha, E_\alpha, U_\alpha)$$

avec un nouveau sommet $v \notin V_\alpha$. Ce graphe est composé de tous les sommets et arcs de G_α, complété avec un nouvel arc entre v, le nouveau sommet initial, et i_α, le sommet initial de G_α. Le sommet i_α devient également un sommet final normal car la sémantique du contrat est qu'il n'est pas obligatoire d'emprunter un comportement. Par conséquent, P peut être évalué mais sans activer le comportement α.

Exemple 5.4 (Graphe d'un contrat avec une clause `@behavior`)**.** La figure 5.5 présente graphiquement le graphe du contrat :

```
@requires P;
@behavior α {
    @requires  Pα;
    @ensures   Qα;
    @throwable TCα with TEα;
}
```

Le contrat du comportement α est atomique et contient une seule clause `@throwable`. Dans la figure, les sommets sont renumérotés à partir de v_1.

5.1.5 Contrat avec plusieurs clauses `@behavior`

La partie 5.1.4 considérait un contrat avec une seule clause `@behavior`. Dans cette partie, nous voyons comment ajouter une clause `@behavior` β à la liste des comportements dont le graphe de contrat est $G = (V, D, i, N, E, U)$. Soit $G_\beta = (V_\beta, D_\beta, i_\beta, N_\beta, E_\beta, U_\beta)$ le graphe de contrat associé au comportement β. *Modulo* quelques renommages, supposons que V contient un sommet ν tel que D contient un arc (i, P, ν). Supposons aussi que tous les sommets de G_β sont distincts de ceux de G ($V \cap V_\beta = \emptyset$). Alors, le nouveau graphe de contrat est $G' = (V \cup V_\beta - \{i_\beta\}, D \cup D'_\beta, i, N \cup N'_\beta, E \cup E_\beta, U \cup U_\beta)$, où D'_β (respectivement N'_β) est obtenu de D_β (respectivement N_β) en remplaçant i_β par ν.

Exemple 5.5 (Graphe d'un contrat avec deux clauses `@behavior`)**.** La figure 5.6 présente graphiquement le graphe du contrat :

```
@requires P;
@behavior α {
    @requires  Pα;
    @ensures   Qα;
    @throwable TCα with TEα;
}
@behavior β {
    @requires  Pβ;
    @ensures   Qβ;
    @throwable TCβ with TEβ;
}
```

Les sous-contrats des deux comportements sont atomiques et ne contiennent chacun qu'une seule clause `@throwable`.

5.1.6 Cas de la clause `@default`

Après un groupe de clauses `@behavior`, nous pouvons avoir une clause `@default`. Rappelons qu'une clause `@default` est strictement équivalente à une clause `@behavior` avec une clause `@requires` implicite (voir la partie 3.2.1 page 34). Ainsi, ajouter une clause `@default` est un cas spécial d'ajout d'une clause `@behavior`, considéré dans la partie 5.1.5.

5.1.7 Cas de la clause `@invariant`

Dans cette dernière partie, nous voyons comment intégrer une clause `@invariant` I (portant sur les attributs de classe) dans le graphe de contrat d'une méthode G_M =

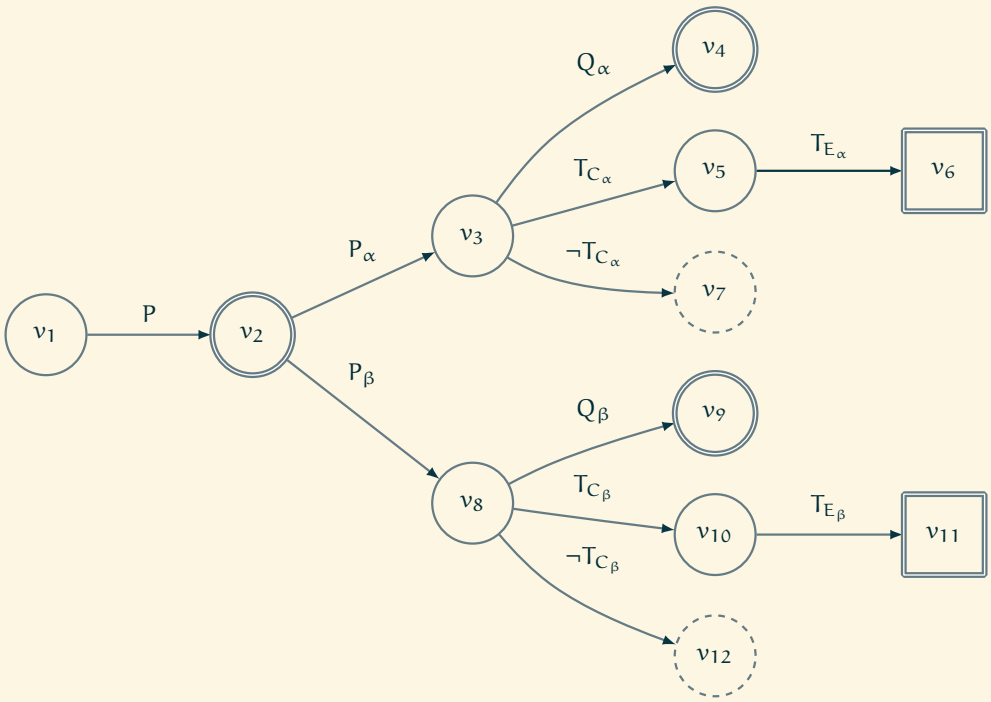


FIGURE 5.6 – Graphe correspondant à un contrat avec deux clauses `@behavior`.

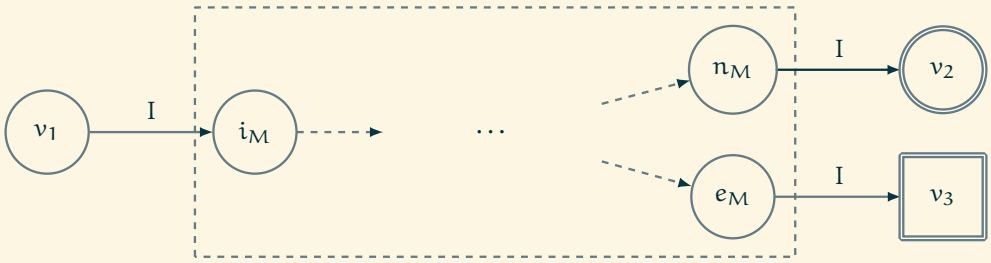


FIGURE 5.7 – Graphe correspondant à contrat avec un invariant.

$(V_M, D_M, i_M, N_M, E_M, U_M)$. Nous supposons que l'ensemble V_M ne contient pas les sommets v_1, v_2 et v_3. Le graphe résultant est :

$$G = (\{v_1, v_2, v_3\} \cup V_M, D, v_1, \{v_2\}, \{v_3\}, U_M)$$

avec

$$D = \{(v_1, I, i_M)\} \cup \{(n, I, v_2) \mid n \in N_M\} \cup \{(e, I, v_3) \mid e \in E_M\}$$

Le graphe G contient trois sommets supplémentaires et plusieurs nouveaux arcs, dont la signification est que l'invariant I doit être satisfait avant et après l'exécution de la méthode, et dans tous les cas, que ce soit une terminaison normale ou exceptionnelle.

Exemple 5.6 (Graphe d'un contrat d'une clause `@invariant`)**.** La figure 5.7 illustre ce principe d'extension d'un graphe de contrat avec un invariant. Dans un souci de simplicité, elle représente seulement un sommet final normal $n_M \in N_M$ et un sommet final exceptionnel $e_M \in E_M$.

5.2 Définitions

Cette partie présente plusieurs définitions sur les chemins, les tests, les cas de tests etc. Ces définitions sont importantes pour exprimer et comprendre correctment les critères de couverture.

Tout au long de cette partie, nous considérons une méthode PHP `f` et nous désignons par $G(f)$, ou G, son contrat (V, D, i, N, E, U), comme défini dans la partie 5.1.

Définition 5.5 (Chemins dans un graphe de contrat)**.** Un **chemin** est une séquence de transitions consécutives séparées par des points. Deux transitions (v, α, w) et (x, β, y) sont consécutives si et seulement si $w = x$. Nous disons qu'un sommet v **est dans** le chemin p si et seulement si le chemin p contient une transition (v, α, w) ou (w, α, v).

Nous associons deux sortes de chemins à un contrat $G = (V, D, i, N, E, U)$.

Définition 5.6 (Routes et sentiers dans un graphe de contrat)**.** Une **route** de G est un chemin dont les transitions sont dans D. Soit r une route de G. Remplaçons toutes les transitions (v, e, w) dans r par le chemin $(v, \varepsilon, i_H) \cdot q \cdot (n_H, \varepsilon, w)$ où q est un chemin de i_H à n_H du graphe $H = (V_H, D_H, i_H, n_H)$ de l'expression e. Le résultat est appelé un **sentier** de r. Un sentier d'un graphe de contrat G est n'importe quel sentier obtenu ainsi, à partir d'une route de G.

Définition 5.7 (Test unitaire, cas de test et suite de tests)**.** Un **test unitaire** pour une méthode `f` est un ensemble de données (appelées entrées), qui fixent les valeurs pour les arguments de `f`. Un **cas de test** est une paire (s, t) composée d'un état s (le contexte dans lequel est exécuté le test) et un test unitaire t. Pour construire l'état s, nous avons deux approches :

1. créer un objet et invoquer ses méthodes afin de changer son état ; ou
2. instrumenter le code pour détourner l'encapsulation et définir l'état de l'objet directement.

Une **suite de tests** est un ensemble de cas de test.

Quand nous exécutons un cas de test, nous sommes capables d'y associer des chemins du graphe de contrat G, appelés chemins **activés par** le cas de test, démarrant depuis l'état initial de $G(\texttt{f})$ et terminant dans un de ses états finaux (soit normaux soit exceptionnels).
Nous représentons par s' l'état après l'exécution du test, par o la valeur retournée par la méthode `f` et par e l'exception levée le cas échéant. Remarquons que si la méthode lève une exception, alors o est indéfinie. À l'inverse, si la méthode termine normalement (c'est à dire sans lever d'exception), e est considérée comme indéfinie.
Afin de définir quels sont les chemins (autant les routes que les sentiers) d'un contrat qui sont activés par un cas de test (s, t), nous explorons le graphe $G(\texttt{f})$ et nous collectons les chemins en évaluant les étiquettes des transitions tout en respectant le test t :

– si la transition est étiquetée par une expression R (respectivement I) provenant d'une précondition (respectivement d'un invariant évalué dans le pré-état), la transition est activée si et seulement si R (respectivement I) est vraie dans le pré-état s ;
– si la transition est étiquetée par une expression E provenant d'une postcondition (normale ou exceptionnelle), la transition est activée si et seulement si E est vraie, quand `\result` est remplacée par la valeur retournée o, les sous-expressions `\old()` sont remplacées par leur valeur dans le pré-état s, et les autres sous-expressions sont évaluées dans le post-état s' ;

- si la transition est étiquetée par une expression I provenant d'un invariant évalué dans le post-état, la transition est activée si et seulement si I est vraie dans le post-état s' ;
- si la transition est étiquetée par une déclaration de domaine réaliste `i: d`, la transition est activée si et seulement si la valeur courante de $i \in t$ appartient au domaine réaliste `d` (grâce à la caractéristique de prédicabilité). Quand cette déclaration provient d'une précondition (c'est à dire dans un graphe d'expression d'une précondition), la valeur de `i` est choisie dans le pré-état s du test si `i` est un attribut de classe, ou dans les arguments de la méthode si `i` est un argument de la méthode. Quand la déclaration est exprimée dans une postcondition (normale ou exceptionnelle), la valeur de `i` est choisie dans le post-état s' ;
- si la transition est étiquetée par un prédicat, la transition est activée si et seulement si le prédicat est satisfait avec l'état courant du système (s dans le cas d'une précondition, s' pour un prédicat *before-after*, c'est à dire contenu dans une postcondition, dont les expressions `\old()` sont évaluées dans le pré-état s) ;
- si la transition est étiquetée par un *exception-identifier* T_C, alors la transition est activée si et seulement si l'exception levée e n'est pas indéfinie et est une instance de T_C.

Il est possible qu'un test active plusieurs chemins en même temps. Par exemple, dans la figure 5.4, si T_{C_1} étend T_{C_2} et qu'une exception de classe T_{C_1} est levée, alors les chemins $v_1 \cdot v_2 \cdot v_4 \cdot v_5$ et $v_1 \cdot v_2 \cdot v_7 \cdot v_5$ seront activés.

Nous représentons également par $R_f(S)$ l'ensemble des routes de $G(f)$ activées par une suite de tests S. Similairement, nous représentons par $T_f(S)$ l'ensemble des sentiers de $G(f)$ activés par une suite de tests S et traversant des graphes d'expressions.

Les exemples suivants sont basés sur le contrat de la figure 5.8 pour la méthode `compute` avec deux arguments : `server` qui doit être une classe de type `\Irc\Server` et `buffer` qui doit contenir un message IRC. Le graphe de contrat associé est présenté dans la figure 5.9. Tous les états ont été renommés en v_i avec $1 \leqslant i \leqslant 22$, pour éviter toute ambiguïté lorsque nous parlerons des chemins. Les graphes d'expressions sont représentés, ce qui permet de parcourir tous les sentiers. Les routes sont représentées par des arcs en pointillés. Certaines étiquettes sont tronquées pour réduire la taille du graphe. Dans cet exemple, nous avons deux comportements, un pour un message privé ou public (représenté par une expression régulière à travers le domaine réaliste `regex` sous sa forme simplifiée) et un pour un « ping ». Dans le dernier comportement, la précondition a un prédicat pour spécifier que le réseau est dans un état spécifique. Enfin, le comportement par défaut spécifie qu'une exception de type `\Irc\Exception\MalformedMessage` doit être levée si le tampon n'est ni un message ni un ping. De plus, le code d'une exception doit être un entier

```
/**
 * @requires server: class('\Irc\Server');
 * @behavior message {
 *     @requires buffer: /^privmessage .+/ or /^message .+/;
 *     @ensures  \result: 1..;
 * }
 * @behavior ping {
 *     @requires buffer: /^ping$/ and
 *               \pred('$server->bufferState   >= 0 or
 *                      network_buffer_state() >  0');
 *     @ensures  \result: 1..;
 * }
 * @default {
 *     @throwable \Irc\Exception\MalformedMessage e
 *                    with e->code: 400..491;
 * }
 */
public static function compute ( $server, $buffer ) { ... }
```

FIGURE 5.8 – Contrat et méthode manipulant un flux IRC.

entre 400 et 491.

Exemple 5.7 (Test et chemin)**.** La syntaxe `object(c)` représente une instance valide d'une classe c. Le test t_1 = `compute(object(\Irc\Server), 'message foobar')` va activer le chemin $(v_1, P, v_4) \cdot (v_4, P_{\texttt{message}}, v_7) \cdot (v_7, Q_{\texttt{message}}, v_{10})$. En effet, la donnée `object(\Irc\Server)` représente la valeur de `$server`. Cet objet est bien une instance de la classe `\Irc\Server`, donc la transition entre v_1 et v_4 est activée. Ensuite, la donnée `'message foobar'` représente la valeur de `$buffer`. Cette donnée active la transition entre v_4 et v_7 étiquetée par `buffer: /^privmessage .+/ or /^message .+/` car la donnée est une chaîne de caractères qui commence par `message`.

5.3 Critères de couverture

Nous définissons maintenant les critères de couverture de contrat, par des propriétés de l'ensemble des chemins du graphe associé à un contrat Praspel annotant une méthode PHP.

FIGURE 5.9 – Graphe de contrat associé à la figure 5.8.

5.3.1 Critère de Couverture de Clause

Le Critère de Couverture Clause vise à couvrir la structure d'un contrat. Ainsi, le Critère de Couverture de Clause est satisfait par une suite de tests S d'une méthode `f` de graphe de contrat $G(f) = (V, D, i, N, E, U)$ si tous les états de $V \backslash U$ sont dans une route de $R_f(S)$.

Exemple 5.8 (Suite de tests satisfaisant le Critère de Couverture de Clause)**.** La suite de tests $\{t_1, t_2, t_3\}$ avec :
– t_1 = `compute(object(\Irc\Server), 'message foobar');`
– t_2 = `compute(object(\Irc\Server), 'ping');`
– t_3 = `compute(object(\Irc\Server), 'xyz')`.
satisfait le Critère de Couverture de Clause, c'est à dire que ses cas de tests couvrent toutes les clauses, avec les chemins respectifs suivants (sur les routes) :

$$\begin{aligned}
R_{\texttt{compute}}(\{t_1\}) = \{\ & (v_1, P, v_4) \\
& \cdot (v_4, P_{\texttt{message}}, v_7) \\
& \cdot (v_7, Q_{\texttt{message}}, v_{10})\ \} \\
R_{\texttt{compute}}(\{t_2\}) = \{\ & (v_1, P, v_4) \\
& \cdot (v_4, P_{\texttt{ping}}, v_{15}) \\
& \cdot (v_{15}, Q_{\texttt{ping}}, v_{18})\ \} \\
R_{\texttt{compute}}(\{t_3\}) = \{\ & (v_1, P, v_4) \\
& \cdot (v_4, P_{\mathcal{D}}, v_{19}) \\
& \cdot (v_{19}, T_{\mathcal{D}}, v_{23})\ \}
\end{aligned}$$

5.3.2 Critère de Couverture de Domaine

Le Critère de Couverture de Domaine raffine le Critère de Couverture de Clause en considérant la déclaration de domaines réalistes à l'intérieur des graphes d'expression. Plus formellement, le Critère de Couverture de Domaine est satisfait par une suite de tests S pour une méthode `f` si toutes les transitions de la forme `i: d` dans le graphe de contrat $G(f)$ (qui sont dans les graphes d'expressions étiquetant les transitions de $G(f)$ et contenant le nœud *dom*) apparaissent dans au moins un sentier de $T_f(S)$.

Exemple 5.9 (Suite de tests satisfaisant le Critère de Couverture de Domaine)**.** La suite de tests $\{t_1, t_2, t_4\}$ avec :
– t_4 = `compute(object(\Irc\Server), 'privmessage foobar')`.
satisfait le Critère de Couverture de Domaine : la variable `server`, les trois déclarations de domaines réalistes de la variable `buffer` et la variable `\result` sont

couvertes. Cette suite de tests produit les chemins respectifs suivants (sur les sentiers) :

$$
\begin{aligned}
T_{\texttt{compute}}(\{t_1\}) = \{\ & (v_1, \varepsilon, v_2) \\
& \cdot (v_2, \texttt{server: class('\Irc\Server')}, v_3) \\
& \cdot (v_3, \varepsilon, v_4) \\
& \cdot (v_4, \varepsilon, v_5) \\
& \cdot (v_5, \texttt{buffer: /\^{}message .+/}, v_6) \\
& \cdot (v_6, \varepsilon, v_7) \\
& \cdot (v_7, \varepsilon, v_8) \\
& \cdot (v_8, \texttt{\textbackslash result: 1..}, v_9) \\
& \cdot (v_9, \varepsilon, v_{10})\ \} \\
T_{\texttt{compute}}(\{t_2\}) = \{\ & (v_1, \varepsilon, v_2) \\
& \cdot (v_2, \texttt{server: class('\Irc\Server')}, v_3) \\
& \cdot (v_3, \varepsilon, v_4) \\
& \cdot (v_4, \varepsilon, v_{11}) \\
& \cdot (v_{11}, \texttt{buffer: /\^{}ping\$/}, v_{12}) \\
& \cdot (v_{12}, \varepsilon, v_{13}) \\
& \cdot (v_{13}, \texttt{\textbackslash pred(...)}, v_{14}) \\
& \cdot (v_{14}, \varepsilon, v_{15}) \\
& \cdot (v_{15}, \varepsilon, v_{16}) \\
& \cdot (v_{16}, \texttt{\textbackslash result: 1..}, v_{17}) \\
& \cdot (v_{17}, \varepsilon, v_{18})\ \} \\
T_{\texttt{compute}}(\{t_4\}) = \{\ & (v_1, \varepsilon, v_2) \\
& \cdot (v_2, \texttt{server: class('\Irc\Server')}, v_3) \\
& \cdot (v_3, \varepsilon, v_4) \\
& \cdot (v_4, \varepsilon, v_5) \\
& \cdot (v_5, \texttt{buffer: /\^{}privmessage .+/}, v_6) \\
& \cdot (v_6, \varepsilon, v_7) \\
& \cdot (v_7, \varepsilon, v_8) \\
& \cdot (v_8, \texttt{\textbackslash result: 1..}, v_9) \\
& \cdot (v_9, \varepsilon, v_{10})\ \}
\end{aligned}
$$

5.3.3 Critère de Couverture de Prédicat

Similairement au Critère de Couverture de Domaine, le Critère de Couverture de Prédicat s'applique aux prédicats dans les graphes d'expression, et vise à tous les couvrir. Plus formellement, le Critère de Couverture de Prédicat est satisfait par une suite de tests S pour une méthode `f` si toutes les transitions qui ne sont pas de la forme `i: d` dans le graphe de contrat $G(f)$ (qui sont dans les graphes d'expressions étiquetant les transitions de $G(f)$ et contenant le nœud *pred*) apparaissent dans au moins un sentier de $T_f(S)$.

Exemple 5.10 (Suite de tests satisfaisant le Critère de Couverture de Prédicat)**.** La suite de tests $\{t_5, t_6\}$ avec :

- t_5 = `compute(object(\Irc\Server), 'ping')` avec `\pred('$server->bufferState >= 0');`
- t_6 = `compute(object(\Irc\Server), 'ping')` avec `\pred('network_buffer_state() > 0')`.

satisfait le Critère de Couverture de Prédicat. En effet, cette suite de tests produit les chemins suivants (sur les sentiers) :

$$
\begin{aligned}
T_{\texttt{compute}}(\{t_5\}) = \{\ & (v_1, \varepsilon, v_2) \\
& \cdot (v_2, \texttt{server: class('\Irc\Server')}, v_3) \\
& \cdot (v_3, \varepsilon, v_4) \\
& \cdot (v_4, \varepsilon, v_{11}) \\
& \cdot (v_{11}, \texttt{buffer: /\^{}ping\$/}, v_{12}) \\
& \cdot (v_{12}, \varepsilon, v_{13}) \\
& \cdot (v_{13}, \texttt{\textbackslash pred('\$server->bufferState >= 0')}, v_{14}) \\
& \cdot (v_{14}, \varepsilon, v_{15}) \\
& \cdot (v_{15}, \varepsilon, v_{16}) \\
& \cdot (v_{16}, \texttt{\textbackslash result: 1..}, v_{17}) \\
& \cdot (v_{17}, \varepsilon, v_{18})\ \} \\
T_{\texttt{compute}}(\{t_6\}) = \{\ & (v_1, \varepsilon, v_2) \\
& \cdot (v_2, \texttt{server: class('\Irc\Server')}, v_3) \\
& \cdot (v_3, \varepsilon, v_4) \\
& \cdot (v_4, \varepsilon, v_{11}) \\
& \cdot (v_{11}, \texttt{buffer: /\^{}ping\$/}, v_{12}) \\
& \cdot (v_{12}, \varepsilon, v_{13}) \\
& \cdot (v_{13}, \texttt{\textbackslash pred('network\_buffer\_state() > 0')}, v_{14})
\end{aligned}
$$

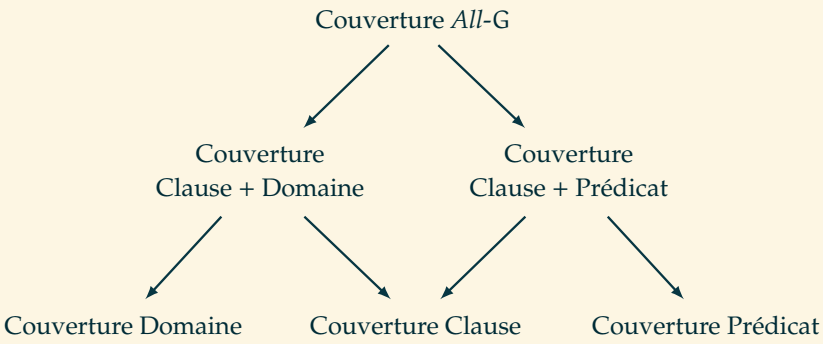


FIGURE 5.10 – Hiérarchie des critères. $A \rightarrow B$ signifie que si le critère A est satisfait, alors B est également satisfait.

$\cdot\ (v_{14}, \varepsilon, v_{15})$
$\cdot\ (v_{15}, \varepsilon, v_{16})$
$\cdot\ (v_{16}, \texttt{\textbackslash result: 1..}, v_{17})$
$\cdot\ (v_{17}, \varepsilon, v_{18})\ \}$

5.3.4 Combinaisons

Avoir de tels critères de couverture n'est pas suffisant pour produire ou qualifier des cas de tests *réalistes*. En effet, couvrir toutes les clauses d'un contrat n'assure pas que tous les domaines réalistes ont été couverts. À l'inverse, tous les domaines n'assure pas que toutes les clauses du contrat aient été exploitées. Ainsi, afin d'avoir des cas de tests plus réalistes, ces critères de couverture de contrat peuvent être combinés ensemble. Nous proposons les combinaisons suivantes :

– le Critère de Couverture Clause + Domaine vise à couvrir la structure et les domaines réalistes d'un contrat ;
– le Critère de Couverture Clause + Prédicat vise à couvrir la structure et les prédicats d'un contrat ;
– le Critère de Couverture *All*-G vise à satisfaire les critères Clause, Domaine et Prédicat en même temps.

Ce **treillis** de critères de couverture de contrats est résumé dans la figure 5.10.

Exemple 5.11 (Suite de tests satisfaisant les combinaisons des Critères de Couverture de Contrat)**.** La suite de tests $\{t_1, t_2, t_3, t_4\}$ satisfait le Critère de Couverture Clause + Domaine. La suite de tests $\{t_1, t_3, t_5, t_6\}$ satisfait le Critère de Couverture Clause + Prédicat. La suite de tests $\{t_1, t_3, t_4, t_5, t_6\}$ satisfait le Critère de Couver-

ture *All*-G.

5.4 Synthèse

Dans ce chapitre, nous avons présenté la transformation d'un contrat Praspel en un graphe de contrat. Nous avons ensuite exploité ces graphes pour définir plusieurs critères de couverture sur les contrats. Nous avons vu que ces critères peuvent également être combinés entre eux.
Ce chapitre clôt nos contributions : un nouveau langage de contrat s'appuyant sur les domaines réalistes, des générations de données pour plusieurs types et enfin des critères de couverture sur les contrats. La « boucle est bouclée » : à partir des critères de couverture, nous pouvons générer et exécuter des suites de tests unitaires automatiquement. Le prochain chapitre s'intéresse à l'implémentation et à l'outillage de nos contributions.

Chapitre 6.
Implémentations et outillage

Table des matières

Dans les chapitres précédents nous avons présenté le langage Praspel et plusieurs algorithmes de générations de données ainsi que des critères de couverture. Ce chapitre est consacré à leur implémentation. Il est organisé en trois parties. Tout d'abord, dans la partie 6.1, nous présentons Hoa, un ensemble

de bibliothèques PHP. Nos contributions sont implémentées sous la forme de bibliothèques au sein de Hoa, qui est un projet *open-source*. Ensuite, dans la partie 6.2, nous présentons notre implémentation de Praspel, son fonctionnement et son découpage. Enfin, dans la partie 6.3, nous présentons l'intégration de ces outils dans atoum, un *framework* de tests unitaires, ainsi qu'une extension qui fait le pont entre Praspel et atoum.

6.1 Implémentation dans Hoa

Hoa [Hoa Project Foundation, 2007] est un ensemble de bibliothèques PHP. De plus, Hoa est un pont entre le monde de la recherche et de l'industrie. Créé par Ivan Enderlin, l'auteur de ce mémoire, ce projet est maintenant porté par une communauté et une association. Il y a plusieurs avantages à utiliser ce projet.

Réutilisation des ressources. Hoa comporte déjà des bibliothèques pour une multitude de domaines. Le projet offre un cadre clair pour développer de nouvelles bibliothèques, ce qui évite des efforts de développement, de distribution et de compatibilité.

Intégration dans une communauté. Hoa est impliqué dans plusieurs consortiums ou groupes de décision concernant plusieurs standards de l'Informatique ou du Web. Par conséquent, les bibliothèques et outils que nous allons utiliser pour construire Praspel nous permettront de mieux le maintenir et d'en assurer une meilleure qualité.

Connaissance des besoins. Grâce à ce projet, il est possible d'avoir une meilleure connaissance des besoins des utilisateurs finaux, à savoir les ingénieurs de test. Et cette thèse tend à y répondre. Hoa et cette thèse sont donc complémentaires, puisque Hoa permettra de valoriser ces travaux de recherche et réciproquement.

Aide et pérennité. Hoa nous offre l'opportunité de faire valider nos expérimentations par sa communauté ou ses utilisateurs. Mais aussi, Hoa est aussi développé sous la licence libre *New BSD License* [1], qui est une licence *open-source*. Cela implique que les outils que nous avons développés durant cette thèse, comme Praspel, le compilateur etc., sont gratuits et libres. Cette approche offre plusieurs avantages. Un majeur est que des contributeurs de tous horizons peuvent nous aider à corriger ou améliorer nos outils. Ce fut le cas à plusieurs reprises, où des contributeurs ont corrigé des erreurs dans Praspel ou ont amélioré le compilateur, notamment en y

1. Voir http://hoa-project.net/About.html#License.

ajoutant le support Unicode, en améliorant les performances de l'analyseur lexical (pour l'analyse de très grandes données) etc.

Lien avec l'industrie. Enfin, puisque le projet est gratuit, l'industrie peut l'utiliser et nous offrir des retours pertinents, comme nous le verrons dans le chapitre 7 avec les expérimentations.

Nos contributions prennent la forme de 4 bibliothèques, présentées dans les parties suivantes : `Hoa\Realdom` pour les domaines réalistes dans la partie 6.1.1, `Hoa\Praspel` pour le langage Praspel dans la partie 6.1.2, `Hoa\Compiler` pour le compilateur et `Hoa\Regex` pour les expressions régulières dans la partie 6.1.3. D'autres contributions plus minimes ne sont pas détaillées ici, comme l'ajout de fonctionnalités dans la bibliothèque `Hoa\Math`.

6.1.1 `Hoa\Realdom`

La bibliothèque `Hoa\Realdom`[2] est la bibliothèque standard des domaines réalistes. Ils sont implémentés comme présenté dans la partie 3.1 page 27, à savoir que chaque domaine réaliste est représenté par une classe. Actuellement, une liste de 28 domaines réalistes est proposée, répartis dans les différentes couches de l'univers $\mathcal{U}_{\text{realdom}}$ des domaines réalistes (voir la partie 3.1.2 page 29).

couche 0 – `Undefined`, représentant des valeurs non-définies ;

couche 1 – `Array`, représentant des tableaux ;
– `Boolean`, représentant les booléens ;
– `Class`, représentant des instances de classes ;
– `Float`, représentant les réels ;
– `Integer`, représentant les entiers ;
– `String`, représentant des chaînes de caractères ;

couche 2 – `Bag`, un sac pouvant contenir plusieurs domaines réalistes de natures différentes ; plus formellement c'est un multiensemble ;
– `Boundfloat`, représentant un intervalle de réels ;
– `Boundinteger`, représentant un intervalle d'entiers ;
– `Color`, représentant des couleurs au format *`#rrggbb`* ;
– `Constarray`, représentant les tableaux de Praspel, c'est à dire une description de tableau ;
– `Constboolean`, représentant les booléens de Praspel ;
– `Constfloat`, représentant les réels de Praspel ;
– `Constinteger`, représentant les entiers de Praspel ;

2. Voir http://central.hoa-project.net/Resource/Library/Realdom.

- `Constnull`, représentant la valeur nulle de Praspel ;
- `Conststring`, représentant les chaînes de caractères de Praspel ;
- `Date`, représentant une date formatée ;
- `Empty`, représentant une donnée vide ;
- `Even`, représentant les entiers pairs ;
- `Grammar`, représentant les chaînes de caractères spécifiées par une grammaire ;
- `Natural`, représentant les naturels (entiers auto-incrémentés) ;
- `Number`, représentant les nombres (entiers ou réels) ;
- `Object`, représentant un objet donné ;
- `Odd`, représentant les entiers impairs ;
- `Regex`, représentant les chaînes de caractères spécifiées par une expression régulière ;
- `Smallfloat`, représentant des petits réels entre -128.0 et 127.0 ;
- `Smallinteger`, représentant des petits entiers entre -128 et 127 ;
- `Timestamp`, représentant des valeurs dans le temps, en secondes depuis l'*Unix Epoch* ;

couche 3 dépendante du projet.

Les domaines réalistes `Constarray`, `Constboolean`, `Constfloat`, `Constinteger`, `Constnull` et `Conststring` représentent les types natifs du langage Praspel. Écrire `12` est automatiquement transformé en `constinteger(12)`. Nous parlons d'*auto-boxing* : toutes les données manipulées sont des domaines réalistes. Cette approche permet d'uniformiser les traitements au sein du langage.
Pour obtenir toutes les informations sur un domaine réaliste, nous pouvons nous aider de l'outil en ligne de commande `hoa realdom:reflection`.

Exemple 6.1 (Informations sur le domaine réaliste `Boundinteger`)**.** La figure 6.1 montre le résultat de l'introspection du domaine réaliste `Boundinteger`. Nous trouvons comme informations : en (1) le nom de la classe qui représente l'implémentation du domaine réaliste et son parent, en (2) toutes les interfaces utilisées par cette implémentation (ici plusieurs de PHP et des domaines réalistes, comme présenté dans la partie 3.1.4 page 31), et en (3) les paramètres avec leurs positions, filtres et valeurs par défaut.

6.1.2 `Hoa\Praspel`

La bibliothèque `Hoa\Praspel` [3] est responsable de tout le support de Praspel. Le fonctionnement de Praspel est détaillé dans la partie 6.2. La bibliothèque est découpée de la façon suivante :

3. Voir http://central.hoa-project.net/Resource/Library/Praspel.

```
$ hoa realdom:reflection boundinteger
Realdom boundinteger {

    Implementation Hoa\Realdom\Boundinteger;

    Parent Hoa\Realdom\Integer;

    Interfaces {

        ArrayAccess;
        Countable;
        IteratorAggregate;
        Traversable;
        Hoa\Realdom\IRealdom\Enumerable;
        Hoa\Realdom\IRealdom\Finite;
        Hoa\Realdom\IRealdom\Interval;
        Hoa\Realdom\IRealdom\Nonconvex;
        Hoa\Realdom\Number;
        Hoa\Visitor\Element;
    }

    Parameters {

        [#0 optional] Constinteger lower = -9223372036854775808;
        [#1 optional] Constinteger upper = 9223372036854775807;
    }
}
```

1 2 3

FIGURE 6.1 – Introspection des domaines réalistes.

- `AssertionChecker` permet l'évaluation du modèle objet ;
- `Bin` contient des scripts, dont un *shell* permettant d'analyser et évaluer du code Praspel à la volée ;
- `Exception` contient toutes les catégories d'exceptions de Praspel ;
- `Iterator` contient entre autre les générateurs de tests unitaires à partir de plusieurs critères de couverture sur un contrat ;
- `Model` contient le modèle objet, à savoir la pièce centrale du langage Praspel ;
- `Preambler` permet de créer un préambule de test pour n'importe quel système sous test ;
- `Visitor` permet de passer d'une forme à une autre du langage (comme nous le verrons dans les parties suivantes).

À la racine de la bibliothèque, nous trouvons entre autre la grammaire au format PP (que nous retrouvons dans l'annexe 1), une classe nommée `Hoa\Praspel` qui rassemble les opérations usuelles sur le langage et une classe représentant une trace d'évaluation d'un contrat.

6.1.3 `Hoa\Compiler` et `Hoa\Regex`

Le compilateur LL(⋆) que nous avons décrit dans la partie 4.3 page 61 est implémenté par la bibliothèque `Hoa\Compiler\Llk`[4] (un compilateur LL(1) existait préalablement). Cette bibliothèque comprend aussi le support du langage PP ainsi que des algorithmes de génération à partir de grammaires : aléatoire uniforme, exhaustif borné et par couverture. Ces algorithmes sont présents dans la sous-bibliothèque `Hoa\Compiler\Llk\Sampler`.

Exemple 6.2 (Génération exhaustive bornée avec Hoa)**.** Pour générer toutes les données au format JSON, dont la taille maximum des séquences de lexèmes ne dépasse pas 10, nous écrirons :

4. Voir http://central.hoa-project.net/Resource/Library/Compiler.

```
$grammar = new Hoa\File\Read('hoa://Library/Json/Grammar.pp');
$parser  = Hoa\Compiler\Llk::load($grammar);
$token   = new Hoa\Regex\Visitor\Isotropic(
    new Hoa\Math\Sampler\Random()
);
$length  = 10;
$sampler = new Hoa\Compiler\Llk\Sampler\BoundedExhaustive(
    $parser, /* (1) */
    $token,  /* (2) */
    $length  /* (3) */
);

foreach($sampler as $i => $data)
    // compute $data
```

La variable `$grammar` contient un flux en lecture de la grammaire JSON. La variable `$parser` contient l'analyseur lexical et syntaxique représentés par la grammaire. La variable `$token` contient le générateur de valeurs pour les lexèmes. Le seul proposé actuellement est isotropique, comme présenté dans la partie 4.3.3 page 75. Enfin, la variable `$length` représente la taille maximum de la séquence. Les trois dernières variables sont distribuées sur le générateur exhaustif borné : les analyseurs en (1), le générateur de valeurs pour les lexèmes en (2) et la taille maximum de la séquence en (3). Ce générateur fonctionne comme un itérateur : chaque donnée est calculée à la demande. En PHP, le meilleur moyen d'exploiter un itérateur est d'utiliser la boucle `foreach`, qui permet de parcourir une structure *traversable*, dont les itérateurs. La variable `$i` contient le numéro de la donnée et `$data` contient toutes les données les unes après les autres.

La grammaire des expressions régulières se trouve également dans la bibliothèque `Hoa\Regex`[5] dans `hoa://Library/Regex/Grammar.pp` (voir l'annexe 3 page 164).

6.2 Praspel

Cette partie présente l'implémentation et l'outillage de Praspel. Tout d'abord, la partie 6.2.1 explique comment le langage est analysé. Ensuite, la partie 6.2.2 présente l'architecture de Praspel et son modèle objet, la pièce centrale. Les parties 6.2.3 et 6.2.4 présentent respectivement comment exporter ou importer le modèle et comment le désassembler. Enfin, la partie 6.2.5 présente comment évaluer le modèle.

5. Voir http://central.hoa-project.net/Resource/Library/Regex.

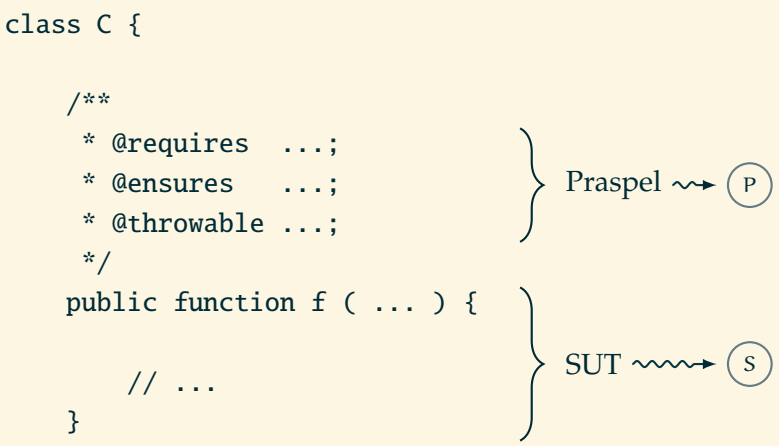


FIGURE 6.2 – Répartitions des composants d'un programme.

La figure 6.2 nous rappelle que, dans une classe, les commentaires qui annotent les méthodes contiennent des contrats écrits en Praspel. Les contrats seront extraits de ces commentaires pour être envoyés dans les outils de Praspel. Les méthodes vont constituer le SUT et seront utilisées lors de l'exécution des tests. Dans les prochaines figures, ces données sont identifiées par une lettre cerclée afin d'indiquer leur flux dans les différents outils. Ainsi, la notation Praspel ⇝ (P) signifie que le composant Praspel sera représenté par (P) dans les figures suivantes. De même, le *System Under Test* sera représenté par (S).
Praspel existe sous différentes formes. Nous les expliquons en nous appuyant sur la figure 6.3 qui montre l'agencement de ces différentes formes et le passage de l'une vers l'autre :

- le langage est la forme textuelle de Praspel, provenant des commentaires des méthodes. Il est défini par la grammaire donnée dans la partie 3.2 page 32. En utilisant des analyseurs, le langage est transformé en modèle objet ;
- le modèle objet (M) est la pièce centrale de Praspel : c'est un ensemble d'objets en mémoire réprésentant chaque partie du langage (clause, variable, opérateur etc.). Tous ces objets sont imbriqués et forment une structure qui s'apparente à un arbre ;
- le modèle objet peut être exporté sous forme de code PHP qui, quand il est exécuté, reconstruit en mémoire le modèle objet depuis lequel il a été généré ; cette exécution est appelée une importation ;
- le désassemblage permet de transformer le modèle objet en forme textuelle.

Les parties suivantes détaillent ces différentes étapes.

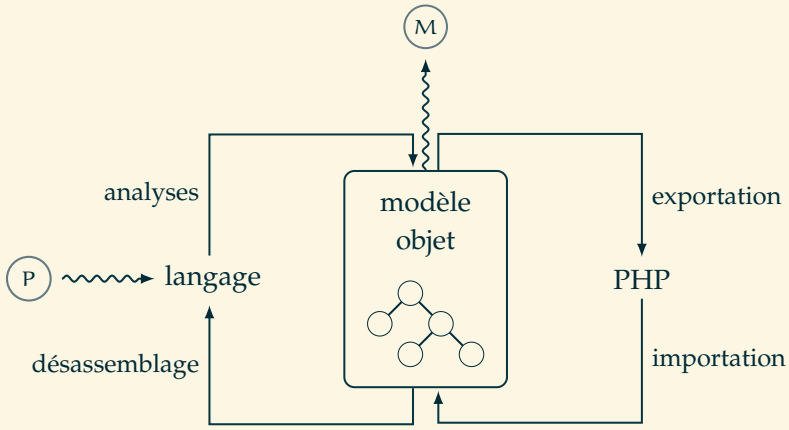


FIGURE 6.3 – Fonctionnement schématique de Praspel.

6.2.1 Analyses

À l'aide de la grammaire de Praspel, définie dans la partie 3.2 page 32, exprimée avec le langage PP (voir l'annexe 1) et un compilateur, défini dans les parties 4.3.1 et 4.3.2 pages 61 et 64, l'*Abstract Syntax Tree* du contrat analysé est produit. Cet AST sera ensuite visité afin de construire le modèle objet. Pour y arriver, nous avons deux façons de faire.
La première détaille toutes les étapes, à savoir l'utilisation du compilateur pour produire un AST, puis l'appel d'un visiteur pour construire le modèle objet.

Exemple 6.3 (Obtenir le modèle objet depuis un contrat)**.** Nous considérons le contrat (P) suivant : `@requires i: 7..42`. Pour obtenir le modèle objet de ce contrat, nous écrirons :

```
$parser      = Hoa\Compiler\Llk::load(
    new Hoa\File\Read('hoa://Library/Praspel/Grammar.pp')
);
$ast         = $parser->parse('@requires i: 7..42;');
$interpreter = new Hoa\Praspel\Visitor\Interpreter();
$model       = $interpreter->visit($ast);
```

(M) (P)

La variable `$parser` contient les analyseurs lexicaux et syntaxiques chargés depuis la grammaire. La variable `$ast` contient l'AST produit par les analyses d'une chaîne de caractères représentant du code Praspel. Ces analyses sont exécutées en appelant la méthode `parse` sur le compilateur. Ensuite, nous instancions un visiteur dans la

variable `$interpreter` qui va visiter l'AST, grâce à la méthode `visit`, pour le transformer en modèle objet.

La seconde façon de faire consiste à utiliser un raccourci qui est la méthode (statique) `interprete` ; ainsi :

```
$model = Hoa\Praspel::interprete('@requires i: 7..42;');
```

produira le même résultat.
Durant la production de l'AST, les erreurs **lexicales** et **syntaxiques** sont détectées.
Durant la production du modèle objet, les erreurs **sémantiques** sont détectées.

6.2.2 Modèle objet

Le modèle objet est la pièce centrale de Praspel. Chaque construction du langage est représentée par une classe. Le diagramme de classes du modèle objet est présenté dans la figure 6.4. Toutes les clauses sont représentées par la classe abstraite `Clause`. La classe `Behavior` représente la clause `@behavior`. Elle peut contenir une clause `@requires` représentée par la classe `Requires`, une clause `@ensures` représentée par la classe `Ensures`, une clause `@throwable` représentée par la classe `Throwable`, une clause `@description` représentée par la classe `Description` et une clause `@default` représentée par la classe `DefaultBehavior`. Une clause `@behavior` peut contenir d'autres clauses `@behavior` *via* la classe `Collection` qui représente une collection de clauses. Une spécification, représentée par la classe `Specification`, est un comportement qui peut contenir des clauses supplémentaires : `@invariant` représentée par la classe `Invariant`.
Certaines clauses contiennent des expressions (voir la règle syntaxique *expression* dans la grammaire de Praspel présentée dans la figure 3.7 page 38). Ces expressions sont représentées par la classe `Declaration`. Une expression est composée de variables, représentées par la classe `Variable`. Une variable est, du point de vue des domaines réalistes, un *holder*, c'est à dire une structure qui porte des domaines réalistes.
Il existe trois sortes de variables : normale, implicite et empruntée. Une variable normale est une variable classique du contrat, représentée par la classe `Variable`. Une variable implicite, représentée par la classe `Implicit`, est une variable non déclarée, comme `this` dans le cas des méthodes. En effet, en PHP, la variable contenant l'objet est toujours implicite, c'est à dire non déclarée. Ce comportement se retrouve dans Praspel où il n'est pas nécessaire de spécifier `this`. Enfin, une variable empruntée est une variable extérieure à la méthode, comme `this->foo` qui représente un attribut de classe, ou une variable extérieure à l'état courant du système, comme `\old(f)` qui fait référence à la variable `foo` dans le pré-état, lorsque que le système est dans son post-état.

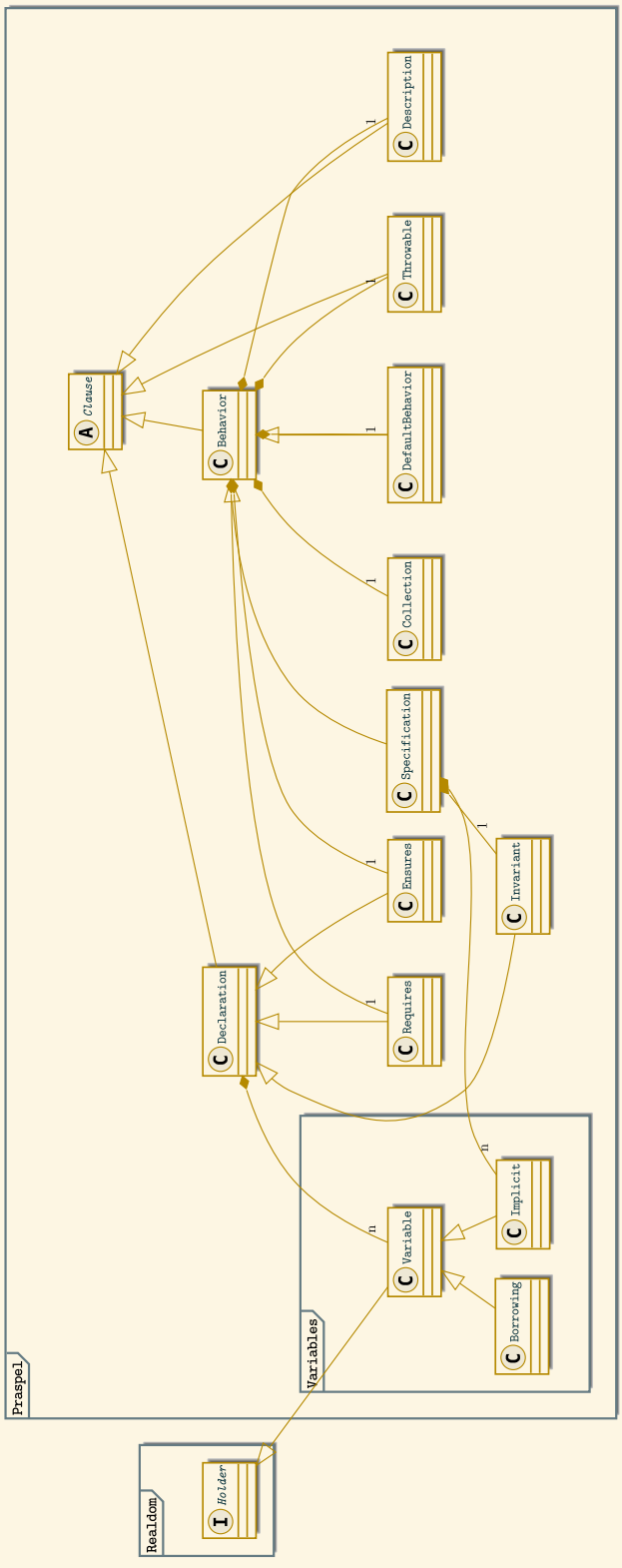


FIGURE 6.4 – Modèle objet de Praspel.

Les détails du modèle objet ne sont pas présents dans la figure 6.4 pour des raisons de simplicité. Le lien entre la bibliothèque `Hoa\Praspel` et `Hoa\Realdom` se fait par les variables qui sont des *holders*. Autant les domaines réalistes peuvent être utilisés sans Praspel, autant l'inverse n'est pas envisageable.
Tous les objets de ce modèle sont imbriqués pour former une structure qui s'apparente à un arbre. La racine de cet arbre est représentée par une instance de la classe `Specification`. Cet arbre est visitable, c'est à dire que nous pouvons lui appliquer des visiteurs, comme nous allons le voir dans les parties suivantes.

6.2.3 Exportation et importation

Le modèle objet peut être exporté de la mémoire vers un fichier sous la forme de code PHP. Quand ce code PHP est exécuté, nous parlons d'importation, qui construit en mémoire le modèle objet depuis lequel il a été généré. Cela permet de « sauvegarder » le modèle en évitant la phase des analyses qui peuvent être coûteuses dans certaines situations.

Exemple 6.4 (Exporter et importer un modèle objet)**.** Ainsi, à partir d'un modèle (M) contenu dans la variable `$model` et d'un visiteur, nous allons produire du code PHP que nous allons afficher :

```
$compiler = new Hoa\Praspel\Visitor\Compiler();
echo $compiler->visit($model);
```

(M)

Cet exemple produira le code PHP suivant :

```
$praspel = new \Hoa\Praspel\Model\Specification();

$requires = $praspel->getClause('requires');
$requires['i']->in = realdom()->boundinteger(7, 42);
```

qui permet de construire le modèle objet contenu dans `$model`, c'est à dire que la valeur de la variable `$model` est strictement égale à la valeur de la variable `$praspel`. Cet exemple nous permet également d'avoir un aperçu de l'API du modèle objet : elle se veut simple et la plus compréhensible possible. Une spécification est instanciée. Sur cette spécification, la clause `@requires` est obtenue pour y déclarer une variable `i` dont les valeurs sont décrites par (ou présentes dans, traduit par `->in`) la disjonction de domaines réalistes, créée à l'aide de la fonction `realdom`.

6.2.4 Désassemblage

Le désassemblage permet de transformer le modèle objet en mémoire sous sa forme textuelle. C'est l'opération inverse des analyses de la partie 6.2.1. Il permet par

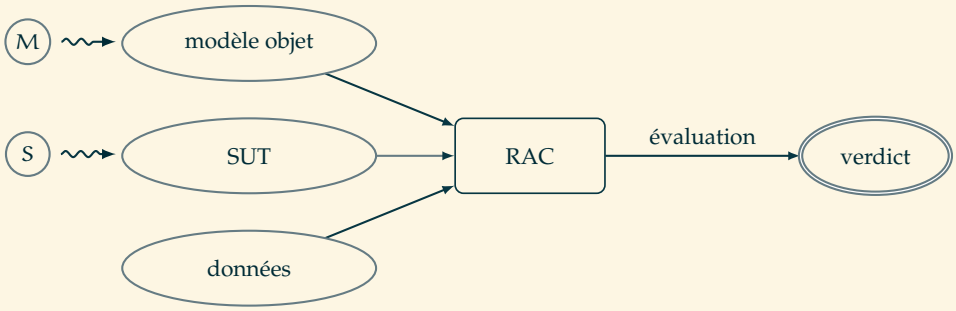


FIGURE 6.5 – Fonctionnement schématique du *Runtime Assertion Checker*.

exemple d'indiquer des erreurs avec précision sous la forme originale de Praspel. C'est tout à fait comparable à un *pretty-printer*.

Exemple 6.5 (Désassembler un modèle objet)**.** En réutilisant la variable `$model` des exemples précédents :

```
$disassembler = new Hoa\Praspel\Visitor\Praspel();
echo $disassembler->visit($model);
```

(M)

Nous instancions un visiteur dans la variable `$disassembler` pour visiter le modèle objet. Cet exemple produira le résultat suivant :

```
@requires i: boundinteger(7, 42);
```

Notons que le sucre syntaxique `7..42` a disparu.

Ce dernier processus, complété des précédents, nous permet de passer d'une représentation vers n'importe quelle autre représentation de Praspel : soit textuelle, soit objet, soit PHP.

6.2.5 *Runtime Assertion Checker*

Les *assertion checkers* sont responsables de l'évaluation de Praspel. Le fonctionnement d'un *assertion checker* est illustré dans la figure 6.5. Il travaille avec 3 composants : un modèle objet (M) qui représente un contrat (provenant par exemple des commentaires des méthodes), un SUT (S) (par exemple la méthode commentée) et des données. Un *assertion checker* est capable de valider le contrat.
Actuellement, nous nous appuyons sur un *Runtime Assertion Checker*, abrégé RAC, pour valider le contrat à l'exécution et ainsi calculer le verdict du test. Quand la validation du contrat échoue, une erreur spécifique est produite. Les erreurs du RAC (aussi appelées *Praspel failures* ou erreurs Praspel) peuvent être de 5 sortes :

1. *precondition failure*, quand une clause `@requires` n'est pas satisfaite lors de l'invocation de la méthode ou qu'elle est manquante et que la méthode a besoin de paramètres ;
2. *postcondition failure*, quand une postcondition n'est pas satisfaite après l'exécution de la méthode ;
3. *throwable failure*, quand l'exécution de la méthode lève une exception inattendue ou qu'une exception est levée et que la clause `@throwable` est manquante ;
4. *invariant failure*, quand un invariant de classe est violé ; et
5. *internal precondition failure*, qui correspond à la propagation d'une *precondition failure* à un niveau supérieur.

Ces erreurs sont inspirées de JML [Leavens et al., 1999] et de son RAC [Cheon and Leavens, 2002b]. Le test réussit si aucune erreur Praspel n'est détectée. Autrement, il échoue, et l'erreur est consignée avec des informations supplémentaires, comme le numéro de ligne du contrat, du fichier, l'assertion qui a échoué etc.

Exemple 6.6 (Évaluer un modèle objet)**.** À présent, illustrons comment utiliser le RAC de Praspel avec un modèle objet (M), lorsque le SUT (S) est la méthode `f` de la classe `C` de la figure 6.2 avec un seul paramètre `$i` :

```
        $rac = new Hoa\Praspel\AssertionChecker\Runtime(
(M) ~~~~> $model,
            xcallable(new C(), 'f') <~~~~ (S)
        );
        $rac->setData(['i' => 13]);
        $verdict = $rac->evaluate();
```

La variable `$rac` contient une instance d'un *Runtime Assertion Checker* qui travaille avec un modèle objet dans la variable `$model` passée en premier argument et un SUT passé en second argument. Le SUT peut être n'importe quelle structure « appelable ». Nous parlons d'un *callable*. La fonction `xcallable` permet de créer un *callable* facilement : ici, la méthode `f` sur une instance de la classe `C`. Ensuite, grâce à la méthode `setData`, nous fixons les données des arguments du SUT, à savoir que son paramètre `$i` aura comme valeur l'entier `13`. Enfin, le SUT est exécuté et le contrat vérifié grâce à l'appel de la méthode `evaluate` sur le RAC. Dans ce cas, la variable `$verdict` contiendra `true` car la précondition (`@requires i: 7..42`) est bien respectée.
Si nous voulons que les données soient générées automatiquement, nous devrons appeler la méthode `automaticallyGenerateData` en plus de définir le générateur numérique des domaines réalistes. Ainsi :

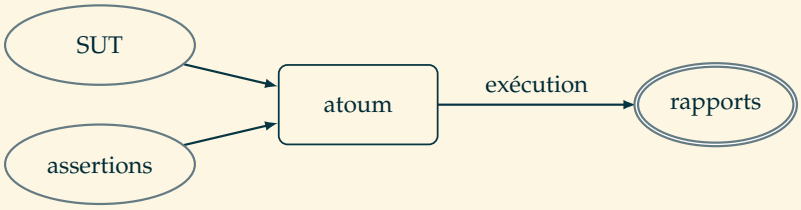


FIGURE 6.6 – Fonctionnement schématique d'atoum.

```
Hoa\Realdom::setDefaultSampler(new Hoa\Math\Sampler\Random());
$rac->automaticallyGenerateData(true);
```

avec la même définition originale de la variable `$rac` que précédemment.

6.3 Intégration dans atoum

atoum [Hardy, 2010], sans majuscule, est aujourd'hui l'un des *frameworks* de tests unitaires pour PHP les plus utilisés dans l'industrie. Gratuit et *open-source*, il compte une communauté active et plusieurs utilisateurs et partenaires industriels importants. atoum est également inclus dans plusieurs IDE et outils majeurs, comme Netbeans [6], Eclipse [7], Vim [8], Travis-CI [9], Jenkins [10], Coveralls [11] etc.

Les suites de tests générées par Praspel doivent être exécutables de plusieurs façons (séquentielles, isolées ou autre), des rapports doivent être produits, les outils doivent être intégrés à des IDE etc. Tout ceci représente un travail important. atoum comporte déjà toutes ces fonctionnalités. Nous avons donc choisi de créer une extension pour introduire Praspel dans atoum. De plus, atoum ne possède aucun générateur de données, alors que Praspel en propose plusieurs. Il a donc été encore plus naturel de rapprocher les deux projets. Cette extension, appelée `atoum/praspel-extension`, est incluse dans la distribution standard d'atoum « sous stéroïde », c'est à dire avec des outils qui ajoutent des fonctionnalités à atoum.

Dans un premier temps, nous allons présenter atoum, puis dans un second temps, nous montrerons les fonctionnalités offertes par l'extension.

6.3.1 Présentation d'atoum

La figure 6.6 illustre le fonctionnement schématique d'atoum et de tous les *framework* de tests unitaires : ils permettent d'exécuter un ensemble d'assertions (pour les vérifier) sur un SUT et sont capables de produire différents rapports. Pour utiliser atoum, il faudra alors basiquement procéder en deux étapes : écrire des assertions et les exécuter. Dans le vocabulaire d'atoum, un test est une méthode qui contient plusieurs assertions sur le SUT et une suite de tests est une classe.

Exemple 6.7 (Premier test avec atoum)**.** Ainsi, nous allons tester que le résultat retourné par une certaine méthode `say`, en (1), d'une classe `Project\HelloWorld` retourne bien la chaîne de caractères `Hello World!`, en (2) :

```
class HelloWorld extends \atoum\test {

    public function testSay ( ) {

        $helloWorld = new \Project\HelloWorld();
        $this->string($helloWorld->say())      /* (1) */
                  ->isEqualTo('Hello World!'); /* (2) */
    }
}
```

Puis nous pouvons exécuter le test en ligne de commande :

```
$ atoum --files Path/To/Our/Test/HelloWorld.php
...
> Total test duration: 0.00 second.
> Total test memory usage: 0.25 Mb.
> Code coverage value: 100.00%
> Running duration: 0.08 second.
> Success (1 test, 1/1 method, 0 void method,
           0 skipped method, 2 assertions)!
```

Le rapport simple nous informe que le test s'est exécuté en moins de 0.005 seconde, en utilisant 0.25 Mb de mémoire, avec une couverture de code (avec le critère tous-les-arcs) de 100%, que l'ensemble s'est exécuté en 0.08 seconde et que l'exécution

6. Voir https://netbeans.org/.
7. Voir http://eclipse.org/.
8. Voir http://vim.org/.
9. Voir http://travis-ci.com/.
10. Voir http://jenkins-ci.org/.
11. Voir https://coveralls.io/.

de la seule suite de tests, ne contenant qu'un seul test représentant deux assertions (`string` et `isEqualTo`), a été un succès.

Au delà de l'intégration avec plusieurs outils industiels ou la production de rapports détaillés, nous pouvons citer trois points forts de cet outil.

Moteurs d'exécution

Tout d'abord, atoum permet nativement (comprendre sans *plugin* supplémentaire) l'exécution des tests de manière séquentielle, en isolation ou en parallèle. Une exécution séquentielle va exécuter tous les tests les uns après les autres de manière séquentielle dans le même processus. Une exécution en isolation va exécuter tous les tests les uns après les autres mais dans un processus nouvellement créé à chaque fois afin d'isoler le test des exécutions précédentes. Une exécution en parallèle va exécuter tous les tests en isolation mais de manière concurrente. Cela permet d'accélérer les exécutions ou d'assurer une fiabilité dans les résultats des tests.

Collection d'asserteurs

Le second point fort d'atoum est sa collection importante d'asserteurs. Bien plus que de simples `assertTrue` ou `assertFalse` que nous retrouvons dans la majorité des autres *frameworks* de tests unitaires, atoum propose des asserteurs personnalisés pour chaque type de données.

Exemple 6.8 (Asserteurs spécifiques avec atoum)**.** Par exemple, une simple assertion `1 - 0.97 === 0.03` sera fausse dans tous les langages malgré le fait que ce résultat nous paraisse naturel. C'est pourquoi atoum propose l'asserteur `isNearlyEqualTo` sur le groupe d'asserteurs `float` :

```
$this->float(1 - 0.97)->isNearlyEqualTo(0.03);
```

De même, nous pouvons trouver l'asserteur `containsValues` sur le groupe `array` qui va vérifier que certaines valeurs sont bien présentes dans un tableau :

```
$this->array([1, 1, 2, 3, 5, 8])->containsValues([1, 3]);
```

Il existe encore plusieurs autres groupes d'asserteurs, notamment `output` qui permet de tester les sorties du programme. Notons que ce groupe hérite du groupe `string` : l'héritage entre groupes est une autre fonctionnalité intéressante.

Mock

Enfin, le dernier point fort d'atoum est son système de *mock* (bouchon). Pour obtenir le *mock* de n'importe quel objet, il suffit de l'instancier sur l'espace de nom `Mock`.

Exemple 6.9 (Bouchonner avec atoum)**.** Si nous voulons obtenir le *mock* de la classe `Project\HelloWorld`, nous instancierons simplement la classe `Mock\Project\HelloWorld`, où nous pourrons ensuite contrôler le résultat de certaines méthodes :

```
class HelloWorld extends \atoum\test {

    public function testSay ( ) {

        $helloWorld = new \Mock\Project\HelloWorld();
        $this->calling($helloWorld)->say = 'Hello!';
        $this->string($helloWorld->say())
                  ->isEqualTo('Hello!');
    }
}
```

Nous avons créé un *mock* puis nous avons spécifié que la méthode `say` retournera toujours `Hello!`.

Nous pouvons également forcer une méthode à retourner une exception à partir de son troisième appel par exemple. Il est aussi possible de bouchonner des fonctions et objets standards de PHP. En somme, les fonctionnalités sont avancées et sont accessibles facilement.

6.3.2 Extension : Praspel dans atoum

Cette extension [12] introduit Praspel dans atoum. Elle propose deux fonctionnalités :
- la génération automatique de données de test, qui n'est pas présente dans atoum ;
- la compilation de suites de tests générées avec Praspel en suites de tests écrites avec l'API d'atoum ; les tests seront donc exécutables avec atoum.

Asserteurs de génération de données de test

Dans un premier temps, nous donnons accès aux domaines réalistes dans atoum à travers l'asserteur `realdom` qui permet de créer des disjonctions de domaines réalistes. La syntaxe utilisée dans PHP est la plus proche possible de la syntaxe utilisée dans Praspel.

Exemple 6.10 (Déclaration de domaines réalistes dans atoum)**.** Pour déclarer l'intervalle d'entiers $[7; 13] \cup [42; 153]$, nous écrirons :

12. Voir http://central.hoa-project.net/Resource/Contributions/Atoum/PraspelExtension.

```
$data = $this->realdom->boundinteger( 7,  13)
                      ->or->boundinteger(42, 153);
```

L'équivalent en Praspel serait `boundinteger(7, 13) or boundinteger(42, 153)` ou plus simplement `7..13 or 42..153`.

Ensuite, nous pouvons utiliser au choix l'asserteur `sample` ou `sampleMany` pour générer une ou plusieurs données.

Exemple 6.11 (Génération d'entiers avec atoum)**.** Pour générer respectivement 1 entier ou 7 entiers dans l'intervalle de l'exemple précédent, nous devons écrire :

```
$integer  = $this->sample($data);
$integers = $this->sampleMany($data, 7);
```

Exemple 6.12 (Génération de données plus fines avec atoum)**.** Nous proposons encore deux exemples pour générer plusieurs dates ou chaînes de caractères spécifiées par une expression régulière. Tout d'abord, nous allons générer des dates au format `d/m H:i` (par exemple `28/03 12:34`) entre hier et lundi prochain, qui sont des repères temporels relatifs :

```
$data = $this->realdom->date(
    'd/m H:i',
    $this->realdom->boundinteger(
        $this->realdom->timestamp('yesterday'),
        $this->realdom->timestamp('next Monday')
    )
);

foreach($this->sampleMany($data, 7) as $date)
    // compute $date
```

En Praspel, la déclaration du domaine réaliste s'écrirait de la façon suivante :

```
date(
    'd/m H:i',
    boundinteger(
        timestamp('yesterday'),
        timestamp('next Monday')
    )
)
```

Ensuite, nous allons générer des chaînes de caractères qui contiennent le mot-clé `foo` suivi des mots-clés `bar` ou `baz` :

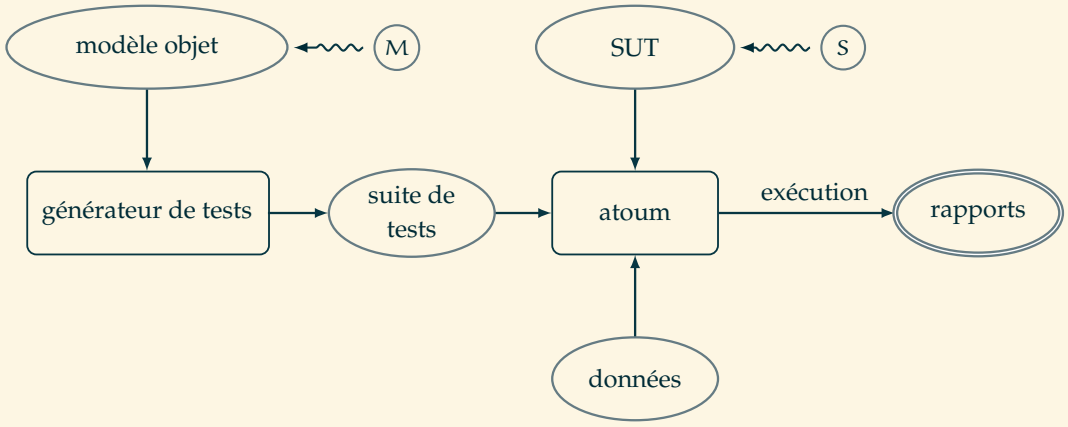


FIGURE 6.7 – Fonctionnement de la génération de tests unitaires avec l'extension.

```
$data = $this->realdom->regex('/.+foo.+ba(r|z).+/');

foreach($this->sampleMany($data, 7) as $string)
    // compute $string
```

Génération de suites de tests unitaires

Dans la partie 5.3 page 95, nous avons défini des critères de couverture sur les contrats. Les critères sont satisfaits par un ensemble de chemins dans les contrats. Un chemin est représenté par un test. Praspel est capable de générer des tests correspondant à ces chemins à partir d'un modèle objet d'un contrat. C'est le point de départ du processus illustré dans la figure 6.7. Nous y retrouvons (M) et (S). À partir du modèle objet, une suite de tests est générée. Ces tests sont compilés vers des tests écrits avec l'API d'atoum (donc exécutables). Cette étape est regroupée dans le nœud suite de tests dans la figure. atoum va ensuite sélectionner chaque test dans cette suite et les exécuter en utilisant le SUT associé et des données. Le nœud atoum dans la figure applique le même processus que dans la figure 6.5.
La compilation de la suite de tests en tests concrets s'effectue en ligne de commande : plusieurs classes sont analysées pour en extraire des contrats Praspel, qui sont à leur tour analysés pour pouvoir appliquer les algorithmes de génération de tests unitaires sur leur modèle objet. Puis, l'extension compile ces tests vers des tests écrits au format atoum. Ces tests sont enregistrés dans les fichiers appropriés grâce à l'exportation, décrite dans la partie 6.2.3. Nous pouvons par la suite exécuter ces fichiers avec atoum.

Exemple 6.13 (Génération automatique de tests unitaires)**.** Si nous reprenons notre classe `Project\HelloWorld` avec sa méthode `say`, nous aurons le contrat suivant :

```
class HelloWorld {

    /**
     * @ensures \result: 'Hello World!';
     */
    public function say ( ) {

        return 'Hello World!';
    }
}
```

Nous allons ensuite générer automatiquement les tests associés en utilisant la ligne de commande suivante :

```
$ praspel generate --class 'Project\HelloWorld' --test-root .
```

L'option `--class` précise la ou les classes à analyser. L'option `--test-root` précise où se trouve la racine des tests, en l'occurence à l'endroit où nous exécutons la ligne de commande. Un fichier par classe sera généré. Le contenu du fichier que nous venons de générer est le suivant :

```
class HelloWorld extends \Atoum\PraspelExtension\Test {

    // ...

    public function test say n°1 ( ) {

        $this
            ->if($this->praspel->ensures['\result']->in =
                realdom()->const('Hello World!'))
            ->then
                ->praspel->verdict('\Project\HelloWorld');

        return;
    }
}
```

Nous pouvons enfin exécuter ce test avec atoum grâce à la ligne de commande suivante :

```
$ atoum --files Path/To/Our/Generated/Test.php
...
> Total test duration: 0.01 second.
> Total test memory usage: 0.50 Mb.
> Code coverage value: 100.00%
> Running duration: 0.06 second.
> Success (1 test, 1/1 method, 0 void method,
           0 skipped method, 1 assertion)!
```

Sans surprise, la suite de tests a été exécutée avec succès.
Nous remarquons une assertion de moins par rapport à l'exemple 6.7. Au moment de la rédaction de ce mémoire, l'extension n'est pas encore totalement branchée avec le composant dédié aux rapports dans atoum. Ainsi, toutes les « assertions » de Praspel ne sont pas comptabilisées (seulement 1 par test).

6.4 Synthèse

Dans ce chapitre, nous avons présenté l'implémentation de Praspel sous forme de plusieurs bibliothèques dans le projet Hoa (`Hoa\Praspel`, `Hoa\Realdom`, `Hoa\Compiler` et `Hoa\Regex`). Nous avons vu que la pièce centrale de Praspel est son modèle objet. Le langage peut être analysé, exporté, importé et désassemblé. Un *Runtime Assertion Checker* est proposé pour évaluer un SUT à l'exécution spécifié par un contrat représenté par son modèle objet. L'évaluation peut se faire avec des données de test fournies manuellement ou générées automatiquement.
Praspel a été intégré dans atoum, un *framework* de tests unitaires, afin de profiter du moteur d'exécution, de la production des rapports, de l'intégration à plusieurs IDE et outils industriels, des API avancées à l'intérieur d'atoum etc. atoum n'avait par ailleurs pas de générateurs de données de test : c'est pourquoi à travers l'extension `atoum\praspel-extension`, nous proposons de nouvelles API pour générer des données de test grâce aux travaux présentés dans ce mémoire. En plus, cette extension permet de transformer une suite de tests générée par Praspel en une suite de tests exécutable avec atoum. Enfin, cette extension est incluse dans la distribution standard d'atoum « sous stéroïde ».
Grâce à cette extension, aux utilisateurs et aux licences *open-source* de Hoa et d'atoum, nous avons pu mener des expérimentations présentées dans le chapitre suivant.

Chapitre 7.
Expérimentations

Table des matières

Cinq expérimentations sont présentées dans ce chapitre. Les deux premières expérimentations sont **quantitatives** et ont été réalisées en interne. Elles ont été publiées dans plusieurs articles [Enderlin et al., 2011, 2012, 2013]. Les trois

suivantes sont **qualitatives** et ont été réalisées en externe, c'est à dire par d'autres personnes que nous.
Nous proposons d'introduire un langage de spécification pour PHP, simple, pragmatique et assemblant plusieurs méthodes du test. Le but est de présenter toutes ces méthodes et de simplifier leur utilisation dans le quotidien des ingénieurs de test et développeurs. Nous allons commencer, dans la première partie de ce chapitre, par valider les techniques introduites avant de nous tourner vers les ingénieurs, dans la seconde partie de ce chapitre, pour qu'ils jugent notre travail. Nous pensons qu'il est pertinent d'obtenir une expertise humaine à travers les retours et remarques d'ingénieurs de test expérimentés.
La première expérimentation, présentée dans la partie 7.1, valide notre approche basée sur les grammaires pour les chaînes de caractères. La deuxième expérimentation, présentée dans la partie 7.2, valide notre approche basée sur le solveur pour les tableaux. La troisième expérimentation, présentée dans la partie 7.3, a pour contexte un projet d'enseignement utilisant un robot, appelé UniTestor, développé par nos soins dans le cadre d'un événement de formation. La quatrième expérimentation, présentée dans la partie 7.4, a été réalisée auprès d'un panel d'ingénieurs de test bénévoles afin d'obtenir des retours concrets sur des programmes du « monde réel ». Enfin, la dernière expérimentation, présentée dans la partie 7.5, présentera un usage inattendu de Praspel pour du test de performance et de non-régression d'une extension officielle de PHP.

7.1 Test à partir de grammaires

Cette partie contient deux expérimentations publiées dans l'article Enderlin et al. [2012]. La première a été conçue pour valider que le compilateur de compilateurs LL($\star$) et les algorithmes de génération de chaînes de caractères fonctionnent correctement. Cette expérimentation est une auto-validation. La seconde expérimentation montre l'utilisation de Praspel pour valider des applications Web.

7.1.1 Auto-validation du compilateur et des algorithmes

Notre première expérimentation vise à valider notre approche *grammar-based testing*, soit à s'assurer que, d'une part, le compilateur de compilateurs LL($\star$) fonctionne correctement, c'est à dire qu'il accepte des données correctes et rejette des données incorrectes, et que, d'autre part, les générateurs de chaînes de caractères fonctionnent correctement également, c'est à dire qu'ils ne génèrent pas de données incorrectes par rapport à une grammaire.
Nous travaillons alors en deux étapes. Premièrement, nous générons des données automatiquement à partir de grammaires qui utilisent toutes les constructions du

Grammaire / n	1	2	3	4	5	6	7
JSON	4	0	6	4	30	20	180
PCRE	1	4	9	36	117	420	1525

8	9	10	11	12	13	14
128	1156	848	8060	6256	59596	TO
5608	21021	79528	304201	1173288	4559049	TO

FIGURE 7.1 – Nombre de données de taille n pour chaque grammaire.

langage de description de grammaires PP. Deuxièmement, toutes ces données seront analysées par notre propre compilateur et d'autres compilateurs. L'objectif est triple :

1. vérifier que les données correctes sont acceptées par tous les compilateurs ;
2. vérifier que les données incorrectes sont refusées par tous les compilateurs ;
3. vérifier que tous les compilateurs ont le même avis sur la validité des données.

Nous considérons une grammaire de JSON (*JavaScript Object Notation* [ECMA, 2013]) et une grammaire simplifiée des PCRE (*Perl Compatible Regular Expressions* [Hazel, 1997]). Ce choix est motivé par le fait que nous travaillons dans le domaine du web et que ces grammaires sont implémentées au sein de langages du Web, comme PHP ou Javascript, avec des API simples nous permettant de valider les données que nous allons générer.
La figure 7.1 montre un aperçu des tailles des grammaires en terme de nombres de données d'une certaine taille n qui peuvent être produites. Dans la figure, TO signifie *Time Out* et indique que le temps de génération des données a dépassé 10 minutes. Ces tailles ont été calculées à partir du nombre de données générées par l'algorithme exhaustif borné.
Nous avons tout d'abord expérimenté avec la grammaire de JSON pour générer des descriptions d'objet JSON. Nous avons utilisé l'algorithme exhaustif borné et l'algorithme basé sur la couverture.

Algorithme exhaustif borné. À cause du nombre important de règles imbriquées, l'algorithme exhaustif borné n'est pas capable de produire des données de taille raisonnable couvrant toutes les règles (et ainsi générer des descriptions complexes d'objet) malgré le fait que nous générions toutes les descriptions d'objets de taille $n \leqslant 9$ en un temps raisonnable (quelques minutes).

Algorithme basé sur la couverture. L'algorithme basé sur la couverture produit moins de données mais il génère des descriptions complexes d'objets contenant jusqu'à 32 lexèmes. Il est intéressant de noter que toutes les règles sont couvertes

avec très peu de données, ici 3 données en moyenne sont nécessaires pour couvrir toute la grammaire de JSON. Nous avons également remarqué que cet algorithme se comporte comme l'algorithme du postier Chinois dans le domaine du test de FSM (*Finite State Machines*) et tend à produire une première donnée longue qui couvre le maximum de règles, puis produit des données plus petites pour couvrir les quelques règles n'ayant pas été couvertes précédemment. Aussi, contrairement aux algorithmes précédents qui sont relativement lents (l'algorithme aléatoire uniforme nécessite une étape de dénombrement exponentielle et l'algorithme exhaustif borné est hautement combinatoire), cet algorithme est capable de produire des données complexes en quelques millisecondes. Ainsi, nous l'avons utilisé de manière répétée pour produire d'importants ensembles de données dont la variété était assurée par les choix aléatoires dans la résolution des points des choix, comme expliqué dans la partie 4.3.3.

Validation et erreurs. Durant cette évaluation, nous avons trouvé une erreur dans deux de nos algorithmes, qui était détectée par le compilateur. L'origine de l'erreur était une mauvaise gestion des caractères échappés, et par conséquent, des données incorrectes étaient produites à partir d'une grammaire valide.
Pour évaluer notre compilateur de compilateurs, nous avons réinjecté ces données dans le compilateur. Nous n'avons relevé aucune erreur durant cette étape : toutes les données générées par nos algorithmes ont été correctement analysées par notre propre compilateur. Additionnellement, nous avons validé ces données avec l'analyseur JSON de Gecko (créé par Mozilla et utilisé par exemple dans Firefox pour Javascript) et l'analyseur JSON de PHP. Toutes les données générées ont été correctement analysées par ces deux autres outils.
Pour tester davantage, nous avons modifié la grammaire en y introduisant des erreurs, afin de générer des données incorrectes. Nous avons considéré des opérations simples de mutation de grammaires [Offutt, Ammann, and Liu, 2006], telles que : remplacement des opérateurs de répétitions (+ devient *, modification des bornes minimales et maximales), suppression d'un lexème ou d'un appel à une sous-règle dans une règle, ajout de point de choix vers des règles existantes etc. Nous avons vérifié si les nouvelles données générées étaient soit acceptées soit refusées par notre compilateur et nous avons comparé les verdicts avec ceux des deux autres outils. Durant cette validation, nous avons trouvé une erreur dans notre compilateur qui considérait incorrectes des données correctes. La cause de cette erreur était une mauvaise gestion du *backtracking*.

Après correction de cette erreur, nous avons recommencé l'expérimentation en entier avec la grammaire des PCRE au lieu de JSON. Aucune erreur n'a été détectée, validant alors nos algorithmes et notre compilateur.

7.1.2 Validation d'applications Web

Nous avons aussi conçu une expérimentation qui utilise Praspel pour la validation d'applications Web. Nous avons sélectionné des projets d'étudiants du cours de « Langage Web », dans lequel les étudiants apprennent le langage PHP et l'utilisent pour générer des documents écrits en HTML. En premier exercice, les étudiants devaient écrire des fonctions qui vérifient des données provenant de formulaires, incluant des adresses emails. Nous avons utilisé le domaine réaliste des adresses emails pour générer des données de test pour ces fonctions. Aucune erreur n'a été détectée : toutes les fonctions validaient correctement les adresses emails générées. Toutefois, nous avons suspecté que ces fonctions étaient en fait trop « faibles » et accepteraient des adresses emails incorrectes (par exemple, contenant deux symboles « `@` »). Pour détecter ces erreurs, nous avons décidé d'utiliser des mutations similaires à l'expérimentation précédente, mais en utilisant une grammaire pour les adresses emails, ce afin de générer des adresses invalides. Nous avons comparé les résultats de toutes ces fonctions avec notre propre fonction qui utilise le domaine réaliste initial validant des adresses emails correctes. Plusieurs erreurs ont été détectées. Ce premier exercice montre qu'il est possible d'utiliser Praspel et les domaines réalistes pour générer des données valides et invalides.
En second exercice, les étudiants devaient générer des blocs de code HTML qui seraient par la suite assemblés. Le SUT est une fonction, existant sous 7 versions différentes et faite par 7 étudiants. Cette fonction est utilisée pour sélectionner une date, représentée par trois balises `select`, respectivement pour sélectionner le jour, le mois et l'année. Les paramètres de cette fonction sont trois entiers représentant les valeurs par défaut de ces trois balises. La spécification informelle de cette fonction est la suivante :
– le code produit doit contenir trois balises `select` en une ligne ;
– le code HTML produit doit être « protégé » (tous les accents doivent être remplacés par des entités XML) ;
– les jours vont de 1 à 31, les mois de 1 à 12 et les années de 2011 à 1911 ;
– uniquement une option (un élément d'une balise `select`) peut être sélectionnée par défaut.
Nous avons conçu plusieurs prédicats basés basés sur plusieurs degrés de granularité.

1. Le premier niveau vérifie que le code HTML est bien structuré. Pour y arriver, nous avons utilisé une grammaire simplifiée du langage XML, vérifiant que toutes les balises ouvertes sont correctement fermées et que les attributs sont syntaxiquement corrects. Tous les étudiants ont passé ce niveau.
2. Le deuxième niveau vérifie que le code HTML généré contient trois balises `select` syntaxiquement correctes. Pour y arriver, nous avons utilisé une gram-

maire d'un sous-ensemble du langage HTML représentant ces trois balises `select`. 4 étudiants n'ont pas passé ce niveau.

3. Finalement, le troisième niveau ajoute un visiteur dédié qui vérifie le contenu du code généré : les valeurs des options pour chaque balise `select`, l'existence d'exactement une option sélectionnée par défaut par balise etc. Seulement 2 des étudiants restants ont passé ce niveau.

En plus de nous montrer que seulement 2 étudiants sur 7 sont capables de suivre des spécifications simples, cette expérimentation nous a montré que Praspel est très commode pour tester des applications. L'approche par les grammaires est utile pour spécifier le format attendu de code HTML. En plus, la flexibilité offerte par les visiteurs permet de valider des cas de tests plus complexes, en vérifiant par exemple des contraintes structurelles qui ne pourraient pas être exprimées au sein d'une grammaire.

7.2 Solveur sur les tableaux

Cette partie présente l'expérimentation que nous avons menée dans l'article Enderlin et al. [2013]. Dans cette partie, nous utilisons le mot **système** pour parler d'un ensemble de propriétés sur les tableaux. Cette expérimentation évalue l'efficacité du solveur présenté dans la partie 4.2 page 48, c'est à dire sa capacité à supprimer ou réduire le rejet lors de la génération de données à partir d'un système. L'expérimentation est composée de trois étapes :

– génération de systèmes (à partir d'une grammaire et des algorithmes présentés dans la partie 4.3.3 page 69) ;
– exécution du solveur sur chaque système généré ;
– mesures du nombre de *backtracks* dans le solveur, le temps de génération de données à partir d'un système satisfaisable et le nombre de systèmes non satisfaisables détectés.

Nous générons automatiquement des systèmes dont les propriétés portent sur des tableaux contenant des chaînes de caractères et des entiers, et dont la taille des tableaux varie entre 5 et 20.

Praspel est décrit par une grammaire. Afin de générer des systèmes, nous réutilisons les travaux présentés dans la partie 4.3 page 61 concernant la génération de chaînes de caractères à partir de grammaire. Trois algorithmes ont été présentés : aléatoire uniforme, exhaustif borné et basé sur la couverture. Puisque la grammaire des propriétés sur les tableaux est petite, nous ne pouvons pas utiliser le dernier algorithme car il ne sera pas capable de générer une collection de systèmes suffisamment importante. L'algorithme exhaustif borné est plus coûteux que l'algorithme aléatoire uniforme mais il est précis et plus adapté à des petites grammaires. Nous

n	systèmes générés	*backtracks*	*backtracks* par système	systèmes rejetés	temps de génération (ms)
10	14	0	0	0	6.484
15	86	34	0.40	0	42.167
18	210	91	0.43	0	141.694
19	275	103	0.37	0	229.001
20	492	114	0.23	0	372.241

FIGURE 7.2 – Résultat de l'expérimentation sur les tableaux

retenons cet algorithme. Ce dernier va générer des séquences de lexèmes dont la taille sera bornée à n. Ces séquences seront ensuite concrétisées pour, dans notre cas, former un système. Nous rappelons que les lexèmes sont concrétisés avec un algorithme aléatoire isotropique.
Avec $n = 3$, nous pouvons générer des propriétés de la forme `a is unique` (avec `a` un tableau). Avec $n = 6$, nous pouvons générer des propriétés de la forme `a[0]: 0`. Avec $n = 8$, nous pouvons générer des propriétés des la forme `a[0]: 0 or 1`. Avec $n = 11$, nous pouvons générer des propriétés comme celle de l'exemple 4.3 page 52 soit `a[0]: 'b' or 'c' and a is unique`.
La deuxième étape invoque le solveur sur chaque système généré afin de générer un tableau satisfaisant ce système. Chaque tableau est validé par les prédicats des domaines réalistes et par les propriétés des tableaux pour vérifier la justesse (*soundness*) du solveur, c'est à dire qu'il génère des données qui sont correctes.
Durant cette étape, des marqueurs permettent de compter uniquement le temps de génération des tableaux sans compter le temps de compilation (génération des systèmes, analyses des systèmes, création du modèle objet de Praspel etc.). Nous mesurons également le nombre de *backtracks* dans le solveur et le nombre de systèmes rejetés. Quand un système est rejeté, ses *backtracks* ne sont pas comptabilisés. Puisque la concrétisation des séquences de lexèmes utilise un algorithme aléatoire isotropique, les valeurs peuvent être très différentes d'un système généré à l'autre. Ces différences peuvent conduire à des amplitudes dans les temps de génération et les nombres de *backtracks*. Pour éviter des pics dans les résultats, nous calculons la moyenne de 100 exécutions.
La figure 7.2 montre les résultats de notre expérimentation. Dans la première colonne, n est la taille maximum de la séquence de lexèmes représentant un système. La deuxième colonne indique le nombre de systèmes distincts générés avec cette taille. La troisième, cinquième et sixième colonne indiquent respectivement les nombres moyens de *backtracks*, de systèmes rejetés pour 100 exécutions et la moyenne des temps de génération des tableaux. La quatrième colonne indique le taux de *backtracks* par système.

Pour $n \leqslant 20$, nous observons qu'aucun système n'est rejeté, ce qui est une importante amélioration par comparaison avec la génération standard qui est aléatoire (voir la partie 4.1 page 46). La génération standard introduisait un taux de rejet supérieur à 90% face à de tels systèmes. Toutes les données générées satisfont leur spécification. Le solveur génère avec succès et rapidement des données avec un faible nombre de *backtracks*. $n = 20$ représente approximativement 3 propriétés avec des disjonctions : l'algorithme exhaustif borné génère 492 systèmes distincts et les exécutions produisent seulement 114 *backtracks*, soit approximativement 1 *backtrack* pour 4 systèmes. C'est un bon résultat. Néanmoins, nous n'avons pas été capables de caractériser le nombre de *backtracks* par rapport au nombre de propriétés. Cela aurait pu permettre d'observer la réaction du solveur face à certaines formes de contraintes, et potentiellement de lui ajouter des optimisations. Enfin, cette expérimentation nous a permis de détecter une erreur dans le solveur. Cette erreur conduisait systématiquement au rejet de systèmes ayant un certain motif. Par conséquent, nous remarquons que cette démarche de validation est efficace pour détecter des erreurs.

7.3 Étude de cas : UniTestor

Les Journées nationales du Développement Logiciel, abrégé JDév' [1], est une Action Nationale de Formation inter-établissements, soutenue par la Mission pour l'interdisciplinité du CNRS, de l'INRIA et de l'INRA, organisée en 2013 à l'École Polytechnique. Une équipe de notre laboratoire a été invitée afin d'y donner des cours et des ateliers. C'est dans ce cadre que nous avons développé UniTestor, un robot écrit en PHP nous permettant d'aborder la majorité des aspects du test unitaire : écrire un préambule de test, définir des assertions, boucher des objets etc. 16 élèves ont alors écrit manuellement une suite de tests pour ce robot. Nous la comparons avec une suite de tests générée automatiquement avec Praspel.

7.3.1 Présentation générale

UniTestor est une simulation de robot pour l'exploration spatiale. Son diagramme de classes UML simplifié est présenté dans la figure 7.3. Le robot est représenté par la classe `Robot`. Il est constitué de plusieurs périphériques, que nous allons détailler, lui envoyant des informations. L'ensemble du robot comporte au total 5 classes et 21 méthodes, ce qui représente 340 lignes de code (sans les contrats Praspel).

1. Voir http://devlog.cnrs.fr/jdev2013.

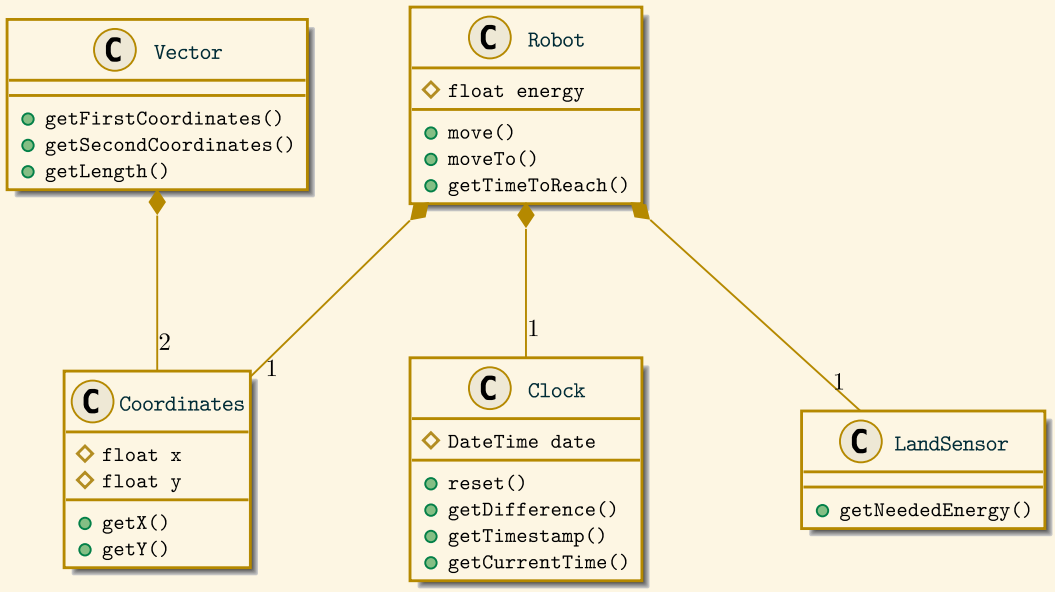


FIGURE 7.3 – Diagramme de classes UML du robot UniTestor.

Équipements et capteurs

Le robot est capable de se déplacer en utilisant des coordonnées relatives ou absolues avec respectivement les méthodes `move` et `moveTo`. Pour se déplacer, il utilise une balise de géolocalisation représentée par la classe `Coordinates`. À tout moment, il peut connaître la nature du terrain avec le capteur de sol représenté par la classe `LandSensor`. La nature et la configuration du terrain influent sur l'énergie utile à un déplacement. Le robot a également sa propre horloge, représentée par la classe `Clock`, lui permettant de connaître des informations sur le temps. Enfin, la classe `Vector` permet de manipuler des paires de coordonnées.

Les opérations principales de UniTestor

La seule opération importante est le déplacement. Lorsque le robot se déplace, il consomme de l'énergie, représentée par un pourcentage. En fonction du terrain, plus ou moins d'énergie sera nécessaire. L'énergie nécessaire est calculée par la fonction `getNeededEnergy` du capteur de sol. Mais le robot a des batteries qui se rechargent en fonction du temps. Le temps qui s'écoule est donné par l'horloge et la recharge est linéaire (paramétrable par une constante).

7.3.2 *Modus operandi* et résultats

Nous avons commencé par annoter le code du robot par des contrats Praspel. Ensuite, nous avons généré automatiquement une suite de tests, appelée la suite AGT (*Automatically Generated Tests*) satisfaisant le critère de couverture de contrat *All*-G (défini dans la partie 5.3.4). Nous avons comparé cette suite de tests avec la suite de tests manuels des étudiants, appelée la suite MT (*Manual Tests*), complétée pour atteindre un taux de couverture de code de 100% quand ce n'était pas le cas.
Nous avons observé que le nombre moyen de tests générés par les étudiants est de 53, alors que pour 21 méthodes, soit 21 contrats, Praspel a généré automatiquement 29 tests, soit une réduction de 45%. Pour caractériser la pertinence des tests, nous avons utilisé comme métrique la couverture de code. Le critère de couverture de code le plus fin que nous propose atoum est le critère tous-les-arcs. Ainsi, nous avons observé que les deux suites de tests avaient un score de 100%. De plus, 1h30 a été nécessaire aux étudiants pour rédiger en moyenne 53 tests, alors qu'il nous a fallu 15 minutes pour écrire 21 contrats. La phase de génération de tests a pris moins d'une seconde : cela comprend la génération automatique des données de test, à savoir des entiers, des réels, des chaînes de caractères spécifiées par des expressions régulières et des objets, en particulier plusieurs instances complètes du robot (avec ses périphériques).
Ainsi, nous obtenons le même score de couverture du code avec pratiquement moitié moins de tests et en 6 fois moins de temps. Ces premiers résultats montrent l'efficacité de l'approche du test automatisé et le gain qu'apportent les contrats : écrire des contrats est moins pénible et nécessite moins de temps que d'écrire une suite de tests complète.
Cependant, cette expérimentation est biaisée : les étudiants étaient en apprentissage de l'outil et un temps d'adaptation leur était nécessaire. De plus, nous sommes des experts de Praspel, donc nous rédigeons des contrats très rapidement. Afin d'évaluer notre approche de manière plus objective, nous avons demandé à un panel de bénévoles, composé d'ingénieurs de test et d'entreprises, d'appliquer une expérimentation similaire sur leur propre code.

7.4 Ingénieurs bénévoles

L'expérimentation précédente a été réalisée auprès d'étudiants. Nous voulons valider nos résultats auprès d'experts industriels sur des programmes du « monde réel ».
La partie 7.4.1 présente les ingénieurs bénévoles et les programmes sur lesquels ils ont joué l'expérimentation. La partie 7.4.2 présente les protocoles expérimentaux qu'ils ont appliqué. Les parties 7.4.3 et 7.4.4 présentent les résultats obtenus avec la

génération automatique de tests. La partie 7.4.5, quant à elle, présente les résultats obtenus avec uniquement la génération de données de test. Beaucoup d'erreurs ont été détectées durant cette expérimentation. La partie 7.4.6 en présente quelques-unes. Enfin, cette expérimentation s'est achevée sur une discussion, synthétisée dans la partie 7.4.8, concernant les bénéfices de l'approche par les contrats pour le test, la pertinence de nos algorithmes de génération de données, de Praspel et ses outils.

7.4.1 Présentation générale

Nous avons lancé un appel sur différents réseaux sociaux et auprès de différentes communautés pour rassembler des bénévoles pour cette expérimentation. 7 ingénieurs de test de nationalités différentes ont répondu à l'appel. Ces ingénieurs font du test depuis en moyenne 5 ans. Ils ont appliqué l'expérimentation sur leurs propres programmes ou sur des programmes auxquels ils participent (en tant que testeurs ou développeurs). Ces programmes constituent nos cas concrets. Voici la description de ces programmes :

- outil de *debugging* : analyse les performances d'exécutions de certaines parties de programmes et génère des rapports dans deux formats différents. Les parties abordées durant cette expérimentation étaient la mesure des données et la génération des rapports.
- outil de test pour le langage JSON : une autre extension à atoum était en développement pour proposer des nouvelles assertions sur le langage JSON. Les parties abordées durant cette expérimentation étaient les assertions et les tests de cette extension, c'est à dire s'assurer que l'outil de test testait correctement le langage JSON.
- outil d'échanges de messages : les messages sont écrits en XML et contiennent des données de toutes sortes, notamment des dates. C'est cette partie qui a été majoritairement abordée durant cette expérimentation car centrale dans cet outil.
- générateur de données de test : pour tester certaines parties techniques d'un outil, un générateur de données de test spécifique a été créé. Afin de s'assurer que ce générateur ne comporte pas d'erreurs et étant lui-même assez volumineux, il a sa propre suite de tests. Les parties qui ont été abordées durant cette expérimentation étaient le calcul de ces données, les tests unitaires et les tests fonctionnels.
- outils de comptabilités : 2 outils développés sur mesure pour des calculs de comptabilité. Les parties abordées étaient multiples.
- outil d'évaluation de formules mathématiques : des utilisateurs écrivent eux-mêmes des formules arithmétiques décrivant l'évolution en temps réel des tarifs de certains produits. Une bibliothèque de fonctions, de constantes et de variables est mise à la disposition de ces utilisateurs. La partie abordée était l'évaluation

de ces formules analysées avec un compilateur et une grammaire spécifiques.
Tous ces programmes font intervenir tous les types de données, à savoir des booléens, des entiers, des réels, des chaînes de caractères, des tableaux et des objets. La plupart des chaînes de caractères pouvaient être spécifiées par une expression régulière, et quelques-unes par des grammaires.
Sur ces programmes variés, nous avons demandé à notre panel d'appliquer deux protocoles expérimentaux.

7.4.2 *Modi operandi*

Cette expérimentation s'est déroulée en deux temps. Tout d'abord, nous avons demandé au panel d'appliquer le protocole expérimental suivant :

1. sélectionner plusieurs méthodes de toutes sortes déjà testées manuellement (avec atoum) ;
2. annoter ces méthodes avec des contrats Praspel ;
3. générer automatiquement une suite de tests satisfaisant le critère de couverture de contrat *All*-G, et l'exécuter ;
4. comparer la suite MT (*Manual Tests*) avec la suite AGT (*Automatically Generated Tests*).

Puis, nous leur avons demandé d'expérimenter les algorithmes de génération de données seuls, sans Praspel. Nous rappelons que l'extension introduisant Praspel dans atoum permet la définition et l'utilisation des domaines réalistes facilement (voir la partie 6.3.2). Il n'y avait pas de protocole expérimental pré-défini mais nous avons demandé aux ingénieurs d'observer l'impact de certains générateurs sur la couverture de code ainsi que l'impact sur les suites de tests (par exemple une réduction du nombre de tests).
L'expérimentation a été ouverte pendant une première semaine avec un premier panel puis pendant deux autres semaines avec un second panel plus important. Les parties suivantes regroupent toutes les données et informations des deux panels.

7.4.3 Couverture de code des suites de tests

Un moyen de comparer les suites de tests est d'utiliser la couverture de code avec le critère tous-les-arcs, comme pour l'expérimentation précédente. Les résultats sont les suivants.
Nous avons observé que les AGT couvrent autant le code que les MT, à $\pm 5\%$ près. Quand les AGT couvrent moins le code que les MT, le panel nous a fait remarquer que c'était à cause de l'utilisation des bouchons (des *mocks*). Un bouchon remplace une partie d'un programme et nous prédéfinissons la manière dont il va se comporter, c'est à dire que nous décrivons un comportement bien spécifique

et testons le SUT dans ces conditions. Les bouchons permettent d'améliorer la testabilité des programmes (en plus d'accélérer l'exécution des suites de tests par effet de bord). Les bouchons ne concernent que les objets ou les fonctions, et Praspel n'est pas capable de générer de tels bouchons. Par exemple, si un contrat précise qu'une exception peut être levée quand une connexion réseau est interrompue, Praspel ne sera pas capable de couper lui-même la connexion. Il faut alors boucher les interfaces réseaux et simuler des déconnexions.

Lorsque les AGT couvrent plus le code que les MT, c'est à cause d'une limitation des outils. En effet, atoum ne permet pas de tester les méthodes avec une visibilité autre que publique, alors que Praspel le permet. Le panel doit alors tester les méthodes protégées et privées à travers les méthodes publiques. Il est évident que certains chemins dans le code n'étaient pas accessibles facilement.

Dans certains cas, malgré une couverture de code de 100% pour les MT et les AGT, des erreurs ont été trouvées. En effet, la couverture tous-les-arcs n'est pas très fine et des erreurs peuvent apparaître derrière des conditions qui n'auraient pas dû être activées. Comme le critère de couverture *All*-G sur les contrats combine les domaines réalistes de toutes les variables, certaines combinaisons peuvent activer ces conditions d'une manière qui n'avait pas été testée manuellement.

Maintenant, comparons le temps de rédaction de ces suites de tests.

7.4.4 Temps de rédaction des suites de tests

Tous les membres du panel sont habitués à écrire des tests. Ils en écrivent des dizaines par jour. Aucun d'entre eux n'avait écrit un contrat en Praspel avant cette expérimentation, mais certains connaissaient toutefois ce paradigme. Nous leur avons demandé de comparer les temps de rédaction des suites MT et AGT.

Pour la partie du panel qui n'était pas habituée à la programmation par contrat, plusieurs heures (entre 4 et 10h) ont été nécessaires pour bien comprendre le principe et savoir comment l'appliquer. Cette période comprend aussi l'analyse des AGT : savoir les lire, les comprendre et les positionner par rapport aux MT. Durant cette période, écrire des MT était bien plus rapide que d'écrire des contrats et produire des AGT. Pour l'autre partie du panel, même si elle connaissait la programmation par contrat, elle ne l'avait jamais pratiquée. Les problèmes rencontrés étaient les mêmes : savoir lire des contrats, comprendre et analyser les AGT. Le panel pondère toutefois cet effort : en comparaison avec les autres outils qu'ils manipulent quotidiennement, l'installation de Praspel et son extension pour atoum est très rapide (autour d'une minute) et le langage est simple à utiliser. Même si les notions sous-jacentes (les algorithmes et les méthodes de test utilisés) ne sont pas connues ou comprises, les domaines réalistes sont également une notion simple à comprendre et à utiliser.

Une fois le cap de l'apprentissage franchi, le panel a observé que pour peu de

code à tester (moins de 5 ou 6 méthodes), il était plus rapide d'écrire des MT. En revanche, lorsqu'il faut tester plus de code (ce qui est pratiquement toujours le cas), l'utilisation des contrats s'avère très efficace. Il faut entre 2 à 4 fois moins de temps pour écrire les contrats et produire les AGT que pour écrire les MT à la main (de 2h pour les MT à 30mn pour les AGT par exemple). Pour mettre ces résultats en perspective avec la partie précédente, le temps gagné est aussitôt réinvesti pour écrire des MT plus « poussés », comprendre plus complets.
Le panel nous a en effet fait remarquer que malheureusemnt, les tests ne sont pas encore une priorité dans les budgets alloués au développement logiciel. Par conséquent, les ingénieurs ont un temps limité pour tester leurs programmes. Pendant le temps imparti, ils n'étaient capables d'écrire que des tests dits « basiques » ou alors ils ne testaient que les parties les plus critiques du code. Avec les contrats, ils spécifient toutes les méthodes car les contrats sont écrits en même temps que le code. La proximité entre le code et les contrats *via* les annotations est un avantage. Par conséquent, plus de code est testé sans nécessiter autant de temps que s'il devait être testé manuellement. Le temps qui a été économisé est ainsi réinvesti pour des tests plus complets, qui n'auraient pas pu être faits en temps normal. Autrement dit, les tests « usuels » sont gérés par Praspel, ce qui permet aux ingénieurs de test de se concentrer sur les parties plus critiques.
Nous aurions aimé faire l'expérimentation 2 fois de suite pour comparer la courbe d'apprentissage, mais nous n'en avons pas eu le temps. Nous rappelons que le panel était constitué de bénévoles.

7.4.5 Génération de données de test

Nous avons précisé dans les parties précédentes que les suites d'AGT ne sont pas toujours complètes. En effet, Praspel n'est pas capable de tout spécifier (nous détaillerons plusieurs cas dans la suite de cette partie) et certaines parties du code ne sont pas testables facilement. Heureusement, la partie précédente nous a montré que le temps gagné grâce aux contrats est réinvesti dans l'écriture de MT. Dans cette étape, les ingénieurs de test ont utilisé nos générateurs de données. Rappelons qu'atoum ne possède aucun générateur de données. Le seul outil qu'il propose consiste à se brancher sur un dictionnaire de données écrit manuellement. Nous présentons trois constats faits par le panel.

Premier constat. Les domaines réalistes permettent de spécifier simplement et facilement une grande majorité des données. Pouvoir utiliser les domaines réalistes dans du code PHP directement permet de mixer et d'orienter la génération facilement. Par exemple, écrire des boucles où chaque pas produit une donnée légèrement différente de la précédente, ou encore utiliser des données générées pour modifier

d'autres données provenant de bases ou d'autres sources.
Par ailleurs, comme les données de test ne sont plus déclarées manuellement, cela simplifie grandement l'écriture des tests. Le panel a précisé que ce n'était « même pas comparable » : il n'y avait rien avant et c'était un travail pénible. De même, cela simplifie aussi la lecture des tests. En effet, le contenu du test se restreint à sa logique plutôt qu'à la manipulation des données de test. Le panel insiste sur le fait que les tests sont par conséquent plus facilement maintenables. Cela implique que les ingénieurs n'hésiteront pas à modifier les suites de tests car ce processus est simplifié.

Deuxième constat. Sans générateur automatique de données, le panel nous avoue que les données de test utilisées étaient très peu nombreuses : entre 1 à 5 données par tests, 1.3 en moyenne. L'objectif était d'assurer une certaine couverture du code avec le critère tous-les-arcs. Avec nos générateurs de données, beaucoup de données sont générables « instantanément » pour reprendre leurs termes. En effet, le temps de génération est suffisamment rapide pour qu'il en devienne « négligeable ». Le panel souligne qu'« aucune différence n'est perceptible » entre l'exécution des suites de tests avec ou sans générateurs de données. Dans certaines méthodologies de développement comme le *Test-Driven Development* ou les méthodes Agile, les tests sont exécutés en permanence, plusieurs dizaines de fois par heure. La rapidité d'exécution des tests est alors importante pour ne pas impacter la vitesse de développement des équipes.

Troisième constat. Générer beaucoup de données de test n'a aucun sens si elles sont toutes inutiles. Le panel a majoritairement apprécié les algorithmes de génération de chaînes de caractères. Pouvoir spécifier des chaînes de caractères avec des expressions régulières ou des grammaires, et ensuite générer toutes les données de manière exhaustive ou alors par couverture de la grammaire, a été remarquablement apprécié. Plusieurs points sont à noter.
Tout d'abord, avoir une exhaustivité ou une couverture sur les données manipulées est un avantage important : cela augmente la confiance dans les tests. La notion de couverture de données est aussi importante que la couverture du code pour plus de la moitié de notre panel.
Ensuite, spécifier des chaînes de caractères avec des expressions régulières est facilité par le fait que, très fréquemment, ces expressions régulières sont déjà définies dans l'implémentation. Il suffit alors de les réutiliser. Quand nous avons demandé comment le panel générait des chaînes de caractères avant, nous avons été surpris de voir qu'un seul ingénieur en générait en concaténant des ensembles de sous-chaînes de caractères. Les autres écrivaient quelques chaînes de caractères manuellement. Dans le cas des grammaires, un ingénieur a fait remarquer qu'en plus d'apprendre

```
/**
 * @requires i: 7..42 and
 *           j: /[a-i][1-5]+/;
 * @ensures  \result: true;
 */
public function testSomething ( $i, $j ) {

    $out = false;

    if(...) ...
    // some asserts
    // compute $out

    return $out;
}
```

FIGURE 7.4 – Un test paramétré.

le langage Praspel, il fallait apprendre le langage PP pour décrire des grammaires. Les autres membres du panel n'ont pas relevé cet inconvénient arguant que, vus les services rendus, l'effort demandé était très rapidement amorti.
Enfin, dans le cas des tableaux, le panel a apprécié le fait de pouvoir générer à moindre coût des centaines de tableaux respectant certaines contraintes. Là encore, très peu de tableaux étaient utilisés avant en tant que données de test manuels.

Dans tous les cas, les membres du panel ont remarqué une diminution du nombre de tests. En effet, comme un test manipule plus de données pertinentes, les autres tests du même genre deviennent inutiles. Cette diminution a été difficile à quantifier : 5% pour certains, 75% pour d'autres, mais il y a toujours une simplification de la suite de tests : la quantité, la lecture, l'écriture et la maintenance.

7.4.6 Tests paramétrés

Le test paramétré est une approche hybride entre le test manuel et automatique. Le principe est d'écrire un test dont les paramètres sont spécifiés par un contrat. À partir d'un test paramétré, nous pouvons générer plusieurs nouveaux tests. Un test annoté par un contrat et qui pré-calcule un verdict est présenté dans la figure 7.4.
Les tests paramétrés permettent deux choses. La première est d'automatiser des tests compliqués à générer automatiquement en décrivant un préambule de test non-trivial ou en facilitant le travail de l'oracle. Par exemple, si le succès ou l'échec

d'un test est déterminé par des informations contenues dans un autre programme ou dans une base de données, le test paramétré va synthétiser ces informations de telle façon que la postcondition soit spécifiable dans le contrat (par conséquent avec Praspel). La seconde est d'utiliser des contrats dans un contexte de test fonctionnel, c'est à dire que le contenu du test sera un assemblage de plusieurs briques logicielles de plus haut niveau.
Durant cette expérimentation, seulement un ingénieur de test du panel a pu écrire des tests paramétrés. L'aspect test fonctionnel n'a pas été abordé. Ce membre n'a pas observé de différences particulières avec des MT dans lesquels il utilisait les générateurs de données. La raison principale est que les contrats qui annotaient les tests paramétrés se ramenaient à décrire le type des données pour la précondition dans un seul et unique comportement et la postcondition étaient très fréquemment `\result: true`. L'effort d'écriture du contrat est donc équivalent à l'effort d'écriture d'un MT sauf que le processus de génération des tests était plus compliqué. En effet, l'extension de Praspel dans atoum est prévue à l'origine pour scanner des implémentations et non pas des tests. Le problème était donc technique car il nécessitait d'instrumenter les tests en plus des implémentations ce qui pouvait parfois poser des problèmes.
Cette partie de l'expérimentation s'est arrêtée sur ce constat : les tests paramétrés n'ont pas apporté de réels bénéfices dans ce contexte.

7.4.7 Erreurs détectées

Cette expérimentation a permis de révéler plusieurs erreurs dans tous les programmes testés. Nous analysons les plus intéressantes d'entre elles.

Erreurs avec les dates. Les dates sont régulièrement des sources d'erreurs dans les programmes. L'utilisation des domaines réalistes `Date` et `Timestamp` permettent de générer et valider finement des dates. Une erreur a été découverte avec des dates relatives comme entre « le dernier vendredi de février » (spécifié avec `timestamp('last Friday of February')`) et « le lundi suivant » (spécifié avec `timestamp('first Monday of March')`). Certains cas avec des samedis et dimanches lors d'années bissextiles étaient sources d'erreurs. L'erreur a été détectée par des gardes dans le programme qui ont levé des exceptions. Un test a donc activé une branche de code jusqu'ici non couverte.
Une autre erreur a été détectée dans un programme qui formatait mal les dates. Le mois de janvier était formaté en `1` au lieu de `01`. La donnée de test manuelle portait sur le mois de décembre, mois durant lequel a été écrit le test (l'ingénieur de test s'est inspiré du mois courant de l'époque pour écrire la donnée de test). Lorsqu'il a spécifié sa donnée avec le domaine réaliste `date('d')`, où `d` représente les mois sur

deux chiffres, l'erreur est apparue. L'erreur a été détectée suite à la violation d'une clause `@ensures` des contrats.
Une fois qu'un contrat a été écrit ou qu'un générateur de données a été utilisé manuellement, 2 minutes en moyenne sont nécessaires pour détecter ces erreurs. Le code impacté n'était pas encore en production mais sur le point d'être livré. L'ingénieur a estimé que si l'erreur était apparue en production, son impact aurait été « important ».

Erreurs avec JSON. Un programme devait valider des données formatées en JSON. Les données de test manuelles ne couvraient pas tous les cas et l'utilisation d'un générateur de chaînes de caractères exhaustif a permis de détecter plusieurs erreurs. Elles ont été détectées en autant de temps qu'il était nécessaire pour écrire le test, c'est à dire moins de 3 minutes. L'erreur a été détectée par PHP à l'exécution lors d'opérations incomplètes. Des tests ont activé des branches de code qui n'étaient pas couvertes.
Ce programme n'était pas encore en production, il était en cours de développement. L'impact des erreurs aurait pu être important puisqu'il s'agissait de valider des données et que le programme retournait parfois des faux-positifs et parfois des faux-négatifs (ce qui est moins grave dans ce contexte). Il a fallu quelques minutes pour corriger les erreurs.

Erreurs arithmétiques. Un programme évaluait des formules arithmétiques écrites par des utilisateurs pour caractériser l'évolution de tarifs en temps réel. Une erreur avec les divisions a été trouvée : elles étaient évaluées comme étant associatives gauches au lieu d'associatives droites, seulement lorsque l'opérande droit avait une forme particulière. Cette erreur a été détectée avec le générateur de chaînes de caractères exhaustif en moins de 15 secondes une fois le test écrit. L'erreur a été détectée par la violation d'une postcondition avec une construction `\pred()`.
Ce programme était en production, avec des centaines d'utilisateurs et plusieurs centaines de milliers de clients (pratiquement un million). Des tests unitaires et fonctionnels étaient même écrits et exécutés très régulièrement. Heureusement, aucune formule de cette forme n'a été écrite par les utilisateurs.

Erreurs dans les tests. Dans les deux derniers points abordés, la correction des erreurs a permis de détecter d'autres erreurs dans les tests. En effet, les tests calculaient mal leurs verdicts ou étaient incomplets. Praspel générait des AGT qui échouaient alors que les MT équivalent étaient des succès. Après analyse des AGT et des MT, il s'est avéré qu'il y avait des erreurs dans les MT. Les contrats avaient des postconditions simples (clause `@ensures`) mais réparties dans plusieurs comportements (clause `@behavior`). Les AGT reflétaient toute la combinatoire possible alors

que certains cas étaient oubliés dans les MT. C'est la raison principale de ce genre d'erreurs dans les tests. Une autre raison est que les AGT utilisent des générateurs de données plus complets que les MT écrits sans nos générateurs de données. Un seul AGT exécuté plusieurs fois couvre alors plus de données qu'un MT. Lorsqu'un AGT détectait une erreur sur les données, cette erreur n'était pas détectée dans les MT car la couverture des données était nettement plus faible.

Autres erreurs. D'autres erreurs ont été détectées notamment avec les domaines réalistes `Regex` et `Array` qui généraient des données non prévues par l'implémentation. Ces erreurs ont principalement été détectées en 2 minutes en moyenne, suite à la violation des postconditions des contrats.

7.4.8 Expertise humaine : discussion

Pour finir cette expérimentation, nous avons discuté en groupe avec les membres du panel. Nous avons synthétisé ces discussions en plusieurs sujets.

Bénéfices des contrats. La remarque la plus fréquente était que les contrats offrent une documentation plus complète. Ils sont écrits en même temps que l'implémentation et sont utiles à court terme, alors qu'écrire une documentation n'a pas de réel intérêt à court terme. Elle était rarement écrite ou complète en plus d'être informelle. Cet aspect formel a été relevé plusieurs fois : le contrat se lit facilement et il n'est plus nécessaire de regarder l'implémentation pour comprendre ce que fait la méthode.
Dans des équipes qui travaillent en *Test-Driven Development*, c'est à dire que les tests sont écrits avant l'implémentation, il y a souvent une notion de binômage. Les tests sont écrits par une personne et l'implémentation par une autre. Cette approche fonctionne bien car les tests et l'implémentation sont écrits dans des fichiers séparés. Avec les contrats, ils sont écrits dans le même fichier. Il a donc été nécessaire de réorganiser le mode de travail en échangeant le clavier. Nous avons fait remarquer au panel qu'il était possible que les contrats soient écrits à part mais la proximité entre le contrat et l'implémentation est désirée. Tout d'abord parce que le contrat joue le rôle de documentation et que, d'autre part, il est plus facile de maintenir un contrat lorsqu'il annote une méthode.
De plus, le panel a remarqué que l'écriture d'un contrat obligeait à réfléchir davantage sur les données que manipule la méthode. Les tests obligent déjà à réfléchir à l'architecture et aux API des programmes, mais les contrats vont encore plus loin car ils nécessitent de formaliser ces réflexions. Le temps dédié à l'écriture des contrats permettrait d'éviter des erreurs de conception par la suite. Toutefois, nous n'avons pas réussi à observer ou quantifier ce phénomène.

Bénéfices du *Contract-based Testing*. Utiliser les contrats pour générer des tests apporte principalement un gain de temps. Même si les tests générés automatiquement sont considérés comme « basiques », ils couvrent tout de même toute la spécification et sont pertinents. Le panel nous a confirmé que la majorité des AGT auraient pu être écrits par eux-mêmes mais qu'il leur aurait fallu beaucoup de temps. Le critère de couverture de contrat *All*-G couvre toute la structure du contrat (qui réflète plus ou moins la structure de l'implémentation) et les combinaisons des données. Écrire un contrat étant plus rapide que d'écrire les tests manuellement, le *Contract-based Testing* apporte un gain de temps et de confiance dans les tests.
Le panel souligne encore que cette utilisation des contrats permet de « déblayer le terrain » pour reprendre leurs termes. Ainsi, ils peuvent davantage se concentrer sur des tests plus avancés tout en respectant le budget alloué à cette phase du développement.

Génération de données de test. Utiliser des générateurs de données permet de générer beaucoup de données pertinentes en très peu de temps. Le panel a ressenti plus de confiance dans les tests avec les générateurs plutôt qu'avec des données de test manuelles. Ce ressenti s'est confirmé par le nombre d'erreurs détectées en peu de temps (en quelques minutes) dans des programmes déjà testés.
L'utilisation de générateurs simplifie également l'écriture des tests, ainsi que leur lecture et leur maintenance. Les générateurs permettent de réduire la taille des suites de tests.
Ne pas connaître les algorithmes sous-jacents n'est pas réellement un problème. Toutefois, utiliser ce type d'algorithmes pousse la réflexion plus loin. En effet, certains des ingénieurs ont commencé à détourner certains algorithmes, notamment sur les grammaires, pour répondre à des besoins très spécifiques. Ces modifications ne sont pas à la portée de tout le monde et il a été demandé une documentation et des informations détaillées sur ces sujets. Le panel a souligné que, sans Praspel, ils n'auraient même pas songé à ce genre de techniques. En effet, comparé aux dictionnaires de données ou aux données manuelles qu'ils utilisent quotidiennement, Praspel va plus loin et offre des services plus efficaces.

À propos de Praspel. Praspel étant un projet de recherche en cours d'élaboration, nous n'avons pas eu le temps d'écrire une documentation utilisateur. Pour que le panel puisse utiliser Praspel durant cette expérimentation, nous lui avons fourni les articles existants [Enderlin et al., 2011, 2012, 2013]. Le panel a reconnu qu'une documentation utilisateur aurait été plus appréciable mais que toutefois le langage était simple et intuitif. Il y a peu de constructions, la syntaxe est simple et le fait qu'il soit possible de composer des domaines réalistes est un réel avantage. Nous n'avons eu que très peu de questions sur le langage.

Une remarque pertinente a été faite par un ingénieur. Praspel convient très bien pour tester du code dit « métier », c'est à dire du code qui manipule des données et qui en calcule d'autres. C'est exactement ce que fait le robot UniTestor présenté dans la partie 7.3. En revanche, pour du code dit « technique », Praspel est plus limité. Un exemple de code technique serait un générateur de bouchons (*mock generator*). Un tel générateur utilise un nom de classe en entrée et va générer du code PHP qui, une fois exécuté, aura un comportement similaire à la classe bouchée *modulo* certaines méthodes ou attributs avec des comportements différents. Praspel pourrait valider que le code PHP produit est correct d'un point de vue syntaxique, mais spécifier qu'il est correct d'un point de vue sémantique est nettement plus difficile. Un autre exemple de code technique est un code créant des *threads*. Praspel est incapable de spécifier que des *threads* ont été créés ou fermés. Actuellement, le seul moyen de spécifier de telles postconditions se ferait par le biais d'un domaine réaliste sur mesure ou alors en utilisant la construction `\pred()`. Si cet effort est ponctuel, il est plus rapide d'écrire un test manuel. Cependant, un test manuel pourrait être limité et l'utilisation d'un test paramétré (c'est à dire annoté par un contrat, voir la partie 7.4.6) serait particulièrement adaptée à ce genre de cas de figure.

7.5 Performance et régression de PHP

Les générateurs de données de Praspel ont également été utilisés dans un cadre de test de performance et de non-régression. PHP a une extension appelée `ext/json` qui permet de manipuler le langage JSON au sein même de PHP (principalement de l'encodage et du décodage). Cette extension devait être remplacée, pour des raisons de compatibilité de licences [2] [3]. Une nouvelle extension, que nous appelerons `ext/jsond`, a dû être développée.

Pour comparer les performances entre ces deux extensions, il était nécessaire d'avoir un grand nombre de données JSON. Nous avons alors proposé d'utiliser la grammaire de JSON et le générateur de chaînes de caractères à partir de grammaire avec l'algorithme exhaustif pour générer des centaines de milliers de données de la plus petite jusqu'à une certaine taille.

Une fois toutes ces données générées, les auteurs de `ext/jsonp` ont pu comparer les performances en terme de vitesse et de mémoire des extensions. De même, ils ont pu comparer que toutes les données étaient manipulées de la même façon par les deux extensions. L'objectif premier n'était pas de détecter des erreurs dans les extensions mais de savoir si `ext/jsonp` n'introduisait pas de régression par rapport à `ext/json`.

2. Voir https://bugs.php.net/63520.
3. Voir https://wiki.php.net/rfc/free-json-parser.

Il nous a fallu approximativement 2 minutes pour écrire le test, qui est un test paramétré. Les auteurs de l'extension ont généré environ 1 million de données en 20 minutes. Ce nombre a été estimé suffisant pour tester les performances et la non-régression.
Cet usage inattendu montre l'intérêt des générateurs automatiques de données de tests dans d'autres types de tests.

7.6 Synthèse

Dans ce chapitre, nous avons présenté 3 expérimentations qualitatives autour de Praspel : sur un projet d'enseignement, sur 7 programmes du « monde réel » grâce à un panel d'ingénieurs bénévoles et sur une extension de PHP. Elles s'ajoutent aux expérimentations quantitatives réalisées dans nos précédents articles [Enderlin et al., 2011, 2012, 2013]. Ces expérimentations nous ont montré l'efficacité des contrats pour la génération automatique de tests unitaires, ainsi que l'efficacité de nos générateurs de données de test. Plusieurs erreurs ont été trouvées dans des programmes qui étaient déjà testés, et parfois déjà en production. Nous avons vu que Praspel et ses outils génèrent des suites de tests simples mais pertinentes couvrant les spécifications. Le temps économisé grâce à la génération automatique de tests unitaires est réinvesti dans l'écriture de tests manuels plus avancés. L'utilisation de générateurs de données de test permet d'augmenter la couverture des données et la confiance dans les suites de tests. Ces générateurs permettent également de réduire le nombre de tests dans des suites. Ces expérimentations nous ont également montré les limites de Praspel, notamment avec des codes dits « techniques », ce qui est une opportunité pour de futurs travaux.

Chapitre 8.
Conclusions et perspectives

Table des matières

Nos travaux s'achèvent avec ce chapitre qui rappelle l'ensemble de nos contributions que nous avons présenté au long de ces chapitres, dans la partie 8.1, et les confronte avec nos objectifs initiaux, dans la partie 8.2. Enfin, ce chapitre termine par énoncer plusieurs perspectives à nos travaux.

8.1 *Recapitulare*

Dans ce mémoire, nous avons présenté trois contributions.

La première contribution est Praspel, un nouveau langage de spécification pour PHP basé sur la programmation par contrat. Praspel spécifie les données avec des domaines réalistes : des structures permettant de valider et générer des données. Les domaines réalistes peuvent hériter entre eux et être emboîtés pour représenter des données plus complexes. La syntaxe et la sémantique de Praspel ont été définies. À partir d'un contrat écrit en Praspel, nous pouvons faire du *Contract-based Testing*, c'est à dire que nous exploitons le contrat pour générer automatiquement des tests unitaires. La précondition est utilisée pour générer des données de test et la postcondition est utilisée comme oracle.

La deuxième contribution concerne la génération de données de test. Pour les booléens, les entiers et les réels, une génération aléatoire uniforme est employée. Cette approche a rapidement montré ses limites pour les tableaux et les chaînes de caractères.
Pour les tableaux, nous avons commencé par mener une étude sur leurs propriétés les plus utilisées. Nous en avons sélectionné trois qui se détachaient réellement des autres. La syntaxe de Praspel a été étendue pour introduire ces propriétés au sein même du langage. Enfin, nous avons défini une sémantique ensembliste de ces propriétés et nous avons construit notre propre solveur de contraintes pour générer des tableaux à partir de ces propriétés.
Pour les chaînes de caractères, nous avons opté pour deux approches : à partir d'une grammaire ou à partir d'une expression régulière. Nous avons défini un nouveau langage de description de grammaire appelé PP, avec un compilateur de compilateurs LL($\star$). Ces derniers permettent de valider une donnée. Pour générer une donnée, nous avons proposé trois algorithmes de génération permettant de répondre à plusieurs besoins différents : aléatoire uniforme, exhaustif borné et basé sur la couverture de la grammaire. Ces algorithmes génèrent des séquences de lexèmes, qui sont concrétisées en chaînes de caractères à partir d'un algorithme de génération aléatoire isotropique. Ce même algorithme est utilisé pour la génération de chaînes de caractères à partir d'une expression régulière.
Enfin, la génération d'objet utilise une stratégie basée sur toutes les générations que nous venons de citer. Cette stratégie considère des références circulaires et permet de réutiliser des objets déjà générés.

La troisième contribution définit des critères de couverture sur les contrats. Ils nous fournissent des objectifs de test. Ces critères sont satisfaits par un ensemble de tests auxquels nous associons des chemins dans les contrats. Nous avons proposé de transformer les contrats en graphes, afin de définir plus facilement les critères. Au total, trois critères, plus des combinaisons, ont été définis : le Critère de Couverture de Clause, qui couvre la structure du contrat, le Critère de Couverture de Domaine,

qui couvre tous les domaines réalistes de toutes les variables, et le Critère de Couverture de Prédicat, qui couvre toutes les combinaisons possibles.

Ces contributions ont été implémentées en tant que bibliothèques dans le projet Hoa. La manipulation de Praspel à travers son modèle objet a été présentée : les différentes analyses, l'exportation et l'importation et le désassemblage. L'évaluation de Praspel, avec un *Runtime Assertion Checker*, a également été présentée. Une extension à atoum, un *framework* de tests unitaires, a été proposée pour compiler les suites de tests de Praspel en suites de tests écrits avec l'API d'atoum.

Nous avons choisi Hoa car l'auteur de ce mémoire est également l'auteur initial de cet ensemble de bibliothèques. Nous avons choisi atoum car c'est un *framework* très utilisé et intégré à l'industrie. Les deux projets ont des licences *open-source* et sont libres. Ils nous offrent ainsi une communauté : des utilisateurs et des contributeurs. Par leur biais, nous avons accès à des entreprises et des utilisateurs qui peuvent nous faire des retours précieux sur nos travaux.

Cette démarche s'est confirmée par la participation d'un panel d'ingénieurs de tests bénévoles pour l'une de nos expérimentations. Ces expérimentations ont répondu à plusieurs questions. Elles ont confirmé l'intérêt de la programmation par contrat dans les méthodologies de développement actuelles. La proximité entre les contrats et le code est un élément crucial : les contrats jouent le rôle de documentation, ils sont plus facilement maintenus et sont toujours écrits en même temps que le code. La génération automatique de tests unitaires permet de générer des tests dits basiques, ce qui laisse le temps aux ingénieurs de tests de se concentrer sur des tests plus avancés. Quand nous savons que leur budget pour la qualité logicielle est limité, c'est un avantage. Pour ces tests plus avancés, les ingénieurs utilisent nos algorithmes de génération de données seuls, qui sont paramétrables manuellement par un ensemble de contraintes défini par l'utilisateur. L'usage de ces algorithmes permet, en plus d'écrire des tests rapidement, de réduire les suites de tests tout en assurant la même qualité et en augmentant la confiance dans les tests restants. Enfin, les suites de tests finales sont plus facilement maintenables et compréhensibles. Ces expérimentations ont permis de détecter des erreurs dans des programmes déjà testés et parfois en production.

Toutes les contributions exceptées les critères de couverture des contrats ont été publiées dans des articles de conférence [Enderlin et al., 2011, 2012, 2013]. L'ensemble des contributions a été soumis au journal *Software and Systems Modeling*, abrégé SoSyM, pour une édition spéciale sur les méthodes formelles intégrées.

8.2 *Summa summarum*

Le langage de spécification que nous avons proposé répond à nos attentes au vu des résultats des expérimentations. Nous voulions que le langage soit simple. Le principe de la programmation par contrat, la syntaxe de Praspel et l'usage des domaines réalistes ont permis de répondre à cette attente. Nous voulions également que le langage soit pragmatique. Nous traitons tous les types de données que les développeurs manipulent au quotidien. Pour les chaînes de caractères et les tableaux, nous avons des stratégies spécifiques qui ne demandent pas d'efforts supplémentaires aux développeurs pour être utilisées correctement. Nous voulions un langage permettant d'assembler plusieurs méthodes du test. Dans Praspel, nous trouvons de la génération aléatoire, de la génération basée sur les solveurs et de la génération basée sur les grammaires. Enfin, pour « boucler la boucle » et rendre l'ensemble du processus automatique, nous avons défini des critères de couverture sur les contrats. Ces derniers nous offrent des objectifs de test. Nous sommes capables de générer des suites de tests satisfaisants plusieurs critères : la structure du contrat, la couverture des données ou encore des prédicats. Il est également possible de combiner ces critères. Toutes ces méthodes fonctionnent ensemble au sein d'un même langage par le truchement des domaines réalistes et leurs deux caractéristiques de prédicabilité et de générabilité. Ces derniers exposent des méthodes avancées du test aux développeurs de manière simple.
L'approche que nous avons présentée dans ce mémoire est satisfaisante. Toutefois, les expérimentations ont montré les limites de Praspel, notamment que le langage n'est pas capable de tout spécifier (comme du code technique manipulant par exemple des *threads*). Nous aimerions générer des tests encore plus pertinents. Il est probable que la solution se trouve dans d'autres méthodes de test ou dans la méthodologie du test paramétré, ce qui ouvre des perspectives intéressantes présentées dans la partie suivante.

8.3 Perspectives

Les perspectives qui se dégagent de ces travaux se classent en trois parties. La première concerne l'utilisation d'autres méthodes de test dans Praspel. La deuxième partie concerne des améliorations possibles sur le langage. Enfin, la dernière partie offre une réflexion sur l'avenir des travaux présentés dans ce mémoire.

8.3.1 Inclure de nouvelles méthodes de test

Grâce aux domaines réalistes et à la conception modulaire et extensible des outils de Praspel, il est possible d'assembler plusieurs méthodes de test. Les expérimen-

tations ont montré certaines limites du langage et des outils. Nous pensons qu'il est pertinent de s'intéresser aux méthodes suivantes.

Test aux limites

Le test aux limites permet de détecter des erreurs usuelles rapidement et efficacement. Nous avons dit que les domaines réalistes utilisaient un générateur numérique aléatoire et uniforme pour générer des données. Nous pensons qu'il est pertinent de réfléchir à une manière de spécifier les limites des domaines réalistes. Par exemple, avec l'intervalle $[-7;42]$, les limites extrêmes sont -7, -6, 41 et 42, et les limites intermédiaires sont -1, 0 et 1. En revanche, définir les limites d'un tableau, d'une chaîne de caractères ou d'un objet n'est pas aussi trivial et mérite plus d'attention. Une fois ces limites connues, nous pourrions utiliser un générateur numérique aux limites dans les domaines réalistes pour produire de nouvelles données aux limites.

Scénarios et propriétés temporelles

Dans le contexte du *Scenario-based Testing*, nous nous intéressons aux travaux de Castillos [2013]. Ces travaux s'inspirent des patrons de propriétés décrits par Dwyer, Avrunin, and Corbett [1999]. Ces patrons permettent de définir une propriété temporelle comme un motif qui doit être satisfait à l'intérieur d'une portée (une portion d'exécution du système). Ils permettent donc d'exprimer simplement un large panel de propriétés temporelles avec un ensemble restreint de constructions. Un langage [Castillos, Dadeau, Julliand, Kanso, and Taha, 2013] basé sur ces patrons a été créé. Ce langage comprend des variantes aux motifs et aux portées afin d'améliorer l'expressivité et corriger certaines incohérences des patrons initiaux. De nouveaux critères de couverture ont alors été définis et permettent de vérifier plusieurs propriétés temporelles.
La démarche de Castillos est similaire à la nôtre : proposer un langage et définir des critères de couverture pour obtenir des objectifs de test, mais ces objectifs sont différents. Cependant, puisque ce langage est très simple et permet d'exprimer un grand nombre de propriétés temporelles, nous pensons que ce langage pourrait être une extension à Praspel. Ainsi, en plus des attributs et des méthodes, les classes pourraient être annotées par ces propriétés. Nous les exploiterions pour générer de nouveaux préambules de test, des objets plus réalistes, des suites de tests fonctionnels ou alors pour vérifier des propriétés temporelles.

Grey-box avec le *Search-based Testing*

Dans les expérimentations, nous avons vu certaines limites de Praspel. La méthode du *Search-based Testing* [McMinn, 2004] utilise des métaheuristiques pour géné-

rer automatiquement des données de test à partir du code source. Cette méthode travaille en boîte blanche. Le SUT est en premier lieu exécuté avec des données aléatoires. Des fonctions de *fitness* (fonctions objectifs) placées sur les points de choix du SUT permettent d'évaluer la « distance » entre les données courantes et les données nécessaires pour activer ou désactiver ce point de choix. En utilisant la programmation génétique, ces données initiales sont mutées pour former une nouvelle population de données. Des algorithmes de sélection vont réduire cette population. Chaque donnée de cette population sera ensuite réutilisée comme donnée de test pour exécuter à nouveau le SUT. Les fonctions de *fitness* vont produire de nouveaux résultats qui serviront à guider l'évolution de la population de données. Par exemple, avec la condition `x == y`, la fonction de *fitness* sera $\text{abs}(\texttt{x} - \texttt{y})$. Quand cette fonction tend vers 0, alors la condition est activée.
Les suites de tests générées par Praspel sont basées sur les critères de couverture des contrats. Les données de test, spécifiées majoritairement avec les domaines réalistes, sont générées aléatoirement. Si le résultat des fonctions de *fitness* pouvait être remonté dans les domaines réalistes pour mieux guider les nouvelles générations de données, nous pensons que nous pourrions détecter plus d'erreurs, ou du moins, couvrir le code source plus efficacement et plus rapidement. En effet, les domaines réalistes ont plusieurs interfaces (comme `Interval`, `Finite`, `Nonconvex` etc.). Ces interfaces imposent l'implémentation de méthodes permettant de contraindre ou manipuler les données contenues dans le domaine réaliste. Il doit être possible de transposer les résultats des fonctions de *fitness* dans les domaines réalistes.
Cette approche, d'une part, permettrait d'exploiter le code source au lieu de rester en boîte noire (nous serions alors en boîte grise), et, d'autre part, ne compliquerait pas le langage en y ajoutant de nouvelles constructions, mais offrirait plus de services à l'utilisateur. Ce dernier n'aurait aucun effort à produire à part mettre à jour ses outils.

Solveurs arithmétiques sur les entiers et réels

Quand des entiers sont manipulés, il est courant de vouloir spécifier des contraintes arithmétiques comme $\texttt{i} < \texttt{j} + 1$. Actuellement, Praspel n'a aucun solveur arithmétique sur les entiers et les réels. Malgré la pertinence d'une telle fonctionnalité, nous n'avons pas eu le temps de la développer. Il serait alors nécessaire d'étudier le domaine, de voir comment nous pouvons y contribuer, et proposer une solution dans Praspel.

8.3.2 Améliorer le langage

Le langage Praspel n'est pas complet. Certaines données pourraient être encore mieux spécifiées. Nous sommes toujours dans cette démarche d'analyser ce que les

développeurs écrivent dans les contrats et d'améliorer le langage en conséquence. Nous avons relevé deux points importants, puis un dernier point que nous aurions aimé traiter mais le temps nous a manqué.

De nouveaux domaines réalistes

Durant l'expérimentation, les ingénieurs de test ont trouvé notre approche pertinente. Toutefois, la bibliothèque standard des domaines réalistes n'était pas suffisante. Les ingénieurs ont demandé la création de plus de domaines réalistes représentant des données métiers, comme des numéros de comptes bancaires, des numéros de téléphones etc. Les ingénieurs ont créé certains de ces domaines réalistes par eux mêmes mais ils n'ont pas eu le temps de tous les créer. Ce ne sont pas des perspectives scientifiques mais industrielles. Toutefois, elles participent à l'adoption de nos outils dans l'industrie.

Pureté des méthodes

À l'origine de Praspel, il existait la clause `@is`. Elle permet de qualifier la catégorie de la méthode. Le seul qualificatif était `pure`, pour contrôler la mutabilité de l'environnement. En effet, une méthode peut modifier ou pas son environnement lors de son exécution. Si elle ne le modifie pas, alors nous parlons d'une méthode pure, sinon nous parlons d'une méthode impure (par défaut). D'une manière plus générale, nous parlons de *frame condition* : quelle partie de l'environement est modifiable ou pas.
Nous pensons qu'il serait pertinent de détecter la pureté d'une méthode. De même, une méthode pure peut être utilisée dans un contrat car elle ne modifie par l'état du SUT. Ainsi, détecter la pureté d'une méthode peut permettre d'écrire des contrats plus élaborés.

8.3.3 Vers des outils industriels ?

Il nous a semblé pertinent de développer nos travaux de thèse dans des projets *open-source* et libres avec une communauté, des utilisateurs et des contributeurs. Cela nous a été bénéfique mais, comme l'ont fait remarquer les ingénieurs de test lors de notre expérimentation, il est nécessaire d'écrire beaucoup de documentation pour permettre une utilisation plus importante de nos travaux au sein de l'industrie.

Conférences industrielles

Toutefois, les communautés de Hoa et d'atoum ont déjà fait plusieurs conférences industrielles pour la promotion du test, de la qualité logicielle et de Praspel. Nous pouvons citer :

– 5 et 6 juin 2012, au ForumPHP à Paris, Ivan Enderlin (auteur de ce mémoire et créateur de Hoa) et Frédéric Hardy (créateur d'atoum) ont présenté une conférence en duo sur l'« Anatomie du test » ;
– 28 octobre 2013, soirée AFUP à Lyon, Ivan Enderlin et Julien Bianchi (contributeur de Hoa et d'atoum) ont présenté une conférence en duo sur Praspel et atoum ;
– 23 et 24 juin 2014, au PHPTour à Lyon, Ivan Enderlin et Julien Bianchi ont présenté une conférence sur Praspel.

Praspel a également été cité dans d'autres conférences sur la qualité logicielle et le test.

Transfert technologique

Il est difficile de mesurer l'impact de nos contributions et l'usage de nos algorithmes dans l'industrie. Les quelques chiffres que nous pouvons obtenir sont le nombre d'installations quotidiennes des bibliothèques que nous avons créées au sein de Hoa ou auxquelles nous avons contribué.
La bibliothèque `Hoa\Compiler` contient notre compilateur de compilateurs LL($\star$), le langage de description de grammaire PP et nos algorithmes de génération de données à partir de grammaires. Elle a été installée plus de 9000 fois depuis sa création, soit 15 installations en moyenne par jour. La bibliothèque `Hoa\Regex` contient la grammaire des PCRE au format PP et l'algorithme de génération isotropique. Elle est installée en moyenne 0.5 fois par jour. Nous avons contribué à la bibliothèque `Hoa\Math` pour des fonctions combinatoires. Des contributeurs ont utilisé notre compilateur de compilateurs pour évaluer des formules arithmétiques à la volée contenues dans des chaînes de caractères. Cette bibliothèque est installée en moyenne 14 fois par jour. La bibliothèque `Hoa\Praspel` est installée en moyenne 1 fois par jour. D'autres outils utilisant nos contributions sur les grammaires sont installés entre 3 et 6 fois par jour en moyenne et comptent des milliers d'utilisateurs. Comme nous avons créé une extension qui fait le pont entre atoum et Praspel, et que atoum est installé plus de 100 fois par jour, nous espérons que Praspel va rapidement se propager et être utilisé. Durant ces travaux de thèse, nous n'avons pas alloué de temps spécifiquement pour la promotion de Praspel auprès de l'industrie. Les implémentations des outils sont terminées depuis peu et sont mis à la disposition de la communauté PHP depuis mars 2014. Au vu des résultats de l'expérimentation, nous espérons que les communautés de Hoa et d'atoum sauront promouvoir Praspel. Nous allons les rejoindre et participer à cet effort au maximum car nous pensons que Praspel et le lien qui s'est créé entre nous et ces communautés sont bénéfiques pour la recherche.

Chapitre

Annexes

Table des matières

Ce chapitre contient des annexes à ce mémoire. Il a été jugé nécessaire de les ajouter pour offrir un complément d'informations. En effet, tous les détails qu'ils exposent ne trouvent pas leur place dans le mémoire mais ils offrent des éléments qui peuvent être utiles à la bonne compréhension des contributions.

1 Grammaire du langage Praspel en PP

Dans le chapitre 3, nous avons présenté la grammaire de Praspel en forme normale, afin de mieux raisonner dessus. Cette partie présente la grammaire de Praspel telle qu'elle est implémentée dans la bibliothèque `Hoa\Praspel`, présentée dans les parties 6.1 et 6.2. Cette grammaire est écrite en langage PP, présenté dans la partie 4.3.1. La dernière version de la grammaire est disponible à l'adresse : http://central.hoa-project.net/Resource/Library/Praspel/Grammar.pp.

```
//
// Hoa
//
//
// @license
//
// New BSD License
//
// Copyright © 2007-2014, Ivan Enderlin. All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are met:
//     * Redistributions of source code must retain the above copyright
//       notice, this list of conditions and the following disclaimer.
//     * Redistributions in binary form must reproduce the above copyright
//       notice, this list of conditions and the following disclaimer in the
//       documentation and/or other materials provided with the distribution.
//     * Neither the name of the Hoa nor the names of its contributors may be
//       used to endorse or promote products derived from this software without
//       specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
// ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS AND CONTRIBUTORS BE
// LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
// CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
// SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
// CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
// POSSIBILITY OF SUCH DAMAGE.
//
// Grammar \Hoa\Praspel\Grammar.
//
// Provide grammar for Praspel.
//
// @author     Ivan Enderlin <ivan.enderlin@hoa-project.net>
// @copyright  Copyright © 2007-2014 Ivan Enderlin.
// @license    New BSD License
//


%skip   s              \s
```

```
%skip   block_comment   /\*(.|\n)*?\*/
%skip   inline_comment  //[^\n]*

// Clauses.
%token  at_is           @is
%token  at_requires     @requires
%token  at_ensures      @ensures
%token  at_throwable    @throwable
%token  at_invariant    @invariant
%token  at_behavior     @behavior
%token  at_default      @default
%token  at_description  @description

// Constructions.
%token  old             \\old
%token  result          \\result
%token  pred            \\pred
%token  nothing         \\nothing

// Symbols.
%token  parenthesis_    \(
%token _parenthesis     \)
%token  brace_          \{
%token _brace           \}
%token  bracket_        \[
%token _bracket         \]
%token  comma           ,
%token  backslash       \\
%token  arrow           \->
%token  resolution      ::
%token  colon           :
%token  semicolon       ;
%token  range           \.\.
%token  count           #|count
%token  heredoc_        <<<                         -> hd
%token  hd:quote        '
%token  hd:identifier   [A-Z]+
%token  hd:content      ((\h[^\n]+)?\n)+
%token  hd:_heredoc     ;                           -> default

// Keywords.
%token  domainof        domainof
%token  from            from
%token  to              to
%token  this            this
%token  self            self
%token  static          static
%token  parent          parent
%token  and             and
%token  or              or
%token  with            with
%token  pure            pure
%token  default         ...|default
%token  contains        contains
%token  is              is
%token  let             let
```

```
// Constants.
%token  null            null|void
%token  true            true
%token  false           false
%token  binary          [\+\-]?0b[01]+
%token  octal           [\+\-]?0[0-7]+
%token  hexa            [\+\-]?0[xX][0-9a-fA-F]+
%token  decimal         [\+\-]?(0|[1-9]\d*)(\.\d+)?([eE][\+\-]?\d+)?
%token  quote_          '                           -> string
%token  string:escaped  \\(['nrtvef\\b]|[0-7]{1,3}|[xX][0-9a-fA-F]{1,2})
%token  string:accepted \\
%token  string:string   [^'\\]+
%token  string:concat   '\s*\.\s*'
%token  string:_quote   '                           -> default
%token  regex           /.*?(?<!\\)/[imsxADSUXJu]*
%token  identifier      [a-zA-Z_\x7f-\xff][a-zA-Z0-9_\x7f-\xff]*

#specification:
    method()*

method:
    (
        is()
      | requires()
      | ensures()
      | throwable()
      | invariant()
    )
    ::semicolon::+
  | ( behavior() ::semicolon::* )+ default()? ::semicolon::*
  | description() ::semicolon::*

#is:
    ::at_is:: <pure>

#requires:
    ::at_requires:: expression()?

#ensures:
    ::at_ensures:: expression()?

#throwable:
    ::at_throwable:: exceptional_expression()?

#invariant:
    ::at_invariant:: expression()?

behavior:
    ::at_behavior:: behavior_content()
    ( ::and:: behavior_content() )*

behavior_content:
    <identifier> ::brace_::
    (
        (
```

```
            requires()
          | ensures()
          | throwable()
        )
        ::semicolon::+
      | ( behavior() ::semicolon::* )+ default()? ::semicolon::*
      | description() ::semicolon::*
    )+
    ::_brace:: #behavior

#default:
    ::at_default:: ::brace_::
    (
        (
            ensures()
          | throwable()
        )
        ::semicolon::+
      | description() ::semicolon::*
    )+
    ::_brace::

#description:
    ::at_description:: string()

expression:
              ( declaration() | constraint() | domainof() | predicate() )
    ( ::and:: ( declaration() | constraint() | domainof() | predicate() ) )*
  | <nothing>

exceptional_expression:
    exception() ( ::or:: exception() )*

exception:
    exception_identifier() ( ::or:: exception_identifier() )*
    ( ::with:: exception_with() )?

#exception_identifier:
    classname() <identifier>

#exception_with:
    expression()

#declaration:
    ( ::let:: #local_declaration )?
    extended_identifier() ::colon:: disjunction()

constraint:
    qualification() | contains()

#qualification:
    identifier() ::is:: <identifier> ( ::comma:: <identifier> )*

#contains:
    extended_identifier() ::contains:: disjunction_of_constants()
```

```
#domainof:
    identifier() ::domainof:: identifier()

#predicate:
    ::pred:: ::parenthesis_:: string()? ::_parenthesis::

disjunction:
    ( constant() | realdom() | extended_identifier() )
    ( ::or:: disjunction() #disjunction )*

disjunction_of_constants:
    constant() ( ::or:: disjunction_of_constants() #disjunction )*

#realdom:
    <identifier> ::parenthesis_::
    ( argument() ( ::comma:: argument() )* )?
    ::_parenthesis::

argument:
    <default> | realdom() | constant() | array() | extended_identifier()

constant:
    scalar() | array()

scalar:
    <null> | boolean() | number() | string() | regex() | range()

boolean:
    <true> | <false>

number:
    <binary> | <octal> | <hexa> | <decimal>

string:
    quoted_string() | herestring()

quoted_string:
    ::quote_::
    ( <escaped> | <accepted> | <string> | ::concat:: #concatenation )
    ( ( <escaped> | <accepted> | <string> | ::concat:: ) #concatenation )*
    ::_quote::

#regex:
    <regex>

#range:
    number() ::range:: number()
  | number() ::range:: #left_range
  | ::range:: number() #right_range

#array:
    ::bracket_::
    ( pair() ( ::comma:: pair() )* )?
    ::_bracket::

pair:
```

```
    ( ::from::? disjunction() ::to:: disjunction() #pair )
  | ::to::? disjunction()

extended_identifier:
    ( ::count:: #count )? arrayaccess()

arrayaccess:
    identifier()
    (
        ::bracket_:: scalar() ::_bracket:: #arrayaccessbykey
      | ::brace_::   scalar() ::_brace::   #arrayaccessbyvalue
    )?

identifier:
    ( <identifier> | <this> )
    ( ::arrow:: <identifier> ( ::arrow:: <identifier> )* )? #dynamic_resolution
  | (
        ::self::
      | ::static::
      | ::parent::
    )
    ( ::resolution:: <identifier> )+ #static_resolution
  | ::old:: ::parenthesis_:: extended_identifier() ::_parenthesis:: #old
  | <result>

#classname:
    ::backslash::? <identifier> ( ::backslash:: <identifier> )*

herestring:
    ::heredoc_::
    (
        ::quote:: <identifier[0]> ::quote:: #nowdoc
      | <identifier[0]>                     #heredoc
    )
    <content>?
    ::identifier[0]:: ::_heredoc::
```

2 Grammaire du langage PP en langage PP

Dans la partie 4.3.1, nous présentons le langage PP à travers ses constructions. Cette partie présente la grammaire du langage PP au format PP. La dernière version de la grammaire est disponible à l'adresse : http://central.hoa-project.net/Resource/Library/Compiler/Llk/Llk.pp.

```
//
// Hoa
//
//
// @license
//
// New BSD License
//
// Copyright © 2007-2014, Ivan Enderlin. All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are met:
//     * Redistributions of source code must retain the above copyright
//       notice, this list of conditions and the following disclaimer.
//     * Redistributions in binary form must reproduce the above copyright
//       notice, this list of conditions and the following disclaimer in the
//       documentation and/or other materials provided with the distribution.
//     * Neither the name of the Hoa nor the names of its contributors may be
//       used to endorse or promote products derived from this software without
//       specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
// ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS AND CONTRIBUTORS BE
// LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
// CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
// SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
// CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
// POSSIBILITY OF SUCH DAMAGE.
//
// Grammar \Hoa\Compiler\Llk.
//
// Provide grammar for the LL(k) parser.
//
// @author     Ivan Enderlin <ivan.enderlin@hoa-project.net>
// @copyright  Copyright © 2007-2014 Ivan Enderlin.
// @license    New BSD License
//


%skip   space          \s

%token  or             \|
%token  zero_or_one    \?
```

```
%token  one_or_more    \+
%token  zero_or_more   \*
%token  n_to_m         \[0-9]+,[0-9]+\
%token  zero_to_m      \,[0-9]+\
%token  n_or_more      \[0-9]+,\
%token  exactly_n      \[0-9]+\

%token  token          [a-zA-Z_][a-zA-Z0-9_]*

%token  skipped        ::
%token  kept_          <
%token _kept           >
%token  named          \(\)
%token  node           #[a-zA-Z][a-zA-Z0-9]+

%token  capturing_     \(
%token _capturing      \)
%token  unification_   \[
%token  unification    [0-9]+
%token _unification    \]

#rule:
    choice()

choice:
    concatenation() ( ::or:: concatenation() #choice )*

concatenation:
    repetition() ( repetition() #concatenation )*

repetition:
    simple() ( quantifier() #repetition )? <node>?

simple:
    ::capturing_:: choice() ::_capturing::
  | ::skipped:: <token> ( ::unification_:: <unification> ::_unification:: )?
    ::skipped:: #skipped
  | ::kept_:: <token> ( ::unification_:: <unification> ::_unification:: )?
    ::_kept:: #kept
  | <token> ::named::
    ( ::unification_:: <unification> ::_unification:: )? #named

quantifier:
    <zero_or_one>
  | <one_or_more>
  | <zero_or_more>
  | <n_to_m>
  | <n_or_more>
  | <exactly_n>
```

3 Grammaire des PCRE en langage PP

Dans la partie 4.3.1, nous précisons que les lexèmes des grammaires sont exprimés avec des expressions régulières. Ces dernières utilisent le vocabulaire des PCRE [Hazel, 1997]. Dans partie 4.3.3, lorsque nous expliquons comment une séquence de lexèmes est concrétisée vers une chaîne de caractère, nous expliquons que nous utilisons la grammaire des PCRE pour obtenir un AST (*Abstract Syntax Tree*) après analyse de chaque lexème. Cet AST est visité pour appliquer un algorithme de génération aléatoire isotropique. Cette partie présente la grammaire des PCRE au format PP. La dernière version de la grammaire est disponible à l'adresse : http://central.hoa-project.net/Resource/Library/Regex/Grammar.pp.

```
//
// Hoa
//
//
// @license
//
// New BSD License
//
// Copyright © 2007-2014, Ivan Enderlin. All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are met:
//     * Redistributions of source code must retain the above copyright
//       notice, this list of conditions and the following disclaimer.
//     * Redistributions in binary form must reproduce the above copyright
//       notice, this list of conditions and the following disclaimer in the
//       documentation and/or other materials provided with the distribution.
//     * Neither the name of the Hoa nor the names of its contributors may be
//       used to endorse or promote products derived from this software without
//       specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
// ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS AND CONTRIBUTORS BE
// LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
// CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
// SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
// CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
// POSSIBILITY OF SUCH DAMAGE.
//
// Grammar \Hoa\Regex\Grammar.
//
// Provide grammar of PCRE (Perl Compatible Regular Expression)for the LL(k)
// parser. More informations at http://pcre.org/pcre.txt, sections pcrepattern &
// pcresyntax.
//
// @author     Ivan Enderlin <ivan.enderlin@hoa-project.net>
```

```
// @copyright  Copyright © 2007-2014 Ivan Enderlin.
// @license    New BSD License
//


// Skip.
%skip   nl                      \n

// 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]{3}|x[0-9a-zA-Z]{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>
```

Chapitre
Bibliographie

J.-R. Abrial. *The B-book : assigning programs to meanings*. Cambridge University Press, 2005.

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. ISBN 3-540-20527-6.

S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. M. Paradkar, and M. D. Ernst. Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. *IEEE Trans. Software Eng.*, 36(4) :474–494, 2010.

D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner : Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *IEEE Symposium on Security and Privacy*, pages 387–401. IEEE Computer Society, 2008.

S. Bardin and A. Gotlieb. fdcc : A Combined Approach for Solving Constraints over Finite Domains and Arrays. In N. Beldiceanu, N. Jussien, and E. Pinson, editors, *CPAIOR*, volume 7298 of *Lecture Notes in Computer Science*, pages 17–33. Springer, 2012. ISBN 978-3-642-29827-1.

M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System : An Overview. In *Proceedings of the International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69, Marseille, France, March 2004. Springer.

M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie : A Modular Reusable Verifier for Object-Oriented Programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005. ISBN 3-540-36749-7.

P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL : ANSI/ISO C Specification Language Version 1.7*, 2013. http://frama-c.com/download/acsl-implementation-Fluorine-20130601.pdf.

K. Beck and E. Gamma. JUnit, 1997. URL http://junit.org/.

M. Benedikt, J. Freire, and P. Godefroid. VeriWeb : Automatically testing dynamic web sites. In *In Proceedings of 11th International World Wide Web Conference (WWW'2002)*. Citeseer, 2002.

S. Bergmann. PHPUnit, 2001. URL http://phpunit.de/.

G. Bernot, M.-C. Gaudel, and B. Marre. A Formal Approach to Software Testing. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *AMAST*, Workshops in Computing, pages 243–253. Springer, 1991. ISBN 3-540-19797-4.

B. Botella, M. Delahaye, S. H. T. Ha, N. Kosmatov, P. Mouy, M. Roger, and N. Williams. Automating Structural Testing of C Programs : Experience with PathCrawler. In D. Dranidis, S. P. Masticola, and P. A. Strooper, editors, *AST*, pages 70–78. IEEE, 2009.

F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. JML-Testing-Tools : A Symbolic Animator for JML Specifications Using CLP. In N. Halbwachs and L. D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 551–556. Springer, 2005. ISBN 3-540-25333-5.

F. Bouquet, F. Dadeau, and B. Legeard. Automated Boundary Test Generation from JML Specifications. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 428–443. Springer, 2006. ISBN 3-540-37215-6.

C. Boyapati, S. Khurshid, and D. Marinov. Korat : automated testing based on Java predicates. In *ISSTA*, pages 123–133, 2002.

C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE : automatically generating inputs of death. In A. Juels, R. N. Wright, and S. D. C. di Vimercati, editors, *ACM Conference on Computer and Communications Security*, pages 322–335. ACM, 2006.

C. Cadar, D. Dunbar, and D. R. Engler. KLEE : Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In R. Draves and R. van Renesse, editors, *OSDI*, pages 209–224. USENIX Association, 2008a. ISBN 978-1-931971-65-2.

C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE : Automatically Generating Inputs of Death. *ACM Trans. Inf. Syst. Secur.*, 12(2), 2008b.

K. C. Castillos. *Génération automatique de scénarios de tests à partir de propriétés temporelles et de modèles comportementaux*. PhD thesis, Université de Franche-Comté, 2013.

K. C. Castillos, F. Dadeau, J. Julliand, B. Kanso, and S. Taha. A Compositional Automata-Based Semantics for Property Patterns. In E. B. Johnsen and L. Petre, editors, *IFM*, volume 7940 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2013. ISBN 978-3-642-38612-1, 978-3-642-38613-8.

CEA and INRIA. Frama-C Software Analyzers, 2008. URL http://frama-c.com/.

O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand. Program slicing enhances a verification technique combining static and dynamic analysis. In S. Ossowski and P. Lecca, editors, *SAC*, pages 1284–1291. ACM, 2012. ISBN 978-1-4503-0857-1.

Y. Cheon and G. T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas*, pages 322–328. CSREA Press, 2002a.

Y. Cheon and G. T. Leavens. A Simple and Practical Approach to Unit Testing : The JML and JUnit Way. In B. Magnusson, editor, *ECOOP*, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255. Springer, 2002b. ISBN 3-540-43759-2.

N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3) :113–124, 1956.

I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. ARTOO : Adaptive Random Testing for Object-Oriented Software. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *ICSE*, pages 71–80. ACM, 2008. ISBN 978-1-60558-079-1.

D. Coppit and J. Lian. yagg : An Easy-To-Use Generator for Structured Test Inputs. In D. F. Redmiles, T. Ellman, and A. Zisman, editors, *ASE*, pages 356–359. ACM, 2005.

P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C - A Software Analysis Perspective. In G. Eleftherakis, M. Hinchey, and M. Holcombe, editors, *SEFM*, volume 7504 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2012. ISBN 978-3-642-33825-0.

L. De Moura and N. Bjørner. Z3 : An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-State Verification. In B. W. Boehm, D. Garlan, and J. Kramer, editors, *ICSE*, pages 411–420. ACM, 1999. ISBN 1-58113-074-0.

ECMA. Standard ECMA-404, The JSON Data Interchange Format, Oct. 2013. http://ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf, http://json.org.

I. Enderlin, F. Dadeau, A. Giorgetti, and A. Ben Othman. Praspel : A Specification Language for Contract-Based Testing in PHP. In B. Wolff and F. Zaïdi, editors, *ICTSS*, volume 7019 of *Lecture Notes in Computer Science*, pages 64–79. Springer, 2011. ISBN 978-3-642-24579-4.

I. Enderlin, F. Dadeau, A. Giorgetti, and F. Bouquet. Grammar-Based Testing Using Realistic Domains in PHP. In G. Antoniol, A. Bertolino, and Y. Labiche, editors, *ICST*, pages 509–518. IEEE, 2012. ISBN 978-1-4577-1906-6.

I. Enderlin, A. Giorgetti, and F. Bouquet. A Constraint Solver for PHP Arrays. In *ICST Workshops, CSTVA*, pages 218–223. IEEE, 2013. ISBN 978-1-4799-1324-4.

J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In *CAV*, pages 173–177, 2007.

R. B. Findler and M. Felleisen. Contracts for Higher-Order Functions. In M. Wand and S. L. P. Jones, editors, *ICFP*, pages 48–59. ACM, 2002. ISBN 1-58113-487-8.

S. Fischer, O. Kiselyov, and C. chieh Shan. Purely Functional Lazy Non-Deterministic Programming. In G. Hutton and A. P. Tolmach, editors, *ICFP*, pages 11–22. ACM, 2009. ISBN 978-1-60558-332-7.

P. Flajolet, P. Zimmermann, and B. V. Cutsem. A Calculus for the Random Generation of Labelled Combinatorial Structures. *Theor. Comput. Sci.*, 132(2) :1–35, 1994.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, January 1995. ISBN 0201633612.

V. Ganesh and D. L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *CAV*, pages 519–531, 2007.

A. Gargantini, M. Guarnieri, and E. Magri. Extending Coverage Criteria by Evaluating Their Robustness to Code Structure Changes. In B. Nielsen and C. Weise, editors, *ICTSS*, volume 7641 of *Lecture Notes in Computer Science*, pages 168–183. Springer, 2012. ISBN 978-3-642-34690-3.

M.-C. Gaudel. Checking Models, Proving Programs, and Testing Systems. In M. Gogolla and B. Wolff, editors, *TAP*, volume 6706 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2011. ISBN 978-3-642-21767-8.

M. M. Geller. Test Data as an Aid in Proving Program Correctness. In S. L. Graham, R. M. Graham, M. A. Harrison, W. I. Grosky, and J. D. Ullman, editors, *POPL*, pages 209–218. ACM Press, 1976.

I. P. Gent, C. Jefferson, and I. Miguel. MINION : A Fast Scalable Constraint Solver. In G. Brewka, S. Coradeschi, A. Perini, and P. Traverso, editors, *ECAI*, volume 141 of *Frontiers in Artificial Intelligence and Applications*, pages 98–102. IOS Press, 2006. ISBN 1-58603-642-4.

M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in UDITA. In J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, editors, *ICSE (1)*, pages 225–234. ACM, 2010. ISBN 978-1-60558-719-6.

P. Godefroid, N. Klarlund, and K. Sen. DART : Directed Automated Random Testing. In V. Sarkar and M. W. Hall, editors, *PLDI*, pages 213–223. ACM, 2005. ISBN 1-59593-056-6.

P. Godefroid, M. Y. Levin, and D. A. Molnar. SAGE : Whitebox Fuzzing for Security Testing. *ACM Queue*, 10(1) :20, 2012.

J. Gosling and B. Joy. Java, 1990. URL https://java.net/.

A. Gotlieb. Euclide : A Constraint-Based Testing Framework for Critical C Programs. In *ICST*, pages 151–160. IEEE Computer Society, 2009. ISBN 978-0-7695-3601-9.

F. Hardy. atoum, 2010. URL http://atoum.org/.

D. Harel. Statecharts : A Visual Formalism for Complex Systems. *Sci. Comput. Program.*, 8(3) :231–274, 1987.

J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. J. Parkinson. Behavioral Interface Specification Languages. *ACM Comput. Surv.*, 44(3) :16, 2012.

D. Hauzar and J. Kofron. On security analysis of php web applications. In X. Bai, F. Belli, E. Bertino, C. K. Chang, A. Elçi, C. C. Seceleanu, H. Xie, and M. Zulkernine, editors, *COMPSAC Workshops*, pages 577–582. IEEE Computer Society, 2012. ISBN 978-1-4673-2714-5.

P. Hazel. PCRE – Perl Compatible Regular Expressions. Technical report, Tech. report, University Computing Service, 1997. URL http://pcre.org/.

P. Heidegger and P. Thiemann. JSConTest : Contract-Driven Testing and Path Effect Inference for JavaScript. *Journal of Object Technology*, 11(1) :1–29, 2012.

Hoa Project Foundation. Hoa, a set of PHP libraries, 2007. URL http://hoa-project.net/.

D. Hoffman, D. Ly-Gagnon, P. A. Strooper, and H.-Y. Wang. Grammar-Based Test Generation with YouGen. *Softw., Pract. Exper.*, 41(4) :427–447, 2011.

W. E. Howden. A Functional Approach to Program Testing and Analysis. *IEEE Trans. Software Eng.*, 12(10) :997–1005, 1986.

D. Jackson. Alloy : a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2) :256–290, 2002.

D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa : the Alloy constraint analyzer. In C. Ghezzi, M. Jazayeri, and A. L. Wolf, editors, *ICSE*, pages 730–733. ACM, 2000. ISBN 1-58113-206-9.

P. Jalote. Specification and Testing of Abstract Data Types. *Comput. Lang.*, 17(1) : 75–82, 1992.

S. C. Johnson. *YACC : Yet Another Compiler-Compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.

A. Kiezun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL Injection and cross-site scripting attacks. In *ICSE*, pages 199–209. IEEE, 2009. ISBN 978-1-4244-3452-7.

B. Korel and A. M. Al-Yami. Automated Regression Test Generation. In *ISSTA*, pages 143–152, 1998.

A. Kumar. CUnit, 2001. URL http://cunit.sourceforge.net/.

D. Kühner. Regular expressions visualization. Bachelor Report TA204, Haute École Arc - Neuchâtel, Jan. 2014. URL https://github.com/davidkuhner/TryVisualization.

R. Lämmel. Grammar Testing. In H. Hußmann, editor, *FASE*, volume 2029 of *Lecture Notes in Computer Science*, pages 201–216. Springer, 2001. ISBN 3-540-41863-6.

R. Lämmel and W. Schulte. Controllable Combinatorial Coverage in Grammar-Based Testing. In M. Ü. Uyar, A. Y. Duale, and M. A. Fecko, editors, *TestCom*, volume 3964 of *Lecture Notes in Computer Science*, pages 19–38. Springer, 2006. ISBN 3-540-34184-6.

G. T. Leavens, A. L. Baker, and C. Ruby. JML : A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from z and b. In L.-H. Eriksson and P. A. Lindsay, editors, *FME*, volume 2391 of *Lecture Notes in Computer Science*, pages 21–40. Springer, 2002. ISBN 3-540-43928-5.

M. E. Lesk and E. Schmidt. *Lex : A lexical analyzer generator*. Bell Laboratories Murray Hill, NJ, 1975.

B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-12112-3.

B. Liskov and S. N. Zilles. Programming with Abstract Data Types. *SIGPLAN Notices*, 9(4) :50–59, 1974.

J.-F. Lépine, D. Seguy, and F. Jean-Marc. Industrialisation PHP, livre blanc, 2009. URL http://alterway.fr/sites/default/files/publications/livre_blanc_industrialisation_php.pdf.

A. K. Mackworth. Consistency in Networks of Relations. *Artif. Intell.*, 8(1) :99–118, 1977.

D. Marinov and S. Khurshid. TestEra : A Novel Framework for Automated Testing of Java Programs. In *ASE*, pages 22–. IEEE Computer Society, 2001. ISBN 0-7695-1426-X.

D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard. An evaluation of exhaustive testing for data structures. Technical report, Technical Report MIT-LCS-TR-921, MIT CSAIL, Cambridge, MA, 2003.

M. Matsumoto and T. Nishimura. Mersenne Twister : A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Trans. Model. Comput. Simul.*, 8(1) :3–30, 1998.

P. M. Maurer. Generating test data with enhanced context-free grammars. *Software, IEEE*, 7(4) :50–55, 1990.

S. McAllister, E. Kirda, and C. Kruegel. Leveraging User Interactions for In-Depth Testing of Web Applications. In R. Lippmann, E. Kirda, and A. Trachtenberg, editors, *RAID*, volume 5230 of *Lecture Notes in Computer Science*, pages 191–210. Springer, 2008. ISBN 978-3-540-87402-7.

W. M. McKeeman. Differential Testing for Software. *Digital Technical Journal*, 10(1) : 100–107, 1998.

P. McMinn. Search-based Software Test Data Generation : A Survey. *Softw. Test., Verif. Reliab.*, 14(2) :105–156, 2004.

B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10) :40–51, 1992.

B. Meyer, J.-M. Nerson, and M. Matsuo. EIFFEL : Object-Oriented Design for Software Engineering. In H. K. Nichols and D. Simpson, editors, *ESEC*, volume 289 of *Lecture Notes in Computer Science*, pages 221–229. Springer, 1987. ISBN 3-540-18712-X.

Microsoft. C#, 2001. URL http://ecma-international.org/publications/standards/Ecma-334.htm.

J. C. Miller and C. J. Maloney. Systematic mistake analysis of digital computer programs. *Commun. ACM*, 6(2) :58–63, 1963.

G. J. Myers. *The art of software testing (2. ed.)*. Wiley, 2004. ISBN 978-0-471-46912-4.

T. Noll and B. Schlich. Delayed Nondeterminism in Model Checking Embedded Systems Assembly Code. In K. Yorav, editor, *Haifa Verification Conference*, volume 4899 of *Lecture Notes in Computer Science*, pages 185–201. Springer, 2007. ISBN 978-3-540-77964-3.

J. Offutt, P. Ammann, and L. L. Liu. Mutation Testing implements Grammar-Based Testing. In *Proceedings of the Second Workshop on Mutation Analysis*, MUTATION '06, pages 12–, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2897-X. doi : http://dx.doi.org/10.1109/MUTATION.2006.11. URL http://dx.doi.org/10.1109/MUTATION.2006.11.

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, *QoSA/SOQUA*, volume 3712 of *Lecture Notes in Computer Science*, pages 242–256. Springer, 2005. ISBN 3-540-29033-8.

PHP Documentor. phpDocumentor. URL http://phpdoc.org/.

A. D. Raghavan and G. T. Leavens. Desugaring JML method specifications. *Department of Computer Science, Iowa State University TR# 00-03e*, 2005.

D. Ritchie. C, 1972. URL http://open-std.org/JTC1/SC22/WG14/www/standards.

S. Sankar and R. Hayes. ADL - An Interface Definition Language for Specifying and Testing Software. In J. M. Wing and R. L. Wexelblat, editors, *Workshop on Interface Definition Languages*, pages 13–21. ACM Press, 1994.

S. Sawyer. Isotropic random walks in a tree. *Zeitschrift für Wahrscheinlichkeitstheorie und Verwandte Gebiete*, 42(4) :279–292, 1978. ISSN 0044-3719. doi : 10.1007/BF00533464. URL http://dx.doi.org/10.1007/BF00533464.

K. Sen, D. Marinov, and G. Agha. CUTE : a Concolic Unit Testing Engine for C. In M. Wermelinger and H. Gall, editors, *ESEC/SIGSOFT FSE*, pages 263–272. ACM, 2005. ISBN 1-59593-014-0.

J. G. Siek and W. Taha. Gradual typing for objects. In E. Ernst, editor, *ECOOP*, volume 4609 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2007. ISBN 978-3-540-73588-5.

E. G. Sirer and B. N. Bershad. Using Production Grammars in Software Testing. In T. Ball, editor, *DSL*, pages 1–13. ACM, 1999. ISBN 1-58113-255-7.

J. M. Spivey. An introduction to Z and formal specifications. *Software Engineering Journal*, 4(1) :40–50, 1989.

K. J. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software assurance by bounded exhaustive testing. In G. S. Avrunin and G. Rothermel, editors, *ISSTA*, pages 133–142. ACM, 2004. ISBN 1-58113-820-2.

The PHP Group. PHP, 2001. URL http://php.net/.

N. Tillmann and J. de Halleux. Pex-White Box Test Generation for .NET. In B. Beckert and R. Hähnle, editors, *TAP*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008. ISBN 978-3-540-79123-2.

E. P. K. Tsang. *Foundations of constraint satisfaction*. Computation in cognitive science. Academic Press, 1993. ISBN 978-0-12-701610-8.

M. Utting and B. Legeard. *Practical Model-Based Testing - A Tools Approach*. Morgan Kaufmann, 2007. ISBN 978-0-12-372501-1.

W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model Checking Programs. *Autom. Softw. Eng.*, 10(2) :203–232, 2003.

S. Viswanadha and S. Sankar. JavaCC, 1996. URL https://javacc.java.net/.

W3C. Extensible Markup Language (XML) 1.1 (Second Edition), 2006. URL http://www.w3.org/TR/xml11/.

W3Techs. Usage of server-side programming languages for websites, 2014. URL http://w3techs.com/technologies/overview/programming_language/all.

G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In J. Ferrante and K. S. McKinley, editors, *PLDI*, pages 32–41. ACM, 2007. ISBN 978-1-59593-633-2.

G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *ICSE*, pages 171–180, 2008.

N. Williams, B. Marre, P. Mouy, and M. Roger. PathCrawler : Automatic Generation of Path Tests by Combining Static and Dynamic Analysis. In M. D. Cin, M. Kaâniche, and A. Pataricza, editors, *EDCC*, volume 3463 of *Lecture Notes in Computer Science*, pages 281–292. Springer, 2005. ISBN 3-540-25723-3.

Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th conference on USENIX Security Symposium*, volume 15, pages 179–192, 2006.

F. Yu, M. Alkhalaf, and T. Bultan. Stranger : An Automata-Based String Analysis Tool for PHP. In J. Esparza and R. Majumdar, editors, *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 154–157. Springer, 2010. ISBN 978-3-642-12001-5.

D. M. Zimmerman and R. Nagmoti. JMLUnit : The Next Generation. In B. Beckert and C. Marché, editors, *FoVeOOS*, volume 6528 of *Lecture Notes in Computer Science*, pages 183–197. Springer, 2010. ISBN 978-3-642-18069-9.

Résumé. Les travaux présentés dans ce mémoire portent sur la validation de programmes PHP à travers un nouveau langage de spécification, accompagné de ses outils. Ces travaux s'articulent selon trois axes : langage de spécification, génération automatique de données de test et génération automatique de tests unitaires. La première contribution est Praspel, un nouveau langage de spécification pour PHP, basé sur la programmation par contrat. Praspel spécifie les données avec des domaines réalistes, qui sont des nouvelles structures permettant de valider et générer des données. À partir d'un contrat écrit en Praspel, nous pouvons faire du *Contract-based Testing*, c'est à dire exploiter les contrats pour générer automatiquement des tests unitaires. La deuxième contribution concerne la génération de données de test. Pour les booléens, les entiers et les réels, une génération aléatoire uniforme est employée. Pour les tableaux, un solveur de contraintes a été implémenté et utilisé. Pour les chaînes de caractères, un langage de description de grammaires avec un compilateur de compilateurs $\mathrm{LL}(\star)$ et plusieurs algorithmes de génération de données sont employés. Enfin, la génération d'objets est traitée. La troisième contribution définit des critères de couverture sur les contrats. Ces derniers fournissent des objectifs de test. Toutes ces contributions ont été implémentées et expérimentées dans des outils distribués à la communauté PHP.

Mots-clés. Praspel, *Contract-based Testing*, génération de données, génération automatique de tests unitaires, PHP.

Abstract. The works presented in this memoir are about the validation of PHP programs through a new specification language, along with its tools. These works follow three axes: specification language, automatic test data generation and automatic unit test generation. The first contribution is Praspel, a new specification language for PHP, based on the Design by Contract. Praspel specifies data with realistic domains, which are new structures allowing to validate and generate data. Based on a contract, we are able to perform Contract-based Testing, i.e. using contracts to automatically generate unit tests. The second contribution is about test data generation. For booleans, integers and floating point numbers, a uniform random generation is used. For arrays, a dedicated constraint solver has been implemented and used. For strings, a grammar description language along with an $\mathrm{LL}(\star)$ compiler compiler and several algorithms for data generation are used. Finally, the object generation is supported. The third contribution defines contract coverage criteria. These latters provide test objectives. All these contributions are implemented and experimented into tools distributed to the PHP community.

Keywords. Praspel, Contract-based Testing, data generation, automatic unit test generation, PHP.