Génie Logiciel Avancé

Université Paris-Cité

Édition 2025-2026

Léo Andrès

Table des matières

1 Avant-propos 5

2 Introduction 7
2.1 Méthodes de détection d'erreurs 7
2.2 Types algébriques et dénombrement 9

3 Production manuelle de tests 13
3.1 Tests manuels 13
3.2 Tests unitaires 14
3.3 Tests d'intégration 16

4 Évaluer la qualité d'une suite de tests 19
4.1 La couverture du code 19
4.2 Graphes d'appel et graphes de flot de contrôle 19
4.3 Critères de couverture formels 19
4.4 Mutation testing 20

5 Génération de tests automatique 21
5.1 Property Testing: QuickCheck 21
5.2 Fuzzing 21
5.2.1 Boîte noire : Radamsa 21
5.2.2 Boîte grise : AFL++ 21
5.2.3 Boîte blanche : libFuzzer 22
5.3 Exécution symbolique et concolique 22
5.4 Bounded Model Checking : CBMC 22

6 TODO 23

6 Bibliographie 25

3.1 « What's even worse is, a month ago they transferred me to work on the game I was already playing, and suddenly I found myself procrastinating by playing the one I'd been assigned before. It's possible they're onto me and this is all part of the plan. » 13

1. Avant-propos

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At.

Remerciements

Merci à Gabriel Scherer, de m'avoir encouragé à donner ce cours.

2. Introduction

Dans ce cours, nous allons nous intéresser au génie logiciel. En particulier, à la façon dont on peut faire au mieux afin qu'un logiciel **fasse ce qu'on attend de lui** en ayant une **consommation de ressources acceptables**. Par « ce qu'on attend de lui », on sous-entend généralement deux choses :

- qu'il n'y ait pas d'**erreur à l'exécution** (i.e. qu'il ne « plante » ou ne « crash » pas) ;
- qu'il respecte sa **spécification** (i.e. que les résultats qu'il donne soient ceux que l'on voulait obtenir).

Par « consommation de ressources acceptables », on parle généralement du **temps** et de la **mémoire**, mais on peut parfois s'intéresser à d'autres métriques : consommation électrique, temps de démarrage plutôt que temps total, nombre de connexions réseau etc.

Cependant, un logiciel ne doit pas être considéré comme un objet figé une fois son développement initial terminé, mais comme quelque chose qui va devoir évoluer au cours du temps. Lors de sa conception, il faut donc s'assurer du respect des deux propriétés mentionnés précedemment, mais également du fait qu'elle vont devoir être maintenues au fil du temps. C'est d'ailleurs, assez souvent, la difficulté principale.

2.1 Méthodes de détection d'erreurs

Il existe différentes méthodes permettant de détecter la présence potentielles d'erreurs à l'exécution ou le non-respect de sa spécification par un programme. On peut les classer dans trois différentes familles :

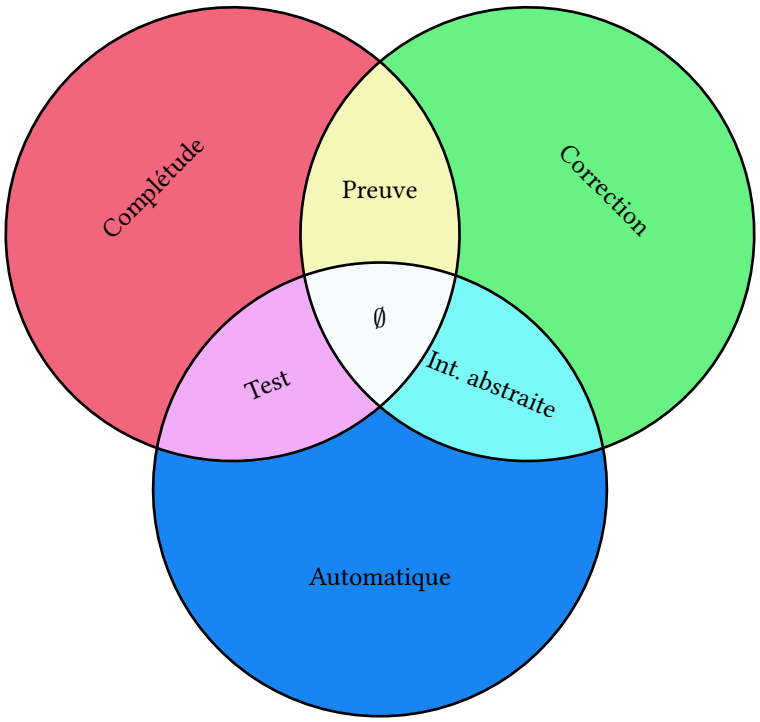
1. Le **test**, qui regroupe de nombreuses méthodes telles que le **fuzzing**, l'**exécution symbolique** ou le **model checking**.
2. L'**interprétation abstraite**.
3. La **preuve de programme**, qui comporte des méthodes telles que la **preuve interactive** et la **vérification déductive**.

Ces trois familles se différencient par certaines propriétés :

1. La **complétude** : la méthode ne produit pas de **fausses alarmes** (on parle aussi de faux positifs) dues à des **sur-approximations**.
2. La **correction** : la méthode ne rate pas de véritables erreurs (on parle aussi de faux négatifs) du fait de **sous-approximations**.
3. L'**automatisation** : la méthode permet d'obtenir des résultats automatiquement, sans intervention humaine.

Il existe plusieurs théorèmes mathématiques qui affirment qu'il est impossible pour une méthode d'analyse d'avoir les trois propriétés à la fois - du moins lorsque l'on cherche à être capable d'analyser des programmes arbitraires. Il s'agit par exemple des théorèmes de Rice et de Gödel. Cela signifie qu'il est possible pour une méthode de détection d'erreur d'avoir, **au mieux**, deux propriétés parmi les trois présentées. C'est justement la présence ou non de ces propriétés qui permet de les classer parmi trois familles différentes :

- le test est automatique et complet, mais pas correct ;
- l'interprétation abstraite est automatique et correcte, mais pas complète ;
- la preuve de programme est complète et correcte, mais pas automatique.



Preuve de programme

La vérification déductive, bien que correcte et complète, repose en grande partie sur l'intervention humaine [76]. Cette méthode exige que le programmeur fournisse des **invariants** sur les boucles et les types, ou prouve la **terminaison** de certaines fonctions, par exemple en fournissant des **variants**. Le programme, sa spécification et les éléments fournis par l'utilisateur sont alors transformés en un théorème mathématique qui est passé à des prouveurs automatiques. Lorsque le prouveur automatique échoue, c'est généralement que l'utilisateur n'a pas fourni les bons invariants (ou bien que le programme est effectivement incorrect). Il est également possible d'effectuer la preuve du théorème généré via un assistant de preuve. Si elle garantit des preuves très rigoureuses de la correction des programmes, sa mise en œuvre demande des efforts considérables, car elle n'est pas entièrement automatisée. Néanmoins, il arrive que sur certains programmes, la preuve soit complètement automatique. 5 L'objectif dans l'implémentation de cette méthode est de minimiser l'intervention manuelle du programmeur et d'automatiser le processus autant que possible. Des exemples d'outils de vérification déductive incluent Why3 [83], Boogie [60] et Viper [92].

La preuve interactive est très similaire à la vérification déductive. Celle-ci s'effectue au travers d'un **assistant de preuve**, qui guide l'utilisateur dans la preuve manuelle d'un théorème. La preuve se fait par l'utilisation de **tactiques** que l'utilisateur doit combiner. Une tactique est par exemple l'application d'un théorème déjà démontré, ou bien l'utilisation d'une hypothèse. Aujourd'hui, il existe des techniques faisant appel à des prouveurs automatiques tels que des SMTs, ce qui permet d'automatiser une partie de la preuve. Des exemples d'outils de preuve interactive incluent Rocq, HOL, Isabelle ou Lean.

Interprétation abstraite

Proposée par Cousot et Cousot en 1977 [6], l'interprétation abstraite est une méthode automatique et correcte. Cependant, elle n'est pas complète, ce qui signifie qu'elle peut signaler des bogues qui n'existent pas réellement, issus des approximations effectuées par l'analyse. Elle fonctionne en construisant une **sur-approximation** des états possibles du programmes au travers de **domaines abstraits**. Cela lui permet par exemple de générer des **invariants** qui garantissent certaines propriétés. Lorsqu'une propriété ne peut pas être garantie, une alarme est levée et signale à l'utilisateur une potentielle erreur. Cependant, lorsque la sur-approximation contenait des comportements impossibles en pratique, cela mène à des fausses alarmes. Cette méthode est donc moins adaptée à la recherche de bogues, car le tri entre les faux positifs et les véritables bogues peut s'avérer fastidieux. Néanmoins, il arrive sur certains programmes que l'approximation ne mène pas à des faux positifs. L'enjeu dans l'implémentation d'un moteur d'interprétation abstraite est de limiter le nombre de faux positifs. Des exemples notables de moteurs d'interprétation abstraite incluent Astrée [56] et Mopsa [158].

Test

Les méthodes de test sont trop nombreuses pour pouvoir être toutes décrites succintement ici. Cependant, c'est sur ces méthodes que l'on va se concentrer dans ce cours et elles seront donc détaillées plus loin. Ces méthodes sont automatiques et complètes. Toutefois elles ne sont pas correctes, car elles peuvent passer à côté de certains bogues ou ne pas aboutir à une conclusion - par exemple si ce sont des méthodes qui peuvent ne pas terminer. Il arrive, dans de rares cas, qu'elles puissent offrir une garantie de complétude et de correction lorsqu'elles terminent. Ces approches sont donc particulièrement adaptées et efficaces pour la recherche de bogues, même si elles le sont moins pour prouver la correction complète d'un programme. L'implémentation de ces méthodes doit chercher à minimiser les sous-approximations, autrement dit, à maximiser le nombre de cas couverts.

2.2 Types algébriques et dénombrement

Beaucoup des méthodes de tests sont des variations sur l'idée d'essayer le programme avec différentes valeurs d'entrées et de vérifier que le résultat est celui attendu. Cependant, le nombre des valeurs d'entrées possible est généralement très grand, lorsqu'il n'en existe pas une infinité. Dans cette section, nous cherchons donc à mesurer le nombre d'entrées possibles afin de fournir des ordres de grandeur qui seront utiles pour développer des intuitions dans le reste du cours.

Type et habitants

Soit t un type. On note $|t|$ le nombre d'habitants du type t. Il s'agit du nombre de valeurs distinctes de type t. Quelques cas simples :

t	Valeurs	$\|t\|$
`|`	$\emptyset$	0
`unit`	`()`	1
`bool`	`false`, `true`	2
`char`	..., `'a'`, `'b'`, `'c'`, ...	256
`int`	..., `-2`, `-1`, `0`, `1`, `2`, ...	$2^{63}-1$

On peut construire des types plus complexes en composant les types simples. Il existe plusieurs moyens de composer les types.

Types produits

On peut composer deux types en formant leur produit, noté : $t_1 \times t_2$. On a alors $|t_1 \times t_2| = |t_1| \times |t_2|$. En OCaml, on note le produit de deux types `t1` et `t2` ainsi : `t1 * t2`. Par exemple :

t	Valeurs	$\|t\|$
`bool * char`	`(true, 'a')`, `(false, '\n')`, ...	$2 \times 256 = 512$
`int * int`	`(0, 1)`, `(3, -2)`, ...	$(2^{63} - 1) \times (2^{63} - 1)$

Types sommes

On peut également composer deux types en formant leur somme, notée : $t_1 + t_2$. On a alors $|t_1 + t_2| = |t_1| + |t_2|$. En OCaml, on peut définir `int + float` ainsi :

```
type t =
  | A of int
  | B of float
```

Il a comme valeurs possibles :

```
...; A -2; A -1; A 0; A 1; A 2; ...
...; B nan; B 0.42; B 12.34; B 100.42; ...
```

Un constructeur constant n'a qu'un seul habitant, par exemple :

```
type t =
  | A
  | B of int
```

Ici, `A` n'a qu'une seule valeur possible : `A`. La définition précédente est donc équivalente à :

```
type t =
  | A of unit
  | B of int
```

Et on peut donc dénoter ce type par `unit + int`. Voici quelques autres exemples utilisant des types prédéfinis :

t	Valeurs	$\|t\|$
`bool Option.t`	`None`, `Some false`, `Some true`	3
`bool List.t`	`[]`, `[false]`, `[true]`, `[false; false]`, `[false; true]`, `[true; false]`, `[true; true]`, ...	∞

> **Tip**
>
> Les enregistrements sont aussi des types produits, par exemple :
>
> ```
> type t = {
>   x : int;
>   y : int;
> }
> ```
>
> Le type `t` est équivalent à `int * int`. Ses champs sont nommés, mais il permet de représenter le même ensemble de valeurs.

Types de fonctions

> ⚠ **Warning**
>
> On ne considère ici que les fonctions **pures**, i.e. sans effet de bord et qui n'échouent pas.

Enfin, on peut s'intéresser aux types de fonctions. Par exemple `bool -> bool` peut prendre les valeurs suivantes :

```
(* Constante `false` *)
let f1 = function
  | false -> false
  | true -> false

(* Identité *)
let f2 = function
  | false -> false
  | true -> true

(* Négation *)
let f3 = function
  | false -> true
  | true -> false

(* Constante `true` *)
let f4 = function
  | false -> true
  | true -> true
```

Regardons un autre exemple, celui du type `unit -> bool`. Il peut prendre les valeurs :

```
let f1 () = false

let f2 () = true
```

On a donc $|t_1 \longrightarrow t_2| = |t_2|^{|t_1|}$.

Génération et énumération

Combien de temps faut-il pour énumérer tous les entiers 32 bits ? Considérons le programme `enum.ml` suivant :

```
let () =
  let start = Int32.to_int Int32.min_int in
  let stop = Int32.to_int Int32.max_int in
  for i = start to stop do
    Sys.opaque_identity ignore i;
  done
```

On peut alors le compiler et mesurer le temps d'exécution avec :

```
$ ocamlopt -O3 enum.ml
$ time ./a.out
./a.out  5.60s user 0.00s system 99% cpu 5.620 total
```

Pour faire la même énumération avec un entier 64 bits, on peut donc estimer que cela prendrait environ $2^{32} \times 5.6\text{s} \approx 760$ ans ! De plus, la fonction testée ici est simplement un appel à `ignore`. On comprend alors pourquoi espérer tester n'importe quel programme avec toutes ses entrées possibles est… **irréaliste**. Et pourtant, il est possible en pratique de tester ses programmes d'une façon telle que l'ont peut espérer, sans essayer toutes les entrées possibles, détecter la grande majorité des erreurs et arriver à un résultat similaire à celui obtenu par énumération brutale et naïve. C'est ce que l'on verra dans la suite de ce cours.

3. Production manuelle de tests

Pour commencer, nous allons nous intéresser aux différentes méthodes de tests où les entrées sont produites manuellement.

3.1 Tests manuels

Le **test manuel** consiste à faire tester le logiciel **par un humain** afin que celui-ci vérifie que tout se passe comme prévu. Il est très utilisé en phase de prototypage, afin de vérifier rapidement que le logiciel a l'air de faire ce que l'on veut et qu'il n'y a pas d'erreurs évidentes. Cependant, cela peut aussi s'avérer pertinent dans d'autres contextes. C'est notamment le cas lorsque le logiciel présente une interface complexe, par exemple parce l'interaction avec l'humain pendant l'exécution du programme est très importante (jeu vidéo), ou encore parce que celui-ci interagit avec un environnement physique difficile à simuler (robot). Cependant, le test manuel présente plusieurs défauts : il est généralement très coûteux (en temps humain) et potentiellement difficile à reproduire. Par exemple, on peut provoquer une erreur en jouant à un jeu vidéo, sans savoir exactement quelle suite d'entrées a mené à ce bogue.

La notion de boîte noire

Le test manuel est qualifié de test en **boîte noire**. Cela signifie que le test se fait au travers d'interactions avec le logiciel sans « regarder à l'intérieur » de celui-ci. Dit autrement, si vous testez un jeu vidéo en y jouant, vous interagissez uniquement avec le binaire exécutable, vous n'effectuez pas vos tests sur le code source de celui-ci. Vous ne pouvez pas savoir comment il est implémenté, mais seulement observer les résultats qu'il produit.

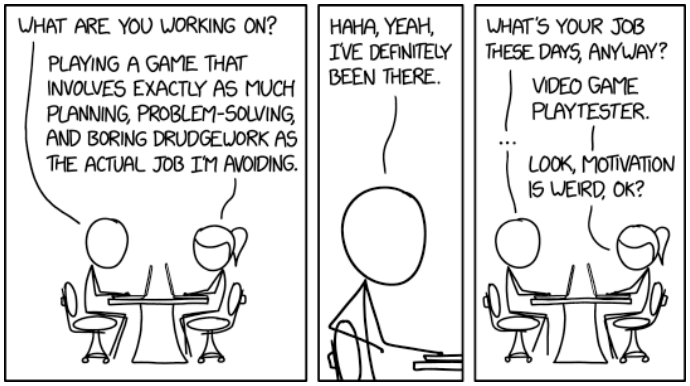


Fig. 3.1. – « What's even worse is, a month ago they transferred me to work on the game I was already playing, and suddenly I found myself procrastinating by playing the one I'd been assigned before. It's possible they're onto me and this is all part of the plan. »

3.2 Tests unitaires

À l'opposé du test manuel, se trouve le **test unitaire**. Le principe de celui-ci est d'écrire de nombreux tests (sous forme de programme), chacun d'entre eux se concentrant sur une petite partie du programme à tester. L'avantage des tests unitaires est qu'ils sont bien moins coûteux en temps humain puisqu'une fois qu'ils sont écrits, les exécuter est très rapide et ne nécessite pas d'intervention humaine. De plus, ils sont généralement reproductibles puisqu'on sait exactement quelles entrées ont été utilisées.

La notion de boîte blanche

Le test unitaire est qualifié de test en **boite blanche**. Cela signifie que, à l'opposé des tests en boite noire, on a accès à « l'intérieur » du programme testé.

Exemple de test unitaire

Prenons un exemple et supposons que l'on soit en train de tester une bibliothèque OCaml appelée `lib`. Celle-ci contient un module `Math`, lequel contient une fonction `square`. Le module est contenu dans le fichier `math.ml` suivant :

```
(** The Math module. *)
let square x = x * x
```

La bibliothèque est construite avec le fichier `dune` suivant :

```
(library
  (public_name lib)
  (modules math))
```

Écrire un test unitaire pour la fonction `square` consiste à écrire un programme de test qui appelle la fonction `Lib.Math.square` sur une entrée connue et à vérifier que le résultat obtenu est bien celui auquel on s'attend. On appelle le programme qui appelle la fonction à tester un **harnais de test**. Voilà un fichier `test_math.ml` qui teste notre fonction sur une entrée donnée :

```
(** Tests for the Math module. *)

(* Test for the square function *)
let test_square () =
  let x = 3 in
  let result = Lib.Math.square x in
  let expected = 9 in
  assert (Int.equal result expected)

let () =
  test_square ();
  Format.printf "All tests are OK!"
```

Exécution des tests

On peut alors écrire le fichier `dune` suivant :

```
(test
  (name test_math)
  (modules test_math)
  (libraries lib))
```

Cela nous permet d'exécuter notre test au moyen de la commande `dune runtest` :

```
$ dune runtest
All tests are OK!
[0]
```

> ⚠ **Warning**
>
> Dans le cas de `dune`, lancer la commande `dune runtest` plusieurs fois de suite ne va pas forcément relancer l'ensemble des tests. En effet, afin d'économiser des ressources, `dune` est capable de se rappeler des tests qui ont réussi et de ne les relancer que si l'une de leur dépendance a changé depuis la dernière exécution. Ici, l'unique dépendance est la bibliothèque `lib`. Il est possible de forcer `dune` à relancer les tests avec l'option `--force`.

Si l'on introduit volontairement une erreur dans le programme ou dans le harnais de test (par exemple en remplaçant 9 par 6), on aura une erreur :

```
$ dune runtest
File "dune", line 2, characters 8-17:
6 |   (name test_math)
           ^^^^^^^^^
Fatal error: exception Assert_failure("test_math.ml", 8, 2)
```

Bibliothèque pour les tests unitaires

En pratique, on utilise souvent des bibliothèques qui aident à l'écriture de tests unitaire. Elles fournissent généralement des primitives qui permettent de comparer des résultats obtenus à ceux attendus, de gérer des fonctions particulières comme celles qui lèvent des exceptions, ou encore d'afficher de manière plus lisible de potentielles erreurs obtenues en exécutant les tests. En OCaml, la bibliothèque de test unitaire la plus répandue est `alcotest`. Lorsque l'on lance `dune runtest`, `alcotest` affiche ainsi les résultats (les exemples qui suivent ont été repris des tests de la bibliothèque `synchronizer`) :

```
Testing `Synchronizer'.
This run has ID `RLSUYDMJ'.

  [OK]          Basic operations              0   Empty queue no pledges.
  [OK]          Basic operations              1   Get single item.
  [OK]          Basic operations              2   Get multiple items.
  [OK]          Pledge mechanics              0   Manual pledge.
  [OK]          Pledge mechanics              1   Get creates pledge.
  [OK]          Pledge mechanics              2   Blocking on pledge.
  [OK]          work_while                    0   work_while simple.
  [OK]          work_while                    1   work_while empty.
  [OK]          Close functionality           0   Close returns None.
  [OK]          Close functionality           1   Close with items.
  [OK]          Concurrent operations         0   Producer/consumer.
  [OK]          Concurrent operations         1   Multiple workers.
  [OK]          Concurrent operations         2   Concurrent write/get.
  [OK]          Concurrent operations         3   Graph traversal.
  [OK]          Stress tests                  0   Stress test.

Full test results in `synchronizer/_build/default/test/_build/_tests/Synchronizer'.
Test Successful in 0.018s. 15 tests run.
```

Son utilisation présente plusieurs avantages au fait de gérer manuellement les tests. En particulier, les tests sont documentés via le groupe et la description qui leur sont attribués. De plus, même si l'un des tests échoue, les autres sont lancés (on ne s'arrête pas à la première erreur). Il permet également de sélectionner un sous-ensemble de tests à lancer. Enfin, lorsqu'un test échoue, il affiche la valeur obtenue et celle qui était attendue, et enregistre les sorties standard et d'erreurs dans un fichier pour permettre d'investiguer :

```
Testing `Synchronizer'.
This run has ID `5MJ9FN0B'.

  [OK]          Basic operations              0   Empty queue no pledges.
> [FAIL]        Basic operations              1   Get single item.
  [OK]          Basic operations              2   Get multiple items.
  [OK]          Pledge mechanics              0   Manual pledge.
  [OK]          Pledge mechanics              1   Get creates pledge.
  [OK]          Pledge mechanics              2   Blocking on pledge.
  [OK]          work_while                    0   work_while simple.
  [OK]          work_while                    1   work_while empty.
  [OK]          Close functionality           0   Close returns None.
  [OK]          Close functionality           1   Close with items.
  [OK]          Concurrent operations         0   Producer/consumer.
  [OK]          Concurrent operations         1   Multiple workers.
```

```
  [OK]          Concurrent operations          2   Concurrent write/get.
  [OK]          Concurrent operations          3   Graph traversal.
  [OK]          Stress tests                   0   Stress test.

 [FAIL]         Basic operations               1   Get single item.

ASSERT get returns Some 42
FAIL get returns Some 42

   Expected: `Some 666'
   Received: `Some 42'

              synchronizer/_build/default/test/_build/_tests/Synchronizer/
Logs saved to `Basic operations.001.output                                  '.

Full test results in `synchronizer/_build/default/test/_build/_tests/Synchronizer'.
1 failure! in 0.018s. 15 tests run.
```

L'écriture d'un test se fait ainsi :

```
let test_get_single_item () =
  let result = (* call to the function being tested *) in
  Alcotest.(check (option int)) "get returns Some 42" (Some 42) result
```

La déclaration d'un groupe de tests est une simple liste où l'on décrit chaque test :

```
let basic_tests =
  [ Alcotest.test_case "Empty queue no pledges" `Quick test_empty_queue_no_pledges
  ; Alcotest.test_case "Get single item"        `Quick test_get_single_item
  ; (* ... *) ]
```

Finalement, on peut lancer l'ensemble de nos groupes de tests de cette manière :

```
let () =
  Alcotest.run "Synchronizer"
    [ ("Basic operations", basic_tests)
    ; ("Pledge mechanics", pledge_tests)
    (* ... *)
    ]
```

Ces bibliothèques pour le test unitaire ne sont généralement pas particulièrement sophistiquées, mais elles rendent le travail quotidien avec les tests unitaires plus agréable.

3.3 Tests d'intégration

Les tests d'intégration sont des tests où l'on considère le comportement d'une grosse portion du programme, et non plus une petite partie de celui-ci comme dans le cas des tests unitaires.

Cram tests

Les ***cram tests*** sont des tests d'intégration pour les outils en ligne de commande. Cram était à l'origine un programme mais différentes variantes ont depuis été implémentées et les *cram tests* désignent aujourd'hui la méthode qui en est issue plutôt que cet outil. En particulier, `dune` intègre nativement un système de *cram tests*.

Au sein de `dune`, les *cram tests* sont des fichiers ayant l'extension `.t`. Toutes les lignes qui ne sont pas indentées sont des commentaires. En revanche, toutes les lignes indentées de deux espaces sont des lignes significatives. L'idée consiste à écrire une série de commandes :

```
On peut appeler l'outil que l'on souhaite tester et sauvegarder sa sortie :
  $ ./my_amazing_tool.exe > output.txt
```

Les lignes significatives commençant par `$` sont des commandes qui vont être exécutées. Mais l'autre atout majeur des *cram tests* réside dans les lignes significatives qui commencent par un autre caractère :

celles-ci représentent en effet la sortie attendue des commandes exécutées. Par exemple, si l'on reprend notre exempe précédent, on peut vérifier que le fichier a bien été créé et que son contenu est le bon :

```
  $ ./my_amazing_tool.exe > output.txt
Vérifions le contenu du nouveau fichier :
  $ cat output.txt
  Hello from my_amazing_tool!
```

Si au cours du temps, le programme est modifié pour ne plus afficher ce message mais « Hello, world! » à la place, voilà la sortie qui sera produite par `dune runtest` :

```
File "amazing.t", line 1, characters 0-0:
------ amazing.t
++++++ amazing.t.corrected
File "amazing.t", line 3, characters 0-1:
  $ ./my_amazing_tool.exe > output.txt
  $ cat output.txt
- Hello from my_amazing_tool!
+ Hello, world!
```

```
File "git_help.t", line 1, characters 0-0:
------ git_help.t
++++++ git_help.t.corrected
File "git_help.t", line 2, characters 0-1:
  $ git --help
+ usage: git [-v | --version] [-h | --help] [-C <path>] [-c <name>=<value>]
+             [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
+
          [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--no-lazy-fetch]
+             [--no-optional-locks] [--no-advice] [--bare] [--git-dir=<path>]
+
            [--work-tree=<path>] [--namespace=<name>] [--config-env=<name>=<envvar>]
+             <command> [<args>]
+
+ These are common Git commands used in various situations:
+
+ start a working area (see also: git help tutorial)
+    clone      Clone a repository into a new directory
+    init       Create an empty Git repository or reinitialize an existing one
+
+ work on the current change (see also: git help everyday)
+    add        Add file contents to the index
+    mv         Move or rename a file, a directory, or a symlink
+    restore    Restore working tree files
+    rm         Remove files from the working tree and from the index
+
+ examine the history and state (see also: git help revisions)
+    bisect     Use binary search to find the commit that introduced a bug
+    diff       Show changes between commits, commit and working tree, etc
+    grep       Print lines matching a pattern
+    log        Show commit logs
+    show       Show various types of objects
+    status     Show the working tree status
+
+ grow, mark and tweak your common history
+    backfill   Download missing objects in a partial clone
+    branch     List, create, or delete branches
+    commit     Record changes to the repository
+    merge      Join two or more development histories together
+    rebase     Reapply commits on top of another base tip
+    reset      Reset current HEAD to the specified state
+    switch     Switch branches
+    tag        Create, list, delete or verify a tag object signed with GPG
+
+ collaborate (see also: git help workflows)
+    fetch      Download objects and refs from another repository
+    pull       Fetch from and integrate with another repository or a local branch
+    push       Update remote refs along with associated objects
+
+ 'git help -a' and 'git help -g' list available subcommands and some
+ concept guides. See 'git help <command>' or 'git help <concept>'
+ to read about a specific subcommand or concept.
+ See 'git help git' for an overview of the system.
```

Et de finir par la commande `dune promote` pour avoir un fichier de test complet. Ce comportement est aussi très utile lorsque la sortie d'un programme évolue au cours de temps : il est bien plus agréable de taper ces quelques commandes pour inspecter le résultat et mettre à jour le test, plutôt que de devoir modifier à la main la sortie attendue par de nombreux tests unitaires manipulant des chaînes de caractères.

Expect tests

- en OCaml, `ppx_expect`

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At.

4. Évaluer la qualité d'une suite de tests

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At.

4.1 La couverture du code

- bisect_ppx en OCaml

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At.

4.2 Graphes d'appel et graphes de flot de contrôle

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At.

4.3 Critères de couverture formels

- mesures formelles de couverture (coverage criteria)
- labels

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At.

4.4 Mutation testing

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At.

5. Génération de tests automatique

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At.

5.1 Property Testing: QuickCheck

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At.

5.2 Fuzzing

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At.

5.2.1 Boîte noire : Radamsa

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At.

5.2.2 Boîte grise : AFL++

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At.

5.2.2.1 Crowbar

5.2.3 Boîte blanche : libFuzzer

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At.

5.3 Exécution symbolique et concolique

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At.

5.4 Bounded Model Checking : CBMC

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At.

6. TODO

- Forge
- documentation (Conventional Commits, CHANGELOG, API, manpage, diataxeis, MDX)
- source of failures (type error, memory error, oob, undefined behaviours, unspecified behaviours, race condition, thread-safety etc.)
- debug
- benchmarks / optimisation
- CI
- deploiement / reproducibilité / packaging / NixOS
- proof of programs (specification language??) (more on abstract interpretation, deductive verification ?)
- assertions, RAC ?
- détection de code mort ?
- solveurs SAT/SMT ?

Bibliographie