Génie Logiciel Avancé

Université Paris-Cité

Édition 2025-2026

Léo Andrès

Table des matières

Liste des figures


THERE'S A LESSON TO BE LEARNED HERE SOMEWHERE, BUT I DON'T KNOW WHAT IT IS...
SCHULZ

1. Avant-propos

Ce document contient les notes de cours que j'ai donné à l'Université Paris Cité à des étudiants de Master 1 en 2026.

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 :

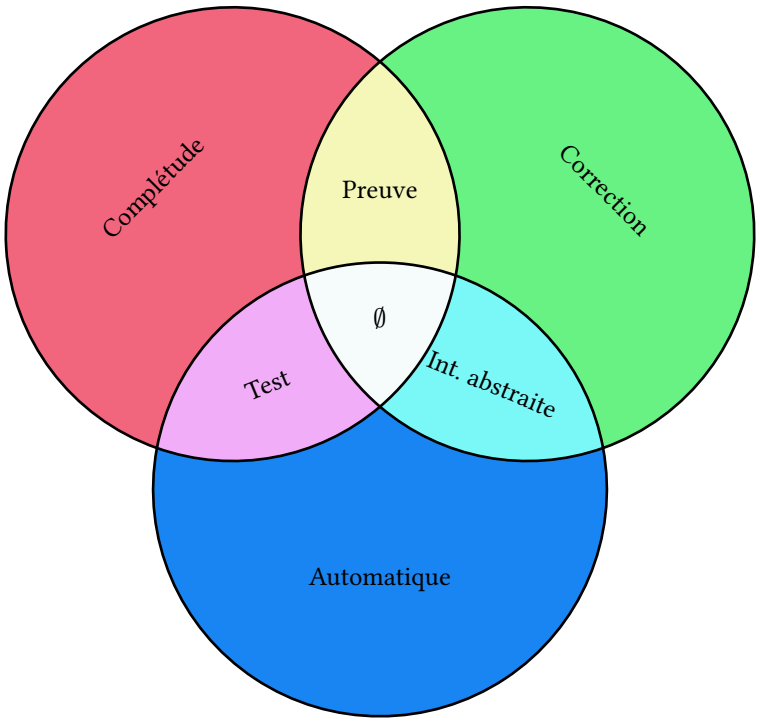
1. Le **test**, qui regroupe de nombreuses méthodes telles que le **fuzzing**, l'**exécution symbolique** ou le **model checking**.
2. L'**interprétation abstraite**.
3. La **preuve de programme**, qui comporte des méthodes telles que la **preuve interactive** et la **vérification déductive**.

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.

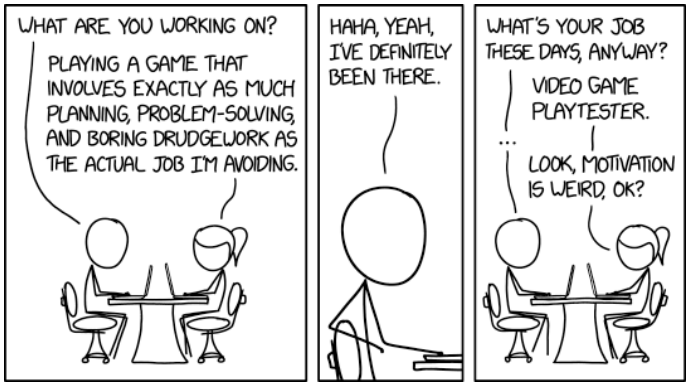


Fig. 3.1. – « What's even worse is, a month ago they transferred me to work on the game I was already playing, and suddenly I found myself procrastinating by playing the one I'd been assigned before. It's possible they're onto me and this is all part of the plan. »

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!
```

Si la différence affichée nous semble conforme au nouveau comportement attendu du programme, il suffit de lancer la commande `dune promote` et le fichier `amazing.t` sera automatiquement mis à jour. Cela est **extrêmement pratique**, par exemple lorsque l'on souhaite écrire des nouveaux tests. Il suffit d'écrire les commandes que l'on veut, sans se préoccuper de la sortie. Par exemple, si pour tester la commande d'aide de `git`, on se contente d'écrire le fichier `git.t` suivant :

```
  $ git --help
```

De lancer la commande `dune runtest`, pour obtenir :

```
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.

4. Évaluer la qualité d'une suite de tests

Une fois que l'on a écrit des tests, on cherche généralement à mesurer la qualité des tests écrits. C'est-à-dire, savoir quelle portion de notre code est effectivement testée. On appelle cette mesure la **couverture du code par les tests**.

4.1 La couverture du code

Commençons par regarder comment une telle couverture du code peut être obtenue. En OCaml, la bibliothèque `bisect_ppx` permet de mesurer cette couverture. Pour l'utiliser, il faut indiquer les différentes bibliothèques et programmes pour lesquels on aimerait mesurer la couverture. Cela se fait en modifiant les fichiers `dune` :

```
; for a library, add:
(library
 ; ...
 (instrumentation
  (backend bisect_ppx))
)

; for an executable, add:
(executable
 ; ...
 (instrumentation
  (backend bisect_ppx))
)
```

Ajouter ces lignes n'a aucun impact sur le projet et les bibliothèques ou exécutables produits, tant qu'on ne les compile pas d'une façon spéciale. Cela ne coûte donc rien de les ajouter. La génération se fait alors en deux étapes. Tout d'abord, on va lancer notre suite de tests, mais en indiquant à `dune` d'activer l'**instrumentation** pour `bisect_ppx` :

```
$ BISECT_FILE=$(pwd)/bisect dune runtest --force --instrument-with bisect_ppx
```

On utilise l'option `--force` afin d'être certain que tous les tests sont bien rejoués et pris en compte dans la mesure de la couverture. Par ailleurs, on indique où les fichiers produits doivent se situer, au moyen de la variable d'environnement `BISECT_FILE`. Cette **instrumentation** va automatiquement modifier l'ensemble de nos fichiers OCaml avant qu'ils ne soient compilés, afin d'y ajouter de nouvelles instructions qui vont permettre de compter le nombre de fois où chaque partie de notre code est exécutée. Ainsi, lorsque `dune` exécute les tests, ils seront exécutés sur la version instrumentée du code. Le code correspondant à l'instrumentation va alors émettre de nombreux fichiers avec les statistiques qui nous intéressent. Pour obtenir un résultat lisible, il faut fusionner toutes ces statistiques (chaque test génère un

fichier de statistiques différent), et générer un rapport de couverture. On peut obtenir le résultat global ainsi :

```
$ bisect-ppx-report summary
Coverage: 9416/12003 (78.45%)
```

Il est également possible d'obtenir un rapport détaillant le score obtenu sur chaque fichier :

```
$ bisect-ppx-report summary --per-file
 93.51 %    245/262     src/bin/owi.ml
 87.02 %    114/131     src/cmd/cmd_c.ml
 73.60 %    131/178     src/cmd/cmd_call_graph.ml
 85.19 %     69/81      src/cmd/cmd_cfg.ml
 87.63 %     85/97      src/cmd/cmd_cpp.ml
 95.00 %     19/20      src/cmd/cmd_fmt.ml
 76.92 %     10/13      src/cmd/cmd_instrument_label.ml
 61.41 %    113/184     src/cmd/cmd_iso.ml
 54.19 %    110/203     src/cmd/cmd_replay.ml
 77.78 %      7/9       src/cmd/cmd_run.ml
 86.96 %     40/46      src/cmd/cmd_rust.ml
100.00 %      5/5       src/cmd/cmd_script.ml
100.00 %     22/22      src/cmd/cmd_sym.ml
  0.00 %      0/43      src/cmd/cmd_tinygo.ml
 77.27 %     68/88      src/cmd/cmd_utils.ml
 75.00 %      3/4       src/cmd/cmd_validate.ml
 21.74 %      5/23      src/cmd/cmd_version.ml
100.00 %     17/17      src/cmd/cmd_wasm2wat.ml
100.00 %      7/7       src/cmd/cmd_wat2wasm.ml
 79.07 %     34/43      src/cmd/cmd_zig.ml
# ... truncated for readability
 78.45 %   9416/12003   Project coverage
```

On peut également générer un rapport bien plus complet au format HTML de cette façon :

```
$ bisect-ppx-report html -o _coverage
```

Le rapport produit est alors inspectable via notre navigateur :

```
$ xdg-open _coverage/index.html
```

Celui-ci comprend un rapport pour tout le projet, avec le pourcentage de couverture de chaque fichier, comme le montre Fig. 4.1. Mais, lorsque l'on clique sur le nom d'un fichier, on arrive sur un rapport détaillé sur ce fichier, où l'on peut voir précisément les portions de code qui ont été testées ou non, et le nombre de fois que chacune a été exécutée, comme on peut le voir sur Fig. 4.2.

Calculer la couverture du code par le tests est un bon moyen pour trouver les parties du code qui ne sont pas testées. Dans certain cas, on peut se rendre compte qu'une partie du code est en fait **inatteignable** et ne pourra jamais être testé. Cela se produit souvent lorsque le code évolue : certaines parties deviennent inutiles sans que l'on s'en rende compte. Avoir une métrique pour la couverture du code permet également de vérifier, lors de l'ajout de code, que celui-ci est bien testé. Si c'est le cas, le pourcentage de couverture ne baisse pas, sinon, c'est qu'on a introduit du code mais pas de tests correspondant.

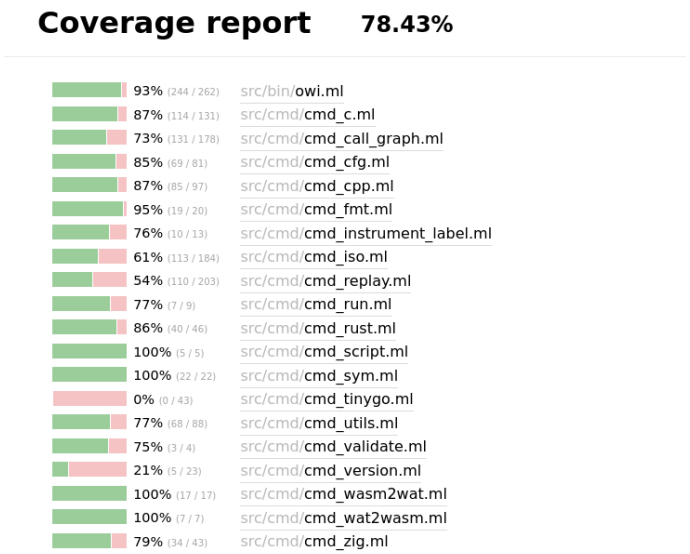


Fig. 4.1. – Rapport de couverture pour le projet.

Fig. 4.2. – Rapport de couverture pour un fichier.

Dans l'absolu, la couverture permet aussi d'avoir une idée de la qualité de la suite de tests, et d'à quel point elle nous donne des garanties sur le code. Cependant, poour l'instant, nous n'avons pas formellement défini à quoi correspond ce pourcentage. Pour être capable de faire cela, on va commencer par s'intéresser à la façon dont on peut représenter des programmes arbitraires, avant de définir des mesures formelles sur ces représentations.

4.2 Représentations structurelles d'un programme

Dans cette section, nous allons présenter plusieurs façons de représenter un programme ou une exécution de celui-ci. Par la suite, ces représentations nous permettrons de mieux comprendre et décrire diverses analyses sur les programmes.

4.2.1 Graphe d'appel

Un graphe d'appel est une représentation **statique** du programme. Cela signifie que la représentation ne dépend pas d'une exécution particulière du programme, mais s'intéresse à la structure de celui-ci, indépendemment des entrées avec lesquelles il sera exécuté. Cette représentation est un **graphe**, où chaque **sommet** représente une fonction. Les **arcs** de ce graphe sont **orientés**. Un arc d'un sommet f vers un sommet g indique que la fonction f peut appeler la fonction g. Par exemple, si on a le programme (invalide) Wasm `cg1.wat` suivant :

```
(module

  (func $start
    i32.const 0
    (if
      (then call $a)
      (else call $b)))

  (func $a
    (block
      call $b)
    call $c)

  (func $b)

  (func $c
    call $c))
```

Sa représentation sous forme de graphe d'appel est visible dans la Fig. 4.3.

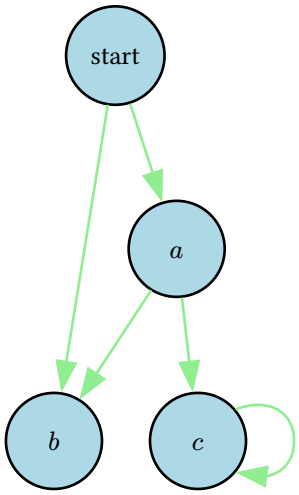


Fig. 4.3. – Graphe d'appel du programme `cg1.wat`

Dans les faits, on ne peut pas calculer le graphe d'appel **exact** de n'importe quel programme. Il s'agit en effet d'un problème **indécidable**. Cette indécidabilité provient de la notion **d'appels indirects**. Les appels indirects sont une notion que l'on retrouve dans la majorité des langages de programmation, mais sous des formes différentes. On parle d'appel indirect lorsqu'un programme n'appelle pas une fonction directement (par exemple par son nom, comme dans `call $f`), mais indirectement, par exemple via une référence, une variable ou un object. Par exemple, on peut imaginer un programme C où la fonction qui sera appelée est choisie aléatoirement :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```c
void hello(void) {
  printf("Hello\n");
}

void goodbye(void) {
  printf("Goodbye\n");
}

void never(void) {
  printf("Never\n");
}

void (*funcs[])(void) = { hello, goodbye, never };

void main(void) {
  fptr = funcs[rand() % 2];
  fptr();
}
```

Si on analyse le programme **syntaxiquement**, les valeurs possibles de `fptr` sont toutes les valeurs du tableau `funcs`. Seulement, si on commence à s'y intéresser **sémantiquement**, on remarque qu'on accède à l'indice `rand() % 2` et on peut donc déduire qu'il ne peut valoir que `0` ou `1`. Donc, la fonction `never` ne sera jamais sélectionnée. On peut donc savoir que `main` appellera `hello` ou `goodbye`. Mais en réalité, il est possible que l'indice qu'on utilise soit inconnu statiquement (par exemple, fourni par l'utilisateur pendant l'exécution, ou bien dépendant d'un calcul mathématique complexe). De plus, le tableau `funcs` peut être modifié pendant l'exécution du programme, ce qui rend les choses encore plus complexes. On pourait imaginer que la fonction `rand` a été remplacée par une fonction qui modifie la table à chaque appel et il est impossible de le savoir statiquement.

Pour cette raison, il existe deux façons de calculer les graphes d'appels. On peut soit procéder à une sur-approximation, et accepter que le graphe d'appel contiendra des arcs en trop par rapport à ce qui est possible lors de l'exécution du programme. Soit, on peut procéder à une sous-approximation, et accepter que le graphe d'appel ne contiendra pas certains arcs qui sont pourtant possibles en pratique. Les deux approches ont leur utilité. Par exemple, dans le cas où l'on écrit un optimiseur qui chercherait à transformer les appels indirects en appels direct (ces derniers étant plus efficaces), on va procéder par sur-approximation, et on ne remplacera l'appel indirect par un appel direct si on peut garantir qu'il n'y a qu'une seule valeur possible pour l'appel. En revanche, dans le cas d'un outil d'exécution symbolique qui cherche trouver des bugs dans le programme, on va plutôt s'intéresser à une sous-approximation et explorer les cas dont on est sûr qu'ils sont possibles - on ne veut pas lever de fausse alarme provoquée par un appel de fonction qui n'est pas possible en pratique.

Dans les deux cas, on dispose de diverses techniques pour minimiser l'approximation et s'approcher de la solution réelle. Par exemple, en WebAssembly, il y a plusieurs restrictions sur les appels indirects, en particulier, le type de la fonction sera vérifié à l'exécution (on peut donc éliminer les fonctions qui n'ont pas le bon type), mais surtout, on doit déclarer dans le programme toutes les fonctions qui pourront être appelée de façon indirecte, ce qui permet d'exclure les autres. Par exemple, si l'on prend le programme `indirect.wat` suivant :

```
(module
  (import "env" "unknown" (func $unknown (result i32)))
  (table $mytable 2 funcref) ;; a function table
  (type $t0 (func))

  (func $hello)
  (func $goodbye)
  (func $never (param i32))

  (func $start
    call $hello
    call $unknown
    call_indirect $mytable (type $t0))

  (start $start))
```

On peut générer son graphe d'appel avec sur-approximation Fig. 4.4 et avec sous-approximation Fig. 4.5. On remarque que la fonction `$never`, qui a l'identifiant 3 n'apparaît pas dans le graphe complet puisque son type ne correspond pas à celui attendu par l'appel indirect.

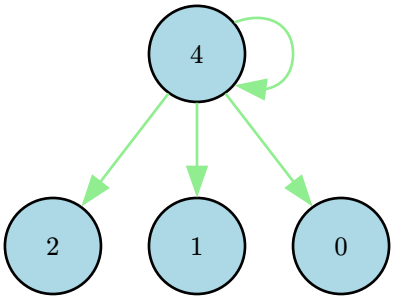


Fig. 4.4. – Graphe d'appel complet du programme `indirect.wat`, généré avec `$ owi analyze cg --call-graph-mode=complete indirect.wat`.

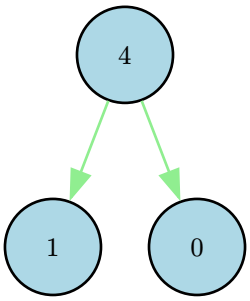


Fig. 4.5. – Graphe d'appel correct du programme `indirect.wat`, généré avec `$ owi analyze cg --call-graph-mode=sound indirect.wat`.

4.2.2 Graphe de flot de contrôle

Un graphe de flot de contrôle est une autre représentation statique d'un programme. Ici, nous allons nous intéresser aux graphes de flot de contrôle permettant de représenter une fonction à la fois, mais il existe des graphes de flot de contrôle qui représentent plusieurs fonctions à la fois. Dans un graphe de flot de contrôle, les sommets sont des expressions (souvent, des suites d'instructions) dans lesquelles il n'y a aucun **branchement**. C'est-à-dire que les différentes instructions s'exécutent l'une après l'autre. Ce sont justement les arcs du graphe qui vont représenter les différents branchements. Par exemple, si on a le programme Wasm `cfg.wat` suivant :

```
(module
  (func $f (param i32) (result i32) (local i32)
    (block
      (block
        (block
          (i32.eqz (local.get 0))
          br_if 0
          (i32.eq (i32.const 1) (local.get 0))
          br_if 1
          (local.set 1 (i32.const 7))
          br 2)
        (local.set 1 (i32.const 42))
        br 1)
      local.set 1 (i32.const 99))
    local.get 1))
```

On peut générer son graphe de flot de contrôle qui est présenté dans la Fig. 4.6.

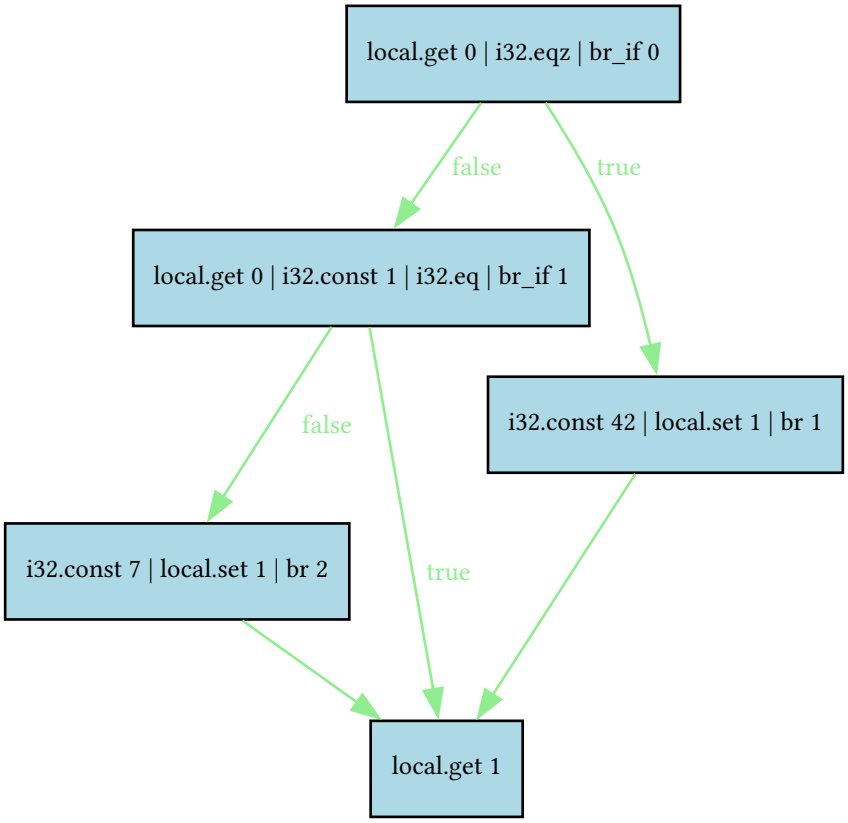


Fig. 4.6. – Graphe de flot de contrôle généré avec `$ owi analyze cfg --entry-point=f cfg.wat`.

On remarque que certains sommets n'ont qu'un seul arc de sortie. Ces arcs correspondent à des branchements inconditionnels. On aurait pu les enlever, mais cela nous oblige alors à dupliquer certaines instructions et il est alors plus difficile de visualiser le fait que différentes expressions finissent au même endroit du programme.

Le flot de contrôle d'un programme peut être **structuré** ou **non structuré**. On dit qu'il est structuré si son graphe de flot de contrôle est **réductible**. Le graphe de flot de contrôle est **réductible** si après avoir supprimé tous les **arcs de retours** (*back edges*) on obtient un graphe **acyclique** (qui ne contient pas de cycle) dont tous les sommets sont accessibles depuis l'entrée du graphe. Un arc de A vers B est un **arc de retour** si et seulement si B domine A. On dit que B domine A si et seulement si tous les chemins depuis le sommet d'entrée du graphe jusqu'à A passent par B. Dit autrement, B domine A si on doit forcément passer par B pour atteindre A. Par exemple dans Fig. 4.7, le graphe de gauche contient un seul **arc de retour** (celui de d vers b). Si on le retire, on obtient le graphe de droite qui contient un cycle. C'est donc un cas irréductible.

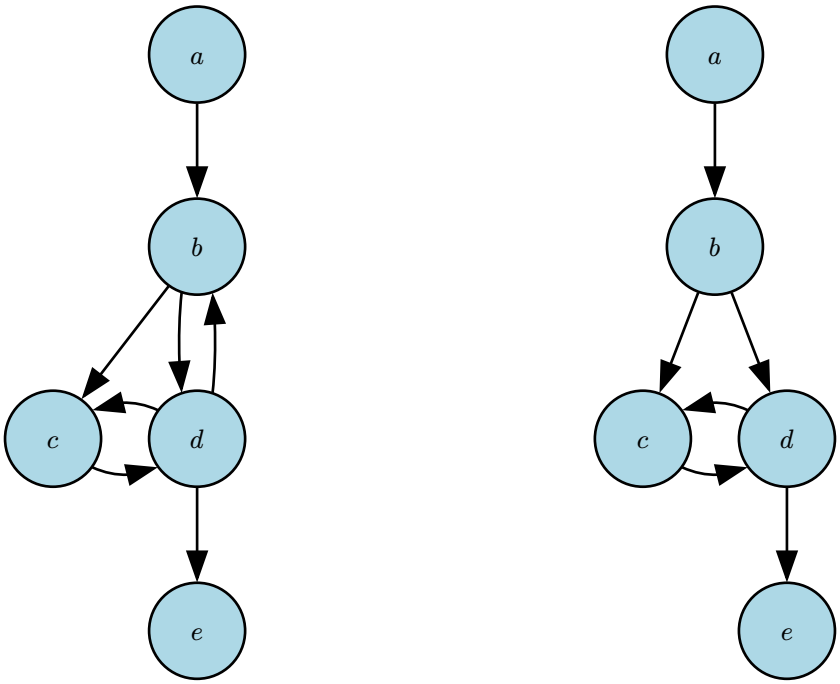


Fig. 4.7. – Graphe irréductible.

Considérons maintenant le graphe de gauche dans Fig. 4.8 qui contient toujours un seul **arc de retour** (celui de d vers b). Si on le retire, on obtient le graphe droite qui est acyclique. C'est donc un cas réductible. Les graphes de flot de contrôle sont une représentation centrale utilisée dans beaucoup de situations. On préfère généralement travailler sur des versions structurées car elles facilitent beaucoup d'analyses et de transformations sur le graphe. Il existe des algorithmes pour passer d'un flot de contrôle non structuré à un flot de contrôle structuré [1]. En particulier, WebAssembly a été conçu de telle sorte que le graphe de flot de contrôle représentant un programme est forcément structuré. Ce n'est pas le cas de langages comme C, ce qui force à transformer le graphe avant de pouvoir les compiler vers WebAssembly.

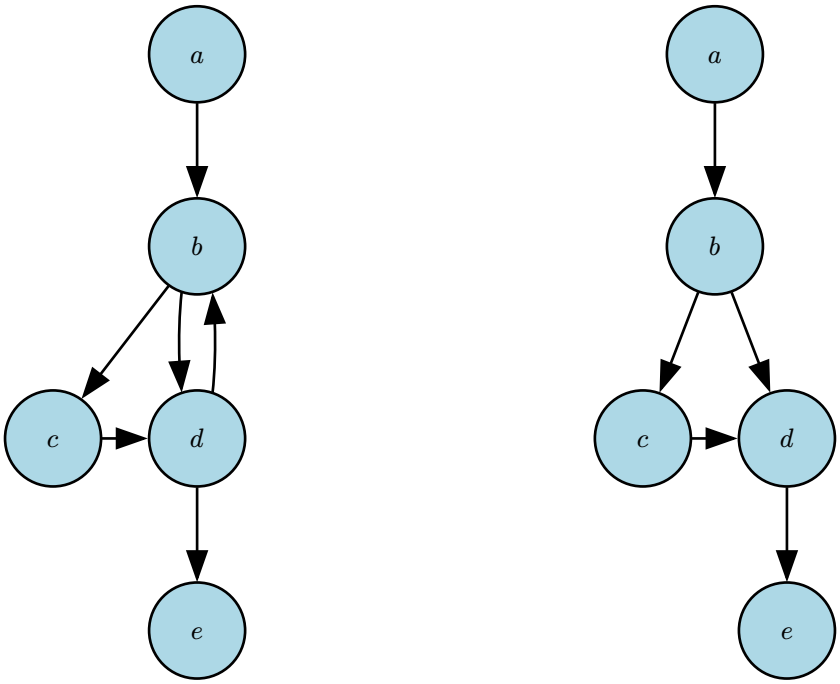


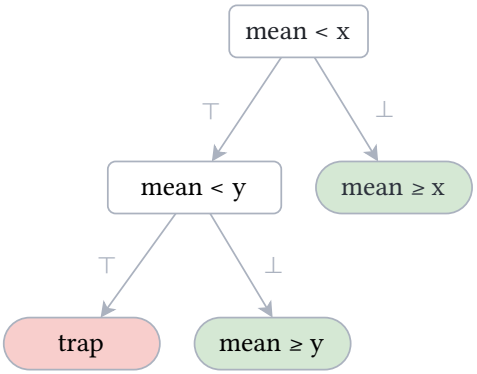
Fig. 4.8. – Graphe réductible.

4.2.3 Arbre d'exécution

Un arbre d'exécution est un moyen de représenter toutes les exécutions possibles d'un programme en fonction de ses valeurs d'entrées. Pour simplifier les choses, on ne va s'intéresser ici qu'aux arbres d'exécution portant sur une seule fonction. Dans ce cas l'arbre d'exécution est similaire, en apparence, au graphe de flot de contrôle. La différence principale est que celui-ci est potentiellement infini, puisqu'il est possible que certaines exécutions d'un programme ne terminent pas. Commençons par un cas fini. On peut voir en Liste 4.1 un programme C assez simple et la représentation de son arbre d'exécution.

```c
unsigned int mean(unsigned int x,
                  unsigned int y) {

  unsigned int mean = (x + y) / 2;
  if (mean < x) {
    if (mean < y) {
      assert(false);
    }
  }
  return mean;
}
```



Liste 4.1. – Arbre d'exécution fini.

Regardons maintenant une fonction avec une boucle :

```c
#include <stdlib.h>

void f(int limit) {
  int i = 0;

  while (i < limit) {
    if (i % 2 == 0) {
      // even case
    } else {
      // odd case
    }
    i++;
  }
}
```

Son arbre d'exécution est représenté dans la Fig. 4.9.

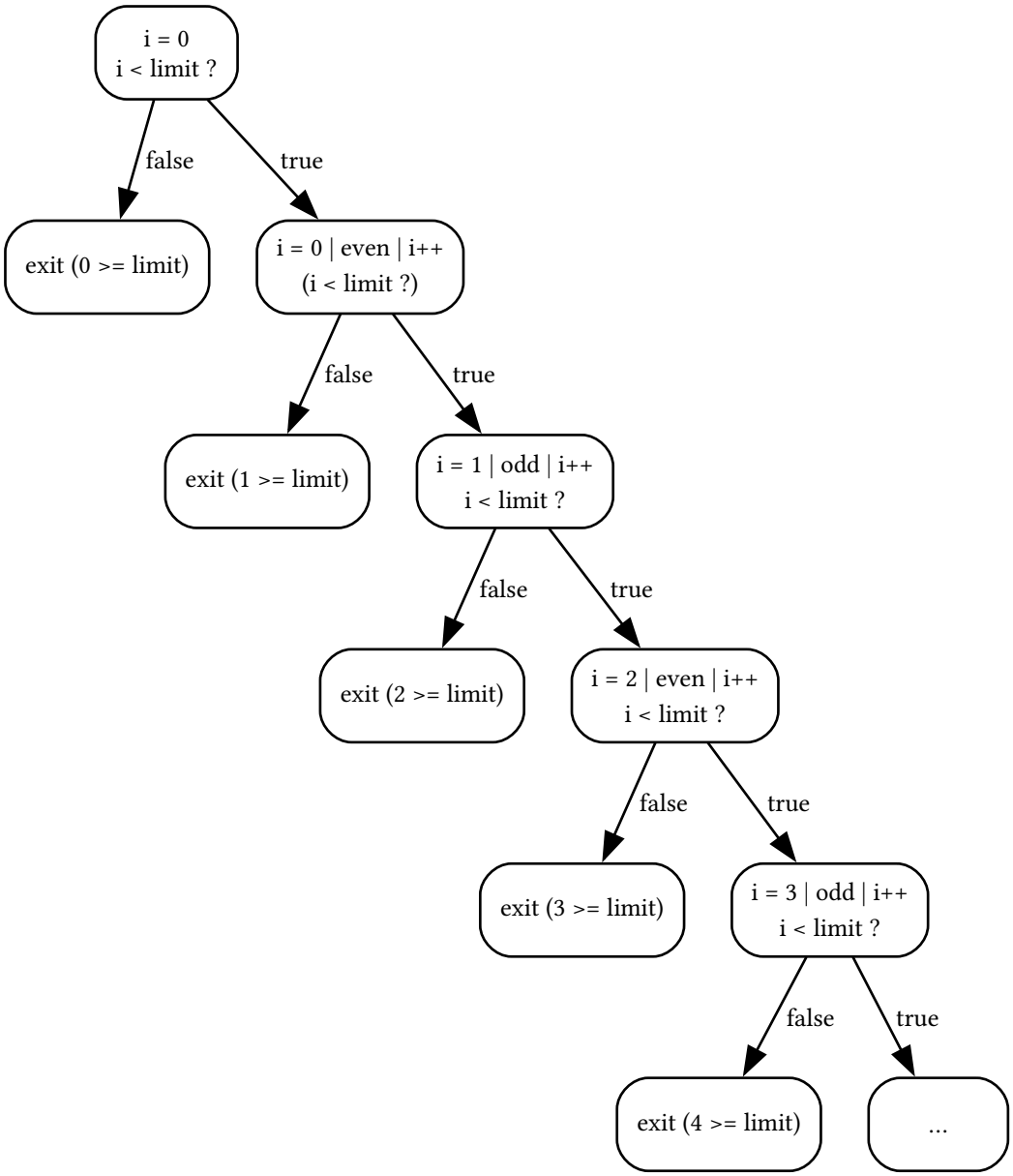


Fig. 4.9. – Arbre d'exécution infini.

On peut donc voir un arbre d'exécution comme un graphe de flot de contrôle dans lequel on a développé les boucles. Par ailleurs, on ne s'intéresse qu'aux cas possibles en pratique. Par exemple, dans l'arbre précédent, nous n'avons pas représenté le cas où `i = 0` et où on aurait exécuté `odd` plutôt que `even`.

4.3 Critères de couverture formels

Nous allons maintenant pouvoir définir différents critères de couverture et faire le lien avec les représentations structurelles vues précedemment. Nous verrons ensuite comment on peut mesure ces différents critères au moyen de l'instrumentation avec des labels.

4.3.1 Définition des critères de couverture formels

Les critères de couverture peuvent être classifiés selon différentes catégories que nous allons étudier séparément.

4.3.1.1 Graphe d'appel

La première catégorie porte sur les fonctions et peut être assimilée à des critères sur le graphe d'appel.

Couverture des fonctions

Ce premier critère est appelé `FC` (*function coverage*). Il cherche simplement à mesurer le nombre de fonctions qui sont exécutées par nos tests. Sur le graphe d'appel, cela correspond à vérifier que chaque sommet du graphe a été exécuté au moins une fois. Par exemple, si notre programme possède deux fonctions f et g et que nos tests n'appellent que la fonction f, on aura alors une couverture de 50%. On suppose bien évidemment qu'on a représenté toutes les fonctions sur le graphe, même celles qui ne sont jamais appelées.

Couverture des appels de fonction

Ce second critère est appelé `FCC` (*function call coverage*). Il cherche à mesurer le nombre d'**appels** de fonctions qui sont exécutés par nos tests. Sur le graphe d'appel, cela correspond à vérifier que chaque **arc** du graphe a été emprunté au moins une fois. Généralement, on simplifie le cas des appels indirects, et un appel indirect correspondant à potentiellement plusieurs fonctions n'est considéré qu'une seule fois.

4.3.1.2 Graphe de flot de contrôle

Cette catégorie porte sur les branchements et les conditions. Elle peut être assimilée à des critères sur le graphe de flot de contrôle.

Couverture des instructions

Ce critère est appelé `Stmt` (*statement coverage*). Il revient simplement à dire que chaque instruction a été exécutée une fois. Ce qui équivaut à peu près à dire que chaque sommet du graphe de flot de contrôle a été exécuté. En effet, si l'on exécute la première instruction d'un sommet, on devrait a priori aussi exécuter toutes les autres, puisqu'il n'y a plus de branchements dans les sommets d'un graphe de flot de contrôle. En réalité, ce n'est pas tout à fait exact, puisqu'on peut imaginer s'arrêter au milieu des intructions d'un sommet, par exemple en cas d'erreur à l'exécution (comme une division par zéro ou une erreur de segmentation).

Couverture des décisions

Ce critère est appelé `DC` (*decision coverage*). Il cherche à mesurer si pour chaque branchement conditionnel, on a pris au moins une fois les deux décisions possibles, à savoir : prendre le branchement ou bien ne pas le prendre. Par exemple, dans le programme C suivant :

```
if (p && q) {
  // ...
}
```

Les tests doivent exécuter le cas où `p && q` est vrai, ainsi que le cas où `p && q` est faux. Par exemple, si l'on exécute seulement l'un des deux, on obtiendra une couverture de 50%. Ce critère correspond donc à un critère qui dirait que chaque arc du graphe de flot de contrôle a été exécuté.

Couvertures des conditions

Ce critère est appelé `CC` (*condition coverage*). Il cherche à mesurer si pour chaque branchement conditionnel, on a exécuté au moins une fois chaque proposition booléenne atomique avec toutes les valeurs possibles. Dit autrement, si l'on reprend notre programme précédent :

```
if (p && q) {
  // ...
}
```

Les tests doivent cette fois-ci exécuter 4 cas différents : le cas où `p` est vrai, le cas où `p` est faux, le cas où `q` est vrai et enfin le cas où `q` est faux. Attention, ce critère n'implique pas le critère `DC`. Par exemple, on peut obtenir 100% sur le critère `CC` avec les exécutions suivantes: `p == true ; q == false` et `p == false ; q = true`, alors que ceux-ci ne donneront que 50% sur `DC` car la condition globale sera toujours fausse. De même, le critère `DC` n'implique pas `CC`, par exemple, avec les exécutions `p == true ; q == true` et `p == true ; q == false`, on obtiendra 100% sur `DC` mais seulement 75% sur `CC`.

On peut donc voir CC comme un critère sur les arcs du graphe de flot de contrôle, mais où l'on utiliserait un graphe de flot de contrôle différent de celui usuel et qui aurait été réécrit pour transformer toutes les conditions en des conditions atomiques. En particulier, dans le cas de Wasm, le critère CC est équivalent à DC puisque toutes les conditions se font sur une valeur booléenne qui a déjà été entièrement calculée. Le critère CC est donc assez particulier dans le sens où deux programmes sémantiquement équivalents peuvent obtenir des scores différents sur des entrées identiques. C'est par exemple le cas de ces deux fonctions C équivalentes :

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

  if (x && y) {
    // ...
  }
}

void g(int x, int y) {
  int condition = x && y;

  if (condition) {
    // ...
  }
}
```

Si l'on exécute f avec les entrées `x == true; y == true` et `x == true; y == false`, on obtiendra 75% de couverture, tandis qu'on obtiendra 100% sur la fonction g avec ces mêmes entrées.

Couvertures des conditions multiples

Ce critère est appelé MCC (*multiple condition coverage*). Il cherche à mesurer si, pour chaque condition, on a exécuté l'ensemble des combinaisons possibles pour toutes les valeurs des conditions atomiques possibles. Par exemple, pour le programme C suivant :

```
if (p && q) {
  // ...
}
```

Pour obtenir 100% de couverture, il faut l'exécuter avec :

1. `p == false; q == false`,
2. `p == false; q == true`,
3. `p == true; q == false`,
4. `p == true; q == true`.

Ce critère implique CC. De plus, il implique généralement DC - sauf dans les cas où la condition globale est en réalité toujours fausse ou toujours vraie, par exemple, si notre condition était `(x && !x)`, auquel cas on ne pourrait jamais rendre la condition vraie et satisfaire DC.

4.3.2 Gestion des critères de couverture par les labels

Nous avons vu différents critères de couverture. Nous allons maintenant voir comment ils peuvent être mesurés à travers la notion de **labels**. L'idée consiste à encoder tout ces différents critères sous une même forme qui va nous permettre de mesurer la couverture lors de l'exécution des tests. Un label est un appel à une fonction qui, lors de l'exécution, va enregistrer le fait que l'on a couvert une partie d'un critère donné.

4.3.2.1 Exemples d'encodage par les labels

Couverture des fonctions

Par exemple, pour le critère FC, qui consiste à couvrir chaque fonction, on peut simplement ajouter un label au début de chaque fonction. Si l'on part du programme C suivant :

```
int f(void) {
  // ...
}
```

```
int g(void) {
  // ...
}
```

On va insérer deux labels, un au début de chaque fonction :

```
int f(void) {
  label("FC", 1, true);
  // ...
}

int g(void) {
  label("FC", 2, true);
  // ...
}
```

La fonction `label` va alors enregistrer l'ensemble des valeurs pour lesquelles elle aura été appelée, et cela nous permettra de dire quel pourcentage des labels correspondant au critère `FC` a été couvert.

Couverture des appels de fonction

Pour le critère `FCC`, si l'on part du programme suivant :

```
int f(void) {
  // ...
}

int g(void) {
  if (...) {
    // ...
    f();
  } else {
    // ...
    f();
  }

  f();
}
```

On insère alors les labels avant chaque appel de fonction :

```
int f(void) {
  // ...
}

int g(void) {
  if (...) {
    // ...
    label("FCC", 1, true);
    f();
  } else {
    // ...
    label("FCC", 2, true);
    f();
  }

  label("FCC", 3, true);
  f();
}
```

Couverture des décisions

Pour le critère `DC`, si l'on part du programme suivant :

```
if (p && q) {
  // ...
}
```

On va alors procéder ainsi :

```
label("DC", 1, p && q);
label("DC", 2, !(p && q));
if (p && q) {
  // ...
}
```

On remarque qu'on a cette fois utilisé l'argument booléen de la fonction `label`. Celui-ci sert à indiquer à la fonction si elle doit considérer le label concerné comme couvert ou non. Un encodage alternatif consiste à placer les labels à l'intérieur des blocs liés à la condition et d'insérer un bloc `else` :

```
if (p && q) {
  label("DC", 1, true);
  // ...
} else {
  label("DC", 2, true);
}
```

Couverture des conditions

Pour le critère `CC`, si l'on part du programme suivant :

```
if (p && q) {
  // ...
}
```

On peut alors encoder le critère ainsi :

```
label("CC", 1, p);
label("CC", 2, !p);
label("CC", 3, q);
label("CC", 4, !q);
if (p && q) {
  // ...
}
```

Couverture des conditions multiples

De même, pour le critère `MCC`, si l'on reprend notre exemple :

```
if (p && q) {
  // ...
}
```

Un encodage possible via les labels est le suivant :

```
label("MCC", 1, !p && !q);
label("MCC", 2, !p && q);
label("MCC", 3, p && !q);
label("MCC", 4, p && q);
if (p && q) {
  // ...
}
```

4.3.2.2 Outils pour la gestion des labels

Bien évidemment, insérer l'entièreté des labels pour un seul critère sur un programme réaliste serait trop laborieux et on ne le fait pas à la main. Il existe des outils qui vont **instrumenter** le programme pour nous. Ces outils attendent en entrée un programme et un ou plusieurs critères, et vont l'analyser pour y insérer les labels correspondant aux critères demandés. Ils vont alors produire un nouveau programme qui contiendra tous les labels.

Langage C

Pour le langage C, l'outil `LAnnotate` de Frama-C permet de faire cela et supporte de très nombreux critères et plein d'options de génération. Il s'utilise ainsi :

```
$ frama-c -lannot=CC,DC file.c
```

On aura alors un fichier `file_labels.c` produit, lequel contiendra les labels pour les critères demandés (ici, `CC` et `DC`). C'est ce fichier qui doit être utilisé lorsque l'on teste notre programme afin de mesurer la couverture.

Wasm

Pour Wasm, Owi supporte les critères les plus courants et s'utilise ainsi :

```
$ owi instrument label --criteria=dc file.wat
```

4.4 Test par mutation

Le test par mutation est une technique de test qui permet de mesurer l'efficacité d'une suite de tests pour détecter des bugs. L'idée est simple :

1. On ajoute des bugs dans le programme.
2. On lance la suite de tests.
3. Si la suite de tests se met à échouer, elle a bien trouvé le bug, sinon, c'est qu'on a trouvé un cas qui n'était pas testé.

Pour ajouter des bugs, on va simplement effectuer des petites modifications dans le programme. Par exemple, on peut :

- échanger des opérateurs booléens comme `&&` et `||` ;
- échanger des opérateurs entiers comme `+`, `-`, `*`.
- changer la valeurs de certaines constantes ;

Il existe énormément de possibilités ! Chaque modification du programme est appelée une **mutation**, et chaque programme contenant une **mutation** est appelé un **mutant**. Une fois que l'on a mis en place l'infrastructure pour générer des mutants, il y a deux approches possibles :

1. Essayer différents mutants jusqu'à trouver un bug, on peut alors écrire un nouveau test à partir de celui-ci.
2. Effectuer l'opération un grand nombre de fois et calculer un **score de mutation** qui correspond au pourcentage de fois où notre suite de test a été capable de détecter un mutant.

4.4.1 Mutaml

En OCaml, un outil appelé **Mutaml** permet de faire du test par mutation très facilement. Il suffit d'indiquer à `dune` que l'on souhaite instrumenter notre projet avec `mutaml` :

```
(executable
  (public_name mytool)
  (instrumentation (backend mutaml)))

(library
  (public_name mylibrary)
  (instrumentation (backend mutaml)))
```

On peut alors compiler notre projet en activant l'instrumentation ainsi :

```
$ dune build --instrument-with mutaml
```

```
        ppx lib.pp.ml
Running mutaml instrumentation on "lib.ml"
Randomness seed: 810959211   Mutation rate: 50   GADTs enabled: false
Created 9 mutations of lib.ml
Writing mutation info to lib.muts
```

Pour un test donné, par exemple `test/mytests.ml`, on peut lancer le processus de test de mutation ainsi :

```
$ mutaml-runner _build/default/test/mytests.exe
```

```
read mut file lib.muts
Testing mutant lib:0 ... failed
Testing mutant lib:1 ... failed
Testing mutant lib:2 ... failed
Testing mutant lib:3 ... failed
Testing mutant lib:4 ... failed
Testing mutant lib:5 ... passed
Testing mutant lib:6 ... failed
Testing mutant lib:7 ... failed
Testing mutant lib:8 ... failed
Writing report data to mutaml-report.json
```

Enfin, on peut générer un rapport avec la commande :

```
$ mutaml-report
```

```
Attempting to read from mutaml-report.json...

Mutaml report summary:
----------------------

 target                          #mutations      #failed      #timeouts      #passed
 ------------------------------------------------------------------------------------
 lib.ml                                   9      88.9%    8     0.0%    0    11.1%    1
 ====================================================================================

Mutation programs passing the test suite:
-----------------------------------------

Mutation "lib.ml-mutant5" passed (see "_mutations/lib.ml-mutant5.output"):

--- lib.ml
+++ lib.ml-mutant5
@@ -11,7 +11,7 @@
 (* A Monte Carlo simulation computing a pi-approximation *)
 let pi total =
   let rec loop n inside =
-    if n = 0 then
+    if n = 1 then
       4. *. (float_of_int inside /. float_of_int total)
     else
       let x = 1.0 -. Random.float 2.0 in

--------------------------------------------------------------------------------
```

Bibliographie

[1] Norman Ramsey. 2022. Beyond Relooper: recursive translation of unstructured control flow to structured control flow (functional pearl). *Proceedings of the ACM on Programming Languages* 6, ICFP (2022), 1-22.