IFSIC

DEA d'informatique

Génération, sélection et optimisation génétique de tests

Présenté par
Benoit Baudry

Effectué au sein de l'équipe PAMPA, IRISA

Encadré par Yves Le Traon

Soutenu le 14 juin 2000

Institut de Formation à l'Informatique et Systèmes de Communication
IRISA – Campus de Beaulieu, Rennes

Remerciements

Je remercie tout d'abord Yves Le Traon, pour ses conseils, ses encouragements et les nombreuses discussions que nous avons eues au cours de ce stage. Je remercie également Jean-Marc Jézéquel pour ses conseils et ses relectures attentives, François Pennaneach, Vu Le Hanh et Hubert Canon.

Enfin, je remercie Jacques Baudry et Françoise Burel qui m'ont aidé à rattacher ce travail au monde du vivant.

Sommaire

Chapitre 1 Introduction

La grande différence entre le paradigme objet et les paradigmes de développement classiques (langages procéduraux et fonctionnels, flots de données) est la notion d'*objet* possédant des caractéristiques partagées (notion de classe) ou héritées (notion d'héritage). La conséquence la plus profonde de la standardisation de ce nouveau paradigme de développement est la remise en cause du cycle de développement logiciel le plus répandu (cycle en V ou « waterfall ») : le test n'est plus séparé de la conception et décomposé en étapes clairement distinctes (test unitaire / intégration / système) mais les étapes de test sont intégrées à la conception et l'accompagnent dans toutes ses phases.

De plus, l'un des principaux intérêts de l'approche objet est la possibilité de réutiliser du code. Ceci se traduit par l'apparition de *composants logiciels réutilisables*. L'idée est alors de développer des logiciels avec ces composants sans connaître leur implémentation. Mais pour cela il faut pouvoir associer au composant une valeur de robustesse, de confiance.

Le sujet étudié au cours de ce stage se place dans le cadre particulier de la conception par contrats [Meyer92 , Jezequel99] et du développement de composants *autotestables*. L'idée principale est que la qualité du composant est liée à la qualité de ses contrats combinée à celle de ses tests. Un composant fiable est alors un composant dont le code est vérifiable à tout instant par des tests efficaces et dont les contrats sont suffisamment complets pour révéler les anomalies.

La méthode développée [LeTraon99] consiste donc à embarquer les tests dans le code du composant, ce qui le rend autotestable puisqu'il peut exécuter lui-même ses tests. Ceci correspond à ce qui était dit plus haut : la planification des tests et la conception du composant vont être liées et ils vont se développer en même temps. Ensuite, la qualité du composant est évaluée par la qualité des tests embarqués calculée grâce à une *analyse de mutation* [DeMillo78]. Pour effectuer cette analyse, des erreurs simples sont injectées dans le programme à tester, ce qui nous donne un ensemble de programmes erronés appelés *mutants*. Les tests sont ensuite exécutés sur chaque mutants et ainsi un *score de mutation*, correspondant à la proportion de mutants détectés par un test, est calculé pour chaque test.

Une étude récente [Deveaux2000] a montré qu'il est assez facile d'obtenir un ensemble de tests initial ayant score raisonnable (60%), mais améliorer cet ensemble initial à la main, pour augmenter le score réclame un surcoût important. En considérant ce problème d'amélioration du score comme un problème d'optimisation de l'ensemble de tests initial, et en prenant cet ensemble comme germe, nous voulons essayer d'utiliser un *algorithme génétique*. Au cours de ce stage nous avons modélisé le problème pour pouvoir appliquer un algorithme génétique, puis un outil mettant en œuvre ce modèle a été développé. Nous présentons ici les résultats des expériences effectuées avec cet outil, et les limites de l'approche que nous avons pu mettre en évidence. Nous proposons ensuite un nouveau modèle qui semble plus adapté à l'optimisation de tests, et qui conjugue le caractère semi-aléatoire des algorithmes génétiques avec des aspects d'optimisation en temps d'exécution et en vitesse de convergence.

Enfin, puisque nous nous plaçons dans un cadre où tests et éléments de spécification sont embarqués, il nous a paru important d'évaluer l'intérêt général d'un telle approche. Ainsi,

dans la dernière partie de ce rapport, nous proposons une analyse du gain en termes de robustesse et de fiabilité d'une approche fondée sur la conception par contrats.

Dans le chapitre 2, nous présentons le cadre général dans lequel s'est déroulée notre étude. Nous rappelons rapidement en quoi consiste l'activité de tests, puis nous détaillons les techniques de mutation. Dans le chapitre 3, nous proposons une première approche d'automatisation de l'optimisation de tests grâce à un algorithme génétique, et présentons des résultats d'expériences qui nous ont conduits à proposer une autre méthode. Dans le chapitre 4, nous présentons cette nouvelle méthode qui limite l'indéterminisme, en introduisant de la mémoire dans le processus, ce qui permet un gain important en temps d'exécution. Enfin, le chapitre 5 aborde les intérêts plus globaux de la méthode des composants autotestables.

Chapitre 2 Cadre général

Dans ce chapitre, nous présentons rapidement le domaine du test, puis nous nous concentrons sur le test des composants objets et sur la méthodologie des composants autotestables qui constitue l'environnement dans lequel se déroule toute l'étude présentée dans ce rapport. Puis nous expliquons comment qualifier et rendre fiable les composants grâce aux techniques de mutation de DeMillo [DeMillo78]. Nous terminons ce premier chapitre par un exposé précis des problèmes que nous abordons dans la suite.

2.1 Présentation générale du domaine du test

Le test constitue le moyen principal de validation d'un logiciel. Il prend place à la fin du processus de développement et s'effectue en étapes successives :

- le *test unitaire* : une unité est la plus petite partie testable d'un programme, c'est souvent une procédure ou une classe dans les programmes à objet
- le *test d'intégration* : consiste à assembler plusieurs unités et à tester les erreurs liées aux interactions entre ces unités
- le *test système* : teste le système dans sa totalité en intégrant tous les sous-groupes d'unités testés lors du test d'intégration

A chaque étape, il s'agit de générer des *jeux d'essais* ou *cas de test*, de les exécuter, d'analyser la validité des résultats produits, et finalement de localiser et de corriger les fautes.

Le test joue un rôle à la fois dans l'obtention et dans l'évaluation de la qualité d'un logiciel. Lors des premières phases de test — mise au point et tests unitaires — l'objectif du test est d'améliorer la correction de l'implémentation. Pour cela le logiciel est exécuté avec l'intention de révéler des fautes c'est-à-dire de provoquer des défaillances. Chaque fois qu'il se produit une défaillance, les fautes à l'origine de celle-ci sont recherchées et supprimées. On attend légitimement de la suppression de ces fautes une amélioration de la correction du logiciel. L'efficacité de la technique dépend de l'aptitude des ingénieurs de test à mettre au point des jeux d'essais ayant une forte probabilité de révéler, par une défaillance, la présence de fautes.

Lors des dernières phases de test — tests d'intégration et tests du système — l'objectif des tests tend à évaluer la qualité (fiabilité, performances, etc.) de la réalisation.

Quelle que soit la technique de test, tester un logiciel consiste à l'exécuter avec des entrées prédéterminées — les jeux d'essai — et à comparer les résultats obtenus à ceux attendus. Il faut donc dans un premier temps avoir construit ou généré ces jeux d'essai.

Une différence entre les résultats obtenus et ceux attendus pour les entrées sélectionnées est une défaillance. Pour déterminer qu'il y a différence, il faut avoir au préalable déterminé un *oracle*, c'est à dire la valeur attendue comme correcte à l'exécution du jeu d'essai.

Une défaillance apporte la preuve que l'implémentation testée est incorrecte. Autrement dit, une défaillance témoigne de la présence dans le texte du logiciel de fautes qui ont été introduites lors du codage ou de la conception. Il s'agit alors de corriger la faute, après l'avoir localisée.

Lorsqu'un logiciel est soumis au test, il n'y a aucun moyen (à moins de pouvoir le tester exhaustivement sur tout son domaine d'entrée) de s'assurer qu'il ne contient absolument aucune faute. Il faut donc décider à un certain moment d'arrêter les tests en espérant avoir obtenu une confiance satisfaisante dans le logiciel sous test. C'est le problème du critère d'arrêt des tests.

En résumé, toute technique de test doit répondre à trois problèmes principaux, et peut être caractérisée en fonction de ceux-ci :

- problème de la génération des jeux d'essai,
- problème de l'oracle,
- problème du critère d'arrêt.

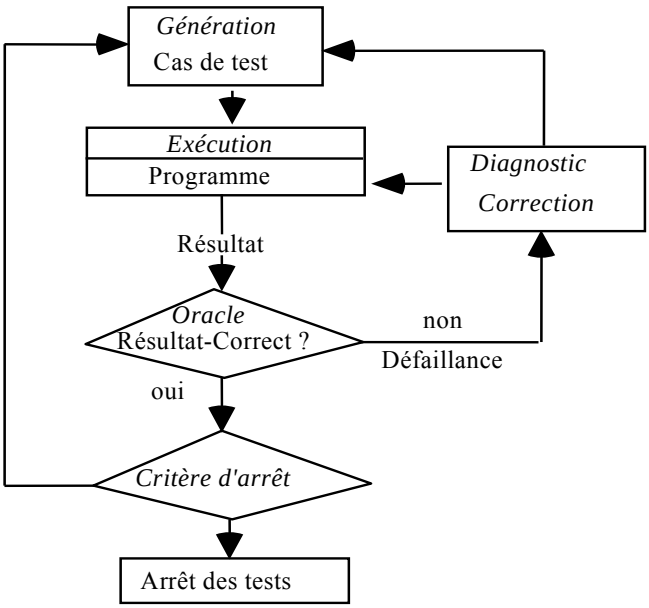


fig. 2.1 schéma global de l'activité de test

La figure 2.1 résume le schéma global de l'activité de test :

- *Génération de cas de tests* : cette étape consiste à trouver des données en entrée pour le programme à tester
- *Exécution* : le programme est exécuté avec les données d'un cas de test en entrée
- *Oracle* : l'oracle est une fonction qui doit permettre de distinguer une sortie correcte du logiciel
- Si l'oracle a détecté une *défaillance*, il faut *corriger et ré-exécuter* le programme avec le même cas de test pour vérifier que la correction a effectivement éliminé la faute
- Si l'oracle n'a pas détecté de défaillance, soit le critère d'arrêt est vérifié et la phase de test du logiciel est terminée, soit le programme est exécuté avec un nouveau cas de test.

Les techniques de test

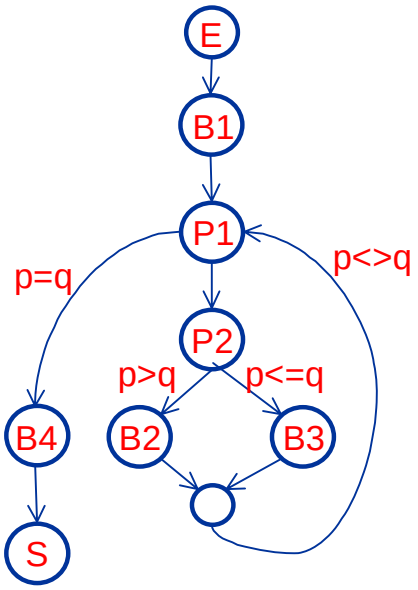
Il existe deux grandes catégories de techniques de test, le test fonctionnel et le test structurel [Beizer90]. Le test fonctionnel est basé uniquement sur la connaissance d'une spécification la plus formelle possible du programme et des services attendus. Cette technique est intéressante car si l'implémentation change, on peut retester avec les mêmes jeux de test, et développer les programmes de test en parallèle avec l'implémentation. Cependant, cette

technique n'assure pas que toutes les parties du code aient été testées et, à l'inverse, ne permet pas d'éviter la redondance de certains tests (en testant deux fonctionnalités différentes, des parties du code peuvent être testées plusieurs fois).

Le test structurel est basé sur la connaissance de la structure interne du logiciel. Cette technique permet d'obtenir un modèle structurel du logiciel (graphe de contrôle, flot de données), et donc de modéliser une structure pouvant être couverte selon certains critères: couverture de toutes les instructions, couverture de toutes les branches. Ceci facilite la conception par la connaissance du but à atteindre et des moyens de l'atteindre (cf. exemple de la fig. 2.2). De plus cette technique renforce la validité des tests puisqu'on sait quelle partie du code a été testée.

Exemple : PGCD de 2 nombres
Précondition: p et q entiers naturels positifs lusau clavier
Effet : result = pgcd(p, q)
En pseudo-langage

```
pgcd: integer is
local p,q : integer;
do
    read(p, q)                B1
    while p<> q do            P1
        if p > q              P2
                              then p := p-q   B2
                              else q:= q-p    B3
        end -- if
    end -- while
    result:=p                 B4
end-- pgcd
```



Toutes les instructions:
(E, B1, P1, P2, B2, P1, B4, S)
(E, B1, P1, P2, B3, P1, B4, S)
Tous les arcs : idem
Tous les chemins élémentaires (1-chemin) :
idem + (E, B1, P1, B4, S)
Tous les 2-chemins :
idem +
(E, B1, P1, P2, B2, P1, B2, P1, B4, S)
(E, B1, P1, P2, B2, P1, B3, P1, B4, S)
(E, B1, P1, P2, B3, P1, B2, P1, B4, S)
(E, B1, P1, P2, B3, P1, B3, P1, B4, S)

fig. 2.2 exemple pour le test structurel

L'exemple de la figure 2.2 illustre la possibilité de construire un graphe de contrôle à partir du code d'un programme. A partir de ce graphe, on peut écrire des tests on fonction du critère de couverture choisi. Pour couvrir toutes les instructions, il faudra deux tests, alors que pour couvrir tous les chemins, il en faudra un troisième.

Une technique structurelle particulière pour la génération de tests est l'analyse de mutation. Cette méthode permet, par injection de fautes simples dans le logiciel, de construire des tests capables de détecter ces fautes. On suppose alors que ces tests sont capables de

détecter les fautes plus complexes présentes dans le logiciel. Nous reviendrons sur cette technique dans la suite de ce rapport.

Dans la section suivante, nous présentons plus particulièrement le test de composants objets dans un cadre de conception par contrats. Nous exposons d'abord une méthodologie basée sur des composants autotestables [LeTraon99], puis nous expliquons comment une analyse de mutation nous permet de qualifier un composant et de renforcer ses contrats.

2.2 Qualification des tests et des contrats pour des composants objets

Nous présentons ici la méthode des composants autotestables. L'idée est que pour réutiliser un composant de manière sûre, il faut pouvoir accorder une mesure de confiance à ce composant. Dans cette section nous présentons l'analyse de mutation comme méthode pour qualifier les composants autotestables en assurant une bonne qualité des tests et des contrats.

2.2.1 Définitions

Avant d'expliquer notre approche, précisons le vocabulaire utilisé.

Définition 2.1 : *Composant*. Dans notre étude nous considérons qu'un composant est une classe.

Nous donnons ensuite les définitions de deux notions importantes de la programmation orientée objets que sont l'héritage et la clientèle. [Meyer88]

Définition 2.2 : *Relation d'héritage*. Une classe hérite d'une ou plusieurs autre(s) classe(s) si elle est conçue comme une extension ou une spécialisation de ces classes. En particulier, ce mécanisme permet le partage de méthodes et d'attributs entre classes.

Définition 2.3 : *Relation de clientèle*. Une classe C1 est cliente d'une classe C2 si elle utilise des services de C2 définis dans l'interface.

Enfin, nous définissons un système.

Définition 2.4 : *Système*. Un système est l'assemblage de plusieurs composants reliés entre eux par des relations de clientèle ou d'héritage.

Nous présentons maintenant les composants autotestables.

2.2.2 Les composants objets auto-testables

Le succès de l'approche objet tient beaucoup de la notion de composant réutilisable. La réutilisation de composants logiciels devient un enjeu crucial tant pour éviter de réécrire inutilement du logiciel, que pour réduire les coûts de développement. On cherche alors à développer la notion de composant objet « sur étagère », et il faut pour cela pouvoir mesurer la confiance que l'on peut avoir en un composant. La solution envisagée dans le cadre du projet Pampa consiste à considérer chaque composant comme un tout possédant une spécification, une implémentation et des tests qui lui permettent de se tester et de tester ses ancêtres [LeTraon99] (fig. 2.3).

Les tests sont donc considérés comme faisant partie du composant. Ceci permet de fournir une première solution au problème de la réutilisation et de l'évolution du composant puisqu'il porte la capacité à se tester (autotest) qui peut être appelée depuis le système.

Trois cas de figures peuvent être pris en compte pour un composant C dans un système S :

- modification de l'implémentation de C : l'autotest permet de tester la nouvelle implémentation, la spécification ne change pas,
- ajout de nouvelles fonctionnalités à C : l'autotest doit être lancé pour assurer le test de non-régression puis complété pour tester les nouvelles fonctionnalités. La spécification doit être complétée.
- réutilisation du composant dans un système S : quand on modifie C, tous les composants clients de C doivent s'autotester pour garantir la non-régression.
- intégration : lors de l'ajout d'un composant C dans un système, l'autotest de C doit être lancé depuis le système pour vérifier sa bonne intégration (héritage et clientèle).

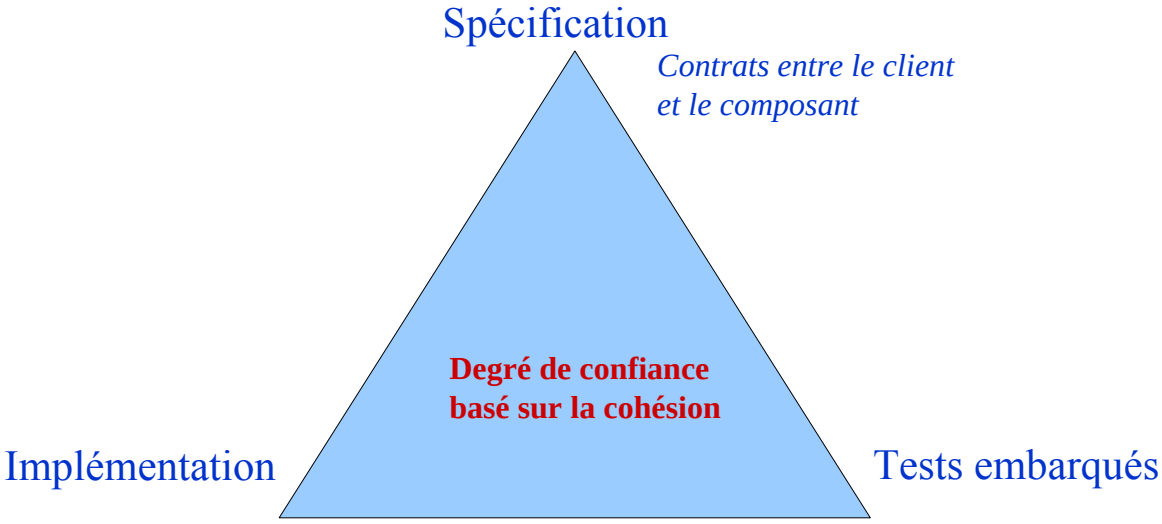


fig. 2.3 Composant autotestable – la vue en triangle

L'approche développée apporte donc des solutions structurelles au problème du test d'intégration, du test de non-régression, de l'évolution des composants et de leur réutilisation. Elle a pour principale limitation le fait qu'elle ne permet pas le test fonctionnel du système, et qu'elle ne modélise pas explicitement les aspects comportementaux, dynamiques du système. C'est ce dernier aspect, qu'il faudrait prendre en compte de manière explicite (aspects de communication entre objets) en proposant des modèles et des méthodes de test spécifiques.

L'un des aspects de la conception de composants autotestables et réutilisables est d'assurer la cohésion entre l'implémentation, les tests et les contrats. Avant d'embarquer les tests dans le composant, il faut indiquer avec quelle qualité le composant a été testé. D'autre part, nous voulons pouvoir assurer une bonne qualité des contrats. Ces deux aspects (qualification des tests et améliorations des contrats) sont mesurés à l'aide d'une analyse de mutation. Nous avons vu plus haut que cette analyse consiste à injecter des fautes dans le programme à tester, nous allons maintenant détailler cette technique.

2.2.3 La qualification des tests par analyse de mutation

Nous présentons ici l'analyse de mutation et la manière dont nous l'utilisons pour qualifier les composants autotestables à partir de la qualification des test et des contrats.

L'analyse de mutation

L'analyse de mutation est une technique qui a d'abord été proposée par DeMillo [DeMillo78] et qui consiste à créer un ensemble de versions erronées du programme à tester appelées des *mutants*. On parle *d'analyse de mutation* lorsque la mutation est utilisée pour qualifier un ensemble de tests. Par ailleurs, lorsque les tests sont conçus en vue de détecter les programmes mutants, on parle de *test par mutation*.

En pratique, l'ensemble des mutants est créé en soumettant le programme à des opérateurs de mutation qui insèrent des erreurs simples (changement de signe des constantes, changement de sens d'une inégalité…). En effet, DeMillo limite l'analyse à l'injection d'erreurs simples et fait l'hypothèse que si un ensemble de tests peut détecter toutes les erreurs simples, alors il pourra détecter des erreurs plus complexes dans le programme initial. Un ensemble de mutants est ainsi obtenu, chacun contenant une erreur simple. Le but pour le testeur est alors d'écrire une série de tests qui permette de détecter tous les programmes mutants. Un test peut détecter un mutant de deux manières.

Définition 2.6 : *Détection de mutants par comparaison des comportements*. Une façon de distinguer un programme mutant du programme initial est d'exécuter un test sur le programme initial puis sur le programme mutant et de comparer les résultats des exécutions. Un test qui s'exécute avec la séquence de valeurs ($x_1,\ldots,x_n$) détecte un mutant Mut de la classe C si :

$$Mut(x_1,...,x_n) \neq C(x_1,...,x_n).$$

Définition 2.5 : *Détection de mutants par un oracle*. Une autre manière de détecter un programme mutant est d'avoir un prédicat qui vaut vrai si une anomalie est détectée. Ce prédicat est appelé un oracle. On distingue deux types d'oracle : des assertions explicites dans le programme de test, ou des contrats (pré post conditions et invariants de classe).

Remarque : la détection de mutants par comparaison de comportements est la technique classique utilisée pour l'analyse de mutation, et c'est aussi cette technique que nous utilisons plus loin pour appliquer un algorithme génétique. Nous avons introduit le deuxième type de détection comme un outil pour l'amélioration des contrats. Ces deux méthodes, dans le cadre particulier de la conception orientée objets, sont détaillées plus loin.

Parmi les mutants générés, certains sont *équivalents* au programme initial.

Définition 2.7 : *Mutant équivalent*. Soit D_i le domaine des valeurs d'entrée , on dit qu'un mutant est équivalent au programme initial s'il n'existe aucune séquence de valeurs de D_i qui permette de distinguer le programme mutant du programme initial. Un mutant Mut est équivalent à une classe C si : $\neg(\exists(x_1,...,x_n) \in D_i) / Mut(x_1,...,x_n) \neq C(x_1,...,x_n)$.

Sur l'exemple de la figure 2.4, le programme mutant a remplacé I par MinVal dans la seconde instruction. Or, l'instruction précédents affecte à MinVal la valeur I. A ce point du programme MinVal et I auront donc toujours la même valeur. Le programme mutant est équivalent au programme initial.

Le programme original

```
Function Min (I , J : integer)
          return integer IS
   MinVal : integer;
Begin
  Minval:=I;
  if (J < I) then MinVal:=J;
  return (MinVal)
End;
```

Le programme mutant

```
Function Min (I , J : integer)
          return integer IS
   MinVal : integer;
Begin
  Minval:=I;
  if (J < MinVal) then MinVal:=J;
  return (MinVal)
End;
```

fig. 2.4 Exemple de mutant équivalent

Il est important de déterminer et d'éliminer les mutants équivalents lors d'une analyse de mutation. En effet, il est impossible d'écrire un test qui distingue un mutant équivalent du programme initial. Dans [Offut97], les auteurs expliquent comment détecter certains mutants équivalents automatiquement. Cependant, cette détection se fait généralement à la main, ce qui augmente le coût de l'analyse de mutation.

Une fois détectés les mutants équivalents, les tests sont exécutés sur le reste des mutants. Si un test détecte un mutant, soit grâce à un oracle, soit par une différence de comportement, le mutant est dit *tué* par le test, sinon le mutant est *vivant*. Quand un mutant a été tué on peut le retirer du processus de test car les fautes qu'il représentait ont été détectées et il a permis de repérer un test pertinent. Cependant, si on laisse le mutant dans le processus, on va exécuter chaque test sur tous les mutants, et on va pouvoir associer à chaque test un *score de mutation.*

Définition 2.8 : *Signature d'un test*. La signature d'un test, *Sign(t)*, est l'ensemble des mutants tués par ce test:

$$Sign(t) = \{mutants\ tués\ par\ t\}$$

Le score de mutation d'un test est calculé à partir de sa signature.

Définition 2.9 : *Score de mutation*. Soit m le nombre total de mutants d'un programme, le score de mutation de t est donné par la formule suivante:

$$SM(t) = \frac{card(Sign(t))}{m}$$

On peut aussi calculer la signature et le score de mutation d'un ensemble de tests.

Définition 2.10 : *Valeurs globales d'un ensemble de tests*. La signature globale d'un ensemble $T = \{t_1 \ldots t_n\}$ de tests est l'union des signatures de tous les tests de l'ensemble:

$$sign(T) = \bigcup_{i=1}^{card(T)} Sign(t_i)$$

et le score global est: $SM(T) = \frac{card(Sign(T))}{m}$

Le processus global de qualification des tests par mutation consiste d'abord à générer tous les mutants du programme. Puis à exécuter les tests sur les mutants. Si certains mutants sont encore vivants, un diagnostique peut permettre de déterminer l'origine de la non détection de certains mutants:

- le mutant peut être équivalent au programme initial

- les tests peuvent être trop faibles pour détecter tous les mutants. Dans ce cas l'utilisateur peut ajouter des cas de test.
- la spécification peut être incomplète, il faut alors renforcer les contrats
-

Une fois que l'utilisateur a effectué ce diagnostique, il réessaie de tuer les mutants vivants. Ce processus, représenté figure 2.5, s'arrête lorsqu'un score de mutation satisfaisant est atteint.

Un outil mettant en œuvre a ce processus a été développé. Cet outil appelé µSlayer, appelle un générateur de mutants pour des programmes Eiffel, développé à l'IRISA par Vu Le Hanh, puis calcul le score de mutation d'un tests. Dans la suite de cette section, nous précisons quelques détails liés à l'utilisation de la mutation dans le cadre particulier des programmes objets conçus par contrats, puis nous détaillons les opérateurs utilisés par µSlayer.

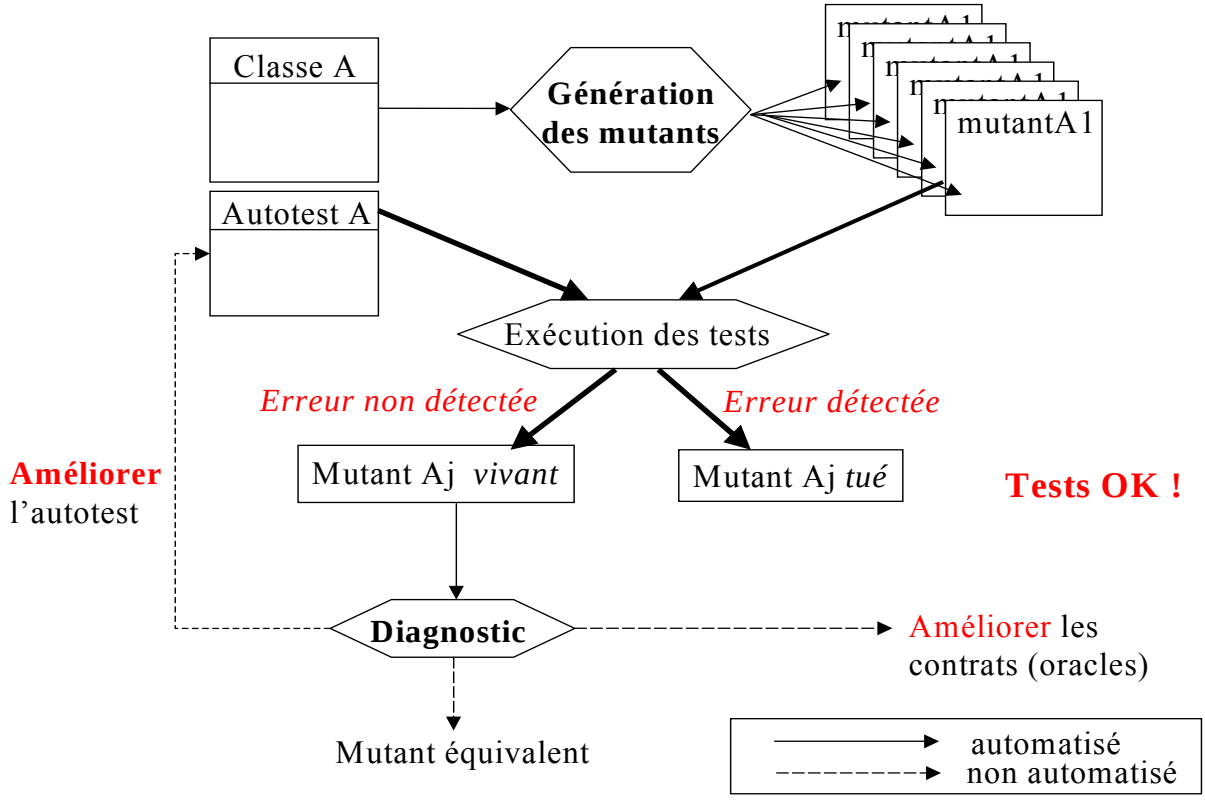


fig. 2.5 : processus global de génération de tests par mutation

La mutation dans la qualification des tests et des contrats

Tout d'abord, nous précisons ici quelles sont les deux façons de tuer les mutants dans le cadre particulier de programmes objets, puis nous expliquons comment nous utilisons l'analyse de mutation et le test par mutation dans le cadre des composants autotestables, pour qualifier les tests et améliorer la qualité des contrats.

Détection d'un mutant par un oracle: dans notre cas, l'oracle est embarqué dans les composants sous forme de contrats. En effet, les contrats (pré et post conditions) décrivent les résultats attendus de le part de l'implémentation avant et après l'exécution d'une méthode, et si ces conditions ne sont pas vérifiées, une exception est levée au cours de l'exécution. Ainsi, si une exception est levée lors de l'exécution d'un test sur un mutant, le mutant est tué.

Définition 2.11 : *Etat d'un objet*. L'état d'un objet à un instant t est l'ensemble des valeurs de ses attributs à cet instant. Pour des attributs complexes comme un tableau, on distingue sa valeur superficielle (l'adresse du tableau), de sa valeur en profondeur (une liste de valeurs décrivant la valeur de tous ses éléments). Nous utilisons ici la valeur en profondeur des attributs.

Détection d'un mutant par comparaison des comportements: pour les programmes procéduraux classiques cette méthode consiste à comparer les traces sur la sortie standard du programme initial et du mutant. Dans notre cas, ceci ne va pas être possible, car un composant, une classe, a rarement de sorties. Par contre, un objet d'une classe se trouve dans un certain état à la fin de l'exécution. Pour tuer des mutants par comparaison de comportements, nous avons donc comparé la valeur des attributs d'un objet de la classe initiale et celle d'un objet de la classe mutante après exécution d'un test sur chacune de ces classes.

Remarque : Grâce à la notion d'état d'un objet, spécifique à la programmation orientée objets, nous avons une comparaison de comportement beaucoup plus forte qu'une simple comparaison de traces. En effet, l'état d'un objet décrit en profondeur toutes les données et structures de données manipulées par l'objet testé, alors qu'une trace ne donne que le résultat d'une exécution mais aucun renseignement sur l'état interne du programme.

Dans le cas des composants autotestables, notre but est de générer un ensemble de test obtenant un bon score global (entre 90% et 95%) qui nous permet d'assurer que le composant a été correctement testé. Nous utilisons une analyse de mutation sur ce premier ensemble pour qualifier les tests d'un composant.

D'autre part, à partir de cet ensemble de tests, nous utilisons le test par mutation pour améliorer la qualité des contrats. Ainsi, nous essayons d'obtenir un score identique avec cet ensemble de tests, mais en ne tuant les mutants que grâce aux contrats. Ce but est atteint lorsque les contrats du composant sont le plus complets possibles.

Les techniques de mutation nous permettent donc d'embarquer des tests de bonne qualité, et nous permettent d'optimiser les contrats qui déterminent la robustesse du composant, comme nous le montrons plus loin dans ce rapport. La *valeur de confiance associée au composant* sera alors liée au score de mutation global de ses tests embarqués et à la qualité de ses contrats.

Nous détaillons maintenant les opérateurs utilisés par μSlayer.

Détail des opérateurs de mutation utilisés pour la qualification des composants

Type	Description
EHF	Exception Handling Fault
AOR	Arithmetic Operator Replacement
LOR	Logical Operator Replacement
ROR	Relational Operator Replacement
NOR	No Operation Replacement
VCP	Variable and Constant Perturbation
MCR	Methods Call Replacement
RFI	Referencing Fault Insertion

Table 1: Opérateurs de mutation pour les programmes objet

Dans notre étude, nous appliquons l'analyse de mutation à la qualification de tests pour des programmes objet. Certains opérateurs sont donc liés à ce type de programme, d'autres sont des opérateurs couramment utilisés par les différents outils de mutation. Tous les opérateurs sont donnés dans la Table 1.

EHF : Déclenche une exception quand il est exécuté. Cet opérateur permet de forcer la couverture du code.

AOR : Remplace les occurrences de "+" par "-" et vice et versa.

Opérateur arithmétique	Remplacé par
+	-, *
-	+, / (or div)
*	/ (or div), +
/	*, -
Div	-, mod
Mod	-, div

LOR : Chaque occurrence d'un opérateur logique (et, ou, non-et, non-ou, ou exclusif) est remplacée par chacun des autres opérateurs. De plus, l'expression est remplacée par VRAI et FAUX.

ROR : Chaque occurrence d'un opérateur relationnel (<, >, <=, >=, =, /=) est remplacée par chacun des autres opérateurs. Pour éviter d'avoir un trop grand nombre de mutants, on applique les règles suivantes :

Opérateur relationnel	Remplacé par
<	<=
>	>=
/=	< et >
<=	<
>=	>
=	>= et <=

NOR : Remplace chaque instruction par l'instruction Null.

VCP : Les valeurs des constantes et des variables sont légèrement modifiées pour effectuer une analyse de sensibilité (sensitivity analysis) analogue à ce que propose Voas dans [Voas92]. Chaque constante ou variable de type arithmétique est incrémentée ou décrémentée de un. Chaque booléen est remplacé par son complément.

MCP : Les appels de méthode sont remplacés par un appel à une autre méthode qui a la même signature.

RFI : Force à void la référence à un objet après sa création. Supprime une instruction de clonage ou de copie. Introduit une instruction de clonage pour chaque affectation d'une référence.

Les opérateurs de mutation AOR, LOR, ROR et NOR sont des opérateurs de mutation traditionnels étudiés dans [DeMillo91] et [Offut96]. Nous avons introduits les autres opérateurs pour le test dans le domaine des programmes objet. L'opérateur RFI injecte des fautes de référence à des objets (fautes d'aliasing) qui sont spécifiques au domaine de la programmation objet :

- la référence à un objet est forcée à void
- les instructions de duplication d'objets sont supprimées
- chaque affectation d'une référence à un objet est précédée de la duplication de cet objet

Les fautes introduites par l'opérateur RFI sont plus difficiles à détecter que celles introduites par les autres opérateurs.

2.3 Le premier problème abordé : l'optimisation automatique de tests

Nous avons observé qu'il est facile d'écrire des tests pour un composant qui ont un score global de 60%, mais qu'augmenter ce score demande plus d'efforts. Le premier problème que nous nous posons dans les chapitres suivants est donc celui de l'optimisation automatique de cet ensemble de tests.

Pour résoudre ce problème, nous proposons d'abord, dans le chapitre 3, une méthode basée sur un algorithme génétique. Nous présentons donc les algorithmes génétiques de manière générale, puis la façon dont nous avons modéliser notre problème pour les appliquer. Nous présentons ensuite des résultats d'expériences faites sur ce premier modèle et les conclusions qui nous ont amenées à modifier ce premier modèle.

Dans le chapitre 4, nous développons les modifications apportées à notre modèle ainsi que le nouvelle méthode qui a découlé de ces modifications. La grande différence entre les deux méthodes est l'aspect de mémorisation que nous avons introduit dans la nouvelle approche. Cet aspect permet surtout des économies en temps de calcul, et diminue le temps de convergence vers la solution.

2.4 Le second problème : intérêt global d'une approche orientée objets avec contrats

Les composants autotestables ont été développés pour permettre la réutilisation sûre et efficace de composants objets. Le but de cette méthode est donc de pouvoir qualifier les composants, et pour cela nous embarquons des tests fiables et des contrats les plus complets possibles. Dans le dernier chapitre, nous analysons l'apport d'une telle approche en termes de fiabilité et de robustesse.

Chapitre 3 Optimisation et génération de tests, une première approche

La confiance que l'on peut avoir en un composant dépend de la fiabilité de ses tests et de ses contrats. Une étude récente [Deveaux2000] a montré qu'il est facile, pour un programmeur, d'écrire, à la main, des tests obtenant un score de mutation de 60%, mais l'obtention d'un score plus élevé réclame un effort important. Le but de notre étude est alors d'automatiser l'optimisation d'un ensemble initial de tests.

Dans ce chapitre, nous proposons une première méthode pour résoudre ce problème. Cette approche est basée sur un algorithme génétique qui devra optimiser le score de l'ensemble initial de tests. Nous commençons par présenter globalement les algorithmes génétiques, puis la manière dont nous les avons appliqués à notre problème. Nous donnons ensuite des éléments d'analyse théorique de ces algorithmes en général et dans notre cas. Nous terminons ce chapitre en présentant les expériences que nous avons effectuées sur ce modèle, et quelles conclusions nous en avons tirées.

3.1 Les algorithmes génétiques

Les algorithmes génétiques [Goldberg89,Alliot94] ont d'abord été développés par John Holland [Holland78] dont le but était d'expliquer les systèmes naturels et de concevoir des systèmes artificiels basés sur ces mécanismes naturels. Les algorithmes génétiques sont donc des algorithmes d'optimisation basés sur les principes de la sélection naturelle. Dans la nature, les individus qui sont le mieux adaptés à leur environnement (qui sont capables d'échapper aux prédateurs, de se protéger du froid…) se reproduisent, et grâce aux croisements et à la mutation, la génération suivante sera encore mieux adaptée. C'est exactement ainsi que fonctionnent les algorithmes génétiques : à partir d'un critère objectif ils peuvent sélectionner les meilleurs individus qui se reproduiront pour fournir la génération suivante.

Les algorithmes génétiques sont relativement simples à programmer, et sont particulièrement efficaces lorsque l'espace de recherche d'une solution est très grand et qu'il existe des minimums locaux. De plus, la recherche d'un optimum se fait sur un ensemble de points et non sur un point isolé.

Pour écrire un algorithme génétique, il faut une population de départ formée d'individus, tirés au hasard ou de manière déterministe, eux-mêmes formés de gènes (un gène peut-être un bit, une lettre…). Il faut aussi définir une fonction d'utilité U qui, pour chaque individu d'une population, rend une valeur U(x) qui correspond à la qualité de l'individu par rapport au problème que l'on veut résoudre. Cette fonction est le critère à maximiser sur notre population de départ. De plus, un algorithme génétique utilise trois opérations:

- la *reproduction*
- le *croisement*
- la *mutation*

Voici, en détail, le rôle de chaque opérateur :

- La *reproduction* a pour but de sélectionner les individus d'une population qui vont participer au croisement. Ces individus sont tirés au hasard par rapport au critère d'utilité. Ce tirage revient au lancement d'une roulette où chaque individu aurait une part proportionnelle à sa valeur d'utilité.

  Sur l'exemple de la figure 3.1, on voit bien qu'en lançant cette roulette, les individus possédant la plus grande valeur d'utilité ont plus de chances d'être tirés, mais la probabilité de tirer un individu moins bon n'est pas nulle: ils peuvent contenir quelques gènes bénéfiques à la population.

No.	valeur d'utilité	% d'utilité
1	169	14.4
2	576	49.2
3	64	5.5
4	361	30.9

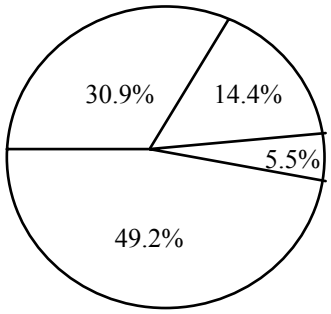


fig. 3.1 fonctionnement de l'opérateur de reproduction

- Le *croisement* consiste à tirer au hasard un nombre k positif inférieur à la taille n des individus. Ensuite, à partir de deux individus A et B deux autres individus sont créés, l'un formé des k premiers gènes de A et de n-k derniers gènes de B, l'autre formé des n-k derniers gènes de A et des k premiers gènes de B.

- La *mutation* consiste à modifier la valeur d'un ou plusieurs gènes d'un individu. Exemple : si un individu est une suite de bits, la mutation d'un gène consiste à lui donner la valeur 0 si sa valeur initiale était 1, ou 1 sinon.

Habituellement, les opérateurs de reproduction et surtout de croisement sont tellement importants pour la convergence vers la meilleure solution (sélection parmi les meilleurs et croisement de ces individus), que l'opérateur de mutation joue un rôle secondaire.

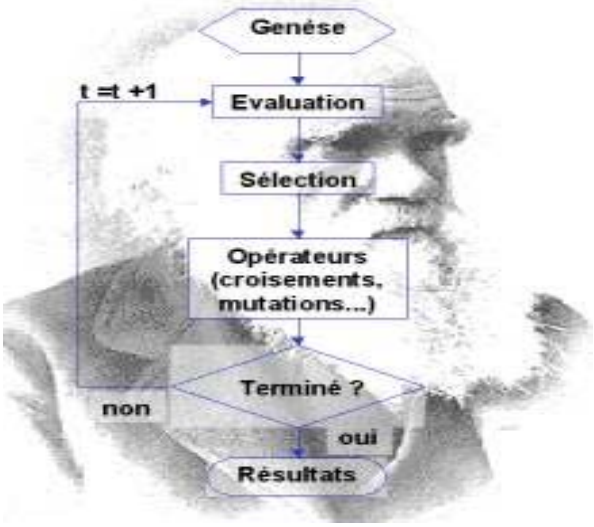


fig. 3.2 schéma global d'optimisation par un algorithme génétique

Etant donnés une fonction d'utilité ainsi que les trois opérateurs, un algorithme génétique se déroule de la manière suivante :

1. fournir une population de départ

tant qu'un optimum pour la fonction d'utilité n'est pas atteint faire :

2. mesurer l'utilité de chaque individu
3. sélectionner une nouvelle population en se basant sur l'utilité des individus. Après cette opération les individus de l'ancienne population disparaissent du processus.
4. remplacer un certain nombre de paires d'individus tirés aléatoirement par les croisements de ces individus
5. Muter un ou plusieurs individus
6. Retourner à l'étape 2

3.2 Application des algorithmes génétiques à l'optimisation des tests

Dans la suite, nous expliquons d'abord comment nous avons adapté les opérateurs génétiques à notre problème et donnons quelques définitions pour l'analyse du modèle. Nous exposons ensuite le processus global de génération et optimisation de tests grâce à un algorithme génétique basé sur une analyse de mutation, et terminons cette section en situant notre approche par rapport à d'autres travaux appliquant les algorithmes génétiques au domaine du test.

3.2.1 Les opérateurs génétiques pour notre problème

Un individu est un ensemble de gènes et les gènes sont des tests. Nous considérons un test comme un couple d'initialisation et appels de méthodes. L'initialisation met le système dans un état tel qu'il peut accepter les appels de méthodes.

Voici les notations que nous utiliserons pour les exemples :

Test : 1 test = 1 gène = [1 séquence d'initialisation , quelques appels de méthodes]
Gène : G = [I , S] et S=($m_1(p_1)$,....,$m_n(p_n)$)
Individu : un individu est un ensemble fini de gènes = {G_1,....G_m}
Population : la population est un ensemble fini d'individus

La taille m d'un individu ne devrait pas changer alors que la taille n d'un gène pourra changer (en modifiant le nombre d'appels effectués par un test).

Un point souvent difficile à résoudre lorsqu'il s'agit d'appliquer un algorithme génétique à un problème particulier, est de trouver la fonction d'utilité. Dans notre cas, cette fonction est évidente, nous prenons comme *critère d'utilité* pour un individu son *score de mutation*.

Les trois opérations de l'algorithme génétique s'effectueront ainsi :

- Reproduction : sélection des individus qui vont participer à la prochaine génération en fonction de leur valeur d'utilité.

- Croisement : soit i un entier tiré au hasard, à partir de deux individus A et B nous obtiendrons deux nouveaux individus en prenant les i premiers gènes de A et

les n-i derniers gènes de B pour le premier et les i premiers gènes de B et les n-i derniers gènes de A pour le second.

$ind_1 = \{G_{1\,1}, ... G_{1\,i}, G_{1\,i+1}, .. G_{1\,m}\}$ $\quad ind_2 = \{G_{2\,1}, ... G_{2\,i}, G_{2\,i+1}, .. G_{2\,m}\}$

$ind_3 = \{G_{1\,1}, ... G_{1\,i}, G_{2\,i+1}, .. G_{2\,m}\}$ $\quad ind_4 = \{G_{2\,1}, ... G_{2\,i}, G_{1\,i+1}, .. G_{1\,m}\}$

fig. 3.3 opérateur de croisement

Commentaire: cet opérateur de croisement nous permet de trouver le ou les ensembles de tests qui tuent le plus de mutants. Par contre il ne modifie pas le score de mutation.

▪ Mutation : nous utilisons deux opérateurs de mutation. Le premier consiste à modifier les valeurs d'appel d'une méthode dans un ou plusieurs gènes.

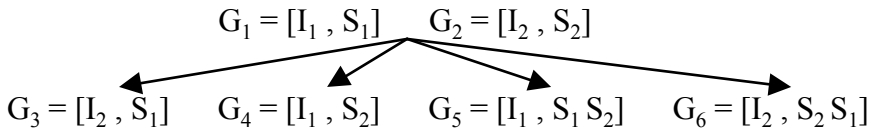


fig. 3.4 premier opérateur de mutation

Commentaire : ce premier opérateur est important, par exemple s'il y a un si-alors-sinon dans une méthode, il faut appeler cette méthode avec une valeur qui nous permet de tester dans la branche si et avec une valeur qui nous permet de tester le sinon.

Le second opérateur de mutation crée des nouveaux gènes à partir de deux gènes, soit en inversant les séquences d'initialisation des gènes, soit en mettant bout à bout les appels de méthodes.

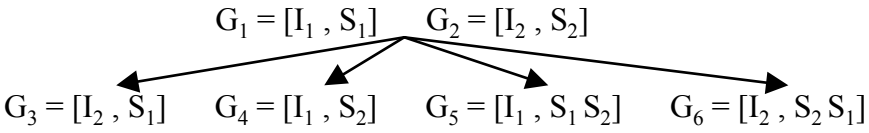


fig. 3.5 second opérateur de mutation

Commentaire : cet opérateur fait varier la taille des gènes. On remarque que ces opérateurs de mutation sont seuls à faire varier le score de mutation des tests. Ces opérateurs nous permettre d'optimiser la population initiale de tests. Donc contrairement à un algorithme génétique classique, nos opérateurs jouent un rôle important pour la convergence vers une solution.

Comme seuls les opérateurs de mutation influencent le score de mutation, nous utilisons un taux de mutation à chaque génération qui se situe aux alentours de 6% alors qu'un algorithme génétique classique mute seulement 1% des gènes à chaque génération. Ceci risque de rendre notre méthode instable, car les bons individus ne sont pas mémorisés, ils passent juste à la génération suivante grâce à la reproduction, un taux de mutation élevé augmente donc fortement la probabilité de perdre ces individus et d'affaiblir globalement la population.

3.2.2 Quelques définitions pour notre modèle

Dans cette section, nous donnons quelques définitions utiles à la compréhension et l'analyse de notre méthode.

Définition 3.1 : *un prédateur*. Dans la première approche que nous présentons, nous considérons un individu comme un prédateur.

Dans la suite, nous utilisons les valeurs suivantes :

- MutantsSet, l'ensemble des mutants de la classe sous test
- NbMut = *card (MutantsSet)*
- GenesPool, l'ensemble des gènes disponibles à un instant donné de l'algorithme
- NbGenes = *card (GenesPool)*
- PredSize = taille d'un prédateur = nombre de gènes d'un prédateur
- PopSize = taille de la population = nombre d'individus dans la population

Nous utilisons aussi les notations *Sign(x)* pour désigner la signature de x et *SM(x)* pour le score de mutation de x. (cf. 2.2.2)

Définition 3.2 : *Signature d'un gène*. La signature d'un gène est l'ensemble des mutants tués par ce gène: *Sign(g_i)* = {mutants tués par g_i}

A partir de la signature des gènes, on peut définir le score de mutation pour un gène, un prédateur et une population de prédateurs.

Définition 3.3 : *Score de mutation d'un gène*: $SM(g_i) = \frac{card(Sign(g_i))}{NbMut}$

Le score de mutation d'un gène est donc la proportion totale de mutants qu'il peut tuer.

Définition 3.4 : *Score de mutation d'un prédateur*:

$$SM(pred_j) = \frac{card(\bigcup_{i=1}^{PredSize} Sign(g_i))}{NbMut}$$

Le score de mutation d'un prédateur est la proportion de mutants qu'il peut tuer. Ce score dépend bien sûr du génotype du prédateur.

Définition 3.5 : *Score de mutation d'une population*:

$$SM(pop) = \frac{card(\bigcup_{i=1}^{NbGenes} Sign(g_i))}{NbMut}$$

Le score global de la population est la proportion des mutants que peuvent tuer les individus, qui dépend en fait uniquement des gènes de GenesPool à un instant donné.

3.2.3 Problème des optimums locaux

Une des principales difficultés lors d'une optimisation, est de ne pas s'arrêter sur un optimum local. Ce problème illustré par la figure 3.6 est difficile à résoudre puisqu'il est impossible de savoir si un optimum est local ou si c'est bien le résultat de notre recherche. Les algorithmes génétiques permettent d'éviter ce problème grâce à l'opérateur de reproduction. En effet, les prédateurs choisis lors de la reproduction pour participer à la génération suivante sont tirés au hasard parmi tous les prédateurs de la population.

Evidemment les prédateurs les plus forts ont plus de chance d'être tirés, mais le fait de permettre à des prédateurs moins bons d'être tirés, permettra, si l'algorithme atteint un optimum local, de revenir en arrière et d'explorer d'autres pistes.

Commentaire : En général, l'optimum absolu n'est pas connu, et il est donc difficile de savoir quand arrêter le processus. Dans notre cas, ce problème ne se pose pas, l'optimum absolu est atteint quand le score global de la population est de 100%. Cependant, ce score est un idéal rarement atteint, et il est possible de fixer un score de mutation inférieur à 100% comme critère d'arrêt. Il est aussi possible d'arrêter le processus après un nombre fixé de générations.

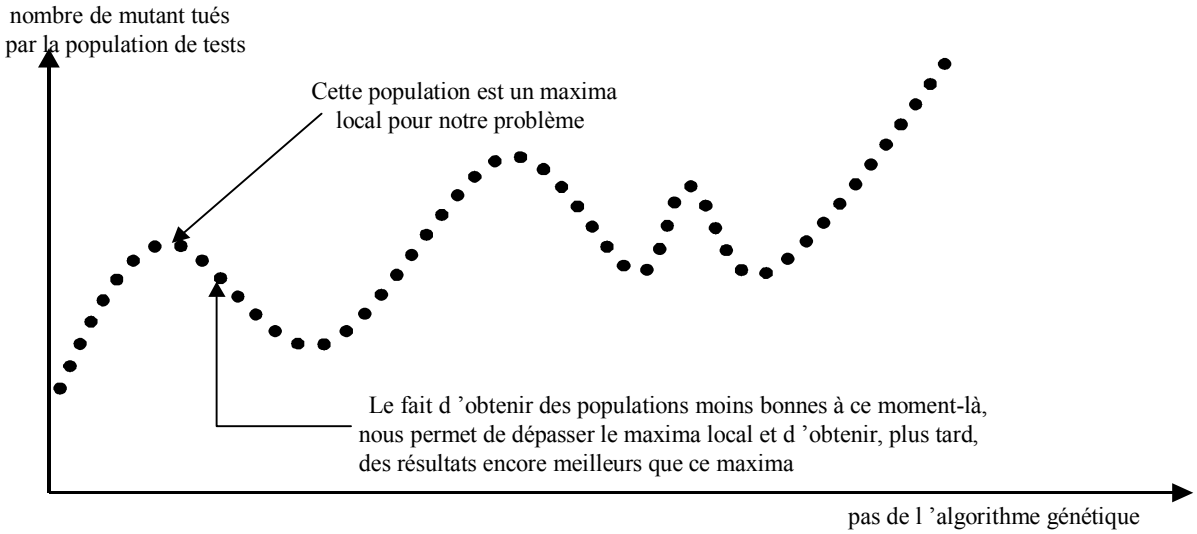


fig. 3.6 problème des optimums locaux

3.2.4 Le processus global

Nous décrivons maintenant le processus global d'optimisation des tests grâce à un algorithme génétique basé sur une étude de mutation. Ce processus se décompose en cinq étapes séquentielles décrites figure 3.7.

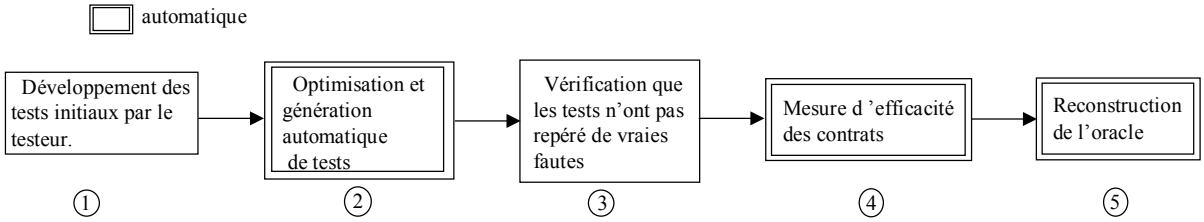


fig. 3.7 Processus global de génération de tests

1. La première étape consiste, pour l'utilisateur, à écrire le premier ensemble de gènes qu'il va fournir à l'algorithme génétique.
2. L'étape suivante consiste à optimiser cet ensemble initial de tests automatiquement. Dans une première approche, nous appliquons un algorithme génétique(fig. 3.8). A partir du premier ensemble tests, la population initiale de l'algorithme est construite,

et les tests sont exécutés sur les programmes mutants pour obtenir leurs scores de mutation, l'algorithme génétique peut alors démarrer. L'algorithme, s'arrête après un certain nombre de générations, ou lorsqu'un certain score est atteint, ou si le score de mutation stagne d'une génération à l'autre. On peut donc optimiser avec un critère de temps maximum autorisé ou de qualité attendue des tests. Quand l'optimisation s'arrête, si le score de mutation global n'est pas assez bon, il faut faire le même diagnostique que lors de l'analyse de mutation. Ensuite, le processus peut redémarrer si l'utilisateur lui fournit de nouveaux tests.

3. Lors de la troisième étape, l'utilisateur doit vérifier que les tests qu'il a obtenu ne repèrent pas de fautes dans le programme initial, et éventuellement corriger ces fautes.
4. La quatrième étape consiste à mesurer la qualité des contrats grâce au test par mutation comme décrit section 2.2.3.
5. Enfin, la dernière étape du processus consiste à créer un oracle complet à partir des tests générés. Pour cela, il faut exécuter chaque test sur la classe initiale, et l'état de l'objet après une exécution correspond à une valeur d'oracle.

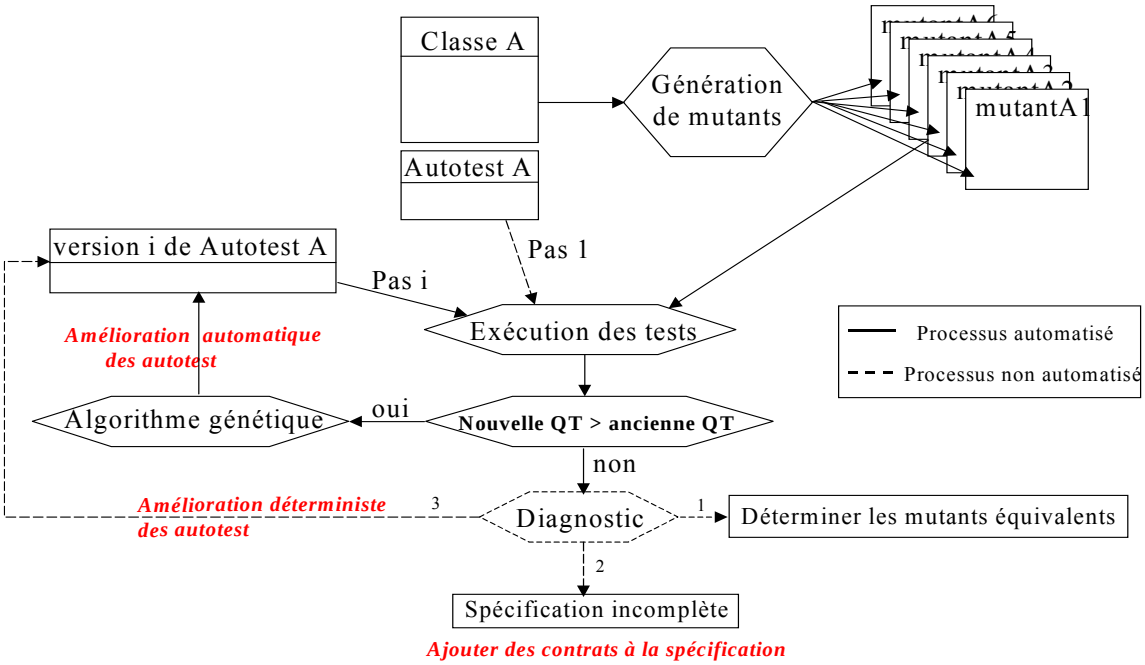


fig. 3.8 processus d'optimisation et de génération automatique de tests

3.2.5 Un bref état de l'art

Dans cette section, nous présentons rapidement quelques travaux basés sur des algorithmes génétiques dans le domaine du test.

Dans [Jones96] et [Harrold99] les algorithmes génétiques sont utilisés pour la génération automatique de cas de tests en utilisant le taux de couverture du graphe de contrôle comme fonction d'utilité. Les deux équipes ont développé un outil, et ont expérimenté leur méthode sur de petits programmes procéduraux.

D'autre part, [Michael97] a repris une étude de Korel [Korel90] qui proposait de ramener la génération de tests au problème de la minimisation d'une fonction. [Korel90] résolvait le problème grâce à une descente de gradient, et [Michael97] a amélioré la méthode en

appliquant un algorithme génétique. L'amélioration ainsi obtenue est non négligeable car les algorithmes génétiques permettent de dépasser les minimaux locaux beaucoup plus facilement qu'avec la descente de gradient.

Enfin, [Wadekar99] utilise un algorithme génétique pour résoudre un problème de fiabilité.

Notre application des algorithmes génétiques à la génération et l'optimisation de tests [Baudry2000] se situe donc dans un cadre différent de ceux proposés dans ces articles pour les raisons suivantes:

- nous utilisons une analyse de mutation pour mesurer l'utilité d'un test
- nous avons des opérateurs de mutation forts
- nous générons des tests à partir d'un ensemble initial déjà bon pour notre critère d'utilité (le score de mutation)
- nous nous intéressons à la génération de tests pour des programmes objets

3.3 Eléments théoriques d'analyse des algorithmes génétiques

Dans cette section, nous présentons une analyse abstraite du fonctionnement des algorithmes génétiques tirée de [Goldberg89], puis nous développons une approximation de la complexité de l'algorithme dans notre cas particulier.

3.3.1 Analyse abstraite de l'évolution d'un algorithme génétique

Pour analyser de manière abstraite les effets d'un algorithme génétique sur une population, nous allons utiliser la notion de schéma introduite par J. Holland en 1968. Un *schéma* permet de représenter un sous-ensemble d'individus qui présentent des similitudes sur certains gènes.

Pour illustrer cette idée, on considère des individus codés sur l'alphabet binaire {0,1}. Pour écrire des schémas, il faut ajouter le caractère * à l'alphabet qui sera le caractère qui remplace tous les autres.

Définition 3.6 : *Schéma*. On appelle schéma de longueur l une suite $H = a_1 a_2 ... a_l$ avec

Définition 3.7 : *Instance*. On dit qu'un individu $A = a_1.....a_l$ est une instance du schéma $H = b_1.....b_l$ si pour tout i tel que $b \neq *$ on a $a_i = b_i$.

Par exemple, le schéma 1*0000 représente le sous-ensemble {110000 ,100000}, et 110000 est une *instance* du schéma 1*0000.

Pour décrire l'effet d'un algorithme génétique sur les schémas, nous aurons besoin de deux propriétés des schémas, l'ordre et la longueur fondamentale.

Définition 3.8 : *Position fixe*, *position libre*. Pour un schéma H, on dit que i est une position fixe de H si $a_i = 1$ ou $a_i = 0$. On dira que i est une position libre de H si $a_i = *$.

Définition 3.9 : *Ordre d'un schéma* H, noté o(H), est le nombre de *positions fixes* dans un schéma (pour un alphabet binaire, c'est le nombre de 0 et de 1). Par exemple, l'ordre du schéma 011*1* est 4.

Définition 3.10 : *Longueur fondamentale* d'un schéma H, dénotée $\delta(H)$, est la distance entre la première et la dernière position fixe du schéma. Par

exemple, pour le schéma 011*1*, on a δ=4, et pour le schéma 0*****, δ=0.

Nous allons maintenant étudier les effets individuels et combinés des opérations de reproduction, croisement et mutation.

Proposition 3.1 : *Effet de la reproduction*. Supposons qu'au pas t de l'algorithme , il y ait m représentants d'un schéma H dans une population A(t), on note m = m(H,t). Soit p_i la probabilité de reproduction d'un individu $p_i = f_i / \sum f_j$. A la fin de la reproduction, on obtient une nouvelle population de n individus, et le nombre de représentant du schéma H est:

$m(H,t+1) = m(H,t) \cdot n \cdot f(H) / \sum f_j$, où f(H) est la valeur d'utilité moyenne des chaînes représentant le schéma H au temps t.

Proposition 3.2 : Comme on sait que l'utilité moyenne de la population est $\bar{f} = \sum f_j / n$ alors on peut écrire l'évolution du schéma après reproduction sous la forme :

$$m(H,t+1) = m(H,t)\frac{f(H)}{\bar{f}}$$

On déduit de la proposition 3.2 que les schémas dont la valeur d'utilité est au-dessus de la moyenne de la population auront plus de représentants à la génération suivante tandis que les schémas dont l'utilité est plus faible que la moyenne auront moins de représentants.

Proposition 3.3 : maintenant, si on suppose qu'un schéma H est toujours au-dessus de $c \cdot \bar{f}$ où c est une constante, alors on peut réécrire l'équation de la manière suivante :

$$m(H,t+1) = m(H,t)\frac{(\bar{f} + c\bar{f})}{\bar{f}} = (1+c) \cdot m(H,t)$$

Et si on commence à t = 0 et que c est constante on a :

$m(H,t) = m(H,0) \cdot (1+c)^t$

Nous venons de montrer qu'un schéma dont la valeur d'utilité est au-dessus de la moyenne voit son nombre de représentants augmenter suivant une progression géométrique. Si la reproduction était seule en jeu, les schémas forts élimineraient très rapidement les faibles. Cependant, le fait de toujours garder les schémas forts ne permet jamais d'explorer de nouvelles solutions, et c'est pour cela qu'il faut croiser les individus. En effet, le croisement permet de créer de nouveaux individus avec un minimum de perturbations puisque cet opération ne fait qu'échanger des informations entre individus.

Exemple : Pour voir quels schémas sont affectés par le croisement, observons un individu de longueur $l = 7$ et deux schémas le représentant :

A = 0 1 1 1 0 0 0

H_1 = * 1 * * * * 0

H_2 = * * * 1 0 * *

Rappelons que le croisement consiste à choisir deux individus et un nombre x au hasard entre 1 et l-1. On échange les parties avant et après le gène x de chaque individu. Supposons maintenant que A ait été choisi pour un croisement avec un autre individu et que x soit égal à 3, l'individu A, et donc ses schémas, seront coupés de la manière suivante (la coupure est marquée par \ ) :

A = 0 1 1 \ 1 0 0 0
H_1 = * 1 * \ * * * 0
H_2 = * * * \ 1 0 * *

Après reproduction, le schéma H_1 va être détruit car le «1» à la position 2 et le «0» à la position 7 vont se retrouver à des positions différentes, alors que le schéma H_2 va rester intact. Quelque soit le point de coupure, il est évident que le schéma H_1 a moins de chances de survivre que le schéma H_2 car les positions fixes extrêmes sont plus espacées pour H_1 que pour H_2.

Propriété 3.1 : *Probabilité qu'un schéma soit détruit lors d'un croisement* est donc la probabilité que le point de coupure tombe entre les positions fixes extrêmes de ce schéma, c'est à dire: $\delta(H)/(l-1)$.

Propriété 3.2 : La probabilité de survie p_s d'un schéma après reproduction est :

$$p_s = 1 - \delta(H)/(l-1)$$

Si un schéma est choisi pour un croisement avec une probabilité p , alors :

$$p_s \geq 1 - p_c \cdot \frac{\delta(H)}{l-1}$$

On peut maintenant calculer le nombre de représentants d'un schéma H à la prochaine génération en prenant en compte les effets de la reproduction et du croisement.

Proposition 3.4 : Si on considère que les opérations de reproduction et de croisement sont indépendantes, on a :

$$m(H,t+1) \geq m(H,t) \cdot \frac{f(H)}{\bar{f}} \left[1 - p_c \cdot \frac{\delta(H)}{l-1} \right]$$

On a obtenu cette expression en multipliant le nombre de schémas après la reproduction seule par la probabilité de survie après un croisement. L'évolution d'un schéma après reproduction et croisement dépend donc de deux choses, si sa valeur d'utilité est au-dessus de la moyenne ou non et si sa longueur fondamentale est élevée ou non.

Considérons maintenant le dernier opérateur, la mutation, c'est-à-dire l'opérateur qui altère la valeur d'un gène avec une probabilité p_{mut}. Un schéma va survivre à la mutation si toutes ces valeurs fixes gardent la même valeur. La probabilité de survie est 1 - p_{mut}, et comme toutes les mutations sont indépendantes, un schéma H survit si toutes ses $o(H)$ positions fixes survivent.

Propriété 3.3 : la probabilité de survie d'un schéma après mutation est $(1 - p_{mut})^{o(H)}$.

Pour une probabilité p_{mut} petite, la probabilité de survie d'un schéma peut être arrondie par $1 - o(H) \cdot p_{mut}$.

Proposition 3.5 : on peut conclure qu'un schéma H aura un nombre de représentants après reproduction, croisement et mutation donné par l'équation suivante :

$$m(H,t+1) \geq m(H,t) \cdot \frac{f(H)}{\bar{f}} \left[1 - p_c \cdot \frac{\delta(H)}{l-1} - o(H)\, p_{mut} \right]$$

Cette dernière proposition nous apprend, d'une part, que les schémas qui ont une longueur fondamentale petite sont plus favorisés que les autres pour survivre au passage à une nouvelle génération. D'autre part, cette proposition nous montre aussi que les schémas qui ont un ordre

petit ont une plus grande probabilité de survie lors de la génération d'une nouvelle population.

3.3.2 Quelques propriétés particulières à notre approche

Dans cette section, nous analysons l'effet des opérateurs génétiques sur le score de mutation de la population, des prédateurs ou des gènes. (cf. 3.2.2)

Proposition 3.6 : effet de l'*opération de reproduction* sur le score de mutation global de la population. Soient pop_{pre} la population avant l'opération et pop_{post} la population après, on a:

$$SM(pop_{pre}) \geq SM(pop_{post})$$

La proposition précédente nous dit que le score de mutation de la population diminue ou stagne après l'opération de reproduction. En effet, cette opération consiste à reproduire certains prédateurs et en éliminer d'autres. Il est donc clair qu'au cours de cette opération aucun gène n'est créé, donc le score ne peut pas augmenter. D'autre part, certains gènes peuvent disparaître, ce qui peut faire diminuer le score de la population.

Proposition 3.7 : effet de l'*opération de croisement* sur le score de mutation global de la population et sur le score des individus. Soient pop_{pre} la population avant l'opération et pop_{post} la population après, on a:

$$SM(pop_{pre}) = SM(pop_{post})$$

De plus, le score de mutation des individus croisés varie mais on ne peut pas savoir dans quel sens.

La proposition 3.7 est évidente. L'opération de croisement ne fait que réorganiser les gènes au sein des individus, aucun gène n'est détruit ni créé, le score de mutation de la population ne varie donc pas. D'autre part, le fait que les individus soient composés de gènes différents fait varier leur score, mais le sens de la variation dépend des gènes et est imprévisible.

Proposition 3.8 : effet de *la mutation d'un gène* g sur le score de mutation global. Soient pop_{pre} la population avant l'opération et pop_{post} la population après, soient g_{pre} le gène avant l'opération et g_{post} le gène après, on a :

si $Sign(g_{pre}) \subseteq Sign(g_{post})$ alors $SM(pop_{pre}) \leq SM(pop_{post})$

sinon $SM(pop_{pre}) \geq SM(pop_{post})$

La proposition 3.8 nous dit que si la signature d'un gène grandit après une opération de mutation, alors le score du gène et de la population peuvent augmenter ou stagner. Par contre, si la signature du gène diminue, les scores de mutation peuvent aussi diminuer. De cette proposition, on déduit l'évolution du score de mutation après mutation d'un ensemble de gènes.

Corollaire : effet *global de l'opération de mutation*. Lors de l'opération de mutation, on mute un ensemble $G = \{g_1 \ldots g_m\}$ de gènes. Soient pop_{pre} la population avant l'opération et pop_{post} la population après, soient g_{pre} le gène avant l'opération et g_{post} le gène après . L'effet global sur le score de mutation de la population est le suivant:

si $\bigcup_{i=1}^{m} Sign(g_{i_{pre}}) \subseteq \bigcup_{i=1}^{m} Sign(g_{i_{post}})$ alors $SM(pop_{pre}) \leq SM(pop_{post})$

sinon $SM(pop_{pre}) \geq SM(pop_{post})$

C'est-à-dire que si l'union des signatures augmente après l'opération de mutation le score de la population augmente ou stagne, sinon le score de la population diminue ou stagne.

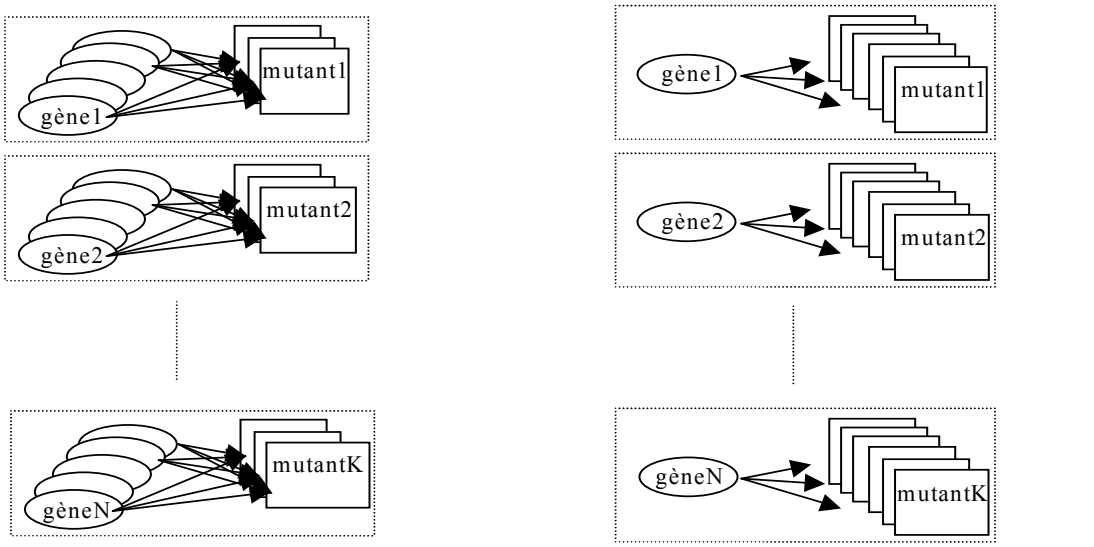
3.3.3 Evaluation de la complexité de la méthode dans le cadre de l'optimisation des tests

Pour évaluer la complexité en temps de notre méthode, nous prenons le nombre de signatures que nous devons calculer au cours d'une exécution. Nous pensons que c'est une bonne approximation de la complexité, car le calcul de signature est très coûteux puisqu'il faut compiler et exécuter le gène avec chaque mutant. De plus les deux autres opérations de reproduction et de croisement ne font que réorganiser les gènes, ce qui correspond à une simple manipulation de pointeurs, la durée de ces opérations est donc négligeable.

Proposition 3.10 : la *complexité de notre algorithme* est le nombre de signatures calculées au cours du processus qui est égal au nombre de gènes créés multiplié par le nombre de programmes mutants :

$$(NbGénérations \cdot NbGènes) \cdot NbMut$$

Cependant, cette complexité pourrait être réduite en parallélisant certains traitements.



N machines calculent des signatures partielles de tous les gènes

N machines calculent chacune la signature d 'un gène

fig. 3.9 deux parallélisations possibles

Les algorithmes génétiques se prêtent bien à la parallélisation. Notre méthode peut être parallélisée de deux manières, soit en calculant la signature des gènes sur des machines différentes, soit en calculant des signatures partielles de tous les gènes sur des machines différentes. La figure 3.9 illustre ces deux types de parallélisation.

3.4 Expériences

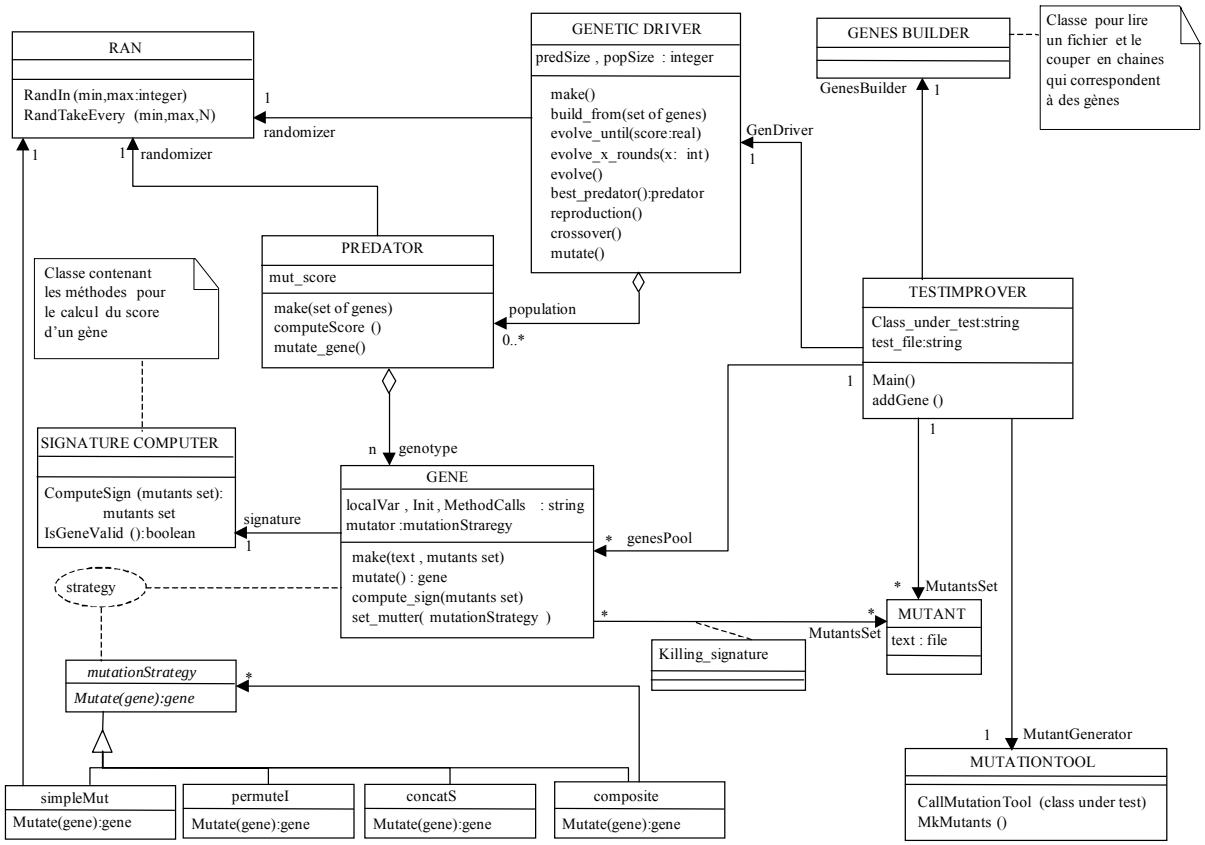


fig. 3.10 diagramme de classes de TestImprover

Le grand problème des algorithmes génétiques est qu'il n'est pas possible de prouver leur efficacité sur un problème particulier de manière formelle. En effet, l'intervention fréquente du hasard dans le processus génétique rend difficile toute prévision de l'évolution de l'algorithme, ou du temps de résolution du problème.

Dans notre problème particulier, on voit bien que les opérateurs de mutation vont permettre d'augmenter la couverture du code par les tests : le fait de faire varier les valeurs d'appel de méthodes, de changer l'initialisation ou de faire plusieurs appels à une méthode va permettre d'explorer d'autres branches du graphe de contrôle , mais on ne peut pas savoir au bout de combien de temps tout le code sera couvert, ni même quand une branche supplémentaire sera explorée.

Pour trouver les bons paramètres d'un algorithme génétique (pourcentage de mutation, nombre de prédateurs, taille initiale d'un gène…) et prouver son efficacité, il faut donc faire des expériences. Pour cela, nous avons développé un outil mettant en œuvre un algorithme génétique pour l'amélioration de tests, basé sur les codages et opérateurs décrits en 4.1.

3.4.1 L'outil développé

Le langage *UML* et les *Design Patterns* nous ont permis de concevoir l'outil appelé TestImprover (fig. 3.10). Nous l'avons développé en Java. Les principales classes de cet outil sont :

- TESTIMPROVER : cette classe contient le programme principal. Elle pilote l'initialisation de l'algorithme génétique (génération des mutants, création des gènes initiaux) puis passe la main à GENETICDRIVER.
- GENETICDRIVER : cette classe pilote la boucle génétique. Elle offre plusieurs méthodes pour la boucle (arrêt après un temps donné, arrêt à un certain score).
- PREDATOR : classe pour un prédateur.
- GENE : classe pour un gène.
- MutationStrategy : cette classe abstraite modélisée par le Design Pattern Strategy, est l'interface pour les différents opérateurs de mutation : changement des valeurs d'appel de méthode, permutation des initialisations, concaténation des appels de méthodes, ou bien une de ces trois opérations tirée au hasard.

Cet outil prend en entrée une classe à tester et un fichier de tests. A partir de la classe sous test, les mutants sont créés. A partir du fichier de tests, les gènes initiaux sont créés et leur signature calculée. Une fois les gènes construits, la population est initialisée : on crée un certain nombre de prédateurs en tirant des gènes au hasard. Une fois cette phase d'initialisation terminée, l'algorithme génétique est lancé. L'algorithme va effectuer l'optimisation génétique sur un nombre fixé de générations, ou s'arrêter lorsque le score attendu est atteint.

Initialisation
1. Génération des mutants de la classe sous test
2. Génération des gènes à partir du fichier de tests et calcul des signatures
3. Génération de la population initiale de prédateurs

Boucle génétique
4. Reproduction des prédateurs en fonction de leur score de mutation
5. Croisement des prédateurs
6. Mutation d'un gène de quelques prédateurs tirés au hasard
7. Retour en 4

Conditions d'arrêt : - un score est atteint
- un nombre de générations est atteint

fig. 3.11 algorithme de TestImprover

L'initialisation

Dans une première phase, TestImprover initialise le système. Pour cela, il prend deux paramètres en entrée, le fichier de la classe sous test et le fichier initial de tests. A partir de la classe sous test, on fabrique tous les mutants possibles en injectant une erreur simple dans la classe pour chaque mutant. Ensuite, à partir du fichier de test initial, on va fabriquer les gènes initiaux. Chaque gène va correspondre à une méthode de test qui devra être écrite sur le modèle décrit sur la figure 3.12.

A partir des mutants de la classe initiale, la signature des gènes est calculée, et la population initiale de l'algorithme génétique est construite. L'initialisation est alors terminée.

```
test_meth1   is
local var1 ;
        var2 ;
do
        init1 ;
        init2 ;
--MethodCalls
        call1 ;
        call2 ;
        call3 ;
end
```

fig. 3.12 exemple de méthode de test dans la classe de test initiale

La boucle génétique

L'initialisation étant achevée, TestImprover active l'algorithme génétique. Une itération consiste à sélectionner une nouvelle population, qui sera la génération suivante, en utilisant l'opérateur de reproduction. A partir de cette nouvelle génération, on croise les prédateurs, et on mute certains gènes (environ 6%). Lors de la mutation d'un gène, on vérifie si le nouveau gène est licite, c'est-à-dire qu'il n'y a pas d'erreur de syntaxe ou d'exécution (violation de certains contrats) dues à la mutation. Si le gène est illicite, on essai de muter le gène initial d'une autre façon jusqu'à production d'un gène licite. On calcule alors la signature du gène muté.

L'utilisateur peut choisir entre deux façons d'arrêter la boucle, soit en faisant tourner un certain nombre de fois, soit en faisant tourner jusqu'à ce qu'un certain score de mutation soit atteint.

3.4.2 Expérimentations

Nous avons effectué des expériences sur les classes p_date, p_time et p_date_time qui sont des classes de la bibliothèque Pylon du langage Eiffel qui permettent de gérer le temps. La relation entre ces différentes classes est décrite par la figure 3.13. La classe p_date_time, qui sert d'interface à ce système de classes, comporte 17 méthodes. Cependant, p_date_time est une classe assez simple car ses méthodes font essentiellement appel à d'autres méthodes des classes p_date et p_time. La classe p_date_time est donc cliente de p_date et p_time, et pour la tester correctement, il faut aussi tester p_date et p_time.

Pour ces expériences il y avait deux ou trois gènes initiaux, quinze prédateurs composés de 5 gènes, et nous avons lancé l'algorithme sur 10 générations pour p_date et p_time puis 20 générations sur p_date_time. Nous avons muté un gène au hasard dans 5 prédateurs à chaque génération.

Les résultats (fig. 3.14) sur le modèle génétique initial sont assez décevants. En effet, les temps de calcul sont très grands (plus de 24 heures pour 10 générations sur p_date), la progression n'est pas strictement croissante et il faudrait certainement un très grand nombre de générations pour obtenir un score global intéressant.

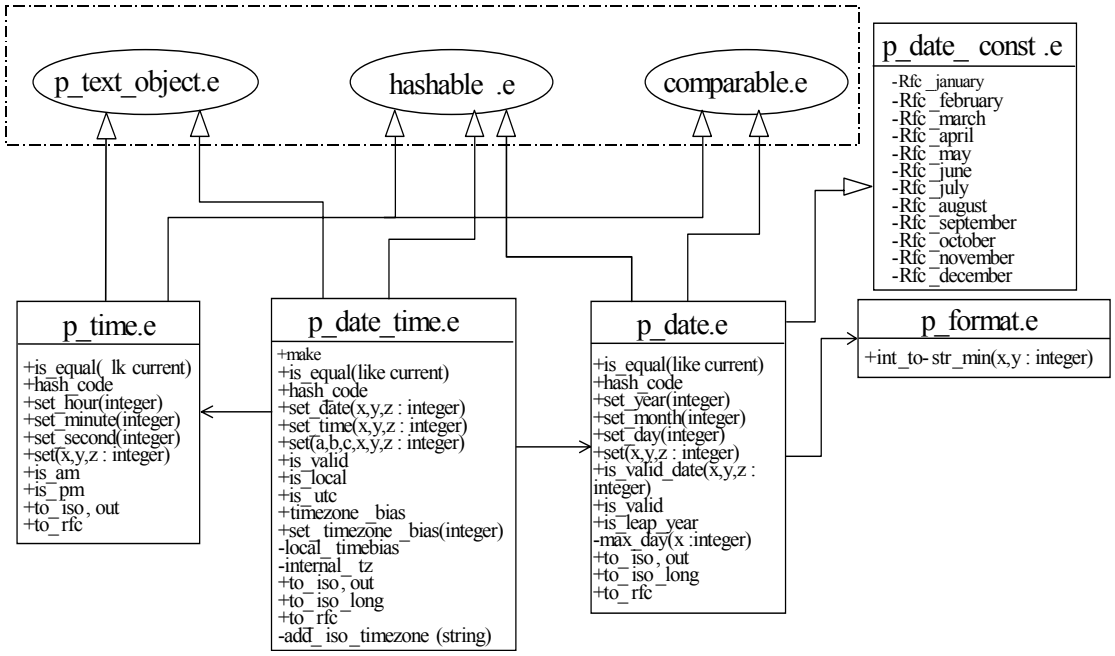


fig. 3.13 Classes qui gèrent le temps dans la bibliothèque Pylon

Cependant, ces expériences nous ont appris plusieurs choses. Tout d'abord, elles nous ont confirmé qu'un algorithme génétique pur, complètement indéterministe est trop coûteux en temps. D'autre part, nous avons montré qu'il est possible d'améliorer automatiquement le score de mutation des tests en les mutant. Enfin, l'analyse de ces résultats nous conduit à améliorer la modélisation des algorithmes génétiques pour augmenter les performances en temps sans perdre le caractère semi-aléatoire des phénomènes génétiques.

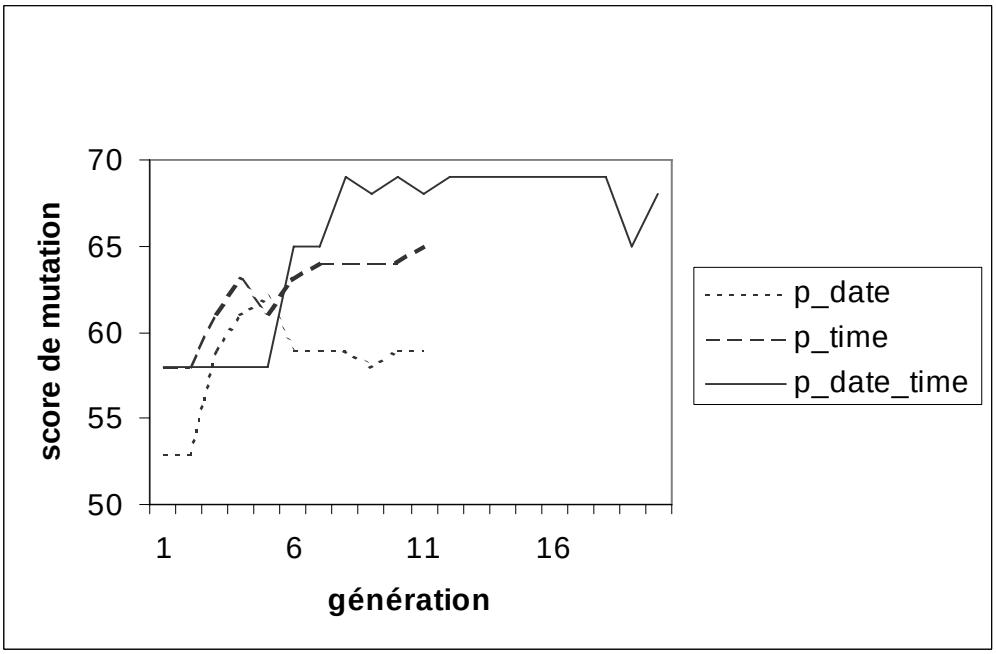


fig. 3.14 Résultats obtenus pour l'amélioration de tests

3.5 Analyse des résultats et remise en cause du modèle initial

Dans cette section, nous décrivons les différentes raisons qui peuvent expliquer les résultats obtenus en appliquant notre modèle d'algorithme génétique initial, et les améliorations qui peuvent être faites.

Tout d'abord, un algorithme génétique classique travaille sur une population de taille fixe, et essaie d'améliorer la qualité globale de celle-ci en modifiant les prédateurs qui la constituent. Ces modifications consistent à sélectionner d'abord les bons prédateurs et à effectuer certaines opérations dessus pour les rendre encore meilleurs, sans les mémoriser. Or, il est impossible de savoir si les modifications apportées (croisement et mutation) vont vraiment améliorer le prédateur. Comme le prédateur n'a pas été mémorisé, l'évolution de la population ne peut pas être toujours croissante.

De plus, le fait de ne pas pouvoir mémoriser les bons prédateurs nous oblige à calculer la signature des nouveaux gènes sur tous les mutants de la classe sous test. En effet, si certains prédateurs étaient mémorisés, la signature des nouveaux gènes pourrait être calculée uniquement sur les mutants qui ne sont pas tués par les prédateurs sélectionnés, ce qui augmenterait énormément l'efficacité de l'approche.

Une nouvelle approche devrait pouvoir permettre de ne pas calculer la signature complète à chaque modification d'un gène, et devrait assurer l'amélioration de la population au passage à la génération suivante.

D'autre part, on sait au départ qu'il faudra de nombreux gènes pour obtenir un bon score final. Comme le nombre de gènes manipulés reste constant, la population initiale doit contenir un grand nombre de fois les même gènes. Pour cela, les prédateurs initiaux sont génétiquement très similaires. Ceci explique que, lors des premières générations le score global évolue lentement, car il y a une forte probabilité de faire plusieurs mutations du même gène, or ce sont les mutations qui font évoluer le score global. Il faudrait donc un nouveau modèle qui limite la redondance des gènes dans le processus.

Enfin, le fait que seul l'opérateur de mutation permette d'augmenter le score de mutation d'un gène, c'est-à-dire la qualité des tests, nous force à muter une plus grande proportion de gènes que dans les algorithmes génétiques classiques (1% habituellement, contre 6% dans nos expériences). Or, cette forte proportion de mutation entraîne une diminution de l'effet de l'opérateur de reproduction qui, en sélectionnant les bons prédateurs, permet d'augmenter le score de la population. En effet, plus le nombre de gènes mutés est grand, plus les prédateurs sélectionnés au cours de la reproduction sont perturbés, au risque de les affaiblir. On se rend donc compte qu'avoir un opérateur de mutation fort ralentit la progression de l'algorithme.

L'analyse des résultats nous montre que les algorithmes génétiques tels que nous les avons appliqués n'offrent pas une solution efficace à notre problème d'optimisation de tests. Cette conclusion offre deux possibilités, soit appliquer un algorithme génétique avec une autre modélisation, soit proposer une autre méthode pour optimiser automatiquement les tests.

La modélisation des prédateurs et des gènes que nous avons choisi est difficile à modifier. En effet, pour appliquer un algorithme génétique, il faut des prédateurs de taille constante, ceci implique de modéliser un test pour un gène, puisque nous voulons pouvoir augmenter sa taille au cours de l'algorithme (on veut pouvoir augmenter le nombre d'appels de méthodes d'un test). A partir de cette modélisation des gènes, un individu ne peut plus être qu'un ensemble fini de gènes.

Ce constat nous fait réfléchir à une autre manière d'optimiser automatiquement des tests. Ce nouveau modèle considère ce qui était ici un gène comme le prédateur, ce qui nous permet d'éliminer la notion d'individu que nous avions dans notre approche initiale. Cette notion

étant éliminée, l'opération de croisement, qui n'apportait aucune information, est aussi éliminée. Ensuite, le nouveau modèle manipule un ensemble de prédateurs, que nous appelons bouillon de bactéries, et essaie d'optimiser ce bouillon en le renouvelant d'une génération à l'autre à partir des meilleures bactéries.

Nous présentons cette méthode dans le chapitre suivant.

Chapitre 4 Une seconde approche, « l'adaptation bactériologique »

La méthode que nous proposons dans ce chapitre consiste à supprimer la notion d'individu et à travailler sur un ensemble de gènes, appelés ici des bactéries, et considérés comme les prédateurs. Les individus, permettaient d'obtenir l'ensemble de tests minimal, mais leur manipulation ralentissait énormément la résolution de notre problème initial qui est l'optimisation de tests. En effet, de nombreux problèmes (départ ralenti par la redondance de gènes, croisement inutile) que nous avons décrits dans l'analyse des résultats de nos expériences sont liés à cette notion. Nous proposons donc une méthode basée sur une *« bouillon bactériologique»*, et nous verrons à la fin comment obtenir l'ensemble minimal de tests à partir de cette soupe.

4.1 Redéfinition du modèle

Avant de présenter le nouveau processus, nous donnons quelques définitions.

4.1.1 Quelques nouvelles définitions

Pour décrire ce nouveau modèle nous utilisons les termes définis en 3.2.2, et nous donnons quelques nouvelles définitions.

Tout d'abord, nous définissons un bactérie.

Définition 4.1 : *Bactérie*. Une bactérie est le prédateur dans notre nouveau modèle, et elle est modélisée comme un gène dans le modèle précédent.

Au cours du déroulement de notre nouveau processus nous distinguons deux types particuliers de bactéries.

Définition 4.2 : *Bactérie nécessaire*. Une bactérie nécessaire à une génération donnée est une bactérie qui est seule à tuer un ou plusieurs mutants, quelque soit son score de mutation.

Définition 4.3 : *Bactérie efficace*. Une bactérie efficace à une génération donnée est une des bactéries qui ont des bons scores de mutation.

Ensuite, notre nouvelle approche permet de rétrécir MutantsSet (cf. 3.2.2) d'une génération à l'autre en n'y mettant que les mutants non tués aux générations précédentes. Ici nous représentons donc l'ensemble des mutants de la classe comme l'union de deux ensembles disjoints:

- MutantsVivants, l'ensemble des mutants qui n'ont pas encore été tués. Initialement, cet ensemble est égal à l'ensemble de tous les mutants.

– MutantsTués, l'ensemble total des mutants tués à un moment donné dans le processus

Cette partition de l'ensemble des mutants nous permet de définir la *signature partielle*.

Définition 4.4 : *Signature partielle*. La signature partielle d'une bactérie notée $Sign_{part}(b)$ est l'ensemble des mutants tués par b dans MutantsVivants :

$Sign_{part}(b)$ = {mutants tués par b parmi MutantVivants}

A partir de cette signature partielle, on définit les scores de mutation pour une bactérie et une population à une génération donnée de la même manière qu'en 3.2.2.

Grâce à ces nouvelles définitions, nous pouvons décrire le fonctionnement de notre nouvelle approche.

4.1.2 Le nouveau modèle bactériologique

L'approche que nous proposons dans ce chapitre se base sur une nouvelle analogie biologique puisque nous considérons ici le prédateur comme l'entité de base. Ce prédateur, qui doit donc évoluer globalement, est appelé ici une *bactérie*. Le phénomène de modification d'une population de bactéries lorsqu'elle plongée dans un nouvel environnement est appelé l'*adaptation bactériologique* [Rosenzweig95].

Le nouvel algorithme (fig. 4.1) manipule un ensemble de bactéries (modélisées de la même façon qu'un gène dans le modèle précédent), et va évoluer en sélectionnant et mutant le sous-ensemble des meilleures bactéries, le critère de sélection étant toujours le score de mutation. De plus, les meilleures bactéries seront mémorisées dans un ensemble qui sera le résultat du processus. La mémorisation de certaines bactéries permet de ne calculer que des signatures partielles aux générations suivantes.

Initialisation
1. Génération des mutants de la classe sous test
2. Génération des bactéries initiales à partir du fichier de tests et calcul des signatures
3. mutation des bactéries et création du bouillon bactériologique initial
4. calcul des signatures

Boucle génétique
5. sélection et mémorisation des bactéries qui ont le meilleur score + bactéries qui sont seules à tuer des mutants
6. mutations
7. calcul des signatures
8. retour en 5

Conditions d'arrêt :
- un score est atteint
- un nombre de générations est atteint

fig. 4.1 nouvel algorithme d'optimisation de tests, le « bouillon bactériologique »

L'initialisation

L'approche est identique à la méthode précédente sauf pour les point 3, 5 et 6. Les deux premières étapes consistent à générer les mutants de la classe sous test ainsi que les bactéries

initiales et à calculer leurs signatures. Comme se sont les premières bactéries, leur signatures sont calculées sur l'ensemble de tous les mutants. A la fin de ces calculs, les ensembles de mutants vivants et tués sont mis à jour.

La suite de l'initialisation est différente ici, puisqu'au lieu de créer la population initiale d'individus à partir des premiers gènes, les premières bactéries sont mémorisées. Ces bactéries sont ensuite mutées plusieurs fois, pour obtenir un ensemble de bactéries qui sera le bouillon initial. La phase d'initialisation se termine par le calcul des signatures partielles des bactéries de ce bouillon.

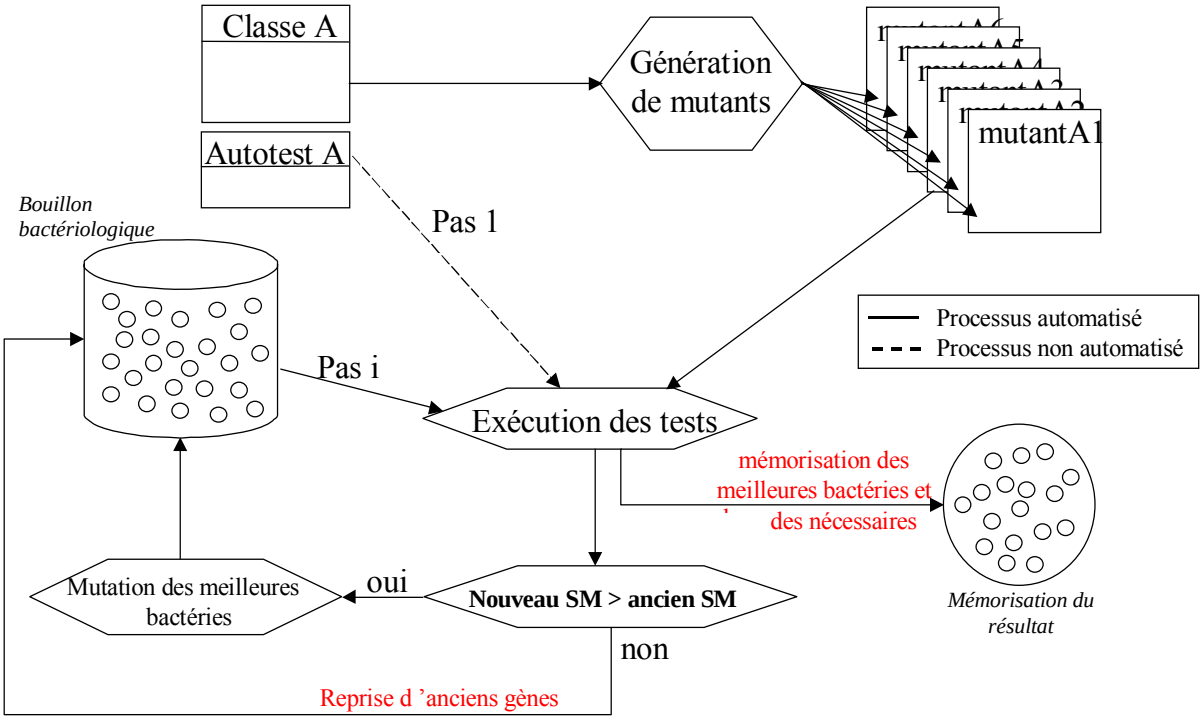


fig. 4.2 schéma global de l'approche basée sur un « bouillon bactériologique »

La boucle génétique

Lors de la première étape de la boucle, les bactéries nécessaires et efficaces sont sélectionnées et mémorisées. Le taux de bactéries efficaces sélectionnées à chaque génération est un paramètre de la méthode qu'il faudra fixer.

La boucle génétique continue en mutant plusieurs fois les bactéries efficaces pour obtenir un nouveau bouillon. Les signatures partielles des nouvelles bactéries sont calculées, les ensembles de mutants vivants et tués mis à jour, et le processus boucle.

La boucle s'arrête si un score satisfaisant est atteint ou si un temps donné s'est écoulé.

Remarque sur les maxima locaux : le fait que cet algorithme n'évolue qu'à partir des meilleurs bactéries, nous empêche de dépasser les maxima locaux, contrairement à l'ancienne approche (cf. 3.2.3). Pour résoudre le problème il faudra tirer d'anciennes bactéries au hasard parmi celles mémorisées lorsque le score stagnera. Cette phase est décrite par la figure 4.3 qui montre le déroulement de notre approche dans le temps.

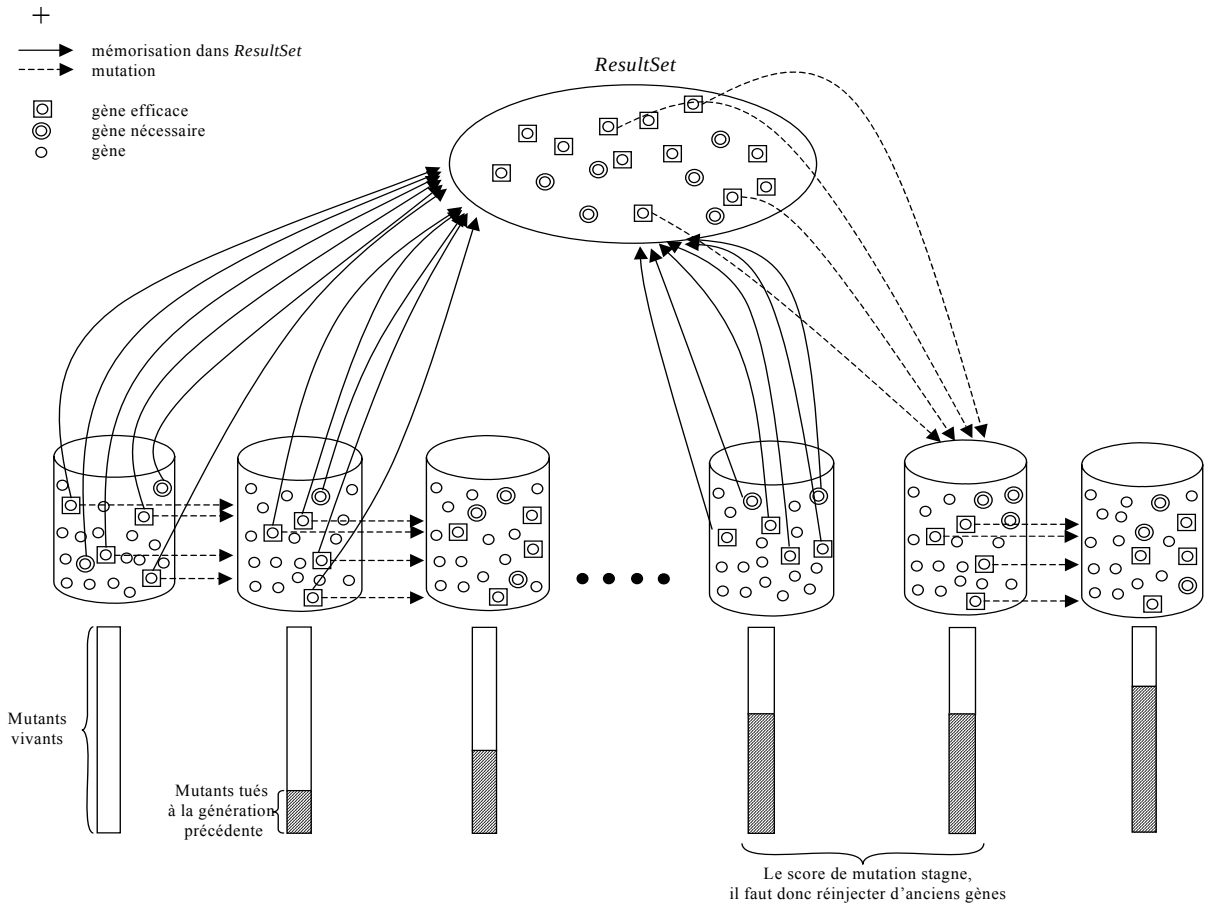


fig. 4.3 évolution du «bouillon » au cours du temps

Calcul de l'ensemble minimal de tests

A la fin de ce processus, l'ensemble solution de toutes les bactéries mémorisées risque de contenir très nombreuse bactéries. Il faut donc calculer l'ensemble minimal de tests obtenant le même score global que cet ensemble solution. Pour cela, nous calculons les signatures complètes de toutes les bactéries sur tous les mutants et nous obtenons une table T de booléens de taille (n+1)*k telle que:

$\forall i \in [1,n], \forall j \in [1,k]$, $T(B_i, M_j) = 1$ si la bactérie B_i tue mutant M_j

0 sinon

$T(n+1, M_j) = \bigcup_{i=1}^{n} T(B_i, M_j)$, c'est-à-dire que $T(n+1, M_j)$ vaut 1 si le mutant M_j est tué par une bactérie. Cette dernière ligne correspond donc à la signature globale de l'ensemble de n bactéries.

La figure 4.4 donne un exemple de table obtenue.

A partir de cette table, nous voulons calculer le sous-ensemble minimal de bactéries qui a le même score que l'ensemble initial.

Proposition 4.1 : *Ensemble minimal de tests*. Soit *ResultSet* l'ensemble résultat de l'adaptation bactériologique, l'ensemble minimal de tests est le sous–ensemble *Bact* tel que :

$$\min(card(Bact)) / ((Bact \subset ResultSet) \wedge SM(Bact) = SM(ResultSet))$$

	M1	M2	M3		Mk-1	Mk
B1	1	1	0		0	0
B2	0	0	1		0	0
B3	0	0	1		1	0
●						
●				● ● ● ●		
Bn-1	1	0	1		0	0
Bn	0	0	0		1	0
	1	1	1		1	0

fig. 4.4 exemple de table pour le calcul de l'ensemble minimal de tests

Le principe pour obtenir l'ensemble minimal de bactéries, est d'essayer d'éliminer le plus de lignes possibles. Après la suppression d'une ligne, la signature globale est recalculée, si elle n'a pas changé, la ligne est définitivement éliminée, sinon elle reste dans la table.

Dans cette première partie, nous avons décrit une nouvelle approche qui doit éliminer les principaux inconvénients du modèle précédent. En effet, cette méthode permet une croissance monotone grâce à la mémorisation, et permet donc une convergence plus rapide.

4.2 Eléments théoriques

La notion de schéma n'est pas valide pour discuter de l'évolution de notre nouvelle méthode puisque nous ne manipulons plus d'individus. Cependant, nous pouvons exprimer la convergence du modèle vers une solution, ainsi que sa complexité temporelle.

4.2.1 Propriété de convergence

Appelons *ResultSet* l'ensemble dans lequel nous sauvegardons les bactéries. Le modèle converge si le score de mutation de cet ensemble augmente d'un génération à l'autre.

Propriété 4.1 : *Apport d'information*. Si un ensemble $\{b_1 \ldots b_n\}$ de bactéries est sélectionné et ajouté dans *ResultSet*, alors le score de *ResultSet* augmente:

$$SM(ResultSet \cup \{b_1 \ldots b_n\}) > SM(ResultSet)$$

En effet, à une génération donnée, la signature partielle des bactéries est calculée sur les mutants encore vivants. Donc, si des bactéries sont sélectionnées, elles tuent des mutants qui n'étaient pas tués par les bactéries déjà présentes dans *ResultSet*, et si elles sont ajoutées à *ResultSet*, le score de cet ensemble va augmenter.

Proposition 4.2 : *Convergence de l'approche*. Soit N le nombre de bactéries dans le bouillon pour une génération, on a:

si $\bigcup_{i=1}^{card(bouillon)} Sign(b_i) \neq \varnothing$ alors $SM(ResultSet_{post}) > SM(ResultSet_{pre})$

sinon $SM(ResultSet_{post}) = SM(ResultSet_{pre})$

Cette proposition se déduit facilement de la propriété précédente. Si la signature globale du bouillon n'est pas nulle, des bactéries vont être sélectionnées et mémorisées dans *ResultSet*. Le score de mutation de *ResultSet* va alors augmenter. Par contre, si les bactéries du bouillon ne tuent aucun nouveau mutant, le score de *ResultSet* ne change pas.

4.2.2 Complexité de l'approche

Pour évaluer la complexité de notre nouvelle approche en termes de temps de calcul de signatures, on prend une approximation de la courbe d'évolution du nombre de mutants vivants au cours du temps. Nous avons obtenu les courbes de la figure 4.5, lors d'expériences sur les classes p_date, p_time et p_date_time (cf. 3.4.2 fig. 3.13). Le nombre de mutants vivants décroît de manière logarithmique suivant une équation du type $-a \cdot \ln(x) + n$. Pour notre problème particulier, x est le nombre de bactéries générées depuis le début du processus, n le nombre initial de programmes mutants et a une constante.

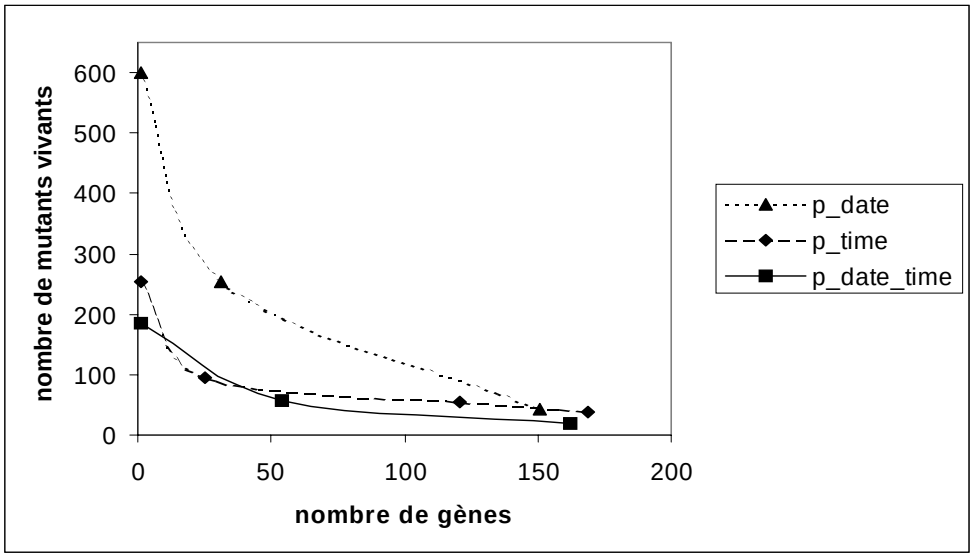


fig. 4.5 évolution du nombre de mutants vivants

Proposition 4.3 : *Complexité de l'approche « adaptation bactériologique »*. La complexité de notre deuxième approche est le nombre d'exécution d'une bactérie sur un mutant c'est-à-dire :

$\sum_{i=1}^{NbGénération} NbBact_i \cdot (-a \ln(\sum_{j=1}^{i} NbBact_j) + NbMutInit)$ où $NbBact_i$ est le nombre de bactéries créés à la i^{ème} génération, et *NbMutInit* le initial de mutants.

Si on considère qu'un nombre égal de bactéries est créé à chaque génération, la complexité s'écrit :

$$\sum_{i=1}^{NbGénération} NbBact \cdot (-a \ln(i \cdot NbBact) + NbMutInit) =$$

$$NbBact \cdot NbGénération \cdot NbMutInit - a \cdot NbBact \cdot \sum_{i=1}^{NbGébération} \ln(i \cdot NbBact)$$

Comme a est une constante positive, dés la seconde génération, cette approche est plus efficace en temps que la première pour un même nombre de bactéries/gènes à chaque génération.

4.3 Premiers résultats

Nous avons expérimenté cette approche sur les classes p_date, p_time et p_date_time de la bibliothèque Pylon en Eiffel (cf. 3.4.2 fig. 3.13). Nous avions les même gènes initiaux que pour l'expérience précédente (3 gènes pour p_date et 2 gènes pour les autres classes). Nous avons généré entre 35 et 45 nouvelles bactéries à chaque génération, suivant les classes, et nous avons fait évoluer l'algorithme jusqu'à obtenir un score supérieur à 90% (3 générations pour p_date et p_date_time, 4 générations pour p_time). La figure 4.6 montre l'évolution du score en fonction du nombre de bactéries.

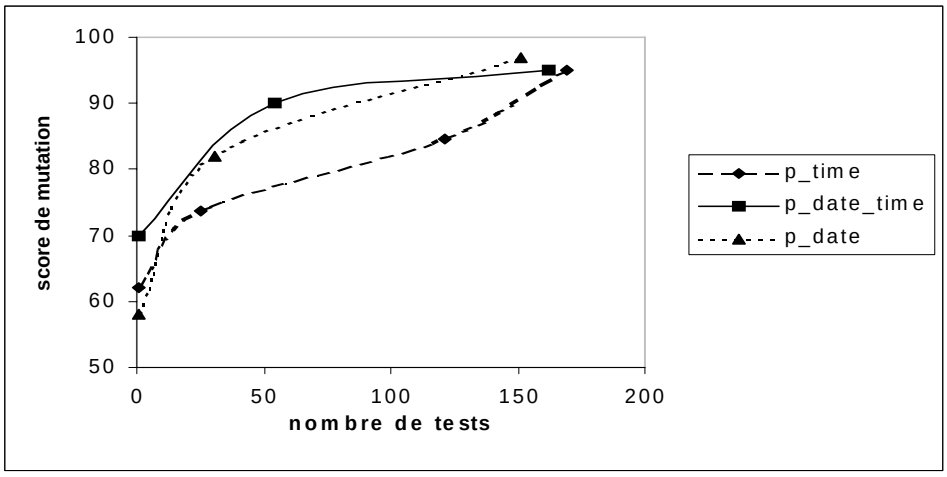


fig. 4.6 expérimentation avec le «bouillon bactériologique »

Les résultats obtenus sont assez encourageants, puisqu'on atteint des scores proches de 100% en générant moins de 200 bactéries. Cependant, il faudra refaire des expériences avec un outil qui automatise ce processus, et qui soit donc plus indéterministe qu'une expérience à la main. Nous pensons développer rapidement un tel outil à partir de TestImprover.

Chapitre 5 Intérêt global d'une approche orientée objets avec contrats

Parallèlement à l'étude de la génération et de l'optimisation automatique de tests dans le cadre des classes auto-testables, nous avons voulu avoir une vision plus globale de la méthodologie proposée. Dans ce chapitre, nous proposons une estimation de la fiabilité basée sur une analyse de la mutation, et nous étudions l'intérêt des contrats pour améliorer la robustesse et la fiabilité d'un système.

5.1 Analyse de fiabilité

Pour effectuer cette analyse, nous devons définir ce qu'est la fiabilité.

Définition 5.1 : *Fiabilité*. On appelle fiabilité *R* la probabilité qu'il n'y ait pas d'erreur à la prochaine exécution du logiciel.

Définition 5.2 : *Taux de défaillance*. Le taux de défaillance *F* est la probabilité qu'il y ait une défaillance à la prochaine exécution du système. $F = 1 - R$.

A la fin de la phase de test et de correction des erreurs détectées, on peut calculer la fiabilité initiale (fig. 5.1). Cette valeur servira à initialiser, certains paramètres nécessaires à la construction d'un modèle de fiabilité [Musa87, Lyu96]. La fiabilité sera ensuite améliorée au cours de la maintenance du logiciel (optimisation de l'implémentation, plus de tests...).

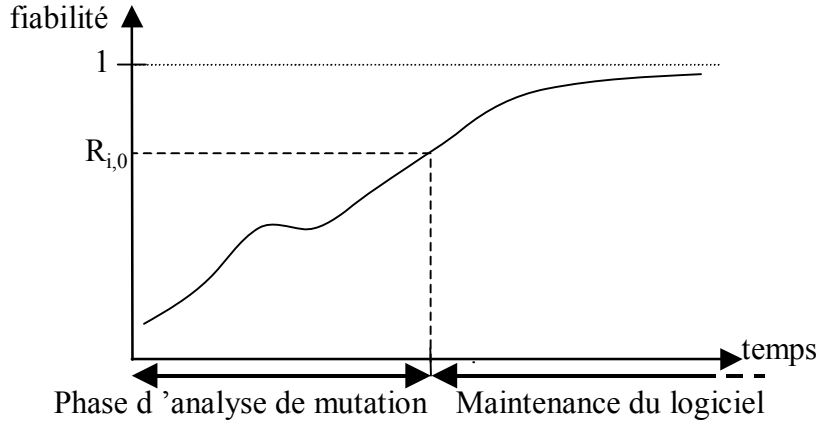


fig. 5.1 évolution de la fiabilité au cours du développement

Il peut paraître étrange de calculer la fiabilité d'un système, c'est-à-dire la probabilité qu'il n'y ait pas d'erreur à la prochaine exécution, dans un cadre de mutation qui consiste justement à injecter des fautes dans le système. Cependant, rappelons brièvement le cadre général dans lequel se place notre étude : nous considérons un composant comme une unité

regroupant sa spécification, son implémentation et ses tests. A partir de ce modèle, nous évaluons la qualité d'un composant grâce au score de mutation de ses tests embarqués. L'analyse de mutation nous sert donc à qualifier un composant, et à partir de cette mesure de qualité nous allons pouvoir calculer le taux de fiabilité du composant puis du système. Le nombre de mutants tués représente alors le nombre de mauvaises implémentations rejetées avec succès par l'ensemble tests et contrats : chaque exécution avec détection d'un mutant signifie une exécution sans défaillance du composant, il s'agit donc d'une exécution fiable.

Pour calculer la fiabilité du système, on considère les C_i composants du système, $i \in [1...n]$.

Soit F_i^0 le taux de défaillance initial, c'est-à-dire la probabilité qu'une défaillance se produise lors de l'exécution de la prochaine instruction. La fiabilité initiale est donc $R_i^0 = 1 - F_i^0$.

Pour mesurer la fiabilité initiale, nous faisons l'hypothèse, pessimiste, suivante, à l'issue de la phase de tests : une défaillance se produit de manière certaine à la prochaine exécution. Sous une telle hypothèse, si $Nstat_i$ instructions sont exécutées avant l'erreur, la probabilité initiale de défaillance est $F_i^0 = 1/Nstat_i$.

La valeur exacte de $Nstat_i$ pourrait être calculée en instrumentant le code source de manière appropriée. Pour simplifier, nous estimons ici la valeur de $Nstat_i$ en multipliant le nombre d'instructions du programme sous test par le nombre de mutants tués :

$$Nstat_i = nb\_de\_mutants\_tués * K_i$$

où K_i est le nombre d'instructions du programme exécutées par les tests. Ceci nous donne une approximation assez satisfaisante du nombre d'instructions exécutées.

Nous pouvons maintenant estimer la fiabilité initiale de deux manières en fonction de l'hypothèse d'indépendance des fautes. D'une part, on peut considérer que les défaillances du système sont indépendantes. Sous cette hypothèse(pessimiste) la fiabilité initiale du système est : $R^0 = \prod_{i=1}^{n} R_i^0$.

D'autre part, on peut faire une hypothèse plus réaliste (qui n'est pas valable dans les systèmes distribués), en considérant qu'une seule instruction est exécutée à la fois.

Proposition 5.1 : Sous l'hypothèse que les instructions du logiciel s'exécutent séquentiellement, nous avons la fiabilité initiale suivante:

$$R^0 = 1 - F^0 = 1 - \frac{1}{\sum_{i=1}^{n} Nstat_i} = 1 - \frac{1}{\sum_{i=1}^{n} \frac{1}{F_i^0}}$$

En effet, si une seule instruction s'exécute à la fois, les n composants vont exécuter leurs $Nstat_i$ instructions avant l'instruction erronée, le taux de défaillance est donc $1/Nstat_i$ d'où l'expression ci-dessus.

Ces deux formules donnent les bornes du domaine dans lequel se trouve la valeur initiale de fiabilité d'un système après une analyse de mutation.

5.2 Robustesse

Définition 5.3 : *Robustesse*. La robustesse Rob_i d'un composant C_i est définie comme la capacité de ce composant à fonctionner même dans des

conditions anormales. Cette capacité permet d'éviter les défaillances catastrophiques. La robustesse peut aussi être vue comme la probabilité qu'une faute soit détectée par un contrat et traitée par un mécanisme d'exception sachant qu'une défaillance se produit certainement sinon. (fig. 5.2)

Définition 5.4 : *Faiblesse*. Par opposition à la robustesse, la faiblesse $Weak_i$ correspond à la probabilité qu'une faute ne soit pas détectée.

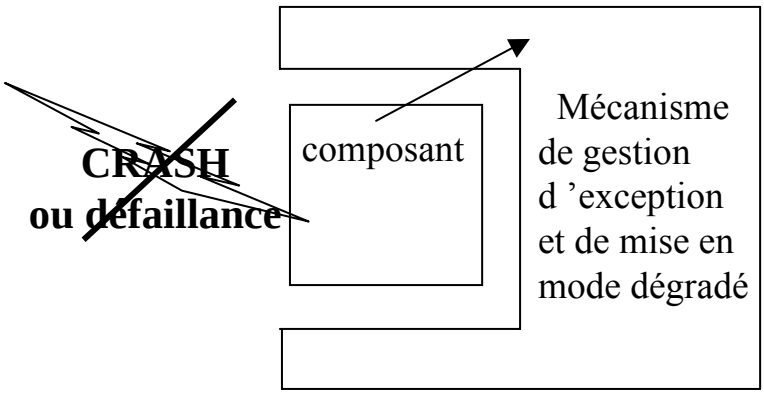


fig. 5.2 Intérêt des contrats pour la robustesse d'un composant

La robustesse correspond donc au pourcentage des fautes détectées par contrats. En effet, si le composant a été conçu par contrats, alors les fautes détectées par les contrats peuvent être retrouvées et un mécanisme d'exception peut éviter qu'une défaillance se produise (fig. 5.2). Dans le cas d'un composant C_i sans contrats, sa robustesse est égale à 0 : $Rob_i = 1 - Weak_i = 0$.

La robustesse d'un composant isolé du système aura une robustesse initiale égale à la force de ses contrats embarqués. Par contre, un composant dans un système aura une valeur de robustesse améliorée par le fait que ses clients ajouteront leurs contrats. Pour expliquer plus précisément l'influence des clients sur la robustesse d'un composant, nous introduisons la notion de *dépendance de test*.

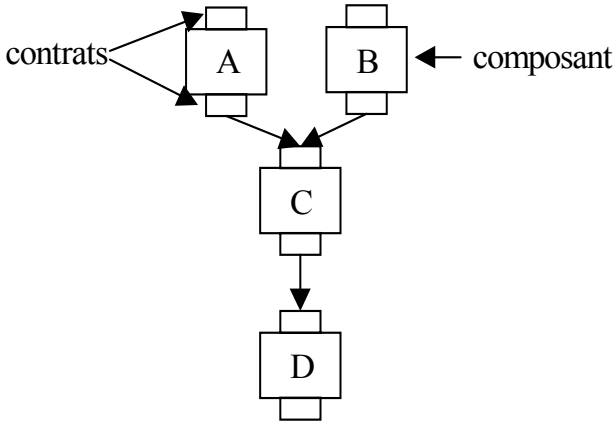


fig. 5.3 exemple pour la dépendance de tests

Définition 5.5 : *Dépendance de test*. Une classe C_i est dépendante d'une classe C_j si elle est cliente de C_j. Cette relation de dépendance est notée: $C_i \, R_{DT} \, C_j$.

Définition 5.6 : Det_i^j . Si C_i R_{DT} C_j, alors la probabilité que les contrats de C_i détectent une faute causée par C_j est notée Det_i^j .

Pour évaluer Det_i^j , on peut utiliser la proportion de mutants détectée par C_i quand il y a des erreurs dans C_j. Même si le relation de dépendance de test est transitive, seules les fautes détectées par un composant client du composant erroné sont prises en compte. Par exemple, sur la figure 5.3 pour le composant D, seules les fautes détectées par C sont prises en compte.

La robustesse $RobDansS_i$ (= 1-$WeakDansS_i$) d'un composant dans un système S est donc la probabilité qu'une erreur ne soit jamais détectée. Cette probabilité est égale à la probabilité qu'une erreur soit détectée localement ($Weak_i$) multipliée par les probabilités que les clients de i ne détectent pas la faute, soit :

$$RobDansS_i = 1 - (Weak_i \cdot \prod_k (1 - Det_i^j)), k / C_k R_{TD} C_i$$

Finalement la robustesse *Rob* du système est donc égale à :

$$Rob = 1 - Weak = 1 - \sum_{i=1}^{n} ProbDef(i) \times WeakDansS_i$$

où *ProbDef(i)* est la probabilité qu'une défaillance vienne du composant sachant qu'une défaillance se produit dans S. Cette probabilité est évaluée grâce à la complexité du composant.

Pour conclure, si on considère qu'une faute détectée par un contrat permet d'assurer que l'exécution ne va pas être interrompue, la fiabilité du système est améliorée de la manière suivante : $R_{new}^0 = 1 - F_{new}^0 = 1 - (F^0 \cdot Weak) = R^0 + F^0 \cdot Rob$

Les paramètres des modèles de fiabilité initiale et de robustesse proposés ici sont faciles à calculer à partir d'une analyse de mutation :

$R_i^0 = 1 - F_i^0$ avec $F_i^0 = 1/Nstat_i$

Rob_i = 1 - $Weak_i$ = pourcentage de mutants détectés par contrats

Det_i^j = pourcentage de mutants de C_i détectés par les contrats de C_j.

ProbDef(i) = 1/n

5.3 Expériences

Pour évaluer le valeur de Det_i^j , une expérience a été réalisée sur les classes qui gèrent le temps de la bibliothèque Pylon(cf. fig. 3.13 dans 3.4.2). L'expérience a consisté à estimer le taux de détection des erreurs par l'autotest de p_date_time.e en ayant injecté des fautes dans p_date.e et p_time.e (fournisseurs de p_date_time). La méthode est la suivante :

- On supprime toutes les méthodes de p_time.e non utilisées par p_date_time.e. Ceci permet de voir l'effet de notre programme de test (de p_date_time.e) sur les mutants des méthodes réellement utilisées (de p_time.e).
- On génère les mutants sur cette partie du programme p_time.e.
- On lance l'autotest de p_date_time.e.
- On observe le pourcentage de mutants tués.

On effectue les mêmes opérations avec p_date.e au lieu de p_time.e. Les résultats sont répertoriés dans la table 5.4.

La raison pour laquelle le score n'est pas de l'ordre de 100 % vient de ce que même si une méthode m est utilisée par une autre classe, il arrive que tout son code ne soit pas utilisé :

dans ce cas, l'autotest local ne détecte pas la faute due à l'environnement fournisseur pour la simple raison que la classe n'utilise pas la portion de code infectée dans m. Les résultats montrent toutefois que 60-80% des fautes prévisibles liées à l'environnement extérieur à une classe ont pu être détectées localement par l'autotest de la classe.

Fichiers mutés	**p_date.e**	**p_time.e**
nb de méthodes au total	19	12
nb de méthodes utilisées	14	11
nb de mutants générés	350	161
nb de mutants équivalents	33	8
Nb de mutants tués	195	114
% de mutants tués = Det_j^i	**61,51%**	**74,50%**

Tableau 5.4 : Robustesse de l'autotest de p_date_time

5.4 Illustration

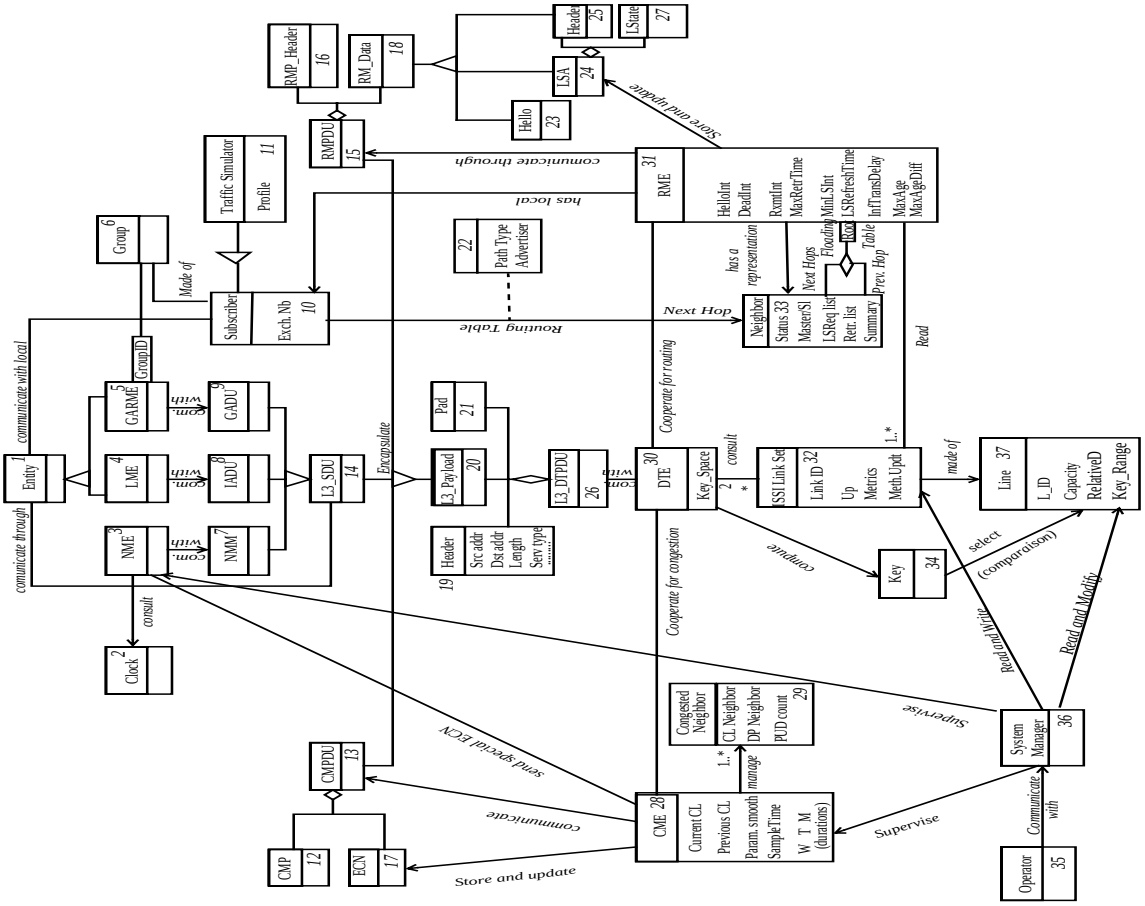


fig.5.5 Diagramme de classe du système SMDS

Grâce à l'expérience précédente, le gain en robustesse d'un système peut être évalué. On considère le système SMDS (fig. 5.5) qui regroupe 37 composants et qui a été étudié dans [Jéron99]. Nous fixons les valeurs comme suit afin d'évaluer l'amélioration globale de la robustesse due à l'usage systématique de contrats :

$Rob_i = 1 - Weak_i = 0.85$, $Det_i^j = 0.7$ et $ProbDef(i)$ = 1/n = 1/37.

Nous considèrons qu'il y a en moyenne 100 tests par cas de test et qu'on génère en moyenne 200 mutants par composant. Enfin, un test exécute environ 10 instructions. Le taux de défaillance initial d'un système de 37 composants est donc égal à : $F^0 = 1.35 \cdot 10^{-7}$.

La faiblesse d'un système avec ses contrats est égale à Weak = 0.031 alors qu'elle vaut 1 sans les contrats, ce qui signifie qu'une erreur sur 30 sera détectée si elle apparaît. La robustesse globale du système a donc été améliorée de 30, ce qui correspond à une usage efficace des contrats. Cependant, un usage simple des contrats permet d'améliorer facilement de 10 la robustesse. On peut donc situer l'amélioration de la robustesse entre 10 et 40, ce qui, dans tous les cas, est une amélioration significative.

Le taux de défaillance initial a été réduit dans le même ordre de grandeur grâce au fait que les fautes peuvent être retrouvées facilement quand on utilise des contrats : le nouveau taux de défaillance est donc $F^0_{new} = 4.2 \cdot 10^{-9}$.

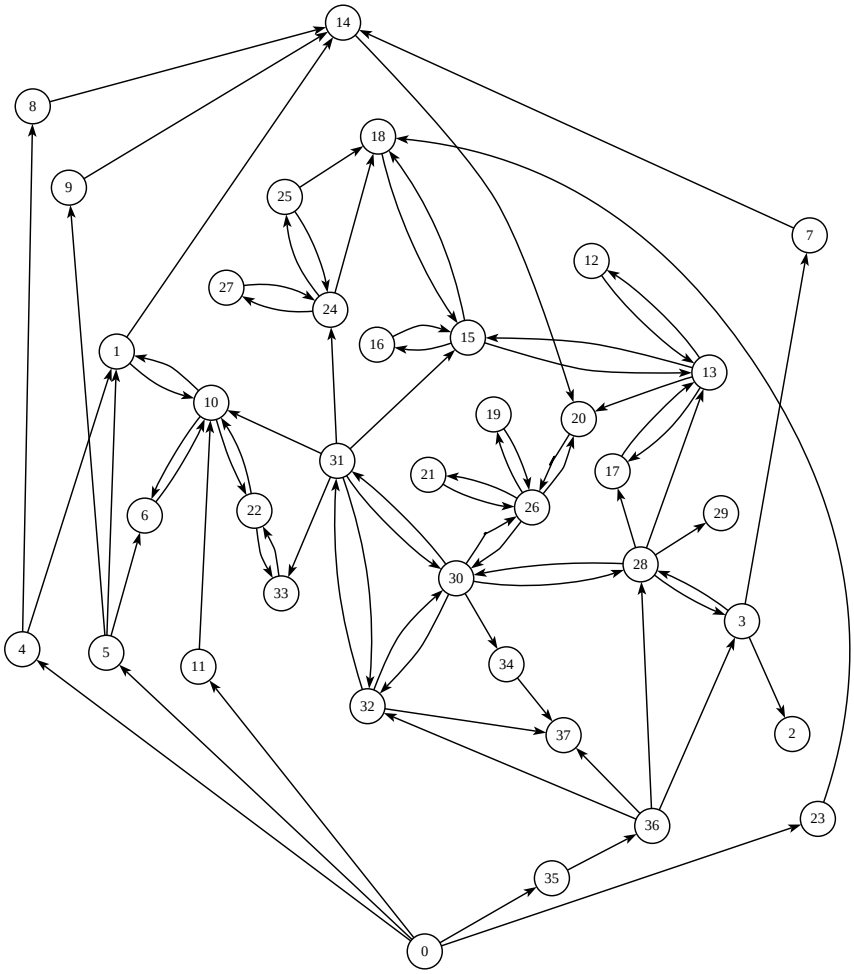


fig. 5.6 Graphe de dépendances de tests pour le système SMDS

Toujours avec le système SMDS, sur la figure 5.7 nous avons observé l'évolution de la robustesse totale du système en fonction de la robustesse des composants. Nous avons tracé plusieurs courbes qui correspondent à différentes qualités de contrats des clients, c'est-à-dire, différentes valeurs de Det_i^j.

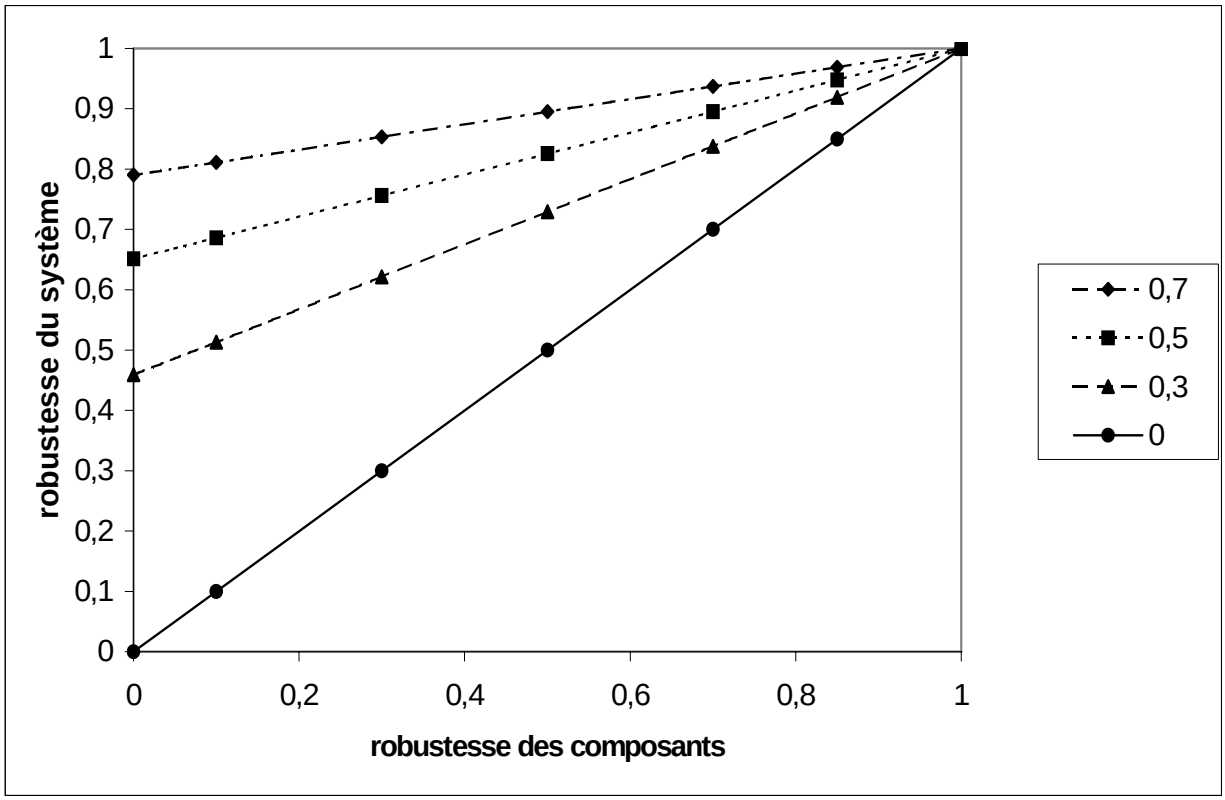


fig.5.7 robustesse du système en fonction de la robustesse des composants et de Det_i^j

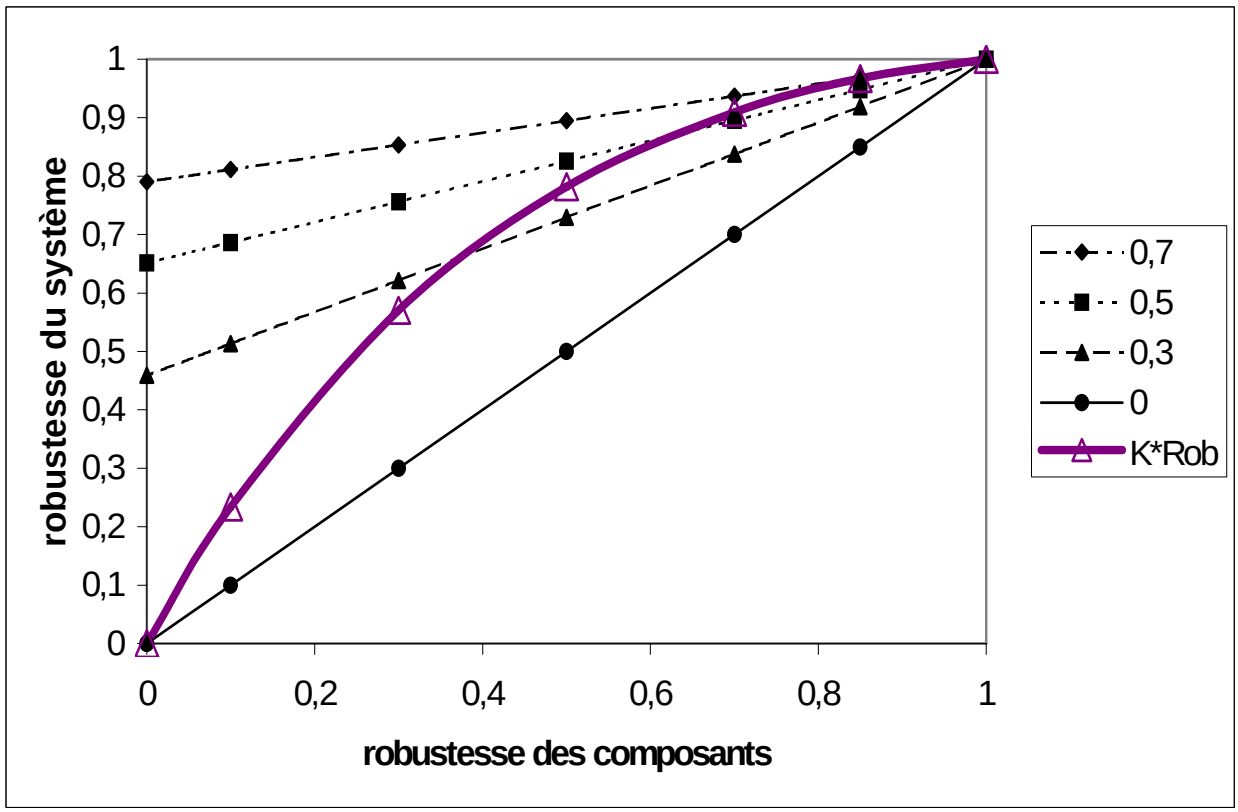


fig. 5.8 évolution réelle de la robustesse du système

Ensuite, si on considère que Det_i^j peut être liée à la robustesse du composant i de la manière suivante : $Det_i^j = K \cdot Rob$, alors la robustesse du système évolue comme le montre la courbe sur la figure 5.8, avec *K=0.8*. Cette courbe correspond en fait à la véritable évolution du système. En effet, les premiers contrats sont souvent faibles, le développement consistant en partie à les rendre plus efficace. Au cours de cette phase de développement, la robustesse des composants et vont augmenter en parallèle, et la robustesse du système évolue selon le modèle de courbe décrit sur la figure 5.8.

Les deux figures 5.7 et 5.8 illustrent bien le fait que les contrats représentent un moyen efficace d'améliorer la robustesse d'un système et, indirectement, la fiabilité initiale. En effet, les contrats permettent d'améliorer la robustesse d'un composant isolé, mais il permettent aussi d'améliorer la robustesse des classes clientes, c'est-à-dire d'augmenter Det_i^j. Nous pouvons en déduire que des expériences plus nombreuses permettront de montrer l'intérêt de la conception par contrats en termes de fiabilité et de robustesse.

Chapitre 6 Conclusion

Partant du constat qu'il est facile d'écrire des test atteignant un score de 60%, mais qu'augmenter ce score réclame un effort important, nous avons essayé, au cours de l'étude présentée ici, d'appliquer un algorithme génétique pour améliorer automatiquement cet ensemble initial de tests. Les expériences sur ce modèle nous ont permis de constater qu'une approche génétique classique n'est pas adaptée à notre problème. Cependant, l'analyse de ces résultats nous a aidé à modéliser une approche plus adaptée, qui améliore l'efficacité des algorithmes génétiques en conservant une exploration semi-aléatoire du domaine des solutions. Par la suite, il faudrait valider cette deuxième approche grâce à une étude de cas plus importante. De plus, ces expériences permettrait de paramétrer plus précisément la méthode.

D'autre part, nous avons présenté l'intérêt global d'une conception par contrats. Nous avons proposé un modèle de fiabilité d'un système en fin de phase de tests. Ce modèle est informé et paramétré grâce aux résultats d'une analyse de mutation. Enfin, nous avons estimé le gain de robustesse d'un composant et d'un système que permet une conception par contrats.

Références bibliographiques

[Baudry2000a] B. Baudry, Y. Le Traon, H. Vu Le, "Testing-for-Trust: the Genetic Selection Model applied to Component Qualification", In proceedings of TOOLS'2000 (Technology of Object Oriented Languages and Systems), pp. 108-119, June 2000.

[Baudry2000b] B. Baudry, Y. Le Traon, H. Vu Le, " Building trust into OO Components using a genetic Analogy", *to be presented* at International Symposium on Software Reliability Engineering 2000, San Jose, October 2000.

[Beizer90] B. Beizer, "Software testing techniques", Van Norstrand Reinhold, 1990. ISBN 0-442-20672-0.

[DeMillo78] R. DeMillo, R. Lipton, F. Sayward, "Hints on Test Data Selection : Help For The Practicing Programmer", Computer, Vol. 11, No. 4, pp. 34-41, April 1978.

[Deveaux2000] D. Deveaux, R. Fleurquin, P. Frison, J.-M. Jézéquel, Y. Le Traon, "Composants Objets Fiables : une approche pragmatique", L'Objet, Vol. 5, No. 34, Mars 2000.

[DeMillo91] R. DeMillo, A. Offutt, "Constraint-Based Automatic Test Data Generation", IEEE Transactions In Software Engineering, Vol. 17, No. 9, pp. 900-910, September 1991.

[Goldberg89] D. E. Goldberg, "Genetic Algorithms in Search, Optimization and Machine Learning", Addison Wesley, 1989. ISBN 0-201-15767-5.

[Harrold99] M. J. Harrold, R. P. Pargas, R. R. Peck, "Test-Data Generation Using Genetic Algorithms", Journal of Software Testing, Verification and Reliability, 1999, to appear.

[Holland70] J. H. Holland, "Robust algorithms for adaptation set in general formal framework", Proceedings of the 1970 IEEE symposium on adaptative processes (9^{th}) decision and control, 5.1 –5.5, December 1970.

[Jéron99] T. Jéron, J-M. Jézéquel, Y. Le Traon, and P. Morel, "Efficient Strategies for Integration and Regression Testing of OO Systems", In proc. of the 10th International Symposium on Software Reliability Engineering (ISSRE'99), November 1999, Boca Raton (Florida), 260-269.

[Jézéquel99] J-M. Jézéquel, M. Train, C. Mingis, "Design Patterns and Contracts", Addison Wesley, 1999. ISBN 0-201-30959-9.

[Jones96] B. F. Jones, H.-H. Sthamer, D. E. Eyres, "Automatic structural testing using genetic algorithms", Software Engineering Journal, Vol.11, No.5, pp. 299-306, September 1996.

[Korel90] B. Korel, "Automated Software Test Data Generation", IEEE Transactions on Software Engineering, Vol.16, No. 8, pp. 870-879, August 1990.

[Le Traon99] Y. Le Traon, D. Deveaux, J.-M. Jézéquel, "Self-testable components: from pragmatic tests to a design-for-testability methodology," In proc. of TOOLS-Europe'99. TOOLS, Nancy (France) pp. 96-107, June 1999.

[Lyu96] M. Lyu, "Handbook of Software Reliability Engineering" McGraw Hill and IEEE Computer Society Press, 1996, ISBN 0-07-0349400-8.

[Meyer88] B. Meyer, "Object-Oriented Software Construction", Prentice Hall, 1988. ISBN 0-13-629040-3.

[Meyer92] B. Meyer, "Applying design by contract", IEEE Computer, pp. 40-51, October 1992.

[Michael97] C. C. Michael, G. E. McGraw, M. A. Schatz, C. C. Walton, "Genetic Algorithms for Dynamic Test Data Generation", Technical Report RSTR-003-97-11, May 1997.

[Musa 87] J. D. Musa, A. Iannino, K. Okumoto, "Software Reliability: Measurement, Prediction, Application", McGraw Hill, 1987, ISBN 0-07-044093-X.

[Offutt96] J. Offutt, J. Pan, K. Tewary, T. Zhang "An experimental evaluation of data flow and mutation testing", Software Practice and Experience, Vol. 26, No. 2, pp. 165-176, February 1996.

[Offut97] J. Offut, J. Pan, "Automatically Detecting Equivalent Mutants and Infeasible Paths", The Journal of Software Testing, Verification, and Reliability, Vol. 7, No.3, pp. 165-192, September 1997.

[Rosenzweig95] M. L. Rosenzweig , "Species diversity in space and time", Cambridge University Press, Cambridge UK, 436p, 1995.

[Voas92] J. Voas, "PIE: A Dynamic Failure-Based Technique", IEEE Transactions on Software Engineering, vol.18, pp. 717-727, 1992.

[Wadekar99] S. A. Wadekar, S. S. Gokhale, "Exploring cost and reliability tradeoffs in architectural alternatives using a genetic algorithm", In proc. of the 10th International Symposium on Software Reliability Engineering (ISSRE'99), Boca Raton (Florida), pp. 104-113, November 1999.