# Techniques de test statique de logiciel

20 avril 1994

**Gilles Bernot*** **
gb@lri.lri.fr

**Marie Claude Gaudel***
mcg@lri.lri.fr

**Bruno Marre***
marre@lri.lri.fr

* **LRI**, URA CNRS 410, Université PARIS-SUD, F-91405 Orsay cedex

** **LIVE**, Université d'Evry, F-91025 Evry cedex

L'objectif de ce rapport est de présenter un état de l'art sur les techniques de *test statique de logiciel*. Ces techniques visent à examiner et analyser les sources du produit logiciel sous test sans véritable exécution de son code.

Ce rapport complète le rapport précédent sur les techniques de *test dynamique* de logiciel (rapport Aérospatiale DCR/Q 120488-91 pour le CNES, centre commun de recherches Louis-Blériot, décembre 1991).

La bibliographie fut établie d'une part à partir des éléments déjà possédés par les auteurs, et d'autre part à partir d'une interrogation de la base de données INSPEC.

Les principales conclusions de ce rapport sont les suivantes : le test statique est une méthode remarquablement efficace et souvent d'un coût raisonnable ; dans le domaine du parallélisme, ce type de méthode va prendre une place prépondérante motivée par les problèmes posés par le test dynamique et les avancées techniques actuelles.

## Introduction

Une technique de test est dite statique si elle ne requiert pas l'exécution du logiciel sous test sur des données réelles. Chaque technique de test statique considérée dans cette étude repose sur le schéma suivant.

- On se donne d'abord des critères de conformité sur les sources du logiciel (voire même sur les spécifications). Ils ne doivent pas porter sur le comportement du produit logiciel lors de son exécution. Par exemple, on peut chercher des anomalies telles que l'existence de portions de code isolées ou un mauvais usage des variables ou encore des déclarations et des appels de procédures incompatibles. Ces critères peuvent cependant porter sur des assertions ou des propriétés des composants logiciels qui peuvent être déduites du texte du logiciel.

- On extrait ensuite du texte du logiciel un modèle "mathématique" (par exemple le graphe de contrôle ou le flot de données). Ce modèle est plus simple que les sources du logiciel et doit permettre de vérifier si les critères fixés sont respectés. Le modèle mathématique retenu doit donc être judicieusement choisi en fonction des critères fixés. Par exemple, s'il s'agit de tester l'existence de code isolé, le graphe de contrôle est un modèle approprié.

Il s'agit donc de techniques de test "boîte blanche". En règle générale, la vérification des critères de conformité est peu coûteuse et s'avère très efficace. Les outils éventuellement nécessaires font appel à des techniques proches de la compilation, ils sont indépendants du produit logiciel sous test et existent pour la plupart des environnements de développement. Les résultats rencontrés dans la littérature font état d'un taux de découverte d'erreurs supérieur à 80%, globalement.

Le test statique couvre un très large spectre allant de l'inspection des sources d'un logiciel, qui ne nécessite quasiment aucun moyen, jusqu'à la preuve de logiciel, qui implique la mise en œuvre de démonstrateurs de théorèmes très puissants ainsi qu'une forte expertise (et créativité) du manipulateur.

Remarquons que certaines techniques comme l'évaluation symbolique simulent une exécution du logiciel sous test, et ce sur des données symboliques. Ces techniques seront donc considérées ici comme des méthodes de test statique, puisqu'elles ne nécessitent pas l'exécution effective du logiciel sur des données réelles. L'évaluation symboliques est même l'une des techniques les plus représentatives du test statique.

Les techniques de test statique retenues pour cette étude peuvent être réparties selon les classes suivantes, par ordre croissant de sophistication.

- L'inspection, c'est-à-dire la vérification de la correction des fonctionnalités d'un morceau de code (ou de spécification) selon des normes et un protocole précis.

- Les critères statiques élémentaires, c'est-à-dire tout critère pouvant être pris en compte dans un "bon compilateur" (syntaxe, typage, cohérence des interfaces entre modules ...).
- Les analyses d'anomalies, telles que les portions de code isolées ou les mauvais usages des variables ou des entrées/sorties.
- L'évaluation symbolique qui consiste à exécuter abstraitement le logiciel sur des valeurs symboliques. (Ceci inclus entre autres l'insertion d'assertions dans les sources du logiciel et leur vérification par évaluation symbolique et résolution de contraintes)

Chacune de ces classes fera l'objet d'une section. Nous ajoutons par ailleurs une section traitant des aspects propres au parallélisme. Une bibliographie succinte est fournie en fin de chaque section.

Presque toutes les techniques présentées ne sont applicables qu'en phase de test unitaire hormis l'inspection (qui peut aussi porter sur les spécifications de conception, intermédiaires ou détaillées), l'analyse de cohérence d'interfaces et certaines techniques de test statique propres au parallélisme.

# 1 Inspection

Après plusieurs années d'expérimentation chez IBM, les inspections ont été proposées par Fagan au milieu des années soixante-dix [Fag 76]. La méthode a été régulièrement améliorée depuis [Fag 86, Fag 89]. Elle part du principe que les erreurs sont souvent des conséquences de mauvaises interprétations entre deux étapes de conception ou de codage. Ceci s'applique aussi bien à des sources de logiciel (par rapport aux spécifications détaillées) qu'aux spécifications détaillées (par rapport aux spécifications intermédiaires) ou aux spécifications intermédiaires (par rapport aux spécifications globales). Les méthodes d'inspection ont pour but de prévenir, ou au moins de minimiser, ces erreurs d'interprétations.

Essentiellement, une inspection est une interprétation partielle du "sens" d'un composant logiciel (ou d'une spécification) à la lumière d'un ensemble de règles bien établies. Nous n'entendons pas ici une vérification systématique de normes ou de règles de programmation élémentaires, mais plutôt, la vérification sémantique des documents traités. L'aspect important dans l'application des méthodes d'inspection est de fournir un point de vue indépendant de l'auteur du document, si bien qu'il est peu probable de reproduire les mêmes erreurs d'interprétations. Ceci implique en pratique de faire intervenir au moins deux personnes, ou plus. Ces méthodes s'appliquent aussi bien à la conception de logiciel qu'à la conception de matériel.

L'équipe d'inspection travaille à partir d'un ensemble prédéfini de règles, prépare un rapport et peu parfois interagir avec les concepteurs ou programmeurs.

D'après Fagan, les équipes les plus efficaces sont constituées comme suit.

- Un modérateur qui doit préserver l'objectivité et garantir l'intégrité de l'inspection, c'est le personnage clef de l'équipe.
- Le chef de projet responsable de la conception du logiciel.
- Le programmeur qui a produit la portion de code traité doit être présent pour sa connaissance détaillée du code.
- Le responsable des tests.

Cette équipe a pour but de trouver des erreurs (c'est donc bien du test). Toute erreur trouvée doit donner lieu soit à une solution immédiate soit à une correction ultérieure (laissée sous la responsabilité du programmeur) et dans ce cas un jeu de tests dynamique ciblé sur cette erreur est prévu.

Avec des règles bien établies (consistant le plus souvent en une liste type de questions et de points à vérifier) et du personnel expérimenté, on traite environ de 50 à 100 instructions par heure. Les nombreuses expériences ont confirmé qu'on arrive à détecter et corriger ainsi de l'ordre d'au moins 75% des fautes. La mise en œuvre systématique de la méthode à JPL est décrite dans [Bus 91] où elle s'avère être très fructueuse sur des projets de la NASA. L'auteur affirme que chaque inspection a économisé environ 25000$. Plus généralement, cette méthode a de toute évidence été un succès chaque fois qu'elle a été mise en œuvre systématiquement [Fow 86, ABL 89, Kel 89, Bus 91, SP 90].

## Références bibliographiques


- [ABL 89] A.F. Ackerman, L.S. Buchwald, F.A. Lewski
  *"Software inspections: an effective verification process"*
  IEEE Transactions on Software Engineering, mai 1989, pp. 31-36

- [Bus 91] M. Bush
  *"Improving software quality: the use of formal inspections at the Jet Propulsion Laboratory"*
  IEEE 12th International Conference on Software Engineering, Nice, mai 1991, pp. 196-199

- [Fag 76] M.E. Fagan
  *"Design and code inspections to reduce errors in program development"*
  IBM System Journal, Vol.15, No 3, pp. 182-211

- [Fag 86] M.E. Fagan
  *"Advances in inspections"*
  IEEE Transactions on Engineering, juillet 1986, pp. 744-751

- [Fag 89] M.E. Fagan
  *"Productivity improvement through defect-free software development"*
  Quality Week 1989, Sotware Research Inc., San Francisco 1989

- [Fow 86] P.J. Fowler
  *"In-process inspections of products at* AT&T*"*
  AT&T Technical Journal, mars-avril 1986, pp. 102-112

- [Kel 89] J.C. Kelly
  *"Formal inspection review technics: experience and results"*
  Quality Week 1989, Sotware Research Inc., San Francisco 1989

- [Mye 78] G.J. Myers
  *"A controlled experiment in program testing and code walkthroughs/inspections"*
  C. ACM, septembre 1978

- [SP 90] C. Sayet, E. Pilaud
  *"An experience of critical software development"*
  Proc. 20th IEEE Int. Symp. on fault tolerant computing (FTCS-20), Newcastle upon Tyne, UK, juin 1990, pp. 36-45

# 2 Critères statiques élémentaires

Ces critères sont vérifiables directement à partir des sources d'un logiciel sans élaboration d'un modèle "mathématique" intermédiaire. Ils sont généralement vérifiés par les outils standards disponibles avec la plupart des compilateurs. Leur objectif est de détecter des usages impropres des constructions du langage de programmation qui peuvent révéler la présence d'erreurs. Les vérifications effectuées concernent les points suivants :

- la correction syntaxique (évidemment) ;
- de bonnes règles de typage (eventuellement plus strictes que celles du langage) : la portée de ces vérifications est fortement dépendante de la permissivité du langage de programmation ;
- le nombre d'arguments d'une fonction ou d'une procédure et leur type ;
- les coercions abusives de types (surcharge d'identificateurs, ...) ;
- certains usages impropres de pointeurs ;
- les variables, constantes, fonctions ou procédures déclarées mais jamais utilisées ;
- dans certains langages (par exemple C) la présence de structures utilisées mais non déclarées ;
- l'usage d'expressions arithmétiques ou logiques ayant une valeur constante (propagation de constantes) ;
- les appels de fonctions dont la valeur retournée n'est pas utilisée (en particulier pour le langage C) ou est utilisée dans un mauvais contexte (mauvais type) ;
- les portions de programme non portables, ou incompatibles avec d'autres dialectes du même langage ;
- les entrées ou sorties de boucles hors des points d'entrée ou de sortie standard ;
- l'existence d'instructions non atteignables (sauts inconditionnels interdisant leur accès) ;

- la présence d'arguments de fonction (ou procédure) jamais utilisés dans son code ;
- la compatibilité avec les bibliothèques ou modules utilisées (en particulier la cohérence des interfaces de modules) ;
- la présence d'une bibliothèque ou d'un module (ou d'une fonction ou procédure) importée mais jamais utilisée ;
- etc.

De plus certains outils produisent des rapports d'analyse des sources du programme permettant d'effectuer des contrôles dépendants directement de l'application. Ces rapports peuvent contenir des diagrammes structurels, des tables de références croisées, les niveaux d'imbrication (et donc la portée) des procédures et fonctions, le nombre d'instructions non commentées, etc.

## Références bibliographiques

Tout manuel d'utilisation d'un langage de programmation.

# 3 Analyses d'anomalies

Nous regroupons ici toutes les anomalies dont la vérification nécessite la construction d'un modèle intermédiaire. Ce modèle est en général un graphe duquel on peut extraire des chemins. Il s'agit le plus souvent du graphe de contrôle ou du flot de données. Le principe général de l'analyse d'anomalies est de maintenir un certain prédicat le long des chemins construits. Ce prédicat traduit un ou plusieurs des critères de conformité retenus. Par exemple, on peut tester pour tout chemin du flot de données si toute variable est définie avant d'être utilisée.

## 3.1 Anomalies du graphe de contrôle

La première étape de l'analyse d'anomalies est de construire un graphe de contrôle dont les nœuds sont les blocs d'instructions séquentiels et les arcs représentent les transferts de contrôle. Il ne comporte généralement qu'un seul nœud d'entrée et peut être complété pour ne contenir qu'un seul nœud de sortie. Les chemins de ce graphe ne sont pas tous exécutables car ils peuvent correspondre à des séquences de conditions contradictoires. Déterminer dans le cas général la faisabilité d'un chemin est impossible. Cependant, la plupart du temps, des techniques de preuve intéractives ou des outils de programmation logique peuvent assister la détection de chemins infaisables.

En exploitant directement le graphe de contrôle on peut calculer les informations suivantes :

- la présence de code isolé (nœuds qui ne sont pas dans la composante connexe du point d'entrée) ;
- certaines métriques comme la profondeur d'imbrication des boucles, la longeur maximale d'un chemin (nombre de transferts de contrôle), le nombre cyclomatique ( $(\textit{nombre de nœuds}) - (\textit{nombre d'arcs}) + 2$ ) ...
- le séquencement correct de certaines instructions, par exemple l'ouverture d'un fichier avant toute lecture ou écriture et sa fermeture avant la fin d'un chemin (un telle approche est utilisé depuis longtemps pour tester des systèmes d'exploitations) [HA 73, OO 86, OO 88].

## 3.2 Anomalies du flot de données

La majorité des analyses d'anomalies repose sur le flot de données. L'analyse du flot de données est une méthode initialement utilisée pour l'optimisation

de code [ASU 89]. Ici l'accent est porté sur la manipulation des variables. On peut extraire du graphe de contrôle les manipulations de données qui relient les données d'entrée aux données de sortie. On s'intéresse en particulier aux nœuds contenant au moins une affectation de variable ou faisant référence à la valeur d'une variable. Les anomalies recherchées concernent essentiellement l'ordre de définition (affectation) et d'utilisation des données dans chacun des chemins du flot de données. Ces anomalies sont :

- une variable est utilisée mais non définie (non affectée) ;
- une variable est définie mais non utilisée, ou non utilisée depuis sa dernière redéfinition ;
- une variable est redéfinie sans avoir été utilisée depuis sa précédente définition.

La première anomalie est clairement une erreur ou un abus de convention par rapport à la définition du langage. Les autres anomalies ne sont pas nécessairement des erreurs mais peuvent révéler une erreur de programmation. Toute portion de programme comportant ces anomalies doit être particulièrement contrôlée.

Remarquons que le traitement d'alias, de pointeurs ou de tableaux n'est pas bien pris en compte par les techniques d'analyse du flot de données.

De même que pour le graphe de contrôle, on peut aussi utiliser une extension du flot de données pour vérifier des contraintes de séquencement d'événements particuliers [00 86, 00 88]. On utilise alors une description des séquences d'événements à observer qui sont comparées avec le flot de données.

La mise en pratique des techniques d'analyse de flot de données sur des programmes de taille réaliste a été rendue possible par la découverte et l'élaboration de techniques dites "interprocédurales" [SP 81] qui permettent d'analyser le flot des données entre procédures en faisant abstraction des variables locales.

Une autre avancée significative est la technique dite de "program slicing" qui permet de délimiter dans un programme les parties dont dépend l'évolution d'une variable. La connaissance de ces dépendances permet des analyses indépendantes et plus efficaces, en particulier quand des changements sont effectués dans un programme [Wei 84].

## Références bibliographiques


- [ASU 89] A. Aho, R. Sethi, J. Ullman
  *"COMPILATEURS Principes, techniques et outils"*
  InterÉditions, Paris, ISBN 2-7296-0295-X, 1989

- [FO 76] L.D. Fosdik, L.J. Osterwell
  *"Data flow analysis in sofware reliability"*
  Computing Surveys, Vol. 8, No. 3, pp. 305-330, 1976

- [HA 73] J.H. Howard, W.P. Alexander
  *"Analyzing sequences of operations performed by programs"*
  Program Test Methods, W.C. Hetzel, Ed. Englewood Cliffs, Prentice-Hall, 1973

- [OO 86] K.M. Olender, L.J. Osterweil
  *"Specification and Static Evaluation of Sequencing Constraints in Software"*
  Workshop on Software Testing, Banff Canada, IEEE Catalog Number 86TH0144-6, pp. 14-22, juillet 1986

- [OO 88] K.M. Olender, L.J. Osterweil
  *"Interprocedural Static Analysis of Sequencing Constraints"*
  A paraître dans TOSEM 1993

- [SP 81] M. Sharir, A. Pnueli
  *"Two approaches to interprocedural data flow analysis"*
  in Program Flow Analysis, Muchnick et Jones ed., Prentice-Hall, 1981, pp. 189-233

- [Wei 84] M. Weiser
  *"Program slicing"*
  IEEE Transactions on Software Engineering, SE-10,1984, pp. 352-357

# 4 Evaluation symbolique

L'évaluation symbolique n'exécute pas le programme au sens traditionnel du terme. La notion traditionnelle d'exécution requiert que des chemins sélectionnés du logiciel soient exercés via des données réelles alors que pour l'évaluation symbolique les données réelles sont remplacées par des valeurs symboliques. Ainsi, l'évaluation symbolique produit en sortie un ensemble d'expressions (une par variable de sortie) au lieu d'un ensemble de valeurs réelles. Cette technique est un moyen terme entre le test dynamique et la preuve. Depuis le milieu des années soixante-dix, il existe bon nombre de systèmes d'évaluation symbolique [BEL 75, Cla 76, Kin 76, RHC 76, VGA 80].

On distingue trois techniques d'évaluation symbolique :

- *l'exécution symbolique* qui, étant donné un chemin dans le graphe de contrôle, construit un prédicat reflétant les conditions du chemins et une ou plusieurs expressions symboliques définissant les valeurs symboliques de sortie en fonctions des valeurs symboliques d'entrée ;

- *l'exécution symbolique dynamique* qui, étant donnés des données réelles et un graphe de contrôle, construit un ou des résultats réels, le chemin emprunté par l'exécution et enfin les expressions symboliques qui reflètent le calcul effectué le long de ce chemin ;

- *l'évaluation symbolique globale* qui, étant donné un graphe de contrôle, construit un ensemble de triplets formés d'un chemin, du prédicat sur les variables d'entrées qui caractérise ce chemin et enfin d'un ensemble d'expressions symboliques (une par variable de sortie) qui reflètent les calculs du chemin.

L'exécution symbolique peut être considérée comme un outil de trace symbolique. L'exécution symbolique dynamique peut être considéré comme un simple outil de trace perfectionné. Enfin, l'évaluation symbolique globale est l'outil le plus général mais subit des limitations relatives au langage de programmation traité ou, au mieux, des limitations sur les constructions reconnues.

La construction d'une expression symbolique peut être effectuée selon deux méthodes.

- L'expension vers l'avant part du nœud d'entrée et modifie successivement l'expression associée à toute variable en accord avec les instructions d'affectations rencontrées. Par exemple, si l'expression associée à la variable

`i` est `exp` et l'instruction rencontrée est `i := i+1` alors la nouvelle expression associée à `i` est `exp+1`. A la sortie du chemin, il suffit alors de ne retenir que les expressions associées aux variables de sortie.

- La substitution vers l'arrière part du nœud de sortie en associant d'abord à chaque variable de sortie "`v`" l'expression "`v`" elle même. Elle remonte ensuite le chemin choisi en effectuant les substitutions indiquées par les instructions d'affectation (ceci pour l'expression associée à chacune des variables de sortie). Par exemple, si l'expression courante associée à la variable `v` est `i+3` et si l'instruction rencontrée est `i := i+1` alors la nouvelle expression associée à la variable `v` est `(i+1)+3`.

Notons que la première approche impose de gérer un plus grand nombre d'expressions, puisqu'elle doit s'appliquer à toutes les variables (pas seulement les variables de sortie). Elle présente par contre l'avantage de faciliter le calcul des prédicats de branchement, permettant éventuellement de détecter plus tôt certains chemins infaisables. C'est par ailleurs la seule approche compatible avec l'exécution symbolique dynamique.
La seconde approche présente l'avantage de focaliser les calculs sur les variables de sortie uniquement. Notons qu'elle est fondée sur les mêmes techniques que la logique de Hoare (décrite dans la fiche sur la preuve du rapport DCR/Q 120488-91).
Enfin il faut souligner que la simplification des expressions est capitale pour que l'évaluation symbolique soit utilisable en pratique.

La prise en compte des boucles ou des itérations est une difficulté importante pour l'évaluation symbolique et requiert souvent l'introduction d'invariants de boucles. Sinon, pour chaque boucle on peut décider arbitrairement d'effectuer un nombre fixé d'itérations (une approche pragmatique est d'effectuer successivement de zéro à deux itérations).

L'examen des expressions d'un chemin ainsi que de son prédicat est souvent utile pour découvrir des erreurs de calcul. Ceci est particulièrement vrai pour des applications scientifiques où il est souvent difficile de calculer à la main (et de façon précise) les résultats attendus (l'expression est plus parlante que les résultats individuellement obtenus par test dynamique). Un autre usage de l'évaluation symbolique est d'incorporer en des points appropriés des expressions décrivant des conditions d'erreur. Ces expressions sont alors interprétées et il est possible alors de tester leur consistance avec un prédicat de chemin. Par exemple, on introduit l'expression `z=0` avant une instruction de la forme `x := y / z` et on cherche à démontrer une inconsistance avec le prédicat de

chemin calculé en ce point ; ceci permet d'éviter un test inutile à l'exécution.

Un autre usage de l'évaluation symbolique, plus proche de la preuve, est la prise en compte d'assertions prédéfinies par le testeur ou le programmeur [How 90]. Ces assertions définissent des conditions qui doivent être vraies en ce point pour tous les éléments du domaine des chemins correspondants. Lorsqu'une assertion est rencontrée pendant une évaluation symbolique, on considère sa négation et on la traite exactement comme une condition d'erreur (cf. l'exemple `z=0` du paragraphe précédent). Une inconsistance du prédicat de chemin résultant signifie donc que l'assertion est valide pour le chemin, tandis que sa consistance signifie que l'assertion de départ n'est pas valide.

Rappelons enfin qu'une application possible de l'évaluation symbolique est la détermination de sous-domaines des valeurs d'entrée, en vue de sélectionner des jeux de tests dynamiques (cf. rapport DCR/Q 120488-91).

Parmi les méthodes d'évaluation symboliques, la théorie de l'interprétation abstraite développée par Patrick et Radhia Cousot [CC 77] a un statut à part. Il s'agit d'une méthode très générale qui consiste à interpréter un programme en donnant aux variables des valeurs qualitatives.

L'exemple le plus simple est le calcul sur le signe des expressions arithmétiques : on considère que les variables entières ou réelles peuvent prendre trois valeurs qui correspondent au cas négatif, nul, ou positif. On se donne des règles de calcul comme : `nul + positif = positif`, `positif + positif = positif`, `positif + négatif = ?`, `nul * ? = nul`, etc. L'interprétation du programme en utilisant ces règles permet alors de conclure sur le fait qu'un résultat est toujours positif, ou qu'on ne peut rien garantir (c'est ce qu'indique le "`?`").

On peut remplacer l'analyse de signe par l'analyse d'intervalles (un exemple de règle est alors : `[n, m] + [p, q] = [n+p, m+q]`, ...) et on obtient alors un outil de vérification des débordements de tableaux ou de variables [CC 76]. Cette méthode permet aussi de déterminer dans certains cas des invariants de boucle minimaux. Cette approche n'a pas été encore, à notre connaissance, intégrée dans des outils du commerce. C'est dommage.

## Références bibliographiques


- [BEL 75] R.S. Boyer, B. Elpas, K.N. Levit
  *"SELECT - a formal system for testing and debugging programs by symbolic execution"*
  Proc. Int. Conf. Reliable Software, pp. 234-244, 1975

- [CC 76] P. Cousot, R. Cousot
*"Static determination of dynamic properties of programs"*
2nd International Symposium on Programming, Dunod, 1976, pp.106-130

- [CC 77] P. Cousot, R. Cousot
*"Abstract interpretation: a unified lattice model for static analysis of programs ..."*
ACM POPL Conference, 1977, pp.238-252

- [Cla 76] L.A. Clarke
*"A system to generate test data and symbolically execute programs"*
IEEE Trans. Soft. Eng., Vol.2, No.3, pp. 215-222, 1976

- [CR 84] L.A. Clarke, D.J. Richardson
*"Symbolic evaluation - an aid to testing and verification"*
Software Validation, H.L. Hausen Ed., Elsevier Sci. Pub., B.V. (North-Holland), pp. 141-167, 1984

- [How 90] W.E. Howden
*"Comments analysis and programming errors"*
IEEE Trans. on Soft. Eng., Vol.16, pp. 72-81, janvier 1990

- [Kin 76] J.C. King
*"Symbolic execution and program testing"*
Commun. ACM, Vol.19, No.7, pp. 385-394, 1976

- [RHC 76] C.V. Ramamoothy, S.F. Ho, W.J. Chen
*"On the automated generation of program test data"*
IEEE Trans. Soft. Eng., Vol.2, No.4, pp.293-300, 1976

- [VGA 80] U. Voges, L. Gmeiner, A. Amschler Von Mayrhauser
*"SADAT - An automated testing tool"*
IEEE Trans. on Soft. Eng., SE-6, 3, pp. 236-246, mai 1980

# 5 Analyse Statique de Programmes Parallèles

Le test dynamique de programmes parallèles, ou plus généralement de systèmes réactifs qui réagissent à des événements externes, est très difficile à cause du non déterminisme introduit par les arrivées non prévisibles de communications ou d'événements. Cela explique les nombreux travaux qui sont menés sur l'analyse statique de tels logiciels. Ces travaux sont encore pour beaucoup du domaine de la recherche, mais certains sont en cours de transfert.

On peut distinguer deux grandes classes de méthodes : celles qui sont des extensions des techniques applicables aux programmes séquentiels, en particulier l'analyse du flot de données, et celles qui sont fondées sur des modéles spécifiques au parallélisme, en particulier les réseaux de Petri ou les systèmes de transitions étiquetés.

## 5.1 Les extensions de l'analyse du flot de données

Les technique d'analyse du flot de données se transposent assez bien aux logiciels comportant du parallélisme.

Rappelons qu'un ensemble de processus parallèles peut coopérer selon deux grands principes, non exclusifs : les processus se partagent une zone de mémoire et intéragissent via des modifications et des accès à cette mémoire ; ou les processus communiquent par messages et comportent des instructions d'envoi de messages et de réception de messages.

Dans le deuxième cas, les instructions de communications peuvent être bloquantes (les processus qui veulent communiquer s'attendent les uns les autres et ne continuent à travailler que quand la communication est terminée) et on parle alors de communication synchrone ; c'est ce choix qui est fait dans la plupart des langages de programmation (Ada, CSP, OCCAM) ; ou alors les communications sont asynchrones, c'est-à-dire qu'un processus n'attend pas que le message qu'il a envoyé soit reçu pour continuer sa propre activité ; cette approche correspond mieux à ce qui se passe dans des systèmes distribués. C'est celle qui est suivie dans les langages à objets comportant du parallélisme.

Dans tous ces cas, les techniques d'analyse interprocédurale de flot de données s'appliquent, avec cependant une très grande complexité des méthodes dans le cas de la communication asynchrone (qui introduit beaucoup de non-

déterminisme, donc beaucoup de cas à prendre en compte). On retrouve évidemment les problèmes classiques vus pour l'aliasing, les pointeurs et les tableaux. On sait donc par exemple vérifier statiquement dans un programme parallèle qu'une variable ne risque pas d'être utilisée avant d'être définie, et ceci depuis longtemps (voir par exemple [TO 80, Tay 83] ). Ces méthodes ont été appliquées à CSP [Apt 84] , et ADA [TS 85] et à d'autres langages par exemple OCCAM.

Assez rapidement, on s'est aperçu que l'on pouvait faire mieux et vérifier des contraintes de séquencement et d'ordonnancement [ABCW 91]. Ces contraintes sont du type "path-expressions" régulières, par exemple : `(réserve libère)*` exprime que les actions `réserve` et `libère` doivent apparaître dans l'histoire des événements du système dans l'ordre : `réserve ... libère ... .... réserve ... libère`, où les points de suspension correspondent à d'autres actions que `réserve` ou `libère`, ou à rien.

L'analyse de flot de données classique est un cas particulier de l'analyse de contraintes de séquencement ou d'ordonnancement : on veut simplement que toute utilisation d'une variable soit précédée d'une définition.

Des propriétés telles que l'absence de blocage entre plusieurs processus qui partagent des ressources peuvent être vérifiées avec ces méthodes.

L'extension de l'interprétation abstraite aux programmes parallèles a été résolue très tôt également [CC 80], mais sa mise en application reste encore du domaine de la recherche [Mer 91].

## 5.2 Modèles Spécifiques au parallélisme

### 5.2.1 Les réseaux de Petri

Les réseaux de Petri sont un formalisme très connu [Bra 83, CV 92] qui permet de décrire des actions (appelées "transitions") qui doivent se synchroniser en certaines circonstances, exprimées par la présence de "marques" dans des "places". Classiquement les marques sont simples, mais il existe de nombreuses variantes des réseaux de Petri : colorés, à prédicats avec des marques valuées, à files, temporisés.

Ces variantes permettent d'éviter d'avoir trop de places et de transitions, ou même d'exprimer des réseaux avec une infinité de places et de transitions.

Il existe de nombreux outils d'analyse de réseaux de Petri. Sur les réseaux "de base" on sait construire un graphe appelé "graphe des marquages accessibles". Sur ce graphe, on sait vérifier des propriétés comme le fait que deux transitions

sont toujours en exclusion mutuelle ou que le système décrit ne peut pas se mettre en état d'interblocage (deadlock). Sur les réseaux plus évolués, l'analyse est plus difficile, voire impossible. On retrouve un dilemme classique entre la puissance d'expression et les possibilités d'analyse.

Les réseaux de Petri se distinguent des modèles et méthodes présentés jusqu'ici par le fait qu'ils sont construits à la main, avant ou pendant la programmation. Le lien avec le programme n'est donc pas garanti.

### 5.2.2 Les systèmes de transition étiquetés

Les systèmes de transitions sont un modèle des systèmes de processus communiquants de plus en plus utilisés, qui permet des analyses très puissantes des propriétés temporelles de ces systèmes [Arn 92].

Un système de transition est très similaire à un automate : c'est la donnée d'un ensemble d'états et de transitions entre ces états. Les transitions sont étiquetées par des actions ou par des événements. L'opération de base sur les systèmes de transitions est le "produit synchronisé" introduit par Arnold et Nivat, qui permet de composer plusieurs systèmes de transitions en un seul en prenant en compte le fait que certaines actions ou certains événements doivent se produire simultanément.

On parle parfois de "graphe des états" à propos d'un système de transition, mais les notions sous-jacentes sont les mêmes.

Quand un système de transition ne comporte qu'un nombre fini d'états, on peut vérifier des propriétés temporelles comme l'absence de blocage ou de famine (un processus n'attend pas indéfiniment), l'équité, la vivacité, ou plus généralement l'impossibilité ou l'inévitabilité de certains évenements [ABBR 92]. Il existe des outils puissants comme MEC, XESAR, AUTO qui permettent de faire ce type de vérification sur des systèmes comportant un grand nombre d'états (plusieurs centaines de milliers pour MEC, un million pou XESAR).

XESAR permet d'analyser des descriptions de systèmes parallèles en ESTELLE ou LOTOS, d'en extraire un graphe d'états fini et de vérifier des propriétés exprimées dans une logique temporelle appelée LTAC prenant en compte le non déterminisme. Le fait de se ramener à un graphe fini oblige souvent à ignorer certains aspects du système (par exemple les valeurs des données). Ce type d'outil ne détecte donc pas toutes les fautes : il s'agit donc bien de test statique. Mais il se confirme que ce sont des outils extrèmement puissants ; par exemple, ils ont déjà permis de détecter des erreurs dans des

protocoles de communication ... normalisés par l'ISO.

Ces outils sont en cours de transfert vers l'industrie (par exemple Verilog et l'IMAG viennent de créer une unité, VERIMAG, pour la commercialisation d'outils dans la lignée de XESAR.

Il faut noter que ces méthodes s'appliquent aux langages synchrones du type ESTEREL, LUSTRE et SIGNAL qui permettent de décrire des applications temps-réel. Par exemple, LESAR est une variante de XESAR qui prend en entrée des programmes en LUSTRE.

## Références bibliographiques


- [ABBR 92] A. Arnold, J. Beauquier, B. Bérard, B. Rozoy
  *"Programmes parallèles, modèles et validation"*
  Armand Colin, 1992

- [ABCW 91] G.S. Avrunin, U.A. Buy, J.C. Corbett, L.K. Dillon, J.C. Wileden
  *"Automated analysis of concurrent systems with the constrained expressions toolset"*
  IEEE Transactions on Software Engineering, SE-17, novembre 1991, pp.1204-1222

- [Apt 84] K. Apt
  *"A static analysis of CSP programs"*
  Workshop on Logics of Programs, Pittsburgh, juin 1984. pp. 1-17

- [Arn 92] A. Arnold
  *"Systèmes de transitions finis et sémantique des processus communicants"*
  Masson, 1992

- [Bra 83] G.W. Brams
  *"Réseaux de Petri : théorie et pratique"*
  tomes 1 et 2, Masson, 1983

- [CC 80] P. Cousot, R. Cousot
  *"Semantic analysis of communicating sequential processes"*
  7th ICALP, LNCS no 85, juillet 1980, pp.119-133

- [FGMRRS 93] J.-C. Fernandez, H. Gavarel, L. Mounier, A. Rasse, C. Rodriguez, J. Sifakis

*"A Toolbox for the Verification of LOTOS Programs"*
Actes ICSE'14, Melbourne, Autralie, mars 1992

- [Mer 91] N. Mercouroff
  *"An algorithm for analysing communicating processes"*
  Mathematical Foundations of Programming Semantics, à paraître en LNCS

- [Tay 83] R.N. Taylor
  *"A general purpose algorithm for analysing concurrent programs"*
  C.A.C.M. 26, mai 1983, pp. 362-376

- [TO 80] R.N. Taylor, L.J. Osterweil
  *"Anomaly detection in concurrent software by static data flow analysis"*
  IEEE Transactions on Software Engineering, SE-6, mai 1980, pp.265-277

- [TS 85] R.N. Taylor, T.A. Standish
  *"Steps to an advanced Ada programming environment"*
  IEEE Transactions on Software Engineering, SE-11, mars 1985, pp.302-310

- [VC 92] G. Vidal-Naquet, A. Choquet-Geniet
  *"Réseaux de Petri et systèmes parallèles"*
  Armand Colin, 1992

# Conclusion

Les techniques de test statique sont en général très peu coûteuses et d'une grande efficacité. Elles présentent l'avantage de faire appel à des outils relativement simples. L'inspection est l'exemple le plus frappant : elle présente un grand potentiel de découverte précoce d'erreurs pour un investissement minime. Il ne faut néanmoins pas négliger l'influence des aspects psychologiques sur l'efficacité de cette méthode.

L'analyse d'anomalie est une technique maintenant bien établie et bénéficie d'outils standards pour la plupart des langages. Tout projet de développement devrait pouvoir accéder à de tels outils.

L'évaluation symbolique est la seule technique faisant appel à des outils relativement sophistiqués. Elle n'est pas, à l'heure actuelle, encore pleinement assistée par des outils standards. Cependant des recherches relativement récentes ont pleinement abouti. L'avancement des prototypes de recherche sur le sujet ne manquera pas, dans un avenir proche, de donner lieu à des produits finis extrêmement puissants. Leur qualité est essentiellement liée à la puissance des modules de simplification d'expressions arithmétiques ou booléennes.

Le parallélisme est probablement le domaine où le test statique répond à un besoin essentiel et urgent : le test dynamique est en effet très difficile à mettre en œuvre et à interpréter. Les résultats théoriques bien établis (plus de dix ans) sur l'analyse du flot de données et des contraintes de séquencement, et les prototypes maintenant bien avancés montrent que des outils vont bientôt être disponibles.

Enfin la vérification de propriétés temporelles à partir de systèmes de transitions est un domaine où les compétences françaises sont nombreuses et des transferts en cours. Les équipes de développement, confrontés à du parallélisme, du temps-réel ou à des protocoles de communications, qui sauront se familiariser rapidement à ces nouvelles techniques de test statique, auront un avantage significatif.