Le test de logiciel : pourquoi et comment

Marie-Claude Gaudel [1]

Remerciements et avertissement. *La première version de cet article est parue dans la revue Rayonnement du CNRS en mars 2012. Je remercie Claudine Hermann qui m'avait aidée à rendre le sujet accessible à un lectorat non spécialiste en informatique, et Michel Petit et Fabrice Bonardi qui m'ont autorisée à le reprendre pour 1024. La présente version a été actualisée, et certains passages (un peu) plus techniques ont été ajoutés. Elle doit beaucoup aux remarques et suggestions de Christine Froidevaux.*

Introduction

Le logiciel est un élément inévitable de notre environnement et nous y sommes confrontés quotidiennement pour l'utilisation de tout système informatisé. Un tel système associe du matériel et du logiciel. Le logiciel, composé d'un ou plusieurs programmes et de données, est stocké dans les mémoires du système. Ce sont ces programmes qui contrôlent et permettent le fonctionnement et l'utilisation du système.

Les systèmes à base de logiciels sont devenus de plus en plus complexes, impliquant par exemple des composants développés par différents fournisseurs, utilisant des approches de programmation différentes, et parfois des plateformes d'exécution différentes. Or toute personne qui a été amenée à développer un logiciel sait à quel point il est facile d'y introduire des fautes. Et toute personne qui utilise un système informatique un peu complexe a constaté qu'il y a des dysfonctionnements, pas autant qu'on le dit car le logiciel sert souvent d'alibi, mais il y en a.

1. Marie-Claude Gaudel est professeure émérite à l'université de Paris-Sud 11 (Orsay) et membre du Laboratoire de Recherche en Informatique (LRI). Page personnelle : https://www.lri.fr/~mcg/.

Comme tout objet manufacturé complexe, un système informatique doit être testé avant d'être mis en service. Matériel et logiciel sont testés selon des méthodes différentes car les types de fautes sont différents. Dans cet article on situe le test parmi les méthodes de vérification du logiciel, on en présente les principales approches et les problèmes qu'il pose.

Spécificités du test de logiciel

Par rapport aux objets matériels, le logiciel présente la particularité de ne pas être sujet aux fautes de fabrication puisque cette fabrication se limite (en général) à une simple duplication. Par contre il souffre très souvent de fautes de conception : le ou les logiciels développés et installés ne font pas ce qui était attendu, ne correspondent pas aux besoins. Plus ces besoins sont complexes et de nature variée, plus le développement et la conception sont propices aux fautes. Ces besoins sont exprimés sous forme de spécifications, appelées aussi modèles, selon les communautés (notons que ce terme est très surchargé, en Informatique et dans les autres disciplines scientifiques).

Voici un exemple – très élémentaire – de spécification :

« Le système lit des nombres et fait leur somme ; quand il lit un zéro, il imprime la somme et s'arrête. Il ne doit jamais provoquer de débordement arithmétique, mais s'arrêter avant. »

La première phrase correspond à ce qu'on appelle une exigence fonctionnelle (elle énonce ce que doit faire le système). La deuxième phrase est une exigence de robustesse. On peut aussi avoir des exigences de performance, ou de facilité d'emploi, d'approximation du résultat, ou d'autres.

Le test fait partie des méthodes de vérification qu'un système informatique satisfait ses spécifications.

C'est une méthode dite « dynamique », car elle est basée sur des exécutions du système. Elle implique de :

– choisir judicieusement un sous-ensemble fini des données possibles du système, qui vont servir aux tests. Ce choix est basé sur des « critères de sélection de tests » ;

– assurer l'exécution des tests retenus. Cette activité nécessite souvent l'utilisation ou le développement d'un environnement de test et l'instrumentation du programme testé ;

– dépouiller les résultats obtenus. Cette activité de décision du succès ou de l'échec des tests est basée sur les spécifications ; elle pose le difficile problème dit « de l'oracle » ;

– d'évaluer la qualité de l'ensemble des tests effectués, qui est un des critères pour décider de l'arrêt du test.

Ces activités soulèvent de nombreux problèmes scientifiques, techniques, et méthodologiques. Un des points difficiles est de contrôler et d'observer l'exécution de certains logiciels, et même de la plupart d'entre eux. Mettre un logiciel en situation d'exécuter un certain test, et observer ensuite ce qui s'est passé, ceci sans biaiser l'expérience, est loin d'être évident en général. C'est encore plus vrai quand le logiciel est destiné à être intégré à un système physique critique (avion ou satellite, centrale, chaîne de production, etc.).

Un autre point dur est le problème de l'oracle mentionné plus haut. Il se pose quand on ne connaît pas le résultat attendu dans tous les cas (et c'est sans doute pour cela qu'on veut développer ce logiciel). Un exemple, parmi beaucoup d'autres, est le calcul d'un résultat numérique à ε près quand on ne connaît pas le résultat exact. Comment tester que la fourchette d'approximation est bien respectée ?

Enfin, ou plutôt avant tout, se pose la question de la sélection d'un ensemble fini de tests en fonction de ce qu'on sait du système sous test, de ses spécifications, et des fautes recherchées.

Méthodes dynamiques ou méthodes statiques ?

Du fait qu'un logiciel en cours de développement n'est pas encore exécutable et difficilement observable, les documents qui accompagnent ce développement ont une grande importance. Ce sont eux qui servent de base aux méthodes de validation et de vérification. Ces documents peuvent être écrits en langage naturel, comme ci-dessus, mais dans ce cas ils sont difficiles à exploiter. Les méthodes de validation et de vérification de logiciel exploitent essentiellement trois types de documents : des spécifications logiques, des diagrammes qu'on appelle aussi modèles, et le texte source des programmes. Dans des cas très particuliers (détection de virus) on travaille aussi à partir du code binaire. Ces documents peuvent être analysés : on parle alors de méthodes de vérification statique, par opposition au test qui, on l'a vu, est une méthode de vérification dynamique basée sur des exécutions. Les spécifications logiques peuvent être analysées via des outils de preuves ou de résolution de contraintes, les diagrammes ou modèles via des outils de model-checking, et les programmes par des analyseurs spécialisés ou par des outils de preuve interactifs.

Pendant des années il y a eu un débat, pas toujours très serein, pour décider si la vérification du logiciel devait être statique ou dynamique [10, 7, 11]. Plus raisonnablement, on travaille maintenant à combiner ces approches [21, 3, 13]. Les détracteurs de la vérification dynamique dénigrent sa non exhaustivité : on ne peut généralement pas tester toutes les données. Les détracteurs de la vérification statique mettent en avant ses limitations dues à des résultats d'indécidabilité et à la difficulté de les mettre en œuvre pour des systèmes où le logiciel est composé, hétérogène, distribué.

Or, ces deux types d'activités portent sur des entités de natures très différentes.

Carte et territoire

Un programme est un texte très détaillé, écrit dans un langage bien défini. Il est parfois annoté par des assertions : pré- et postconditions, invariants, qui sont des formules, écrites dans un autre langage bien défini, qui donnent des indications sur les propriétés attendues du programme. Il est possible de raisonner sur des programmes annotés en les considérant comme des formules [12].

Un système, informatique ou non, est une entité dynamique insérée dans le monde physique. Il n'est observable que via une interface ou des procédures limitées. Il n'est pas toujours facilement contrôlable. Par essence il est très différent d'un texte, d'une formule, ou d'un diagramme. Le texte d'un programme, ou sa spécification, ou un modèle, ne sont pas le système, mais une description du système. Il ne faut pas oublier le (pas si) vieil adage :

La carte n'est pas le territoire [18].

Un système où intervient du logiciel est obtenu par des opérations complexes (génération de code, compilation, édition de liens, déploiement sur plusieurs supports d'exécution) à partir du texte du ou des programmes et des données nécessaires à leurs exécutions. Ces opérations complexes peuvent introduire des fautes [15, 20]. Par ailleurs, on ne peut l'observer qu'en soumettant des données et en constatant des résultats. Même si son état interne est accessible, il n'est souvent que partiellement observable ou interprétable (ah, les obscurs « memory dumps » !) ce qui pose des problèmes d'analyse et d'oracle.

Indécidabilités et programmes

Nous connaissons tous les résultats fondamentaux d'indécidabilité en informatique (sur la terminaison d'un programme, sur la satisfiabilité des prédicats dans certaines logiques...). Dans le cas de la vérification de logiciel, il y a un théorème déprimant, le théorème de Rice [22], qui établit que toute propriété non triviale d'un langage reconnu par une machine de Turing est indécidable. Cet énoncé très informel mériterait d'être précisé, mais dans le cadre de cet article, il suffit à donner l'intuition d'un certain nombre de corollaires négatifs : pour les langages de programmation dits « Turing-complets », c'est-à-dire ceux qui ont un pouvoir d'expression intéressant, la satisfaction d'une spécification par un programme est indécidable.

Dire qu'un problème est indécidable ne veut pas dire qu'il n'y a pas d'espoir de trouver des solutions, mais seulement qu'il n'existe pas dans le cas général de méthode applicable d'une façon systématique pour le résoudre. De nombreux résultats existent sur comment se ramener à des sous-problèmes décidables en fixant des paramètres ou en acceptant des approximations. On peut aussi accepter des interventions humaines ou des heuristiques dans la recherche de solutions.

De ce fait, un outil automatique d'analyse statique qui considère des propriétés intéressantes sur des programmes réalistes repose soit sur des cas particuliers, soit sur

des approximations : dans ce dernier cas cela peut mener à des fausses alarmes (sur-approximation), ou à des réponses inconclusives (sous-approximation). Les chercheurs du domaine privilégient la sur-approximation, tout en étudiant comment limiter la prolifération des alarmes qu'il faut ensuite valider ou réfuter par du test ou par des méthodes statiques adaptées. On dispose maintenant d'analyseurs statiques automatiques performants pour d'importantes classes de fautes [6].

Par ailleurs, des environnements de preuve de programmes basés sur des systèmes interactifs de preuve comme Coq, Isabelle/HOL, ou PVS, permettent, moyennant une certaine expertise de leurs utilisateurs, de prouver des propriétés de programmes complexes [12].

En conclusion, on a des méthodes statiques qui permettent d'analyser des programmes et des spécifications, qui sont soit spécialisées, soit pas complétement automatiques. On a des méthodes dynamiques qui permettent de tester les systèmes informatiques et de détecter efficacement des dysfonctionnements pour certaines données, mais qui sont incomplètes, sauf si on les couple avec des hypothèses, dites « hypothèses d'uniformité », que l'on peut vérifier par des méthodes statiques ou dynamiques comme suggéré dans [5].

Des avancées essentielles pour les deux domaines ont été obtenues ces dernières années par combinaison de ces deux types de méthodes : on peut citer entre autres le test dit « concolique » qui combine exécutions de tests et analyses des évaluations symboliques correspondantes [24], et le « test passif » ou la « vérification dynamique » où on analyse les traces d'exécution de tests en fonction du programme ou d'une spécification ou d'un modèle.

Comment teste-t-on les systèmes informatiques ?

De très nombreuses solutions ont été proposées pour la sélection des tests. On peut les classer selon les informations qui servent de point de départ à la sélection et selon la manière dont on se ramène à un ensemble fini de cas. Cette section présente une classification et l'illustre en détaillant quelques solutions, sans tenter une énumération exhaustive. On va distinguer les méthodes basées sur le domaine d'entrée du système, celles basées sur le programme, et celles basées sur les spécifications ou modèles. Pour un panorama complet et récent des recherches dans ce domaine, on peut se reporter à [2]. Notons que la sélection peut se faire avant l'exécution des tests (on parle alors de sélection« off-line ») ou pendant cette exécution, en s'adaptant aux résultats des tests précédents (sélection adaptative ou « on-line »).

L'oracle lui, est toujours basé sur une spécification ou un modèle. Pour des raisons de place, on n'en parlera pas ici. On se reportera à [16] pour une récente et très complète synthèse des avancées sur ce sujet.

Méthodes basées sur le domaine d'entrée

Le domaine d'entrée est en général infini, ou en tout cas beaucoup trop grand pour envisager un test exhaustif. Il y a deux grandes approches pour se ramener à un jeu de tests fini : le tirage aléatoire et la décomposition en sous-domaines. Ces approches sont généralement combinées avec d'autres : tirage ou décomposition peuvent être guidés par la structure du programme ou la spécification (cf. sous-sections suivantes).

Le tirage aléatoire peut se faire selon différentes distributions. Une distribution uniforme est souvent simple à mettre en œuvre mais donne des résultats décevants, du fait que la plupart des programmes sont organisés pour traiter un très gros sous-ensemble de données qui correspondent au cas normal, et beaucoup de très petits sous-ensembles qui correspondent à des cas particuliers qui ont peu de chance d'être atteints par cette méthode. Une distribution basée sur un profil d'usage présente l'avantage de détecter avec une très bonne probabilité les dysfonctionnements les plus susceptibles de se produire quand le système sera en opération, et permet une estimation de sa fiabilité. Une telle distribution associe à chaque donnée sa probabilité d'occurrence quand le système sous test sera utilisé. La difficulté est d'établir un profil d'usage fidèle. C'est possible dans certains domaines, comme les télécommunications, où on peut archiver de nombreuses exécutions.

La décomposition en sous-domaines d'uniformité, appelée aussi test par partition, consiste à déterminer un nombre fini de sous-ensembles du domaine d'entrée pour lesquels il paraît suffisant d'avoir un seul test : on fait une hypothèse d'uniformité sur ces sous-domaines, c'est-à-dire que l'on considère que le succès ou l'échec du test y est identique pour toute valeur. On sélectionne donc une valeur par sous-domaine.

Le choix de la décomposition peut être guidé par le type des données : par exemple pour des entiers on sélectionne une valeur négative, zéro, et une valeur positive. Mais il est souvent guidé par le domaine d'application, la spécification ou le programme (voir ci-dessous). Cette sélection est souvent complétée par des valeurs spéciales (c'est le cas du zéro dans l'exemple ci-dessus) ou aux limites, ou au voisinage des limites.

Méthodes basées sur le programme

Ces méthodes sont essentiellement de deux types : soit elles sont basées sur des critères de couverture des éléments du programme, et on parle de test structurel, soit elles sont basées sur des fautes introduites dans le programme, et on parle de test par mutation.

Les critères de couverture les plus classiques pour le test structurel sont : la couverture des instructions qui requiert que lors de l'exécution du jeu de tests on passe au moins une fois par chaque instruction ; la couverture des enchaînements, qui requiert de passer au moins une fois par chaque enchaînement du programme entre deux instructions ; la couverture des chemins du programme, qui est souvent trop coûteuse en nombre de tests, le nombre de chemins ayant tendance à exploser, et pas toujours

possible quand le programme comporte des boucles, sauf si on borne le nombre des itérations. Il existe plus d'une centaine de tels critères selon les caractéristiques des programmes considérés et des fautes recherchées...

En fait, ces méthodes induisent la définition de sous-domaines d'uniformité sur les données : par exemple le sous-domaine des données provoquant l'exécution d'une instruction particulière. Mais à la différence de ce qu'on a vu plus haut, cette définition est déduite de la structure du programme. L'hypothèse d'uniformité correspondante peut être validée par analyse statique de celui-ci, en montrant par exemple que dans tous les cas on exécute la même suite d'instructions avec le même effet, ou par preuve [5].

La mise en œuvre du test structurel nécessite des données provoquant l'exécution des chemins sélectionnés. Cela se fait par exécution symbolique puis résolution de contrainte : à partir des prédicats de ses instructions conditionnelles, de ses boucles et autres points de choix du chemin, et de ses instructions, on construit une contrainte qui caractérise les valeurs qui provoquent son exécution ; on doit ensuite résoudre cette contrainte [14]. Les difficultés viennent à la fois des programmes, qui comportent des chemins infaisables à cause de conditions contradictoires, et des limitations des solveurs de contraintes : là aussi on est confronté à de l'indécidabilité.

Certaines approches de sélection aléatoire prennent en compte la structure du programme de manière à garantir un critère de couverture avec une bonne probabilité. C'est le cas des méthodes basées sur le tirage de chemins avec des probabilités adaptées à certains critères de couverture [8] ou de certaines méthodes évolutionnaires [1].

Le test par mutation a été défini à l'origine comme une méthode de sélection de tests. Mais il s'est avéré très utile pour évaluer l'efficacité des jeux de test. Il s'agit d'une méthode d'injection de fautes : on crée des mutants du programme à tester en modifiant systématiquement certaines instructions ou certaines expressions. Par exemple on remplace une condition par *true*, puis *false*, ou $\leqslant$ par $<$, etc. On obtient ainsi un ensemble de mutants, chacun résultant de l'injection d'une faute. Le critère de sélection est que chaque mutant doit être détecté par au moins un test. Le test par mutation pose plusieurs problèmes. Le plus sérieux est celui des mutants équivalents : une mutation peut être sans conséquence sur le comportement du programme. Dans ce cas, on ne peut pas atteindre un taux de détection de 100%.

Le modèle de fautes sur lequel est basé le test par mutation est très incomplet : seules des fautes mineures de programmation sont considérées. Les fautes de conception ne sont pas envisagées. Sans que l'on puisse vraiment l'expliquer, on a constaté expérimentalement qu'elles sont cependant détectées de manière satisfaisante par ces méthodes, dont les fondements théoriques restent à établir. Notons que si on a des modèles de fautes bien établis pour le matériel qui ont mené à des méthodes de test très efficaces, ce problème est ouvert pour le logiciel : la palette des fautes en ce domaine reste à établir et à classifier.

Les méthodes de test structurel sont essentielles, mais présentent un défaut de base, qui est qu'elles ne testent que ce qui est dans le programme. Elles ne détectent pas les cas oubliés ou mal formulés. D'où la nécessité d'autres méthodes basées sur les spécifications ou les modèles.

Méthodes dites « boîte noire »

Ces méthodes ne prennent pas en compte la structure du logiciel. Appelées aussi « basées sur les modèles » ou « basées sur les spécifications », elles utilisent soit des diagrammes, soit des spécifications logiques. On en trouve une synthèse dans [23].

Dans le cas de diagrammes comme des automates, des diagrammes UML, des machines à états ou des systèmes de transitions (cf. Figure 1), on retrouve des méthodes similaires à celles qui ont été présentées à propos du test basé sur la structure des programmes, avec cependant tout un corpus de résultats spécialisés sur la couverture des transitions. L'intérêt pour ce critère de sélection vient du fait que ces diagrammes sont en général conçus sans mémoire globale, et donc que l'effet des transitions est indépendant du chemin suivi pour les atteindre. Il suffit donc de couvrir chaque transition une fois. Dans le cas du diagramme de la Figure 1 qui comporte 2552 transitions, cela implique plusieurs centaines de tests (car un test couvre plusieurs transitions) qui peuvent être générés automatiquement à partir du modèle en utilisant par exemple un outil comme TGV [17]. Le problème de ces approches est l'explosion du nombre des états ; le modèle de la Figure 1 est très petit par rapport à ceux que l'on rencontre couramment. De nombreuses recherches visent à réduire le nombre des états, par exemple par abstractions, ce qui coûte en précision du modèle et mène à des tests infaisables. On peut aussi renoncer au traitement exhaustif des modèles et développer des méthodes randomisées avec garantie probabiliste de couverture [8].

Dans le cas de formules logiques, les outils de choix pour sélectionner des tests sont des prouveurs de théorèmes ou des solveurs de contraintes. Les solveurs de contraintes sont d'ailleurs des outils essentiels pour la génération de tests, que l'approche soit structurelle ou basée sur des spécifications. On a vu que dans le cas du test structurel ils permettent de fournir des données correspondant à certains chemins. C'est aussi le cas pour les traces et les comportements de diagrammes. De plus, dans le cas de spécifications logiques, pour des logiques adéquates, ils permettent de décomposer les formules correspondant aux cas à tester sous une forme normale disjonctive [9] qui énumère une liste de sous-cas disjoints qui fournissent des tests (sous l'hypothèse qu'ils soient satisfiables et qu'ils correspondent à des sous-domaines d'uniformité). Cette approche donne généralement lieu à une explosion du nombre des sous-cas que l'on maîtrise en inventant des stratégies de décomposition et de résolution intelligentes. Les avancées spectaculaires dans le domaine des solveurs SMT et des solveurs SAT ont permis, et vont permettre sans aucun doute, de grands progrès pour les outils de test.

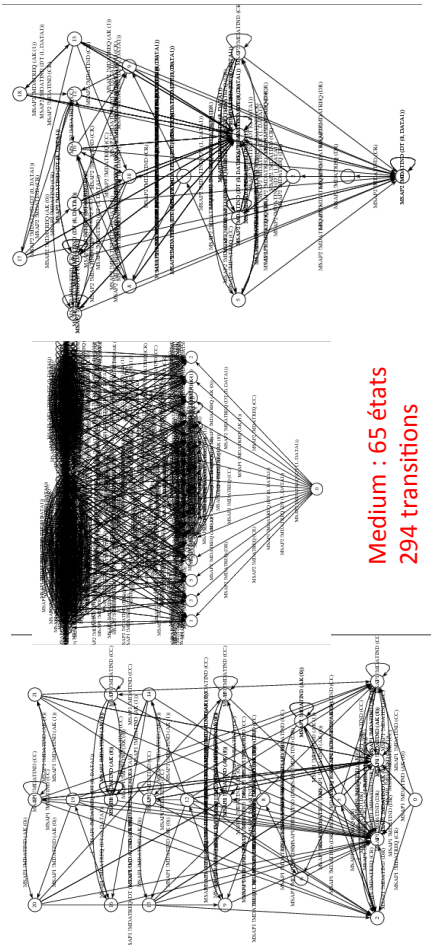

FIGURE 1. Exemple de modèle : les trois composants du protocole INRES

Le grand avantage de ces méthodes boîte noire est leur indépendance par rapport aux détails d'implémentation et d'organisation du système sous test : elles permettent de concevoir des tests pour des systèmes hétérogènes qui utilisent des approches de

programmation différentes, des supports d'exécution variés et distribués. Ces approches sont d'ailleurs très utilisées pour les protocoles de communication et dans les industries où les logiciels sont embarqués [19]. La contrepartie est qu'il est parfois compliqué de concrétiser les tests obtenus à partir d'un modèle abstrait pour les rendre exécutables et exploitables au niveau du système.

Conclusion

De plus en plus, des systèmes où le logiciel est prépondérant jouent un role essentiel et critique dans notre environnement. De ce fait les exigences de sûreté de fonctionnement de ces systèmes sont très élevées, et il en est de même pour les logiciels qui y sont intégrés. Toute une variété de méthodes sophistiquées sont apparues pour s'en assurer, qu'il s'agisse des programmes, qu'on analyse ou qu'on prouve, de modèles du système, dont on vérifie des propriétés, et des systèmes eux-mêmes qu'on teste. Ces méthodes ont montré qu'elles peuvent s'apporter beaucoup mutuellement, qu'elles soient utilisées comme outils de l'une pour l'autre, ou de manière complémentaire [13]. Beaucoup reste à explorer dans ce domaine. On sait d'ailleurs depuis longtemps qu'il n'y a pas de solution miracle et universelle aux problèmes que pose le logiciel (« no silver bullet » [4]).

Parmi ces méthodes, la spécificité du test vient des contraintes physiques des systèmes, qui soulèvent des problèmes d'observation et de contrôle, et de sa dépendance par rapport à l'environnement. Un système est toujours une boîte opaque, même si on connaît le texte source des programmes qui y sont enfouis : le test permet une vérification globale de l'ensemble de ces boîtes, y compris des tables internes (qui sont à l'origine des fautes découvertes dans le système SACEM de la RATP [15] et le Pentium [20]), des optimisations des compilateurs, des choix d'ordonnancements et de configuration.

C'est un domaine qui demande des connaissances sur des sujets théoriques aussi divers que : sémantique, logique, probabilités, modèles de fautes, et dont les retombées pratiques sont de toute première importance.

Références

[1] Shaukat Ali, Lionel C. Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Trans. Software Eng.*, 36(6) :742–762, 2010.

[2] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8) :1978–2001, 2013. Edited by Antonia Bertolino, J. Jenny Li and Hong Zu.

[3] Andrea Asperti, Herman Geuvers, and Raja Natarajan. Social processes, program verification and all that. *Mathematical Structures in Comp. Sci.*, 19(5) :877–896, 2009.

[4] Frederick P. Brooks Jr. No silver bullet - essence and accidents of software engineering. *IEEE Computer*, 20(4) :10–19, 1987.

[5] Achim D. Brucker, Lukas Brügger, and Burkhart Wolff. Verifying test-hypotheses : An experiment in test and proof. *Electr. Notes Theor. Comput. Sci.*, 220(1) :15–27, 2008.

[6] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In *ESOP-ETAPS 2005 Proceedings*, volume 3444 of *L.N.C.S.*, pages 21–30. Springer, 2005.

[7] Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *C.A.C.M.*, 22(5) :271–280, 1979.

[8] Alain Denise, Marie-Claude Gaudel, Sandrine-Dominique Gouraud, Richard Lassaigne, Johan Oudinet, and Sylvain Peyronnet. Coverage-biased random exploration of large models and application to testing. *International Journal on Software Tools for Technology Transfer (STTT)*, 14(1) :73–93, 2012.

[9] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *FME '93 : 1st Int. Symposium of Formal Methods Europe*, volume 670 of *L.N.C.S.*, pages 268–284. Springer, 1993.

[10] Edsger W. Dijkstra. The humble programmer. *C.A.C.M.*, 15(10) :859–866, 1972.

[11] James H. Fetzer. Program verification : The very idea. *C.A.C.M.*, 31(9) :1048–1063, 1988.

[12] Jean-Christophe Filliâtre. Deductive software verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 13(5) :397–403, 2011.

[13] Marie-Claude Gaudel. Checking models, proving programs, and testing systems. In *Tests and Proofs (TAP 2011)*, volume 6706 of *L.N.C.S.*, pages 1–13. Springer, 2011.

[14] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. In *ISSTA'98*, pages 53–62, 1998.

[15] Gérard D. Guiho and Claude Hennebert. SACEM software validation (experience report). In *Int. Conf. on Soft. Eng. (ICSE) 1990*, pages 186–191. IEEE Comp. Soc., 1990.

[16] Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. A comprehensive survey of trends in oracles for software testing. Technical Report CS-13-01, Univ. of Sheffield, Dep. of Comp. Sc., 2013. http ://philmcminn.staff.shef.ac.uk/publications/pdfs/2013-techreport.pdf.

[17] Claude Jard and Thierry Jéron. TGV : theory, principles and algorithms, a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Software Tools for Technology Transfer (STTT)*, 6, October 2004.

[18] Alfred Korzybski. *Science and Sanity : A Non-Aristotelian System and its Necessity for Rigour in Mathematics and Physics*. Institute of General Semantics, 1933.

[19] Bruno Marre and Benjamin Blanc. Test selection strategies for lustre descriptions in gatel. *Electr. Notes Theor. Comput. Sci.*, 111 :93–111, 2005.

[20] Vaughan R. Pratt. Anatomy of the Pentium bug. In *TAPSOFT'95 Proceedings*, volume 915 of *L.N.C.S.*, pages 97–107. Springer, 1995.

[21] Brian Randell. Facing up to faults. *The Computer Journal*, 43 :95–106, 2000.

[22] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74 :358–366, 1953.

[23] Mark Utting and Bruno Legeard. *Practical Model-Based Testing : A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[24] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. PathCrawler : Automatic generation of path tests by combining static and dynamic analysis. In *5th Europ. Dependable Computing Conf. (EDCC-5)*, volume 3463 of *L.N.C.S.*, pages 281–292. Springer, 2005.