HAL
archives-ouvertes.fr

Sûreté de Fonctionnement du Logiciel

Jean-Claude Laprie, Karama Kanoun

▶ **To cite this version:**

Jean-Claude Laprie, Karama Kanoun. Sûreté de Fonctionnement du Logiciel. La Revue de l'Electricité et de l'Electronique, 2005, pp.37-41. hal-01978498

HAL Id: hal-01978498
https://hal.laas.fr/hal-01978498

Submitted on 11 Jan 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Sûreté de Fonctionnement du Logiciel

Jean-Claude Laprie et Karama Kanoun
LAAS-CNRS
7 avenue Colonel Roche — 31077 Toulouse

État actuel

Les exemples spectaculaires de défaillances informatiques ayant le logiciel pour origine ont émaillé les deux denières décennies : report du premier lancement de la navette spatiale (avril 1981), doses excessives de radiothérapie aux Etats-Unis et au Canada (Therac-25, juin 1985 à janvier 1987), téléphone interurbain indisponible pendant 9 heures aux Etats-Unis (15 janvier 1990), écroulement du système de communication des ambulances de Londres (novembre 1992), défaillance du vol inaugural d'Ariane 5 (4 juin 1996), perte du *Mars Climate Orbiter* (septembre 1999), rôle dans la propagation de la panne électrique dans le Nord-Est des Etats-Unis et au Canada (août 2003). Au-delà de ces exemples spectaculaires, et médiatisés, le logiciel s'avère statistiquement comme la première source de défaillance des systèmes informatiques.

La défaillance d'un logiciel en opération est consécutive à l'activation d'une faute (ou défaut, ou faille) résiduelle, c'est-à-dire ayant échappé aux diverses vérifications effectuées tout au long du processus de développement, destinées à éliminer les fautes créées au cours de ce processus. Afin de fixer les idées, et de caractériser l'état de l'art actuel, la densité de fautes créées au cours du processus de développement est de l'ordre de 10 à 200 fautes par millier de lignes de code exécutable, ou kLOC, et la densité de fautes résiduelles en opération est de l'ordre de 0,01 à 10 fautes/kLOC. La large variation dans ces valeurs des densités de fautes (créées, résiduelles) accompagne des variations tout aussi larges en termes d'effort consacré au développement : depuis une fourchette de 0,1 à 0,5 personne.année pour les valeurs supérieures des densités de fautes, relatives à des logiciels de grande taille (quelques centaines de milliers à quelques millions de lignes de code) et à des applications critiques en termes économiques (télécommunications, systèmes transactionnels) [Donnelly et al. 92], à une fourchette de 5 à 10 personne.année pour les valeurs inférieures des densités de fautes, relatives à des logiciels de taille beaucoup plus modeste (quelques milliers ou dizaines de milliers de lignes de code) et à des applications critiques au sens de la sécurité des personnes (avionique, surveillance de centrales nucléaires, signalisation ferroviaire) [Craigen et al. 93], ou à des applications telles que les interventions en opération sont limitées voire impossibles (spatial). Enfin, le pourcentage de l'effort de développement consacré aux vérifications va de 40% au moins pour les premiers types de logiciels mentionnés ci-dessus, à 75% pour les seconds.

Dans la suite de cet article, nous examinons tout d'abord les approches pour diminuer les densités de fautes créées et résiduelles. Une conséquence des données chiffrées dont il a éte fait état ci-dessus est que, sauf exception qui reste entachée d'incertitude, tout logiciel est immanquablement le siège de fautes résiduelles, et il importe donc de le munir de procédures et mécanismes pour tolérer ces fautes en opération, c'est à dire pour que l'activation d'une faute n'entraîne pas la défaillance ; nous examinons donc ensuite les approches de tolérance aux fautes

des logiciels, destinées à améliorer la sûreté de fonctionnement en dépit des fautes résiduelles. L'incertitude dans l'absence de fautes résiduelles et les inévitables imperfections de la tolérance aux fautes conduisent à évaluer la sûreté de fonctionnement des logiciels, ce qui est l'objet de la troisième partie, qui traite de l'analyse du comportement des logiciels par rapport aux fautes, tant créées que résiduelles. Nous examinons ensuite la situation par rapport à la réutilisation de composants logiciels pour produire un logiciel, et particulièrement l'utilisation de composants dits disponibles sur étagère, ou COTS (*Commercial Off-The-Shelf*). Nous concluons enfin en attirant l'attention sur la défaillance des processus de développement, donc n'aboutissant pas à produire les logiciels recherchés.

Réduction des fautes créées et des fautes résiduelles

A densité de fautes créées donnée, la réduction de la densité de fautes résiduelles passe par une amélioration des activités destinées à éliminer les fautes, donc des activités de vérification. Il est maintenant largement admis que des vérifications doivent avoir lieu tout au long du processus de développement, et accompagner chaque étape de la création du logiciel. En effet, le coût d'élimination d'une faute est d'autant plus élevé que l'on attend pour la corriger. En fait, on considère généralement qu'éliminer une faute au cours de la phase où elle a été créée est d'un ordre de grandeur moins coûteux que de l'éliminer lors de la phase suivante. Par exemple, corriger une faute de spécification lors de la spécification plutôt que de la corriger lorsque la conception, ou, a fortiori, lorsque le codage a été effectué.

Il existe deux grandes classes de méthodes de vérification, statiques et dynamiques. Les méthodes statiques les plus employées sont les inspections (revue des spécifications, relecture du code), et les méthodes dynamiques recouvrent les diverses formes de test (déterministe, statistique). Alors qu'initialement seul le test était pratiqué, les inspections ont fait la preuve de leur efficacité, tant par leur aptitude à révéler des fautes dès les premières étapes de la création d'un logiciel, que par leur efficience supérieure à celle du test, en termes de fautes révélées par rapport à l'effort de vérification [Grady 1992]. De plus, les fautes de spécification constituent statistiquement la principale classe de fautes créées, que seules les vérifications statiques sont à même de révéler au cours, ou à l'issue, de la phase de spécification. De ce qui précède, il serait cependant erroné de déduire que le test est inutile : il reste un moyen ultime, et indispensable, de vérification du logiciel.

La réduction du nombre de fautes créées passe par l'amélioration du processus de développement lui-même. Les références dans ce domaine sont sans nul doute le *Capability Maturity Model* (CMM) [Paulk et al. 1993] et le *Capability Maturity Model-Integrated* (CMMI) [Ahern et al. 2001], qui insistent sur les mesures quantitatives qu'il est nécessaire d'effectuer afin que les points faibles du processus soient identifiés et corrigés. La diminution du nombre de fautes créées consécutive à la mise en œuvre de ces modèles a été démontrée statistiquement [Diaz & Siglio 97]. Une autre source significative d'amélioration, non exclusive de la maîtrise quantitative du processus, est le recours à la mécanisation de certaines étapes du développement du logiciel, en particulier l'emploi de générateurs automatiques pour produire le code source à partir des spécifications.

Le moyen ultime pour réduire tant les fautes créées que les fautes résiduelles est le recours à des approches mathématiquement formelles : diminution du nombre de fautes créées par la rigueur et l'absence d'ambiguïté qui accompagnent des notations mathématiques, diminution des fautes résiduelles par des vérifications également mathématiques. Les vérifications formelles peuvent être des vérifications de modèle (particulièrement adaptée à la vérification des spécifications) [Schnoebelen 1999], des analyses statiques du logiciel produit (visant alors à compléter ou à remplacer des tests effectués sur le code source [Cousot & Cousot 2004]), ou des démonstrations d'équivalence entre les états successifs du logiciel au cours de son développement, vu alors comme une suite d'affinements successifs [Abrial 2003]. Les méthodes mathématiquement formelles ont été longtemps l'objet d'un fort scepticisme de la part des praticiens, mais leur utilisation est indéniablement en croissance, qu'il s'agisse de logiciels critiques, mettant en jeu la sécurité, et donc de complexité modeste [Dollé et al. 2003], ou significativement plus complexes comme dans le domaine des télécommunications.

Tolérance aux fautes résiduelles

Le caractère inéluctable des fautes résiduelles, joint à notre dépendance croissante dans les systèmes informatiques, et donc dans les logiciels, conduit au souci de tolérer les fautes logicielles.

La tolérance aux fautes logicielles a été longtemps l'apanage de systèmes mettent en jeu la sécurité, en particulier l'avionique civile [Brière & Traverse 1993] ou la signalisation ferroviaire [Kantz & Koza 1995]. Ces systèmes procèdent de la même approche, la diversité, dans laquelle deux ou plusieurs variantes du logiciel sont développées séparément [Laprie et al. 1990]. Le but des développements séparés est d'obtenir des variantes qui soient indépendantes vis-à-vis des fautes résiduelles et de leur activation. Clairement, des développements séparés entraînent une telle augmentation du coût que cette approche est limitée aux systèmes à haute criticité.

Par ailleurs, il a été constaté que la plupart des fautes résiduelles sont suffisamment subtiles pour que leur activation dépende de combinaisons également subtiles de l'état interne et des sollicitations de l'environnement. Ces fautes présentent alors toutes les caractéristiques de fautes temporaires, y compris la difficulté à reproduire leurs conditions d'activation [Gray 90]. Les approches pour tolérer de telles fautes logicielles temporaires sont d'un coût significativement plus faible que l'approche précédente [Huang & Kintala 1995]. Des points de reprise dans des architectures faiblement couplées, a priori destinées à tolérer des fautes matérielles ont démontré leur efficacité [Gray 1990].

Analyse du comportement par rapport aux fautes créées et résiduelles

L'évaluation est effectuée soit dans l'objectif d'analyser qualitativement le comportement du logiciel soit dans l'objectif de quantifier, en termes de probabilités, la fiabilité du logiciel.

Les évaluations qualitatives (via des analyses des modes de défaillance ou des arbres de fautes) peuvent être des aides à la décision, par exemple lors du développement pour guider la mise en œuvre de techniques de tolérance aux fautes ou orienter le test vers les modes de défaillances critiques, en maintenance pour définir les procédures de surveillance et analyser l'origine des défaillances.

La quantification de la fiabilité du logiciel est assurée d'une part par l'analyse de la tendance et par la modélisation probabiliste du comportement du logiciel d'autre part. Ces deux méthodes utilisent des informations recueillies sur le logiciel concerné (observation des défaillances et des corrections, en phase de validation ou en opérationnel). L'élimination progressive des fautes entraîne théoriquement une croissance de fiabilité. Cependant, les conditions d'environnement et de test peuvent mettre en cause la croissance de fiabilité, soit par une absence d'effet perceptible (fiabilité stabilisée), voire par des effets contraires, conduisant alors à une décroissance de fiabilité. L'analyse de la tendance [Lyu 1995, Chapitre 10] permet de déterminer le sens d'évolution de la fiabilité. Elle constitue une aide précieuse à la gestion de la validation d'un logiciel, par l'identification de l'impact des phases successives de test sur la fiabilité, et permet donc de mettre en évidence d'éventuels problèmes dans les activités de validation.

La modélisation de la fiabilité vise à estimer la fiabilité actuelle ou à prévoir la fiabilité future, en s'appuyant sur les données de défaillances observées, en termes de temps jusqu'à défaillance, taux ou intensité de défaillance, ou nombre cumulé de défaillances [Xie 1991]. Il existe actuellement plus d'une cinquantaine de modèles dits de croissance de fiabilité. Certains modélisent une croissance pure, d'autres une décroissance suivie d'une croissance. Aucun modèle ne se distingue par sa capacité de prévision. Le choix du modèle doit être guidé par les résultats de l'analyse de tendance, ainsi que par la qualité et la représentativité des prévisions qui doivent être vérifiées par des critères de validation statistique. Les modèles de croissance de fiabilité considèrent le logiciel comme une boîte noire et ne tiennent pas compte explicitement de son architecture. Les modèles structurels permettent d'évaluer la sûreté de fonctionnement d'un logiciel à partir de celle de ses composants (y compris dans le cas de la tolérance aux fautes), ces derniers pouvant être en fiabilité stabilisée ou croissante [Lyu 1995, Chapitre 2]. Les résultats des modèles s'avèrent significatifs surtout en fin de validation et en exploitation opérationnelle, et ne permettent pas en tout état de cause d'effectuer des prévisions pertinentes pour les logiciels critiques [Butler & Finelli 1993].

On peut avoir recours à des expériences contrôlées pour caractériser la fiabilité d'un logiciel, de façon qualitative ou quantitative, afin de compléter les informations obtenues par les analyses présentées ci-dessus. De telles expériences se basent essentiellement sur l'injection de fautes. Elles peuvent servir lors de la validation du logiciel pour identifier des faiblesses dans le logiciel qui pourraient entraîner des défaillances catastrophiques, ainsi que pour localiser les portions du logiciel qui contiennent ces faiblesses [Voas et al. 1997]. De tels résultats complètent les évaluations qualitatives effectués sur le logiciel ou sa spécification. Les fautes injectées peuvent simuler uniquement des fautes externes au logiciel, via son exécution avec des entrées erronées, l'évaluation visant alors à caractériser la robustesse du logiciel par rapport à son environnement. La pertinence des informations fournies par ces expériences contrôlées est encore accrue lorsqu'elles sont conduites dans un cadre favorisant leur reproductibilité, conduisant à la notion d'étalonnage ("benchmarking") de la sûreté de fonctionnement [Kalakech et al. 2004].

Utilisation de COTS dans le développement de logiciels

L'utilisation de COTS n'est pas un phénomène nouveau, surtout si l'on se réfère au logiciel dit de base, soit en ligne comme les systèmes d'exploitation, soit hors ligne comme les compilateurs.

Leur utilisation dans des systèmes soumis à des exigences de sûreté de fonctionnement, nécessite de s'assurer que les COTS sont cohérents avec la démarche d'ensemble :

- connaissance de leurs densités de fautes résiduelles et, le cas échéant, informations précises sur leur développement et en particulier les vérifications qu'ils ont subies ;
- connaissance de leurs mécanismes de tolérance aux fautes et des hypothèses de fautes, tant internes (pour juger de leur réaction à leurs propres fautes résiduelles), qu'externes (pour juger de leur robustesse vis-à-vis de fautes externes), sur lesquels ces mécanismes sont basés, ou mise en œuvre de tels mécanismes par empaquetage du COTS [Rodriguez et al. 2002].

De la défaillance des logiciels à la défaillance de leur processus de développement

Nous avons traité dans cet article des approches pour améliorer et évaluer la sûreté de fonctionnement de logiciels qui ont été effectivement produits. Il existe cependant toute une population de logiciels qui ne voient jamais le jour de façon opérationnelle, du fait de la défaillance de leur processus de production. Des sommes considérables sont alors investies en pure perte, estimées à plusieurs dizaines de milliards de dollars par an aux seuls Etats-Unis [Johnson 1995] et, pour certains cas, à plusieurs milliards de dollars pour un seul projet avorté, comme la tentative infructueuse de modernisation du système de contrôle de trafic aérien.nord-américain. Ces défaillances du processus de développement n'ont généralement pas une cause unique, et résultent d'une combinaison de décisions malheureuses quant aux approches techniques, à la gestion de projet, aux modèles économiques. Dans le contexte de la sûreté de fonctionnement, des exigences irréalistes, des spécifications piètrement formulées ou en perpétuelle évolution, ou l'utilisation de technologies immatures se retrouvent souvent, qui ont généralement pour conséquences un nombre excessivement élevé, et non maîtrisable, de fautes.

Références bibliographiques

Abrial 2003 J.R. Abrial, “B : passé, présent, futur”, *Technique et science informatiques*, vol. 22, no. 1, 1993, pp. 89-118.

Ahern et al. 2001 D. Ahern, A. Clouse, R. Turner, *CMMI Distilled*, Addison-Wesley, 2001.

Brière & Traverse 1993 D. Brière, P. Traverse, “Airbus A/320/A330/A340 electrical flight controls — A family of fault-tolerant systems”, *Proc. 23rd IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-23)*, Toulouse, France, June 1993, pp. 616-623.

Butler & Finelli 1993 R.W. Butler, G.B. Finelli, “The infeasibility of quantifying the reliability of life-critical real-time software”, *IEEE Trans. on Software Engineering*, vol. 19, no. 1, January 1993, pp. 3-12.

Cousot & Cousot 2004 P. Cousot, R. Cousot, “Basic concepts of abstract interpretation”, in *Building the Information Society*, R. Jacquart (ed.), Kluwer, pp. 359-366, Proc. of the 18th IFIP World Computer Congress (WCC 2004), Toulouse, August 2004.

Craigen et al. 1993 D. Craigen, S. Gehrart, T. Ralston, "An international survey of industrial applications of formal methods", report NIST GCR 93/626, National Institute of Standards and Technology, March 1993.

Diaz & Siglio 1997 M. Diaz, J. Siglio, "How software process improvement helped Motorola", *IEEE Software*, vol. 14, no. 5, Sept.-Oct. 1997, pp. 75-81.

Dollé et al. 2003 B. Dollé, D. Essamé, J. Falampin, “B dans le transport ferroviaire. l’expérience de Siemens Transportation Systems”, *Technique et science informatiques*, vol. 22, no. 1, 1993, pp. 11-32.

Donnelly et al. 1992 M. Donnelly, W. Everett, J. Musa, G. Wilson, "Best current practices: Software reliability engineering", AT&T Bell Labs, 1992.

Grady 1992 R. B. Grady, *Practical Software Metrics for Project Management and Process Improvement,* Software Engineering Metrics, Prentice Hall, Englewood Cliffs, NJ, 1992.

Gray 1990 J. Gray, "A census of Tandem system availability between 1985 and 1990", *IEEE Trans. on Reliability*, vol. 39, no. 4, Oct. 1990, pp. 409-418.

Huang & Kintala 1995 Y. Huang, C. Kintala, "Software fault tolerance in the application layer", in *Software Fault Tolerance*, M. Lyu (ed.), Wiley, 1995.

Johnson 1995 J. Johnson, “Chaos: the dollar drain of IT project failures”, *Application Development Trends*, January 1995, pp. 41-47.

Kalakech et al. 2004 A. Kalakech, K. Kanoun, Y. Crouzet,, A. Arlat, “Benchmarking the Dependability of Windows NT, 2000 and XP”, in *Proc. 2004 Int. Conf. on Dependable Systems and Networks*, Florence, Italie, 2004, pp. 681-686.

Kantz & Koza 1995 H. Kantz and C. Koza. “The ELEKTRA Railway Signalling-System: Field Experience with an Actively Replicated System with Diversity,” in *Proc. 25th Int. Symp. on Fault-Tolerant Computing (FTCS-25),* Pasadena, CA, USA, June 1995, pp. 453-458.

Laprie et al. 1990 J.C. Laprie, J. Arlat, C. Béounes, K. Kanoun, "Definition and analysis of hardware- and software-fault-tolerant architectures", *IEEE Computer,* vol. 23, no. 7, July 1990, pp. 39- 51.

Lyu 1995 M.R. Lyu (ed.), *Software Reliability Engineering*, McGraw-Hill, 1995.

Paulk et al. 1993 M.C. Paulk, B. Curtis, M.B. Chrissis, C.V. Weber, “Capability maturity model for software”, Software Engineering Institute, report CMU/SEI-93-TR-24, ESC-TR-93-177, Feb. 1993.

Rodriguez et al. 2002 M.Rodriguez, J.C.Fabre, J.Arlat, "Wrapping real-time systems from temporal logic specifications", in *Proc. 4th European Dependable Computing Conference (EDCC-4)*, Toulouse, October 2002, pp.253-270.

Schnoebelen 1999 P. Schnoebelen (coord.), *Vérification de logiciels — Techniques et outils du model-checking*, Vuibert , 1999.

Voas et al. 1997 J. Voas, F. Charron, G. McCraw, K. Miller, M. Fiedman, "Predicting How Badly “Good” Software Can Behave", *IEEE Software*, juillet/août 1997, pp. 73-83.

Xie 1991 M. Xie, Software Reliability Modeling, World Scientific Publisher, 1991.