Elimination des fautes : contribution au test du logiciel

Hélène Waeselynck

UNIVERSITE DE TOULOUSE

Manuscrit présenté par :

Hélène WAESELYNCK

en vue de l'obtention d'une :

Habilitation à Diriger les Recherches de

l'INSTITUT NATIONAL POLYTECHNIQUE DE TOULOUSE

Elimination des fautes : contribution au test du logiciel

Soutenue le 9 décembre 2011 devant le Jury composé de :

Mme	Karama Kanoun	Directeur de Recherche CNRS	Présidente et Garante
M.	Richard Castanet	Professeur Emérite à l'Institut Polytechnique de Bordeaux	Rapporteur
Mme	Ana Cavalli	Professeur à TELECOM SudParis	Rapporteur
M.	Yves Ledru	Professeur à l'Université Joseph Fourier de Grenoble	Rapporteur
M.	Lionel Briand	Professeur à l'Université d'Oslo	Examinateur

Remerciements

Cette Habilitation à Diriger les Recherches a été présentée devant le Jury composé de :

- Karama Kanoun, Directeur de Recherche CNRS (Présidente et Garante),
- Richard Castanet, Professeur Emerite à l'Institut Polytechnique de Bordeaux (Rapporteur),
- Ana Cavalli, Professeur à TELECOM SudParis (Rapporteur),
- Yves Ledru, Professeur à l'Université Joseph Fourier de Grenoble (Rapporteur),
- Lionel Briand, Professeur à l'Université d'Oslo (Examinateur).

Je tiens à les remercier vivement de l'honneur qu'ils m'ont fait en participant à ce Jury. Ayant beaucoup d'estime pour leurs travaux et leur parcours, je suis heureuse qu'ils aient bien voulu s'associer à cette étape importante de ma vie professionnelle. Je remercie tout particulièrement mes trois rapporteurs pour le temps consacré à la lecture attentive de ce mémoire. Lors de la soutenance, l'ensemble du Jury m'a soumis à un feu roulant de questions que j'ai vraiment trouvées très intéressantes. Karama Kanoun s'était portée Garante de ma candidature HDR auprès de l'INPT et a ensuite accepté de présider le Jury. Je la remercie pour ses encouragements et son soutien amical.

Le groupe TSF du LAAS offre un environnement humain et scientifique très enrichissant. J'adresse une pensée chaleureuse à tous mes collègues. Parmi ceux-ci, je suis particulièrement redevable aux "premiers parmi les pairs" qui ont assumé la responsabilité de groupe, soit par ordre chronologique : Jean-Claude Laprie, David Powell, Jean Arlat et Karama Kanoun. Jean-Claude Laprie était le fondateur de TSF. C'était un très grand chercheur, avec une vraie vision prospective et une éthique de la recherche exemplaire. Son décès en octobre 2010 nous a tous bouleversés. Jean Arlat est maintenant en charge de la Direction du LAAS suite à un autre décès tragique, celui de Jean-Louis Sanchez. Je souhaite lui exprimer ma gratitude pour son implication au service du laboratoire, dans un moment particulièrement difficile. Je ne saurais non plus oublier le Directeur qui m'avait fait confiance en m'accueillant au LAAS, Alain Costes : je peux témoigner de sa bienveillance envers les jeunes chercheurs dont j'étais.

Mes premiers pas dans la recherche ont été guidés par Pascale Thévenod-Fosse, qui a dirigé mes travaux de doctorat. Cela induit à vie des liens tout à fait privilégiés, à la fois intellectuels et amicaux. Je ne saurais trop souligner à quel point ses compétences, sa rigueur scientifique et la qualité de ses conseils ont influé durablement sur moi. Aujourd'hui, c'est à mon tour de guider des étudiants. Je souhaite remercier tous les doctorants passés et actuels avec lesquels j'ai eu le plaisir de travailler : Yvan Labiche, Salimeh Behnia, Marc Mersiol, Olfa Abdellatif-Kaddour, Guillaume Lussier, Minh Duc N'Guyen, Thomas Bochot, Zoltán Micskei, Jimmy Lauret, Robert Guduvan et Pierre André. J'espère qu'ils garderont de leur doctorat le souvenir d'une période enrichissante, comme le mien l'avait été pour moi. Je souhaite également remercier les collègues du LAAS, de l'ONERA et de l'Université Technologique de Budapest

avec lesquels certains de ces doctorants ont été co-encadrés : Pascale Thévenod-Fosse, Jean Arlat, David Powell, Nicolas Rivière, Jean-Charles Fabre, Virginie Wiels, Guy Durieu et István Majzic. Une collaboration avec un autre collègue du LAAS, Yves Crouzet, a porté sur l'analyse de mutation.

Mes différentes expériences professionnelles, les projets et groupes de travail auxquels j'ai participés, m'ont permis de collaborer avec de nombreuses personnes que je ne peux toutes citer nommément. Qu'il me soit donc permis de remercier collectivement :

- les membres du centre de Recherche & Développement *(ZFE)* de *Siemens AG*, à Munich, qui m'ont accueillie lors d'un séjour post-doctoral ;
- les membres de l'Unité de Recherche *ESTAS* de l'*INRETS* (maintenant devenu l'IFSTTAR), où j'ai obtenu un premier poste avant de rentrer au CNRS ;
- les partenaires des projets européens *PDCS, PDCS2, CASCADE, DeVa, DSoS, ASSERT, HIDENETS* et du Réseau d'excellence *ReSIST* ;
- les partenaires industriels de mes contrats d'expertise ou de recherche : *Airbus, Alstom Transport, Cassidian Test & Services, Siemens Transportation Systems*, ainsi que la *Communauté Urbaine de Lille* ;
- Les membres industriels du *Laboratoire d'Ingénierie de Sûreté de Fonctionnement*, venus travailler en immersion au LAAS ;
- les partenaires du projet ANR *DALI* ;
- les membres du *B User Group*, de l'*Action Spécifique STIC CNRS n°23*, du groupe de travail "*Logiciel Libre et Sûreté de Fonctionnement*" du *Réseau d'Ingénierie de la Sûreté de Fonctionnement*, du groupe *MTV*^{2} du *GDR GPL*, et du groupe toulousain "*Spécification et Vérification Formelles*" ;
- Les chercheurs de l'*Université de Witwatersrand* (Afrique du Sud), de l'*Université de Campinas* (Brésil) et de l'*Université de York* (Royaume Uni) avec lesquels j'ai eu des collaborations.

Indépendamment de l'aspect scientifique, je n'aurais pu mener mes travaux dans de bonnes conditions sans le soutien des services techniques, administratifs et supports du LAAS : *Informatique et instrumentation*, *Documentation-Édition*, *Relations Contractuelles et Partenariales*, *Gestion*, *Personnel*, *Magasin*, *Systèmes d'Information*, *Infrastructure et Logistique*, *Assistantes de Direction*. Je dois également remercier les secrétaires successivement rattachées au groupe TSF : Joëlle Penavayre, Marie-José Fontagne, Gina Briand et Sonia De Sousa.

Plus ponctuellement, il me faut absolument mentionner l'aide de mon collègue Yves Crouzet, qui s'est installé au pupitre sono et vidéo pour filmer ma soutenance de HDR. Je le remercie d'avoir ainsi permis d'immortaliser l'événement…

A Laurent, Hugo et Clara.

Sommaire

Introduction générale

Ce mémoire présente un ensemble de travaux que j'ai réalisés au sein du groupe "Tolérance aux fautes et sûreté de fonctionnement informatique" du LAAS, depuis mon recrutement au CNRS en 1996. Les fautes affectant les systèmes informatiques sont au cœur des problématiques abordées dans ce groupe. Le développement d'un système sûr de fonctionnement passe par l'utilisation combinée d'un ensemble de méthodes qui peuvent être classées en quatre grandes catégories [ALR+ 04] : prévention des fautes, tolérance aux fautes, élimination des fautes et prévention de fautes. Dans ce cadre, ma contribution personnelle porte sur l'élimination des fautes. Je m'intéresse plus particulièrement au test du logiciel.

Le test est une méthode de vérification dynamique. Elle consiste à exécuter un logiciel en lui fournissant des valeurs d'entrée. Les résultats de l'exécution sont observés pour élaborer un verdict d'acceptation ou de rejet. Sauf cas trivial, le test exhaustif est impossible ; le test n'effectue donc qu'une vérification partielle. Pour révéler une faute, il faut que plusieurs conditions soient réunies : (i) les entrées fournies pour l'exécution permettent d'activer la faute, produisant une erreur ; (ii) par propagation, cette erreur crée d'autres erreurs jusqu'à affecter une sortie observable ; (iii) l'analyse des résultats de test détecte une déviation par rapport au comportement attendu. La conception du test est ainsi confrontée à deux problèmes majeurs, celui de la sélection des entrées et celui de la détermination du comportement attendu en réponse à ces entrées (également appelé problème de l'oracle). Ces problèmes se déclinent à différents niveaux de test, selon une stratégie d'intégration où des composants sont typiquement testés de façon unitaire puis graduellement assemblés jusqu'à tester le système complet.

Les critères de test permettent d'aborder de façon méthodique le problème de la sélection d'entrées de test. Basés sur l'analyse de la structure du logiciel, ou sur l'analyse des fonctions qu'il doit réaliser, ils définissent un ensemble d'éléments à couvrir pendant le test. Des exemples de critères sont la couverture des branches du graphe de contrôle (un modèle structurel) et la couverture des transitions d'une machine à état (un modèle fonctionnel). L'ouvrage [AO 08] présente une synthèse des critères de couverture existants. Un point mis en avant par les auteurs est que tous ces critères, qu'ils soient structurels ou fonctionnels, s'appuient en fait sur un petit nombre de types de modèles : des graphes, des expressions logiques, des décompositions de domaines en classes de valeurs, et des descriptions syntaxiques (grammaires formelles). Un critère peut être utilisé *a priori*, les entrées étant spécifiquement sélectionnées pour satisfaire le critère retenu, ou *a posteriori*, le critère servant alors à mesurer la couverture offerte par un ensemble d'entrées sélectionnées par d'autres méthodes. Dans une perspective d'automatisation, ceci correspond à deux familles d'outils, les outils de génération de test et les outils d'analyse de couverture. La génération de test étant la plus difficile à automatiser, de nombreux travaux de recherche se sont attaqués à ce défi (voir par exemple les outils de génération mentionnés dans [BFS 05]). Certains outils sont

spécialisés pour des critères prédéfinis, d'autres laissent à l'utilisateur la flexibilité de spécifier les éléments à couvrir. En pratique, malgré des succès dans des domaines spécialisés comme le test des protocoles, la génération de test reste problématique avec peu d'utilisations opérationnelles dans l'industrie.

La solution la plus répandue au problème de l'oracle consiste à déterminer manuellement les valeurs de sorties attendues. Cette approche est cependant fastidieuse et source d'erreur. Une automatisation est préférable. La solution la plus satisfaisante suppose l'existence d'une spécification formelle complète. La notion de conformité de l'implémentation par rapport à sa spécification prend un sens mathématiquement bien défini, fournissant l'oracle de test quelles que soient les entrées sélectionnées. Par exemple, plusieurs des outils de génération de test mentionnés ci-dessus, qui se basent sur une modélisation complète du comportement attendu, produisent des cas de test incorporant la détermination de leur verdict. En l'absence d'une spécification formelle complète, il est encore possible d'envisager un oracle partiel ciblant un ensemble de propriétés. Cette solution est notamment employée dans les approches de test passif (voir par exemple [LAC 05]), qui observent la trace d'exécution d'un système en fonctionnement. Le principe d'une vérification partielle est par ailleurs courant dans les mécanismes de détection en-ligne pour la tolérance aux fautes [AL 81] [RFR 08]. Les propriétés ciblées par un oracle partiel vont de simples contrôles de plausibilité (appartenance à une plage de valeurs, cohérence entre différentes données) à des invariants sophistiqués comprenant des aspects temporels [FFJ+ 10]. L'instrumentation du code avec des assertions exécutables permet des vérifications à un niveau de granularité fin, pour détecter des états internes erronés ou des violations de pré/postconditions au sein de séquences d'appels d'opérations [LBS 02] [BLM+ 04] [LBJ 06].

Ces deux problèmes de conception du test, sélection et oracle, sont récurrents dans mes travaux. Un souci majeur est notamment l'imperfection des critères de test pour révéler les fautes dans le logiciel. Dans les années 80 et jusqu'au début des années 90, une question débattue au sein de la communauté du test était si l'utilisation méthodique de critères était réellement plus efficace qu'une sélection aléatoire aveugle sur le domaine d'entrée[1] [DN 84] [HT 90] [WJ 91]. Ce débat a été en toile de fond de mes travaux de thèse de doctorat [Wae 93], qui ont porté sur une approche originale combinant utilisation de critères et procédé de génération probabiliste : le *test statistique*. Notons que cette approche, qui peut conduire à une taille de test élevée, nécessite une solution automatique au problème de l'oracle. Depuis ma thèse sous la direction de Pascale Thévenod-Fosse, le test statistique n'a plus été pour moi un objet de recherche en tant que tel. Néanmoins, comme il est utilisé dans le cadre de plusieurs travaux présentés dans ce mémoire, et qu'il sera mentionné dans mon programme de recherche, j'en fais ici une brève introduction.

Le principe du test statistique est d'exploiter l'information apportée par le critère retenu, sans pour autant trop cibler la sélection des entrées de test. Pour un critère donné, l'approche requiert : 1) la recherche d'une distribution d'entrée qui maximise la probabilité d'activer l'élément du critère le moins probable et 2) le calcul d'une longueur de test N associée à cette distribution, pour satisfaire une exigence de qualité q_N vis-à-vis du critère – q_N étant la probabilité d'activer l'élément le moins probable au moins une fois au cours de N exécutions. La sélection d'un jeu de test consiste alors à tirer aléatoirement N entrées de test selon le profil défini. En pratique, pour des qualités de test élevées (> 0.9), chaque élément du critère va être

[1] La question resurgit parfois lors d'expérimentations de techniques de génération automatique de test. Par exemple dans [HDW 04] : "To our surprise (and dismay) the randomly generated tests performed better than the generated structural tests", "Our experiences from this experiment raises some concern about the use of automated test case generation from formal specifications".

en moyenne activé plusieurs fois avec des entrées aléatoires différentes. Plusieurs études expérimentales ont montré l'efficacité de cette approche, qui permet de compenser l'imperfection des critères vis-à-vis des fautes recherchées. Suite à nos travaux, d'autres auteurs ont exploré des mises en œuvre différentes du test statistique, utilisant la génération aléatoire de structures combinatoires [DGG 04], une extension probabiliste de la programmation par contraintes [PG 07] ou encore des métaheuristiques de recherche [PC 10].

Ainsi qu'indiqué plus haut, on ne trouvera pas dans ce mémoire de nouvelles contributions sur le test statistique en tant qu'objet de recherche. Par contre, je continue à utiliser cette approche pour la génération de test, par exemple dans le cadre d'expérimentations de nouveaux critères. Plus généralement, la filiation à l'école probabiliste du test m'a naturellement conduite à m'intéresser à différents procédés aléatoires ou métaheuristiques. Ces procédés sont mis en œuvre pour implémenter des profils de charge, évaluer l'apport de méthodes de test proposées (qui sont alors comparées au test aléatoire uniforme), rechercher des entrées de test possédant certaines propriétés, ou tout simplement obtenir à faible coût un large échantillon d'entrées de test. La plupart de mes travaux présentent une forte composante expérimentale. Ils sont étayés par des études de cas – pour certaines industrielles – avec des fautes à révéler. Ces études de cas concernent des logiciels applicatifs critiques (typiquement des logiciels de contrôle/commande) ou des services de tolérance aux fautes. Dans ce dernier cas, on distingue d'une part les fautes de conception affectant les services, que le test cherche à révéler, et d'autre part les fautes à tolérer, considérées comme des entrées pour le test.

D'un point de vue thématique, mes contributions peuvent être classées en quatre grandes catégories, correspondant aux chapitres de ce mémoire.

Le premier chapitre rassemble des travaux pour adapter la conception du test aux technologies de développement de logiciels. Les contributions portent sur deux exemples de technologies : la technologie orientée objet et la méthode formelle B. Un point commun de ces contributions est qu'elles considèrent des spécificités telles que les liens architecturaux propres à chaque technologie, les artefacts de développement exploitables pour définir des critères de couverture, et les relations de conformité convenables (par exemple, basées sur les contrats des objets ou sur la notion de raffinement entre machines abstraites B).

Le deuxième chapitre étudie des associations entre techniques de test et de vérification formelle. L'association test et preuve de théorème est utilisée pour consolider la vérification d'algorithmes partiellement prouvés. La conception du test est alors guidée par la structure logique de l'arbre de preuve et par la connaissance des branches non prouvées : on cherche ainsi à cibler les lacunes de la preuve. L'association test et model checking est utilisée pour analyser des contrexemples, lorsque la vérification formelle trouve une violation de propriété. Il s'agit d'une part de faciliter la compréhension des causes de cette violation, et d'autre part de permettre la génération de plusieurs contrexemples, illustrant des scénarios de violation différents. L'approche proposée repose sur l'identification de chemins activés par un contrexemple, et présente des liens étroits avec des travaux sur le test structurel.

Le troisième chapitre traite de la génération de test par des procédés métaheuristiques, en prenant l'exemple du recuit simulé [KGV 83]. Une métaheuristique correspond à une stratégie générale, dont on doit régler les paramètres selon le problème à résoudre. L'accent est mis sur l'utilisation de mesures pour guider le paramétrage. Les mesures retenues sont traditionnellement utilisées dans le cadre de problèmes combinatoires liés à la recherche opérationnelle ou à la théorie de la complexité des algorithmes. Nous montrons leur intérêt dans le cadre d'un problème de génération de test.

Le quatrième chapitre aborde le test de systèmes mobiles, incluant des dispositifs qui se déplacent dans le monde physique tout en étant connectés aux réseaux par des moyens sans

fil. Nous avons défini un langage formel basé sur UML pour représenter des scénarios d'interaction au sein d'un tel système. Ces scénarios correspondent à des propriétés que l'on souhaite vérifier sur les traces de test. Ils considèrent à la fois les configurations spatiales des nœuds mobiles et leurs communications. Les configurations spatiales montrent la topologie de connexion sous forme de graphes étiquetés, le mouvement des nœuds étant abstrait par la séquence de configurations traversées. L'analyse d'une trace de test par rapport à un scénario combine alors des algorithmes d'appariement de graphes et des algorithmes de calcul d'ordres partiels d'événements.

Chaque chapitre démarre par une introduction qui justifie l'intérêt de la thématique abordée, et qui situe nos contributions par rapport à d'autres travaux précédents ou contemporains. A la fin du chapitre, les résultats principaux et les questions ouvertes sont discutés et mis en perspective avec d'autres travaux postérieurs. Le mémoire termine par une présentation de mon programme de recherche pour les années à venir.

Chapitre I. Prise en compte des technologies de développement des logiciels testés

Le test intervient au sein d'un processus de développement, et les choix technologiques retenus pour ce développement ont nécessairement un impact sur la conception et l'implémentation du test. Ce chapitre présente mes contributions pour deux exemples de technologies : la technologie orientée-objet (§I.1 et I.2) et la méthode formelle B (§I.3). Dans les deux cas, les travaux présentés ont été menés dans la deuxième moitié des années 90. Il s'agit d'une période où ces technologies se diffusaient de façon opérationnelle dans l'industrie, massivement pour l'orienté-objet et dans un cadre plus restreint pour la méthode B (logiciels critiques du domaine ferroviaire en France).

La technologie orientée-objet a introduit une rupture par rapport aux logiciels procéduraux. Le développement n'est plus centré sur les fonctions, mais sur des entités – les classes – encapsulant des données et offrant des opérations pour accéder à ces données. Les liens entre classes (héritage, association, agrégation, …) induisent des architectures éclatées et non hiérarchiques. Les classes sont instanciées par des objets, avec du polymorphisme et des liaisons dynamiques à l'exécution. Ces caractéristiques ont nécessité de revisiter les approches de test, initialement définies pour un cadre procédural. De fait, lorsque nous avons démarré nos travaux, le test de logiciels orientés-objet était un domaine de recherche en pleine expansion. Les auteurs s'intéressaient à la prise en compte de l'héritage [PK 90] [HMF 92], au test structurel de classes [PBC 93] [HR 94], au test fonctionnel [TR 93] [DF 94] [HKC 95], et au test d'intégration des classes [JE 94] [KGH+ 95]. L'article [Bin 96] donne une vue détaillée des travaux en cours à cette époque.

Mes travaux sur le test de logiciels orientés-objet ont été réalisés en collaboration avec Pascale Thévenod-Fosse (LAAS). Ils ont suivi deux directions. La première est le test de robustesse d'objets concurrents (§I.1), de façon à faciliter leur réutilisation dans des contextes applicatifs différents. Cette problématique réunit des aspects relatifs à la conception objet (modèles d'interface, contrats) et des aspects relatifs à la programmation concurrente (entrelacements d'événements, files d'attente). L'approche proposée se base sur des modèles issus d'une méthode de conception orientée-objet, Fusion [CAB+ 94], et adopte une génération probabiliste de tests avec différents profils de charge. La deuxième direction suivie par nos travaux est la définition et l'ordonnancement d'étapes de test d'intégration pour une application donnée (§I.2). Elle reprend et étend les travaux pionniers de Kung *et al.* [KGH+ 95], qui ont été les premiers à s'intéresser à un ordre de test pour des architectures orientées-objet. Une de nos contributions a été de distinguer les étapes ciblant des dépendances statiques entre classes et celles ciblant des dépendances dynamiques liées au polymorphisme. De plus, nous avons considéré la possibilité de supprimer des étapes infaisables (du fait de la présence d'une classe abstraite non instanciable), et de reporter leurs

objectifs sur d'autres étapes. Ces travaux sur le test d'intégration de classes ont fait l'objet de la thèse d'Yvan Labiche [Lab 00], en partenariat avec Airbus.

La deuxième technologie abordée dans ce chapitre est la méthode B, conçue par Jean-Raymond Abrial [Abr 96]. Cette méthode formelle couvre toutes les étapes de développement du logiciel, par raffinements successifs d'une spécification initiale jusqu'à la génération de code source. Des preuves sont effectuées à chaque étape. En France, l'industrie ferroviaire a été un acteur majeur pour la promotion et la maturation de la méthode B, qui a été utilisée avec succès dans plusieurs applications critiques pour des métros ou des trains [BDM 1997] [BBF+ 99] [DEF 03]. Dans les développements industriels basés sur B, les tests unitaires sont supprimés. Des tests fonctionnels de plus haut niveau sont cependant conservés pour révéler d'éventuelles fautes liées à une formalisation incorrecte des exigences de l'application. Par essence, le passage du cahier des charges informel à la modélisation B ne peut pas être vérifié par des preuves. Les tests fonctionnels sont conçus à partir du cahier des charges, et sont passés sur le code généré en fin de développement formel. Il m'a paru intéressant d'étudier comment remonter la validation fonctionnelle à une étape plus amont, en considérant le test de modèles B. Ces travaux (§I.3) ont fait l'objet de la thèse de Salimeh Behnia [Beh 00], en partenariat avec l'INRETS.

Une ligne directrice de nos travaux a été de tirer parti du cadre uniforme qu'offre B pour modéliser une application à différents niveaux d'abstraction. Nous nous sommes attachés à définir un cadre théorique de test lui-même uniforme, que la cible du test soit un modèle abstrait, ou un modèle comportant des étapes de raffinement. En particulier, la notion d'oracle de test a été liée à la notion de raffinement : si, à une étape de développement, l'animation de modèles fournit des résultats corrects selon le cahier des charges, alors les obligations de preuve de B garantissent que tout raffinement ultérieur fournira aussi des résultats corrects pour les mêmes entrées de test. Ceci nous a notamment conduit à prendre en compte certains problèmes posés par les liens architecturaux présents dans B [Rou 99], et à identifier des insuffisances dans les outils d'animation de modèles B existants. Enfin, nous avons proposé des analyses de couverture de modèles B qui unifient des critères applicables à un niveau concret (couverture de graphes de contrôle) et des critères applicables à un niveau abstrait (couverture de prédicats avant/après).

I.1. Test de robustesse d'objets concurrents réutilisables

Dans ces travaux, nous considérons des applications consistant en des objets concurrents liés par des relations client/serveur. Les objets considérés sont des petits sous-systèmes construits par instanciation de quelques classes, suffisamment significatifs pour avoir un modèle d'interface définissant les services offerts à leurs utilisateurs. L'accent est mis sur la notion de robustesse d'un objet. Il s'agit de tester la conformité par rapport au modèle d'interface dans un environnement d'utilisation le moins contraint possible, de façon à faciliter la réutilisation dans des applications différentes. Dans le cadre d'applications concurrentes, un objet robuste doit notamment pouvoir gérer des sollicitations arrivant dans un ordre et à une fréquence arbitraires. Les résultats de test sont alors exploités pour rejaillir sur la conception de l'objet, le rendant ainsi plus robuste, ou pour expliciter des contraintes d'utilisation de l'objet.

I.1.1. Approche proposée

L'approche proposée exploite des modèles issus de la méthode Fusion [CAB+ 94]. Selon cette méthode, le modèle d'interface d'un objet comprend :

- Un *cycle de vie*, spécifiant l'ordre de traitement et d'émission de messages au moyen d'une expression régulière ;
- Un *modèle des opérations*, spécifiant des contrats de type pre/post conditions[2].

La spécification de contrats est un point fort de Fusion. Notons que l'ordre général défini dans le cycle de vie d'un objet ne garantit pas toujours la précondition des opérations. Le modèle des opérations ajoute donc des contraintes supplémentaires sur l'ordre autorisé.

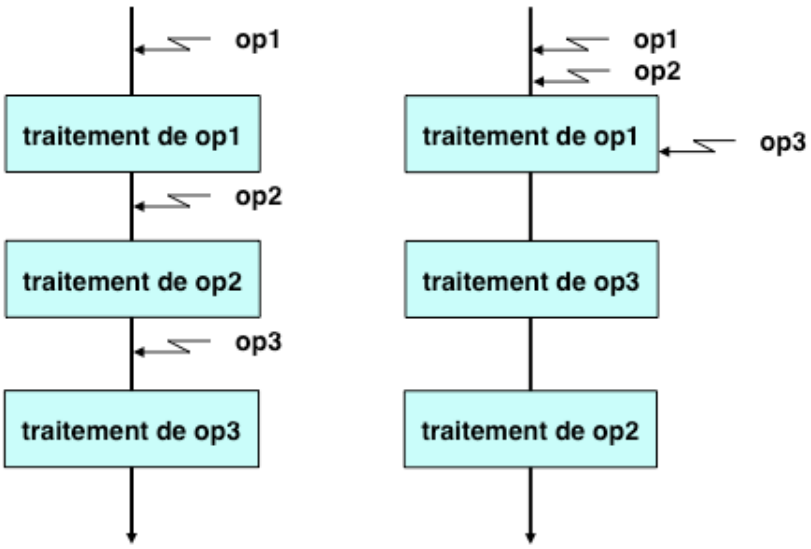
Dans le cadre d'applications séquentielles, la précondition d'une opération représente une obligation pour le client : il doit s'assurer qu'il n'appelle jamais l'opération de l'objet en dehors de sa précondition. Dans le cadre d'applications concurrentes, cette interprétation des préconditions n'est plus adaptée. Le client n'a aucun contrôle sur l'état de l'objet, qui peut être modifié à tout moment par l'action d'autres clients concurrents. Meyer suggère alors d'interpréter la précondition non comme une obligation, mais comme un mécanisme de synchronisation [Mey 93] [NMO 09] : la requête du client est mise en attente jusqu'à ce que la précondition devienne vraie. En fonction des requêtes en attente, l'objet choisit la prochaine à traiter. Son choix doit satisfaire le cycle de vie et le modèle des opérations.

Pour modéliser un tel comportement, nous reformulons le modèle d'interface par un modèle comprenant :

- un modèle à états combinant les ordres spécifiés dans le cycle de vie et le modèle des opérations,
- des files d'attentes attachées à chaque état.

Un critère de couverture de ce modèle peut alors être défini en considérant la structure états / transitions (ex : couverture des transitions) et certaines classes de configurations de files (ex : 0, 1, 2, 3 ou un nombre ≥ 3 de requêtes en attente pour telle opération). Contrôler la couverture est cependant complexe, car l'exécution d'une séquence de test peut induire différents entrelacements des réceptions des requêtes et de l'activité de l'objet. Le problème est illustré en Figure I-a, pour un objet testé avec une séquence de requêtes *op1*, *op2* et *op3*.

[2] Les notations Fusion étant aujourd'hui obsolètes, une reformulation moderne du problème considèrerait une modélisation UML, avec notamment des diagrammes états-transitions et des contrats OCL.



(1) Environnement lent **(2) Environnement quelconque**

Figure I-a Différents entrelacements d'une séquence de test

Dans le cas (1), l'environnement est lent par rapport au temps de réaction de l'objet. L'objet est *quiescent* [Vaa 91] quand il reçoit une nouvelle requête : il a fini le traitement de toutes les requêtes précédentes éligibles. On lui laisse toujours le temps de traiter la nouvelle requête, ainsi que les requêtes en attente devenues éligibles suite à ce traitement, avant d'envoyer la prochaine requête. Si l'on connaît la stratégie implémentée pour vider les files, avec par exemple des stratégies FIFO ou des priorités fixes entre *op1*, *op2* et *op3*, alors le comportement est déterministe. De manière générale, même s'il reste de l'indéterminisme, l'environnement lent est celui qui permet le plus de contrôle sur le comportement de l'objet. Cette approche est donc généralement adoptée pour le test de conformité.

Dans le cas (2), on ne fait plus l'hypothèse d'un environnement lent. Plusieurs requêtes éligibles peuvent arriver avant que l'objet ne choisisse d'en traiter une ; une nouvelle requête peut également arriver en cours de traitement. Du fait de tous les entrelacements susceptibles de se produire lors de l'exécution de la séquence de test, le comportement n'est pas contrôlable. Notons que dans notre cas, la conception du test ne peut pas se limiter à la considération d'un environnement lent : le but est précisément de vérifier le comportement d'objets plongés dans un environnement de clients concurrents, envoyant des requêtes dans un ordre et à des intervalles de temps arbitraires.

Pour assurer une couverture satisfaisante du modèle comportemental, tout en n'excluant pas des entrelacements tels que ceux de la Figure I-a.2, notre génération de test procède en deux étapes. Dans une première étape, on fait l'hypothèse d'un environnement lent, et on génère des séquences de requêtes selon un critère de couverture du modèle comportemental (par exemple, la couverture des transitions et des configurations de files). Dans la lignée de nos travaux antérieurs sur le test statistique fonctionnel [TW 93], nous proposons une génération probabiliste des séquences de requêtes à tester, chaque élément du modèle étant en moyenne activé plusieurs fois au cours du test. Dans une deuxième étape, on complète les séquences de test en générant les intervalles de temps entre deux requêtes consécutives. Nous proposons de considérer trois profils de charge :

- les intervalles de temps sont longs par rapport au temps de réaction de l'objet (environnement lent) ;

- les intervalles de temps sont courts par rapport au temps de réaction de l'objet, ou du même ordre de grandeur (charge élevée de l'objet) ;
- les intervalles de temps peuvent être longs ou courts (charge variable au cours de la séquence).

Ainsi, toutes les séquences de requêtes de la première étape seront en fait exécutées trois fois, avec potentiellement des entrelacements différents dans chaque cas.

L'oracle de test est défini quel que soit le profil de charge. Lors de l'exécution d'une séquence, il acceptera le comportement de l'objet si : (i) l'ordre de traitement des requêtes est accepté par le modèle à états ; (ii) chaque traitement de requête assure la postcondition spécifiée ; (iii) à la fin de l'exécution de la séquence, aucune requête non traitée n'est éligible.

En résumé, la notion de contrat – revisitée dans le cadre d'objets concurrents – amène à considérer explicitement la mise en attente de requêtes. L'accent mis sur la robustesse amène à considérer différents entrelacements possibles lors de l'exécution des tests. Notre approche traite ces deux aspects en introduisant des files dans la modélisation comportementale, et en ayant différents profils de charge pour exécuter les tests sélectionnés à partir du modèle.

I.1.2. Application à une étude de cas : la cellule de production

Cette approche de test de robustesse a été appliquée à une étude de cas : la cellule de production [LL 95]. L'étude de cas a été proposée par le FZI (Forschungzentrum Informatik, Allemagne), qui a fourni une spécification du logiciel de contrôle/commande et un simulateur des équipements physiques pilotés par ce logiciel. Des chercheurs de l'EPL (Suisse) ont mis à notre disposition un dossier complet de développement du logiciel de contrôle/commande [BBP 98] : analyse et conception Fusion, implémentation en Ada 95 connectée au simulateur.

Le but de la cellule de production est de forger des plaques métalliques. La Figure I-b montre le diagramme de contexte système. On voit que l'analyse Fusion distingue 6 agents concurrents, chacun assurant le pilotage d'un dispositif physique de la cellule : deux tapis convoyeurs (*FeedBelt*, *DepositBelt*), une table (*Table*), un robot (*Robot*), une presse (*Press*), une grue (*Crane*). Ces agents peuvent s'envoyer des requêtes. Par exemple, le robot est client des deux opérations offertes par la presse, *go_load_position* et *forge*. Inversement, la presse est cliente de deux des opérations offertes par le robot, *load_press* et *pick_from_press*. Chaque agent est décrit par un modèle d'interface et correspond à un sous-système de complexité modeste. Ainsi le robot, qui est l'agent le plus complexe, est implémenté par un objet agrégeant deux bras, un potentiomètre et un moteur ; chaque bras est également un agrégat avec un moteur, un potentiomètre et un électro-aimant. En combinant les informations issues du cycle de vie et du modèle des opérations, l'ordre de traitement des requêtes par le robot peut être décrit par un automate comportant seulement 8 états et 13 transitions.

Le test de robustesse des agents, ainsi que les tests d'intégration au sein de la cellule de production, sont décrits dans [WT 99]. La génération des tests de robustesse a suivi les deux étapes présentées précédemment :

1. Génération probabiliste de séquences de requêtes, offrant une qualité de test $q_N = 0.999$ par rapport à un critère de couverture (transitions et configurations de files) dans un environnement lent.
2. Génération d'intervalles de temps pour les séquences, selon trois profils de charge calibrés en fonction du temps de réaction des objets.

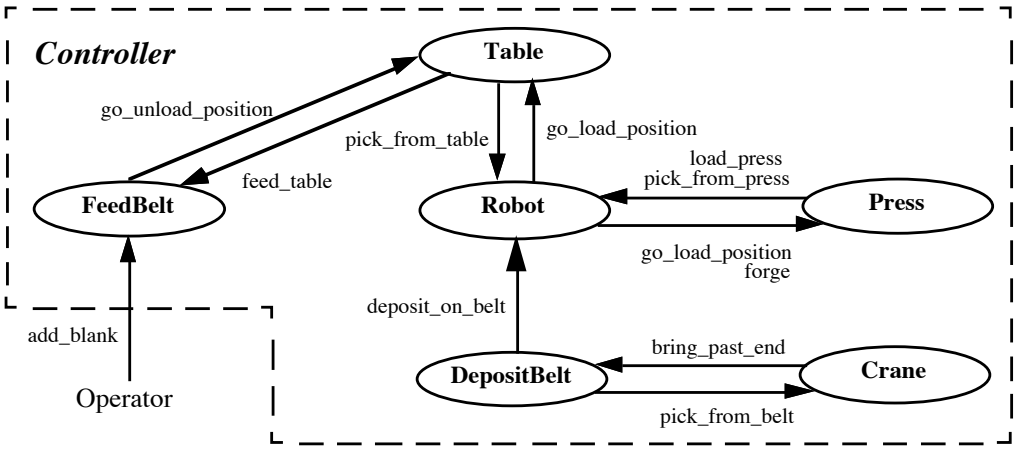


Figure I-b Diagramme de contexte système (vue interne)

L'implémentation des tests a été réalisée de façon à poursuivre l'exécution d'une séquence lorsque la requête courante est mise en attente.

Nos tests ont mis en évidence trois problèmes, induisant des défaillances d'autant plus fréquentes que la charge était élevée. Le premier problème affectait tous les agents, et concernait la synchronisation avec le simulateur. Il a pu être résolu par la modification de deux classes de base gérant la communication avec les capteurs et actionneurs. Les deux autres problèmes affectaient respectivement *Robot* et *DepositBelt*.

Dans le cas de *Robot*, nous avons observé le non-traitement de requêtes pour des configurations de files avec plusieurs *deposit_on_belt* en attente. L'agent n'est pas suffisamment robuste pour gérer des requêtes *deposit_on_belt* multiples : il perd toute nouvelle requête envoyée tant que la précédente n'a pas été traitée. Une modification mineure du code de l'opération a permis de rendre l'agent robuste aux requêtes multiples.

Dans le cas de *DepositBelt*, nous avons observé la violation de conditions, avec notamment des chutes de plaques en fin de tapis. Il s'agissait d'un problème de robustesse lié au convoi de plusieurs plaques. Sa résolution aurait nécessité une modification importante de l'interface de l'agent. Nous avons choisi de ne pas entreprendre de modification, mais d'ajouter une contrainte explicite sur l'environnement d'utilisation : aucune nouvelle plaque ne peut être déposée avant que la précédente ne soit arrivée en fin de tapis. La satisfaction de cette contrainte a pu être vérifiée lors des tests d'intégration de la cellule de production.

Au travers de cette étude de cas, on voit les bénéfices qui peuvent être tirés du test de robustesse d'un objet : soit on rejaillit sur la conception de l'objet, pour qu'il puisse gérer des sollicitations arrivant dans un ordre et à une fréquence arbitraires ; soit on explicite des contraintes d'utilisation à vérifier lors de l'intégration au sein de toute application.

I.2. Tests d'intégration inter-classes : définition et ordre des étapes de test

Les logiciels orientés-objet sont caractérisés par des architectures décentralisées et non hiérarchiques. Une classe dépend généralement de plusieurs autres classes, avec des liens d'héritage, d'agrégation et d'association. Prise isolément, une classe peut sembler élémentaire (avec quelques lignes de code pour chaque méthode) mais la complexité émerge par composition avec le reste du système. Une conséquence de cette complexité émergente est l'importance des tests d'intégration inter-classes.

La stratégie d'intégration doit définir plusieurs étapes de test, avec des sous-ensembles de plus en plus grands de classes. Idéalement, la conception du test à chaque étape peut alors s'effectuer de façon incrémentale, en considérant ce qui a déjà été testé aux étapes précédentes, et ce qu'il y a à tester (ou re-tester) du fait des classes nouvellement introduites. Déterminer l'ordre d'intégration des classes est cependant non trivial : l'architecture n'étant pas hiérarchique, il n'y a ni "bas" ni "haut" par lequel commencer. De plus, il faut tenir compte des différents types de liens entre classes, et des couplages plus ou moins forts. Kung *et al.* ont été les premiers à aborder ce problème de l'ordre de test [KGH+ 95]. Après avoir rappelé brièvement leurs travaux (§I.2.1), je présente les extensions que nous avons proposées et qui forment l'approche TOONS (*Testing level generator for Object-OrieNted Software*, §I.2.2). TOONS a été appliquée à une étude de cas fournie par Airbus (§I.2.3). Les étapes de test ainsi identifiées ont permis une mise en œuvre incrémentale du test.

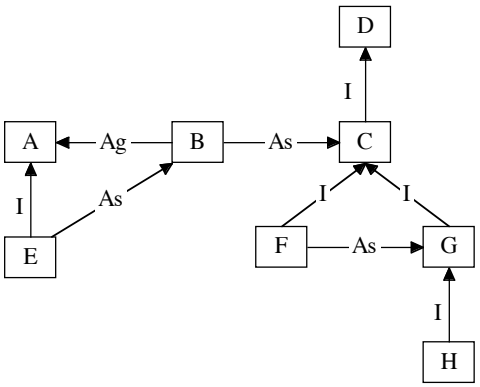
I.2.1. Travaux pionniers de Kung *et al*.

Le but de Kung *et al.* était de trouver un ordre d'intégration des classes qui minimise le nombre de *stubs*. Un stub est un composant de test placé comme substitut à un composant externe dont dépend le sous-système testé. Le développement de stubs est coûteux, difficilement automatisable et source de fautes, d'où le souhait de minimiser leur nombre.

L'approche proposée dans [KGH+ 95] se base sur un graphe orienté montrant les dépendances entre classes, l'ORD (*Object Relation Diagram*). Un exemple d'ORD est montré dans la partie gauche de la Figure I-c. Un ORD comporte trois types d'arcs, représentant des relations d'héritage (*I*), d'agrégation (*Ag*) et d'association (*As*). Dans les travaux de Kung *et al.*, le graphe est obtenu par rétro-conception de code C++ mais on peut également l'obtenir à partir de modèles de conception (diagrammes de classes UML). Pour chaque classe *X*, le graphe permet de déterminer l'ensemble des classes *CFW(X)* qui dépendent transitivement de *X* (voir Figure I-c.2).

Si l'ORD est acyclique, comme dans la Figure I-c, un tri topologique inversé permet de déterminer l'ordre de test des classes : une classe X est toujours testée avant les classes *CFW(X)* qui dépendent d'elles et il n'y a pas besoin de stubs. Dans l'exemple de la Figure I-c, on commence par tester les classes *A* et *D* qui ne dépendent d'aucune autre classe ; puis on teste la classe *C* qui hérite du code de *D* déjà testé, etc.

Si l'ORD contient des cycles, Kung *et al.* proposent de combiner tri topologique inversé et coupure de cycles. La coupure de cycles consiste à identifier les composantes fortement connexes de l'ORD, et à supprimer des arcs d'association au sein de ces composantes pour se ramener à un cas acyclique (des stubs devront donc être développés). Les arcs d'association sont choisis de préférence aux arcs d'héritage ou d'agrégation car ils correspondent à un couplage plus faible entre classes. L'ordre global des classes est alors calculé à partir des ordres entre (i) composantes fortement connexes et (ii) classes au sein d'une composante.



(1) ORD

Classe X	CFW(X)
A	B, E
B	E
C	B, E, F, G, H
D	B, C, E, F, G, H
E	ø
F	ø
G	F, H
H	ø

(2) Analyse de dépendance (*Class Firewall* CFW)

Niveau de test	Classe(s)
1	A, D
2	C
3	B, G
4	E, F, H

(3) Ordre de test

Figure I-c Ordre de test de *Kung et al.*

Ces travaux pionniers ont été le point de départ de nombreux développements, dont les nôtres [LTW+ 00]. Dans la période contemporaine à nos travaux, plusieurs auteurs se sont concentrés sur la coupure de cycles [JJL+ 99] [TD 99], avec différents procédés de calcul de l'ordre de test global. Pour notre part, nous nous sommes intéressés au traitement des ORD acycliques, c'est-à-dire après coupure de cycles. En effet, nous avions identifié des points à améliorer dans l'approche de Kung *et al.* :

- La focalisation sur l'ordre des classes, considérées une par une, masque la réalité des étapes de test qui impliquent en fait des ensembles de classes (par exemple, le test de *B* requiert au minimum les classes *A*, *C*, *D*). Il serait intéressant de garder l'information de cette complexité croissante des étapes, pour éventuellement modifier l'architecture en phase de conception, et en tout cas pour planifier l'effort de test.
- Les liaisons dynamiques (à l'exécution) ne sont pas prises en compte dans l'analyse de dépendance, ni donc dans le calcul de l'ordre de test. Dans l'exemple de la Figure I-c, on considère que *F* et *H* sont indépendantes alors que *F* peut être dynamiquement associée à *H*, du fait du polymorphisme.
- Certaines étapes peuvent être infaisables par la présence de classes abstraites non instanciables. Par exemple, si *A* est une classe abstraite, la première étape de test de *A* ne peut être réalisée. De plus, le test de *B* au niveau 3 requerra l'instanciation de la classe fille *E* (au lieu de *A*) ; mais le test de *E* est prévu après *B*, puisque *E* est associée à *B*. De façon générale, il faudrait pouvoir supprimer certaines étapes de test, et adapter l'ordre et le contenu des étapes restantes en conséquence.

Notre approche TOONS traite ces différents points.

I.2.2. L'approche TOONS

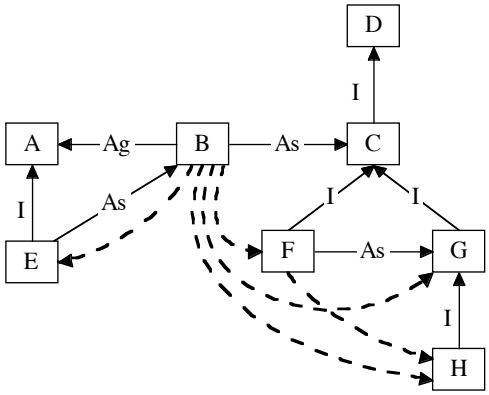
TOONS [LTW+ 00] procède comme suit :

1. Analyse des dépendances statiques et dynamiques entre classes, dont on déduit un ensemble d'étapes de test à réaliser ;
2. Ordonnancement des étapes de test ;
3. Analyse du rôle des classes impliquées dans chaque étape ;
4. Suppression des étapes infaisables et report de leurs objectifs sur les étapes restantes.

La sortie est une visualisation graphique de l'ordre partiel des étapes, avec une notation qui montre le rôle des classes impliquées dans chaque étape. Les traitements 1 à 4 sont détaillés dans le manuscrit de Yvan Labiche [Lab 00]. La complexité algorithmique du pire cas est $O(n^4)$, où n est le nombre de classes de l'architecture. Un prototype a été implémenté et connecté à l'environnement UML *Rational Rose*[3]. Les paragraphes suivants donnent une présentation informelle des traitements réalisés et les illustrent sur l'exemple de la Figure I-c.

I.2.2.1.Analyse des dépendances statiques et dynamiques

Le premier traitement considère deux types de dépendance : statique (D_1) et dynamique (D_2). La relation D_1 est l'inverse du Class Firewall considéré par Kung *et al.* : pour une classe X, $D_1(X)$ est l'ensemble des classes dont dépend X via un chemin dans l'ORD. $D_1(X)$ peut être vu comme l'ensemble minimal de classes devant être présentes pour la compilation et le test de X, lorsque l'on ignore le polymorphisme. La relation D_2 étend D_1 en ajoutant toutes les dépendances dues au polymorphisme : à l'exécution, la cible Y d'un lien d'association ou d'agrégation peut être intanciée par des objets de n'importe quel type dérivé de Y par héritage. La Figure I-d.1 reprend l'ORD de l'exemple et montre les arcs à rajouter (en pointillés). L'ORD ainsi complété est appelé C-ORD. D_2 est alors calculée par transitivité sur ce C-ORD, en prenant en compte tous les types d'arc. Par construction, $D_2(X)$ inclut $D_1(X)$. Une inclusion stricte indique la présence de dépendances dynamiques vers de nouvelles classes. Une égalité ne signifie cependant pas l'absence de dépendances dynamiques, car celles-ci peuvent apparaître entre classes déjà liées par des dépendances statiques. Pour repérer ces cas, un Booléen $B_d(X)$ est mémorisé : il indique si X dépend dynamiquement d'au moins une classe. La Figure I-d.2 montre le résultat de l'analyse de dépendance sur l'exemple.



(1) C-ORD

Classe X	D1(X)	D2(X)	Bd(X)
A	ø	ø	0
B	{A,C,D}	{A,B,C,D,E,F,G,H}	1
C	{D}	{D}	0
D	ø	ø	0
E	{A,B,C,D}	{A,B,C,D,E,F,G,H}	1
F	{C,D,G}	{C,D,G,H}	1
G	{C,D}	{C,D}	0
H	{C,D,G}	{C,D,G}	0

(2) D1(X), D2(X) et Bd(X)

Figure I-d Analyse des dépendances selon TOONS

A l'issue de cette analyse, on construit un ensemble d'étapes de test. Pour chaque classe X, on prévoit une étape ciblant les dépendances selon D_1, puis une autre étape prenant en compte toutes les dépendances additionnelles selon D_2 (dans le cas où $B_d(X)=1$). Notons que pour cette seconde étape, les dépendances dynamiques ajoutées à l'ORD peuvent avoir introduit des cycles (voir le cycle entre B et E dans l'exemple de C-ORD). Nous choisissons alors de ne pas couper ces cycles, et de construire une seule étape pour tester toutes les dépendances dynamiques au sein d'une composante fortement connexe (ex : une seule étape pour tester les

[3] Rational Rose est maintenant distribué par IBM : http://www-01.ibm.com/software/rational/uml/.

dépendances dynamiques de *B* et *E*). A ce stade, chaque étape de test *T* est représentée de façon interne comme un triplet *(T.goal, T.need, T.type)* où :

- *T.goal* est l'ensemble des classes ciblées par le test ;
- *T.need* est l'ensemble des classes nécessaires à la réalisation de l'étape *T* ;
- *T.type* indique si *T* teste des dépendances statiques (*T.type = Sta*) ou dynamiques (*T.type = Dyn*).

Par exemple, les deux étapes à prévoir pour le test de la classe *B* sont :

- *T1 = ({B}, {A,B,C,D}, Sta)* pour les dépendances statiques,
- *T2 = ({B,E}, {A,B,C,D,E,F,G,H}, Dyn)* pour les dépendances dynamiques au sein de la composante fortement connexe à laquelle *B* appartient.

I.2.2.2.Ordonnancement des étapes de test

Le traitement suivant consiste à ordonner les étapes de test ainsi obtenues. La Figure I-e montre la relation de précédence '<' entre deux triplets *T* et *T'*, et son application à l'exemple. Nous avons montré que la relation de précédence définit un ordre partiel, et qu'elle satisfait certaines propriétés souhaitées. Ainsi, le test ciblant les dépendances statiques (avec un *T.need* minimal) est toujours réalisé avant celui ciblant les dépendances dynamiques (avec un *T.need* maximal, puisqu'il faut prendre tous les types d'objets possibles). Par exemple, pour la classe *F* on a : *({F}, {C,D,F,G}, Sta) < ({F}, {C,D,F,G, H}, Dyn)*. Surtout, nous avons gardé le principe général de toujours tester une classe après les classes dont elle dépend :

- Toute linéarisation de l'ordre partiel des étapes de type *Sta* est un ordre topologique inversé de l'ORD acyclique.
- Lorsque l'on arrive à une étape de type *Dyn*, toutes les classes de *T.need* ont déjà fait l'objet d'une étape de type *Sta*. Pour les classes de *T.need* qui exhibent également des dépendances dynamiques :
    - Soit elles ont déjà fait l'objet d'une étape de type *Dyn*, par exemple étape *Dyn* de *F* avant l'étape *Dyn* de *B* et *E*,
    - Soit elles sont une des cibles de l'étape courante (ex : la dernière étape ciblant *B* et *E*).

$$T < T' \Leftrightarrow \begin{cases} T.type = T'.type \text{ et } T.need \subset T'.need \\ \text{ou} \\ \left(T.type = Sta \text{ et } T'.type = Dyn\right) \text{et } T.need \subseteq T'.need \end{cases}$$

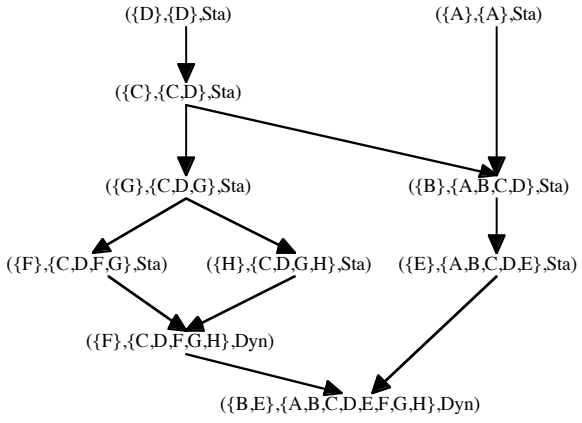


Figure I-e Ordre partiel des étapes de test (représentées sous forme de triplets)

I.2.2.3.Analyse du rôle des classes dans les différentes étapes

La représentation sous forme de triplet n'est pas celle qui est montrée à l'utilisateur. Un traitement est effectué pour réécrire le graphe avec une notation montrant le rôle des classes aux différentes étapes. Il est bien sûr impossible de représenter de façon compacte tous les rôles au sein de *T.need*. On peut cependant rendre certaines informations explicites, notamment quelles classes seront instanciées ou non, et quelles classes serveur (lien *As* ou *Ag*) introduisent du polymorphisme. La notation que nous avons définie est montrée dans la Figure I-f, ainsi que le résultat de la réécriture du graphe des étapes.

Par exemple, l'étape *({F}, {C,D,F,G,H}, Dyn)* devient *(C),(D),F#,G*,H* ce qui permet de voir que :

- *F* est la cible de cette étape de test.
- Il s'agit d'une étape de type *Dyn*, avec une classe serveur *G* introduisant du polymorphisme. Les liens vers G devront donc être testés en considérant tous les types possibles de serveur (instances de *G* et de ses descendantes).
- Il n'y aura pas d'instance des classes *C* et *D*, leur code est présent uniquement en tant que classe mère.

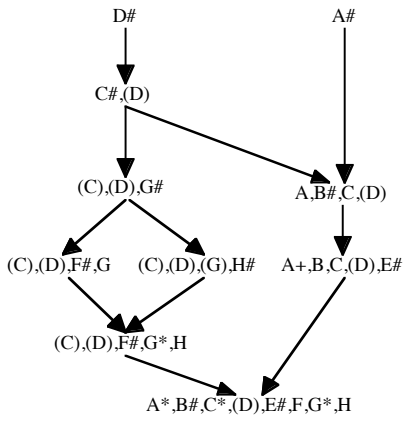
Soit *X* une classe :

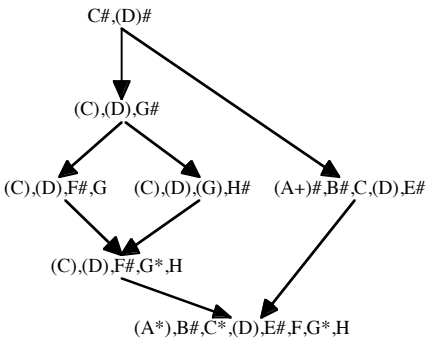
X# : *X* est une cible du test.

(X) : il n'y a pas d'instance de *X* à cette étape (*X* a le seul rôle de classe mère).

X* : *X* est à la fois classe mère et serveur, ce qui introduit du polymorphisme à tester (étape *Dyn*).

X+ : *X* est à la fois classe mère et serveur, mais on ne teste pas le polymorphisme du rôle serveur (étape *Sta*).





(1) Notation montrant le rôle des classes **(2) Graphe des étapes avec la notation** **(3) Modification du graphe pour A et D abstraites**

Figure I-f Ordre partiel des étapes de test (avec une notation montrant le rôle des classes)

I.2.2.4.Suppression d'étapes de test

On sait maintenant quelles sont les classes à instancier dans chaque étape. Or, si certaines classes sont des classes abstraites, les étapes requérant leur instanciation sont en fait infaisables. Il faut donc supprimer ces étapes, et reporter leur cible sur des étapes faisables.

Ce traitement est ici illustré sur l'exemple, en supposant *A* et *D* abstraites. La partie droite de la Figure I-f montre les modifications apportées au graphe des étapes. Trois étapes infaisables ont été supprimées : les étapes unitaires *A#* et *D#,* ainsi que l'étape *A,B#,C,(D)*. Trois étapes faisables sont modifiées :

- *C#,(D)* devient *C#,**(D)#*** puisque le test de *D* ne peut s'effectuer qu'avec le test de sa fille *C*.

- *A+,B,C,(D),E#* devient ***(A+)#,B#,C,(D),E#***. Il s'agit de la première étape où l'on peut tester *A* et *B*, d'où le marquage # pour ces classes. Notons que *A* avait initialement le double rôle de mère pour *E* et de serveur pour *B*. Comme *A* n'est pas instanciable, elle perd son rôle de serveur : *A+* devient *(A+)*. Le lien entre *A* et *B* doit maintenant être testé en prenant une fille de *A*, ici *E*.
- *A*,B#,C*,(D),E#,F,G*,H* devient ***(A*)***,*B#,C*,(D),E#,F,G*,H* puisque *A* perd son rôle de serveur.

I.2.3. Application de TOONS à une étude de cas

L'approche TOONS a été appliquée à une étude de cas R&D fournie par Airbus : l'*Automatic Dependent Surveillance Function* (ADSF). Il s'agit d'une application pour automatiser la surveillance des avions depuis le sol. L'avion reçoit des requêtes qu'il traite en transmettant au sol des données élaborées à partir des équipements embarqués. Le logiciel testé comporte 18 classes implémentées en C++. La Figure I-g montre le diagramme des classes issu de la conception. En coupant la dépendance de la classe *Chainon* vis-à-vis d'elle même, on obtient un ORD acyclique.

La classe *Evenement* est abstraite. En prenant en compte cette information, le prototype TOONS produit un graphe de test avec 23 étapes. L'analyse manuelle du graphe a conduit à la décision de supprimer encore quelques étapes. Par exemple, la classe *Liste* peut être vue comme un driver de *Chainon* : on décide de supprimer les étapes relatives à *Chainon*, et on reporte les cibles sur les étapes relatives à *Liste*. Au final, on garde 18 étapes de test, dont 7 de type *Dyn*. Notons que la complexité de cette étude de cas pour le test provient essentiellement du polymorphisme introduit au niveau de *Object* : dans la Figure I-g, toutes les classes de la partie droite (*Chainon*, *Liste*, *ListeId*, …) dépendent dynamiquement de toutes les classes de la partie gauche (arbre d'héritage de *Object*).

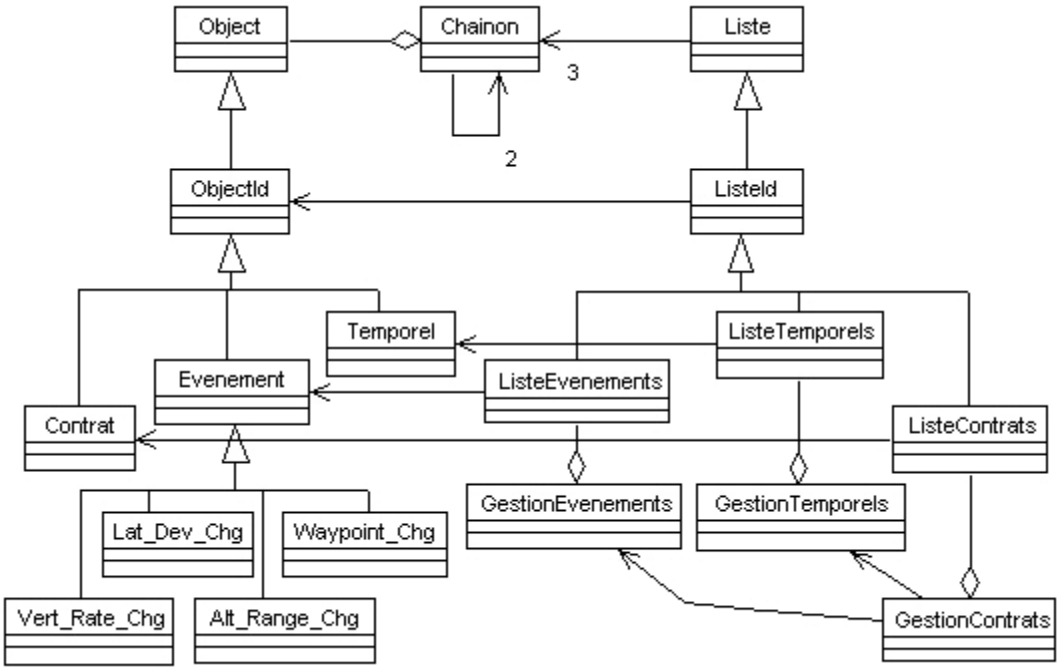


Figure I-g Architecture des classes de l'ADSF

Les 18 étapes de test ont été réalisées, permettant de révéler quelques fautes dans l'implémentation de l'ADSF (rappelons qu'il s'agit d'une étude R&D, et non d'un système en opération). Les critères de sélection de test sont décrits dans le manuscrit de Yvan Labiche [Lab 00]. Ils incluent :

- des critères de couverture structurelle, avec notamment des chemins de définition/utilisation d'attributs entre paires de méthodes ;
- des critères de couverture de diagrammes états-transitions pour les classes ayant fait l'objet d'une modélisation comportementale ;
- l'identification de cas d'instanciation pour tester les relations d'agrégation et d'association (ex : nombre d'instances associées, présence ou non d'aliasing, …),
- un critère de couverture du polymorphisme pour les étapes *Dyn*, en considérant l'ensemble des types possibles des objets agrégés ou associés.

Un point intéressant est que l'ordre des étapes a permis une approche incrémentale de test. Le test d'une classe fille après sa mère permet des stratégies basées sur les éléments redéfinis ou ajoutés dans la fille, avec réutilisation d'une partie des tests de la mère [HMF 92]. Le test des dépendances dynamiques réutilise les cas précédemment sélectionnés lors du test des dépendances statiques, en changeant simplement le type des objets agrégés ou associés. Cette approche incrémentale permet un gain de temps non seulement lors de la conception du test mais aussi lors de son implémentation, beaucoup de scripts de test pouvant être réutilisés tels quels ou avec de légères adaptations.

I.3. Test de modèles B

La méthode B appartient à la famille des méthodes formelles dites orientées-modèle, comme VDM [Jon 90] et Z [Spi 94]. La modélisation utilise la théorie des ensembles et la logique du premier ordre. Des étapes de raffinement, avec des obligations de preuves associées, permettent de passer graduellement d'un modèle abstrait à un modèle concret à partir duquel le code est généré. L'ouvrage de référence de la méthode B est le B Book [Abr 96]. Je me bornerai ici à faire quelques rappels nécessaires à la compréhension de nos travaux (§ I.3.1).

Nos travaux visaient à vérifier la prise en compte des besoins fonctionnels dans les modèles formels, ce qui ne peut être couvert par la preuve. Ils se démarquaient donc de travaux visant à tester un programme vis-à-vis de sa spécification formelle, et en particulier ceux relatifs au test à partir d'une spécification orientée-modèle VDM [DF 93], Z [Hie 97] ou B [VBL 97]. Dans notre cas, c'est le modèle B lui-même qui est testé. Par rapport aux autres travaux, on va retrouver des problématiques communes autour de la couverture structurelle de modèles, que je développerai en fin de cette partie (§I.3.4) [BW 99]. Mais il y a également des problématiques différentes liées à la définition de l'oracle de test, et à l'interprétation exécutable donnée aux modèles pour obtenir les résultats de test. Ces problématiques sont au cœur du cadre théorique que nous avons proposé pour formaliser le test de modèles B (§I.3.2 et §I.3.3) [WB 98].

I.3.1. Brève introduction à la méthode B

La brique de base d'un développement en B est la machine abstraite, qui encapsule un état et offre des opérations à ses utilisateurs. La Figure I-h montre la spécification d'une machine abstraite modélisant une pile. Le texte est structuré en clauses (MACHINE, CONSTRAINTS, SEES, …), et on distingue quatre parties :

- l'en-tête, avec le nom du composant et éventuellement des paramètres,
- Des clauses permettant d'établir des liens avec d'autres composants (ici, SEES),
- Une partie déclarative, qui définit l'état de la machine (avec notamment un invariant), au moyen de prédicats et de constructions ensemblistes,
- Une partie exécutive, qui définit l'initialisation et les opérations au moyen d'un langage basé sur la notion de substitution généralisée (une extension des substitutions simples *x := expr*).

La sémantique des substitutions généralisées est celle des transformateurs de prédicats de Dijkstra [Dij 76] : *[S] R* dénote la plus faible précondition pour que la substitution *S* établisse la postcondition *R*. Le Tableau I-A montre la sémantique d'un sous-ensemble des substitutions généralisées. La plupart de ces substitutions n'apparaissent pas directement dans le texte d'une machine abstraite : pour faciliter le développement, du sucre syntaxique est fourni. Ainsi, dans l'opération *PUSH* de la pile, la construction PRE *P* THEN *S* END correspond à une substitution avec précondition : *P* | *S*. D'autres exemples de sucre syntaxique sont :

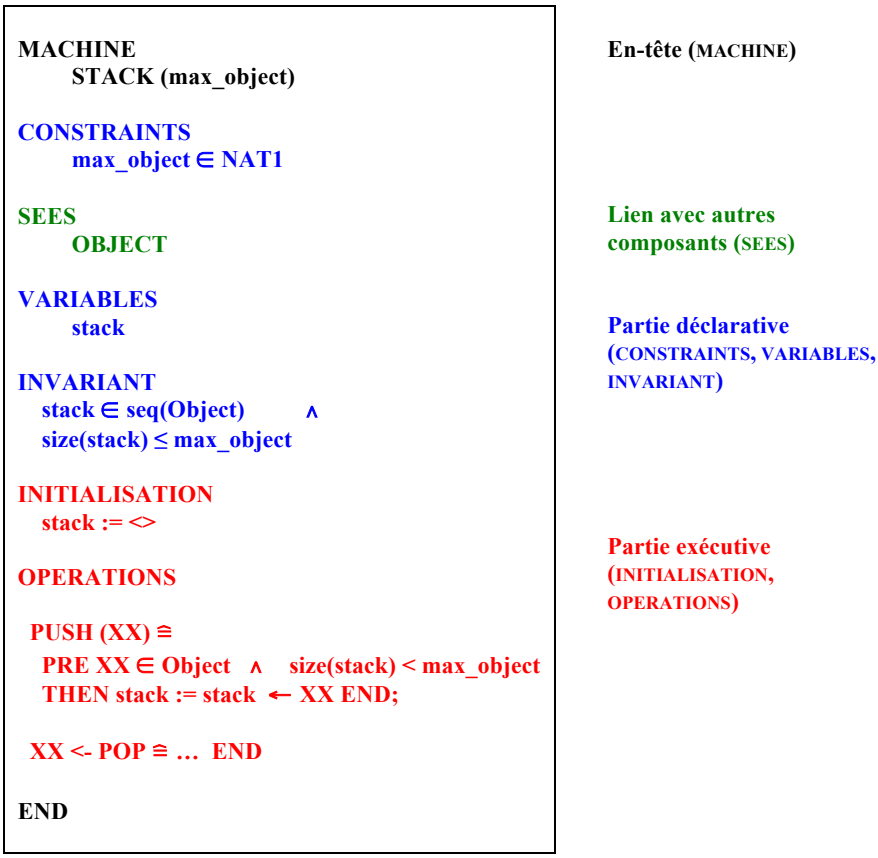


Figure I-h Spécification d'une pile en B

- IF *P* THEN *S1* ELSE *S2* END pour $(P \Longrightarrow S1) \square (\neg P \Longrightarrow S2)$
- VAR *x* in *S* END pour $@x \cdot S$
- $y :\in E$ pour $@x \cdot (x \in E \Longrightarrow y := x)$.

Au niveau d'un composant MACHINE tel que l'exemple de la pile, on a deux types d'obligations de preuve : (1) l'initialisation doit établir l'invariant, (2) chaque opération appelée dans sa précondition doit préserver l'invariant.

	Syntaxe	Sémantique
Substitution simple	$x := E$	$[x := E]\ R \Leftrightarrow$ remplacer toutes les occurrences libres de x dans R par E
Ne rien faire	skip	$[\text{skip}]\ R \Leftrightarrow R$
Substitution avec précondition	$P \mid S$	$[P \mid S]\ R \Leftrightarrow P \wedge [S]\ R$
Choix borné	$S \square T$	$[S \square T]\ R \Leftrightarrow [S]\ R \wedge [T]\ R$
Substitution gardée	$P \Longrightarrow S$	$[P \Longrightarrow S]\ R \Leftrightarrow P \Rightarrow [S]\ R$
Séquence	$S1 ; S2$	$[S1 ; S2]\ R \Leftrightarrow [S1]\ [S2]\ R$
Choix non borné	$@x \cdot S$	$[@x \cdot S]\ R \Leftrightarrow \forall x.[S]\ R$ où x n'est pas libre dans R

Tableau I-A Sous-ensemble des substitution généralisées (x est une variable, E une expression, R et P sont des prédicats, S et T des substitutions)

Partant d'un composant MACHINE racine, le développement en B s'effectue selon un processus incrémental. Ce processus repose sur deux mécanismes essentiels, qui sont le mécanisme de raffinement (lien REFINES) et le mécanisme de décomposition en couches logicielles (lien IMPORTS). Le raffinement permet de concrétiser les données et les opérations de la machine, jusqu'à arriver à une description qui puisse être automatiquement traduite dans un langage de programmation. On passe ainsi graduellement d'un composant MACHINE à un composant IMPLEMENTATION, avec des obligations de preuve de raffinement à chaque étape. La décomposition en couches se matérialise par un lien IMPORTS entre un composant IMPLEMENTATION de la couche supérieure et des composants MACHINE de la couche inférieure. Par exemple, dans la Figure I-i, l'implémentation de *Main* utilise les services de deux machines abstraites, *FCT1* et *FCT2*. Ces deux machines peuvent être raffinées indépendamment l'une de l'autre jusqu'à leur implémentation. On a ainsi décomposé un problème complexe (implémenter le comportement global de *Main*) en sous-problèmes plus simples (implémenter *Main* en supposant que les services *FCT1* et *FCT2* sont fournis, implémenter *FCT1*, implémenter *FCT2*). Les liens REFINES et IMPORTS forment l'ossature d'un développement en B selon une architecture arborescente. D'autres liens accessoires (INCLUDES, USES) permettent d'enrichir localement le texte d'un composant avec celui de composants auxiliaires (*A*, *B* et *C* dans la Figure I-i). Les liens SEES permettent des accès en lecture. Nous verrons ultérieurement que ces liens SEES, qui établissent des courts-circuits entre branches de développement indépendantes, posent certains problèmes (§I.3.3).

Conceptuellement, au fur et à mesure qu'on développe l'arbre des composants selon les liens REFINES et IMPORTS, on construit des modèles de *Main* de plus en plus "gros", jusqu'à arriver au modèle concret qui contient toutes les implémentations. Dans les applications industrielles que nous avons pu voir, il existe une étape intermédiaire où toutes les exigences fonctionnelles ont été formalisées, de sorte que les développements ultérieurs de l'arbre ne servent qu'à introduire des détails de programmation. Dans la Figure I-i, cette étape

intermédiaire pourrait par exemple correspondre au modèle 2, mais dans la réalité la capture des exigences nécessite typiquement plusieurs raffinements et décompositions en couches. La validation du modèle intermédiaire par rapport au cahier des charges s'effectue d'abord par relecture des textes formels des composants. Des tests fonctionnels sont ensuite réalisés sur le code généré à l'issue du développement complet. Nos travaux étudient comment introduire les tests au plus tôt, sur le modèle contenant toutes les exigences fonctionnelles.

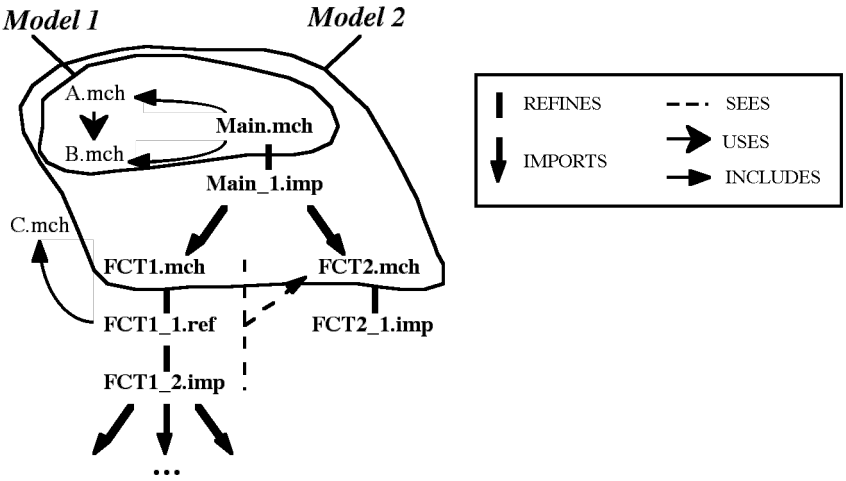


Figure I-i Architecture arborescente d'un développement en B

I.3.2. Test d'un composant MACHINE isolé

Avant d'aborder le test de modèles comportant plusieurs niveaux de raffinement et de décomposition en couches, considérons le cas simple d'un composant MACHINE isolé. Peut-on formaliser les séquences de test que l'on souhaite appliquer, et les résultats attendus, en termes de notions issues de la méthode B ? Notamment, peut-on établir un lien entre l'oracle de test et la notion de raffinement ? De la sorte, les preuves effectuées sous l'Atelier B[4] devraient garantir que tout raffinement du composant MACHINE préserve les propriétés testées. Ceci suppose cependant que les résultats calculés par animation de modèle soient corrects par rapport à la sémantique de transformation des prédicats. Les deux paragraphes suivants présentent respectivement le lien entre test et raffinement, et les conséquences que nous en tirons sur l'animation d'une machine abstraite.

I.3.2.1.Oracle de test et raffinement

Dans le B Book [Abr 96], le raffinement d'une machine abstraite est introduit en se référant à la notion de *substitution externe* implémentée sur cette machine. Une substitution externe contient une séquence d'appels aux opérations de la machine. Elle ne peut pas faire directement référence aux variables d'état de la machine : du fait du principe d'encapsulation, l'état est accessible uniquement via les opérations offertes par la machine. Dans l'exemple d'un composant *STACK* spécifiant une pile d'entiers, une substitution externe pourrait être :

```
T ≅   BEGIN
      PUSH (1) ;
      res ← POP
      END
```

[4] Outil support de la méthode utilisé dans l'industrie ferroviaire : http://www.atelierb.eu/

L'implémentation de *T* sur *STACK* consiste à initialiser la pile, puis à expanser les appels aux opérations en remplaçant les paramètres formels par les paramètres d'appel. Soit T_{STACK} la substitution résultante. Du fait de l'encapsulation, le comportement observable est en fait : $@\ stack\ .\ (stack \in Seq(Nat) \Longrightarrow T_{STACK})$ où '@' cache l'effet de la substitution sur l'état interne *stack* (voir le Tableau I-A). Seul l'effet sur *res* est observable. Du point de vue de la substitution externe *T*, on ne sait pas distinguer *STACK* d'une autre version *STACK'*, avec des variables d'état différentes pour encoder la pile, mais une interface *PUSH* et *POP* permettant d'établir les mêmes postconditions sur *res*. Deux machines abstraites sont observationnellement équivalentes s'il n'existe pas de substitution externe permettant de les distinguer.

La relation de raffinement entre machines abstraites peut également être définie en comparant les comportements observés depuis des substitutions externes. Ainsi, *STACK'* est un raffinement de *STACK* si et seulement si pour toute substitution externe *T* on a :

$$@\ stack\ .\ (stack \in Seq(Nat) \Longrightarrow T_{STACK}) \quad \sqsubseteq \quad @\ stack'\ .\ (stack' \in Type' \Longrightarrow T_{STACK'})$$

où $\sqsubseteq$ est la relation de raffinement algorithmique classique (affaiblissement de la précondition de terminaison, diminution de l'indéterminisme).

Cette définition n'est pas opératoire pour dériver des obligations de preuve, car on a une quantification sur toutes les substitutions externes possibles. En pratique, le B Book montre que l'on peut se ramener à des preuves de raffinement ne portant que sur les machines comparées. Cela est rendu possible en forçant le spécifieur B à exhiber une relation totale connectant les états concrets aux états abstraits. J'ai insisté ici sur la définition initiale du raffinement, du fait des liens évidents avec le test. Une séquence de test peut en effet être vue comme une substitution externe, qui doit établir des résultats attendus (postconditions) sur les sorties des opérations de la machine testée. En choisissant l'oracle de test pour qu'il compare les résultats attendus et obtenus selon une relation de raffinement, on peut entièrement formaliser le test en B. Cette idée avait été conjointement introduite par Jean-Louis Boulanger et moi-même, lors de nos travaux communs à l'INRETS. Le principe est montré en Figure I-j.

Soit M une machine à tester, de variable d'état $s \in Type_M$. Soit *T* une séquence de test pour *M*. On commence par définir un composant abstrait *Test_Driver* qui spécifie les résultats attendus pour *T* (d'après le cahier des charges) sous la forme d'une substitution $T_{attendu}$. Dans l'exemple précédent de la pile, on pourrait avoir $T_{attendu} \mathrel{\widehat{=}} res := 1$. Dans le cas général, la sémantique de $T_{attendu}$ détermine les postconditions à établir pour toutes les sorties observables $o_1, \ldots o_n$ de la séquence *T*, avec éventuellement de l'indéterminisme si plusieurs valeurs sont acceptables.

On définit ensuite un composant IMPLEMENTATION, *Test_Driver_1*, censé raffiner *Test_Driver*. L'importation de la machine à tester *M* permet d'effectuer l'implémentation de *T* sur *M*. L'oracle de test est défini pour accepter les résultats de test si et seulement si :

MACHINE Test_Driver OPERATIONS $o_1, ..., o_n \leftarrow$ test_sequence $\mathrel{\widehat{=}}$ $T_{attendu}$ END	IMPLEMENTATION Test_Driver_1 REFINES Test_Driver IMPORTS M OPERATIONS $o_1, ..., o_n \leftarrow$ test_sequence $\mathrel{\widehat{=}}$ T END

Figure I-j Formalisation du test en B

$$T_{attendu} \sqsubseteq @\, s \,.\, (s \in Type_M \Longrightarrow T_M)$$

Les preuves de *Test_Driver_1*, pour montrer qu'il raffine bien *Test_Driver*, vont justement garantir cette relation. De plus, par transitivité de la relation de raffinement, l'acceptation des résultats du test *T* sur *M* garantit l'acceptation sur tout raffinement prouvé de *M*.

On voit ici comment on peut envisager une approche de "test" entièrement basée sur des preuves réalisées sous l'Atelier B. Cette approche est cependant peu réaliste pour des modèles multi-composants tels que ceux que nous visons (I.3.3), pour lesquels les preuves seraient trop complexes. Nous allons maintenant discuter d'une approche plus classique pour le test, où les sorties du modèle sont calculées à l'aide d'un outil d'animation. Nous souhaitons conserver les bonnes propriétés de l'oracle de test, qui permettent de remonter la validation au niveau de la machine abstraite *M* en étant sûr que tout raffinement ultérieur restera conforme aux résultats attendus. Pour cela, il faut que la sémantique opérationnelle utilisée par l'animateur soit correcte vis-à-vis de la sémantique originelle de transformation des prédicats.

I.3.2.2.Interprétation exécutable correcte d'une machine abstraite

Les travaux de Breuer & Bowen [BB 94] ont proposé une formalisation de la correction des animations de modèles, dans le cadre d'une notation voisine de B (la notation Z). Il est possible de transférer ces travaux à B : ce paragraphe présente informellement les conséquences que nous en tirons sur l'animation de machines abstraites.

La notation B comportant des prédicats du premier ordre et des constructions ensemblistes, la non terminaison de l'animation doit être un comportement prévu. De plus, une spécification pouvant être indéterministe, il y a parfois plusieurs valeurs possibles pour une variable de sortie o_i. [BB 94] introduit la notion d'approximation correcte de cet ensemble de valeurs. Il y a 3 classes d'approximations : (1) résultat indéfini noté $\perp$, (2) résultat partiel noté $\{v_1, \ldots v_k\}_\perp$, c'est à dire que l'animateur produit un sous-ensemble de *k* valeurs possibles mais le calcul ne termine pas, (3) résultat exact, c'est-à-dire que le calcul termine et produit toutes les valeurs possibles. Prenons des exemples simples d'opérations avec une sortie o_1 :

- Opération indéterministe $o_1 :\in \{0,1\}$ (o_1 prend n'importe quelle valeur de l'ensemble). D'après [BB 94], les approximations correctes sont $\perp$, $\{0\}_\perp$, $\{1\}_\perp$, $\{0,1\}_\perp$, ou $\{0,1\}$.
- Opération PRE $i_1 \in dom(f)$ THEN $o_1 := f(i_1)$ END appelée en dehors de sa précondition. La seule approximation correcte est $\perp$.
- Opération miraculeuse $o_1 :\in \varnothing$. Cette opération est infaisable, les approximations correctes sont $\perp$ et $\{\}$ car il n'y a aucune valeur possible pour o_1.

Cette interprétation en termes d'ensembles de valeurs – complets ou partiels – nous permet de définir un oracle de test compatible avec la notion de raffinement. L'oracle est basé sur l'inclusion entre ensembles calculé et attendu, en prenant en compte le caractère éventuellement partiel des calculs. Par exemple, supposons qu'on attende un ensemble de valeurs possibles $\{0, 1\}$ pour o_1. Pour différents résultats calculés en appliquant la séquence de test, on aura les verdicts suivants :

$\{0\}$ $\{1\}$ $\{0, 1\}$	L'oracle de test accepte n'importe lequel de ces résultats.
$\{\}$	Détection d'une spécification B miraculeuse.
$\{2\}$ $\{2\}_\perp$ $\{0, 2\}$ $\{0, 2\}_\perp$	L'oracle de test rejette ces résultats.
$\perp$ $\{0\}_\perp$ $\{0, 1\}_\perp$	Rejet si on détecte la violation d'une précondition, test non concluant sinon.

Le point important est qu'un verdict d'acceptation ne peut jamais être émis si le résultat de l'animation est partiel. Ainsi, $\{0\}_\perp$ est traité différemment de $\{0\}$, car on doit considérer la possibilité que la machine abstraite permette une autre valeur de sortie incorrecte, qui serait choisie lors du processus de raffinement en B. Cela induit des exigences fortes sur le traitement de l'indéterminisme par l'animateur de modèle : d'une part il faut pouvoir distinguer les résultats complets et partiels (pour avoir la correction de l'interprétation), et d'autre part il faudrait idéalement produire un résultat complet (pour avoir un test concluant).

L'animateur fourni dans l'Atelier B ne répond pas à ces besoins. C'est un outil interactif qui utilise des règles de réécriture pour évaluer les prédicats et les expressions de la théorie des ensembles, souvent avec l'aide de l'utilisateur. L'indéterminisme est toujours levé en demandant à l'utilisateur de rentrer une valeur. Un tel outil ne peut pas garantir la préservation des résultats attendus après raffinement de la machine testée. De façon générale, l'Atelier B met l'accent sur l'outillage de la preuve, plutôt que sur la validation par rapport aux exigences. D'autres outils à destination industrielle, comme VDMTools [FLS 08] pour le support de VDM, correspondent à des choix stratégiques différents axés sur le prototypage et la validation. Il est tout à fait envisageable de renforcer l'Atelier B pour permettre la validation de machines abstraites, en s'inspirant d'outils académiques pour exécuter les spécifications orientées modèles (par exemple, [WLB 00] et [Utt 01] semblent intéressants).

I.3.3. Test de modèles multi-composants

Le cadre théorique étant posé pour le test d'un composant MACHINE, considérons maintenant les modèles multi-composants. Comme indiqué dans l'introduction à la méthode B (§I.3.1), la formalisation des exigences nécessite typiquement plusieurs raffinements et décompositions en couches. A titre d'exemple, la Figure I-k montre l'architecture d'un sous-système d'un logiciel ATP (*Automatic Train Protection*) développé par Alstom Transport, et mis en opération sur les lignes A et B du métro de Lyon. L'ensemble des composants en grisé sur la figure correspond à la formalisation des exigences. La machine abstraite résultante peut être construite en combinant les textes formels des différents composants, pour produire un texte formel unique, dans lequel les liens architecturaux (REFINES, IMPORTS, …) ont été éliminés par mise à plat. La thèse de Salimeh Behnia définit un tel algorithme de mise à plat de modèles, dont un prototype a été réalisé [Beh 00].

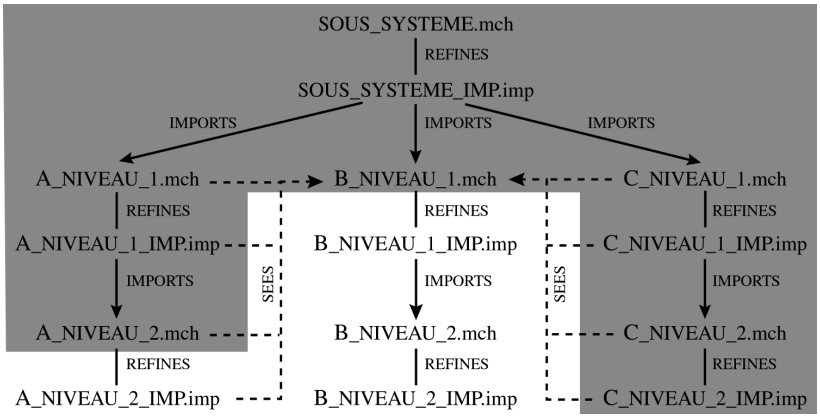


Figure I-k Architecture du développement B d'un sous-système du MPL75

La notion de mise à plat de modèle permet de raisonner formellement sur le comportement observable induit par une architecture de composants. Ainsi, nous pouvons reprendre le cadre théorique de test défini pour un composant MACHINE isolé : une séquence de test est une substitution externe implémentée sur la machine abstraite aplatie, et l'oracle est choisi comme précédemment. De la sorte, en supposant que l'animateur utilisé interprète correctement les constructions de la notation, les obligations de preuve B devraient garantir la préservation des propriétés testées. Etant donnée une séquence de test, l'acceptation des résultats fournis par un modèle intermédiaire aplati implique l'acceptation des résultats du modèle final (c'est-à-dire le modèle correspondant à l'architecture complète, à partir duquel le code est généré).

Mais au fait, est-ce garanti par les obligations de preuve ? Malheureusement non, à cause des liens SEES entre sous-arbres. Les travaux de Rouzaud [Rou 99] ont montré la dangerosité de ces liens. Il est possible d'exhiber des architectures complètement prouvées telles que le modèle final viole la spécification de la machine racine. Ce problème n'avait pas été identifié dans le B Book et a conduit Rouzaud à proposer une condition architecturale restreignant l'utilisation des SEES. Nous avons alors été amenés à examiner l'impact de ces liens SEES sur notre cadre théorique de test. Par exemple, l'architecture montrée en Figure I-l respecte la condition architecturale de [Rou 99] : on est sûr que le modèle final raffine le modèle racine *A*. Mais est-ce que le modèle final raffine n'importe lequel des modèles intermédiaires M_1 et M_2 ? Si ce n'est pas le cas, certains modèles pourraient être inadéquats pour le test.

Notre analyse est que M_1 est bien une abstraction du modèle final, mais que M_2 ne l'est pas. Intuitivement, cela est dû au lien SEES entre B_I_1 et B_2. Par définition, ce lien ne permet que des accès en lecture à l'état de B_2. Mais les accès à B_2 peuvent indirectement induire des accès en écriture à B_3, pourvu que les variables modifiées soient indépendantes de l'état abstrait de B_2. Ainsi, dans le modèle final, certaines variables de B_3 peuvent être modifiées par l'implémentation des opérations de B_1. Dans le modèle intermédiaire M_2, ces variables sont non modifiables par B_1 : les résultats du test de M_2 ne seront pas représentatifs – au sens de la relation de raffinement – des résultats obtenus par test du modèle final (ou par test du code généré à partir de ce modèle).

Nos travaux pour identifier les modèles intermédiaires adéquats pour le test sont décrits en détail dans le manuscrit de Salimeh Behnia [Beh 00]. Ils nous ont conduit à préconiser des restrictions supplémentaires sur l'utilisation des liens SEES. Notons que le développement du logiciel ATP, dont un sous-système est montré en Figure I-k, respecte ces restrictions. La pratique industrielle s'est ici avérée compatible avec notre cadre théorique de test.

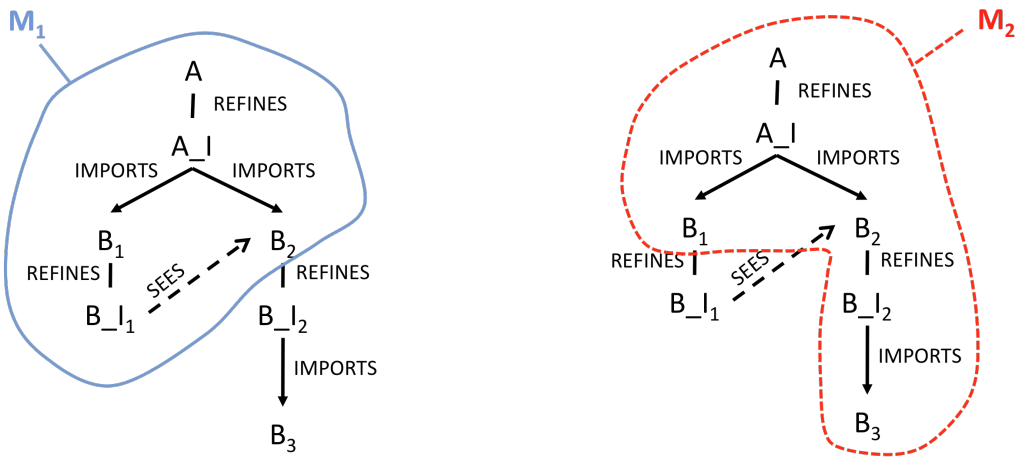


Figure I-l Modèles intermédiaires dans une architecture de développement

I.3.4. Couverture structurelle de modèles B

Supposons que des séquences de test soit sélectionnées à partir des exigences fonctionnelles du cahier des charges. Peut-on analyser la couverture structurelle du modèle sous test ? Le modèle sous test résulte de la mise à plat d'un ensemble de composants, avec plusieurs niveaux de raffinement et de décomposition en couche. Le texte formel correspondant mélange des constructions abstraites et concrètes de la notation. Nous avons besoin d'une notion unifiée de couverture pour toutes ces constructions.

Notre contribution s'est concentrée sur la couverture structurelle des opérations. Les critères existant dans la littérature peuvent être classés en deux familles :

- A un niveau abstrait, les critères structurels définis pour les spécifications orientées-modèles (Z, VDM, B) couvrent la structure propositionnelle des prédicats avant/après des opérations [DF 93] [HP 95] [Hie 97] [VBL 97].
- A un niveau concret, les critères structurels définis pour les programmes procéduraux couvrent le graphe de contrôle construit à partir du code source [Nta 88].

Pour les opérations B, la première famille de critères est en fait définie quel que soit le niveau d'abstraction. Comme indiqué dans le B Book, on sait dériver le prédicat avant/après $prd_X(S)$ d'une substitution S portant sur la liste X de variables[5]. Au sein d'une machine abstraite, une opération est une substitution portant sur les variables de sortie et les variables d'état. Le prédicat relie leur valeur après X' aux entrées de l'opération et à l'état avant de la machine. Une fois le prédicat avant/après obtenu, différentes stratégies de décomposition de sa structure propositionnelle peuvent être envisagées, la stratégie la plus sévère étant celle proposée par Dick & faivre [DF 93]. Pour extraire le sous-domaine d'entrée correspondant à chaque cas de la décomposition, on effectue ensuite une quantification existentielle sur les sorties et l'état après. Dans le B Book, $\exists X' . prd_X(S)$ est noté $fis(S)$.

La deuxième famille de critères ne semble *a priori* applicable qu'aux composants IMPLEMENTATION. Le sous-ensemble de la notation B utilisable à ce niveau est appelé B0. Les substitutions permises sont semblables à des instructions d'un langage de programmation : on sait alors construire le graphe de contrôle d'une opération. Chaque appel de l'opération va activer un chemin d'exécution dans ce graphe, et on a différents critères de couverture associés aux parcours du graphe.

Nos travaux ont réalisé une unification des deux familles de critères pour les opérations B. Cela a notamment nécessité de ré-interpréter la notion de chemin d'exécution, pour qu'elle s'applique aux substitutions généralisées. Avec cette interprétation, le critère de tous les chemins (*All-paths*) peut être vu comme une certaine décomposition du prédicat avant/après. Cette décomposition est bien moins sévère que celle proposée par Dick & Faivre, et nous avons proposé quelques critères intermédiaires entre *All-paths* et [DF 93]. La partie droite de la Figure I-m illustre cette hiérarchie de critères. Tout en haut de la hiérarchie, nous avons [DF 93]. Tout en bas, les critères usuels de couverture du graphe de contrôle. Les critères médians font le lien entre les deux, en introduisant graduellement des décompositions basées soit sur les opérateurs du langage des substitutions, soit sur les opérateurs propositionnels des prédicats de garde.

[5] Dans nos travaux comme dans ceux sur la couverture prédicats avant/après, les boucles et définitions récursives sont dépliées un nombre borné de fois.

Réécriture de l'opération sous la forme $\square_i S_i$.

All paths. Les cas à couvrir sont $fis(S_i)$.

All path combinations. Les cas à couvrir sont toutes les combinaisons de chemins faisables. Ex pour 2 chemins :

$$fis(S_1) \wedge \neg fis(S_2)$$
$$\neg fis(S_1) \wedge fis(S_2)$$
$$fis(S_1) \wedge fis(S_2)$$

All extended path combinations. Dans les combinaisons précédentes, $\neg fis(S_i)$ est décomposé sur la structure de S_i.
$\neg fis(P \Longrightarrow S_{i1})$ donne 2 sous-cas $\neg P$ et $P \wedge \neg fis(S_2)$.
$\neg fis(S_{i1} \parallel S_{i2})$ donne 3 sous-cas $\neg fis(S_{i1}) \wedge fis(S_{i2})$
$fis(S_{i1}) \wedge \neg fis(S_{i2})$
$\neg fis(S_{i1}) \wedge \neg fis(S_{i2})$

Guard coverage. Tous les critères précédents peuvent être raffinés en décomposant les prédicats des gardes.

$A \vee B$	donne	A	et	$\neg A \wedge B$
$\neg(A \wedge B)$	donne	$\neg A$	et	$A \wedge \neg B$
$A \Rightarrow B$	donne	$\neg A$	et	$A \wedge B$

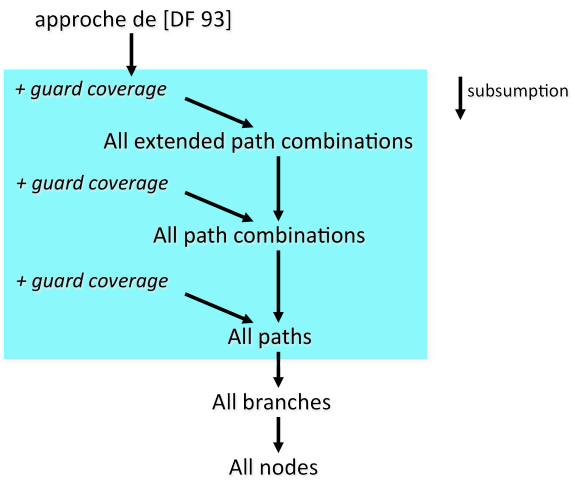


Figure I-m Critères de couverture structurelle d'une opération

Pour introduire nos critères de couverture, commençons par ré-interpréter la notion de chemin. Toute substitution S peut être mise sous la forme $\square_i S_i$ en utilisant des règles de réécriture qui "remontent" les opérateurs de choix bornés $\square$. Des exemples de règles sont :

$$(S_a \square S_b) ; S_c \quad \rightarrow \quad (S_a ; S_c) \square (S_b ; S_c)$$
$$@x \,.\, (S_a \square S_b) \quad \rightarrow \quad (@x \,.\, S_a) \square (@x \,.\, S_b)$$

Dans le cadre du B0, les S_i ainsi obtenus correspondent aux chemins du graphe de contrôle, et la condition d'activation d'un chemin est $fis(S_i)$.
On a : $fis(S) \Leftrightarrow fis(\square_i S_i) \Leftrightarrow \vee_i fis(S_i)$. Pour le sous-ensemble B0 de la notation, la couverture de *All-paths* revient donc à décomposer le prédicat $fis(S)$ en une disjonction de cas selon les opérateurs de choix bornés.

Nous étendons cette approche à l'ensemble de la notation (cf. *All paths* dans la Figure I-m). Un "chemin" S_i n'est alors plus interprétable comme une séquence de nœuds et d'arcs dans un graphe. On a notamment des substitutions parallèles, avec la règle de réécriture :

$$S_a \parallel (S_b \square S_c) \quad \rightarrow \quad (S_a \parallel S_b) \square (S_a \parallel S_c)$$

Cette règle induit deux "chemins" S_i qui contiennent des composantes parallèles. Au niveau du graphe de contrôle, nous avons maintenant deux types arcs : les arcs de branchement classiques, et les arcs correspondant à la division et au regroupement de flots parallèles. La propriété $fis(S_a \parallel S_b) \Leftrightarrow fis(S_a) \wedge fis(S_b)$ montre que la condition d'activation d'un "chemin" avec du parallélisme n'introduit pas de disjonction : soit toutes les composantes s'exécutent, soit aucune ne s'exécute.

Une autre conséquence de l'extension au-delà du B0 est que $\vee_i fis(S_i)$ ne réalise plus une partition du domaine d'entrée de l'opération. Par exemple, pour la substitution indéterministe : $(i < 10 \Longrightarrow o := i+1) \square (o := 10)$, l'entrée $i = 0$ active les *deux* chemins et donne un ensemble de valeurs possibles $\{1, 10\}$ pour o. On peut alors définir des critères plus sévères que All paths, en considérant les combinaisons de chemins actifs (voir la Figure I-m).

Par rapport à [DF 93], nos décompositions ignorent certains cas qui pourraient générer des prédicats dépourvus de sens (voir [BBM 98] pour la bonne définition du B, et [BW 99] pour les conséquences sur nos critères). Et surtout, [DF 93] est plus sévère car il ajoute des décompositions prenant en compte le domaine de sortie. Dans le petit exemple précédent, le

sous-domaine d'entrée *i<10* serait décomposé en *i<9* et *i=9*, pour distinguer les cas où la sortie est différente ou égale dans les deux composantes du choix borné.

Notons qu'en pratique, le choix d'un critère va être fortement contraint par la combinatoire. A titre d'exemple, le sous-système montré en Figure I-k offre une opération dont la mise à plat donne 24 000 chemins syntaxiques. Pour cette opération, on devra se contenter d'un critère peu exigeant comme la couverture des branches (48 branches).

I.4. Conclusion et discussion

Les trois parties de ce chapitre correspondent à des objectifs de test différents. Dans la première partie, l'accent est mis sur la robustesse d'objets destinés à servir les requêtes de clients concurrents. Dans la deuxième, on s'intéresse au test d'intégration inter-classes, et à la définition d'un ordre de test qui facilite la définition d'une stratégie incrémentale. Dans la troisième, il s'agit de tester des modèles formels pour révéler des fautes introduites lors de la capture des exigences fonctionnelles. Un point commun est que nous avons dû considérer les spécificités des technologies sous-jacentes. Ainsi, nous avons notamment considéré les liens architecturaux présents dans les différents types de développement : relations clients/serveurs, agrégation, héritage pour la technologie objet ; raffinement, décomposition en couches et liens SEES pour la méthode B. L'oracle de test prend en compte les relations de conformité adaptées à chaque cas : conformité vis-à-vis de modèles comportementaux issus des dossiers de conception OO ; relation de raffinement algorithmique entre une substitution $T_{attendu}$ et une substitution T implémentée sur la machine abstraite à tester, pour un développement en B. Enfin, les critères de couverture exploitent classiquement les modèles disponibles pour chaque technologie.

Le test de robustesse d'objets est basé sur une interprétation des contrats telle qu'une requête soit mise en attente jusqu'à ce que sa précondition devienne vraie [Mey 93] [NMO 09]. Ce n'est pas la seule interprétation possible. Notamment, pour des systèmes temps-réels durs, la mise en attente n'est pas une solution acceptable et il est préférable de rejeter la requête avec un message d'erreur. De plus, nous ne considérons qu'une notion limitée de robustesse : il s'agit de vérifier que l'objet sait traiter des requêtes arrivant dans un ordre et à une fréquence arbitraires. D'autres propriétés de robustesse sont évidemment envisageables, par exemple la non-dégradation des capacités d'un système sur de longues exécutions [AW 95], ou la tolérance aux fautes survenant dans l'environnement de l'objet[6]. Dans le cadre des travaux présentés ici, il faut souligner que nous ne cherchions pas à répondre à des problèmes de tolérance aux fautes. Les tests effectués sous les différents profils de charge visaient à permettre la réutilisation d'un objet serveur au sein de contextes applicatifs variés.

La définition d'un ordre de test pour les logiciels orientés objet est restée un sujet d'actualité plusieurs années après la fin des travaux présentés en deuxième partie de ce chapitre. Les contributions nouvelles se sont principalement intéressées à la coupure de cycle dans le graphe des dépendances. Selon les approches, le choix des dépendances à couper est basé sur la connectivité du graphe [BLW 03, MCL 03], sur des métriques de couplage entre classes [BFL 02] [ML 05] [HSR 05], voire sur des analyses de dépendance sémantique entre méthodes [ZR 07]. Dans tous les cas, l'information prise en compte vise à mieux appréhender l'effort nécessaire pour le développement des stubs, de façon à permettre le choix des coupures les moins coûteuses.

---

[6] Pour une discussion générale du test de robustesse, voir le rapport [WAM+ 06], issu de réflexions menées au sein d'une Action Spécifique du CNRS (co-animée avec Richard Castanet, http://www.laas.fr/TSF/AS23/), puis au sein des projets européens ASSERT (http://www.assert-project.net/) et ReSIST (http://www.resist-noe.org/).

Une technologie de développement complètement formelle telle que la méthode B pose la question du rôle dévolu au test. Un rôle majeur est de permettre une validation de la prise en compte des exigences fonctionnelles, en adoptant un point de vue complémentaire à celui de la formalisation. Celle-ci induit un point de vue axé sur des propriétés (invariants, pre- et postconditions des opérations), alors que le test adopte un point de vue axé sur des cas d'utilisation (séquence de sorties attendues pour des séquences d'entrée données). De plus, la formalisation introduit les propriétés graduellement lors des raffinements et décompositions en couches, alors que le test donne une vue transversale et "à plat" du comportement modélisé. Nos travaux ont exploré la possibilité d'introduire la validation fonctionnelle dans des phases amont, par test de modèle. Nous avons alors identifié ce que serait la cible du test (des modèles multi-composants, satisfaisant certaines conditions architecturales) et proposé une définition de l'oracle de test en liaison avec la relation de raffinement. Nous avons également proposé des critères de couverture structurelle de modèles B, avec une unification de deux familles de critères, basés sur la structure propositionnelle de prédicats avant/après et basés sur le graphe de contrôle. Des travaux postérieurs aux nôtres ont complété ce type de critères avec des stratégies de couverture des valeurs aux limites [LPU 04].

Le test d'un modèle B suppose que l'on puisse lui donner une interprétation exécutable. Certains auteurs considèrent qu'une spécification n'a pas à être exécutable, car cela conduit à restreindre l'expressivité du langage utilisé et à avoir un style de spécification trop proche de l'implémentation [HJ 89]. D'autres [Fuc 92] soutiennent au contraire qu'il est important de pouvoir valider la spécification, et qu'en pratique l'expressivité reste suffisante pour permettre un style de spécification abstrait. Clairement, j'adhère à cette dernière opinion. On peut faire le constat qu'il existe de nombreux travaux sur l'exécution de spécifications orientées-modèle, avec notamment des outils qui prennent en compte l'indéterminisme [BB 94] [WLB 00] [Utt 01] [BLL+ 08]. Notre cadre théorique de test trouvera peut-être un jour des applications industrielles.

Chapitre II. Test en support à la vérification formelle (et vice-versa)

Le test et la vérification formelle sont deux approches pour réduire la présence de fautes dans un système. Le test est une méthode de vérification partielle, qui implique l'exécution du système en lui fournissant des entrées valuées. La vérification formelle permet une analyse exhaustive, mais elle est réalisée statiquement sur une abstraction du système et cible certaines propriétés. Les deux approches sont généralement considérées comme complémentaires plutôt que rivales.

Si on regarde plus en détail les contributions sur le test, on voit apparaître de nombreuses relations avec la vérification formelle. En particulier, l'article [HBB+ 09] présente un état de l'art sur le test à partir de spécifications formelles. De telles spécifications peuvent être utilisées pour formaliser l'oracle de test, ou pour servir de base à la génération de cas de test. L'état de l'art montre que la génération automatique de tests exploite toute une palette de techniques issues de la vérification formelle : réécriture, slicing, analyse d'accessibilité dans un graphe d'états, résolution de contraintes, … Certains travaux utilisent les services d'outils avancés de vérification tels que des démonstrateurs de théorèmes [Cuk 97] [BCM 00], des model checkers [ABM 98] [GH 99] [HMR 04] [FG 09], ou les deux [CR 02].

Il y a comparativement moins de travaux sur l'utilisation du test en support à la vérification formelle, mais on peut néanmoins citer quelques exemples. [HSS 02] propose un outil pour tester les preuves en tant que programmes fonctionnels, ce qui est possible dans des logiques constructives. Le but du test est de révéler des fautes dans les preuves en cours de développement (preuve fausse) ou même dans les preuves achevées (formalisation erronée). [Rus 02] s'intéresse à la vérification de systèmes de transitions symboliques (*IOSTS*), et utilise des techniques de synthèse de test pour faciliter la preuve : des objectifs de test sont spécifiés pour extraire les sous-graphes de l'*IOSTS* relatifs aux propriétés à prouver. D'autres travaux considèrent l'exécution de tests comme un moyen d'acquérir des informations sur un système, et d'en construire une abstraction pour la vérification formelle. [GPY 02] et [Pel 03] exploitent les observations sous test pour construire un modèle à états vérifié par model checking. Les travaux de [WE 02] généralisent les résultats de test sous forme d'invariants que l'on introduit ensuite comme lemmes intermédiaires dans des preuves de théorèmes plus complexes.

Tous ces travaux montrent qu'on peut aller au-delà du constat général d'une complémentarité pour l'élimination des fautes, et considérer des couplages étroits au niveau des techniques mises en œuvre. Ce chapitre présente deux contributions s'inscrivant dans cette perspective. La première étudie un couplage test et preuve de théorème pour la vérification d'algorithmes de tolérance aux fautes (§II.1). La deuxième associe le model checking à des techniques issues du test, dans le but d'analyser des contrexemples (§II.2).

Les travaux sur le couplage test et preuve ont été menés au début des années 2000. Ils ont fait l'objet de la thèse de Guillaume Lussier [Lus 04]. Dans ces travaux, le test est vu comme un moyen pragmatique de compléter une preuve existante, lorsque celle-ci est sujette à caution (démonstration "papier" informelle), ou n'a pu aboutir (preuve formelle partielle comportant des branches non prouvées). L'idée est d'étudier comment guider la conception du test pour cibler les lacunes de la preuve. Une approche de *test guidé par la preuve* a été proposée et expérimentée sur plusieurs algorithmes de tolérance aux fautes.

Les travaux sur l'analyse de contrexemples ont été menés dans la deuxième moitié des années 2000. Ils ont fait l'objet de la thèse de Thomas Bochot [Boc 09], co-encadrée avec Virginie Wiels (ONERA), au sein d'une collaboration avec Airbus. Les travaux portaient sur l'utilisation du model checking au niveau de la conception détaillée des logiciels de commande de vol. Nous avons mis l'accent sur le problème de l'exploitation des contre-exemples obtenus en phase de mise au point des modèles. Il s'agit d'une part de faciliter la compréhension des causes de la violation d'une propriété, et d'autre part de permettre la génération de plusieurs contrexemples, correspondant à différents scénarios de violation. L'approche proposée repose sur l'identification de chemins activés par un contrexemple, et présente des liens étroits avec des travaux sur le test structurel. Un prototype a été réalisé et utilisé dans une étude de cas fournie par Airbus.

II.1. Test guidé par la preuve

Ces travaux visaient la vérification d'algorithmes de tolérance aux fautes, qui constituent des briques de base pour construire des architectures sûres de fonctionnement ; leur correction doit donc être établie de manière rigoureuse. La grande majorité des preuves développées dans ce but sont des preuves informelles, c'est-à-dire des démonstrations données en langage naturel et basées sur le raisonnement humain. On peut alors s'interroger sur la confiance que l'on peut leur accorder. On a malheureusement plusieurs exemples de "preuves" d'algorithmes de tolérance aux fautes qui se sont avérés incorrects après publication. Une preuve formelle, dont le développement est assisté par un démonstrateur de théorèmes, offre une confiance bien supérieure. En contrepartie, l'effort de formalisation et de développement de la preuve est très important, sans garantie d'aboutissement.

Pour pallier ces problèmes, notre proposition a été d'utiliser le test pour compléter une preuve existante, lorsque celle-ci présente des lacunes. L'hypothèse sous-jacente est que les éventuels cas de défaillance de l'algorithme devraient être liés aux lacunes de la preuve, d'où le principe d'un test guidé par la preuve. La Figure II-a illustre ce principe. En se basant sur la structure logique de la preuve (visualisée ici par un arbre de preuve schématique), et connaissant les parties faibles dans cette structure (une branche de l'arbre non prouvée dans la figure), on devrait pouvoir guider la conception du test. La thèse de Guillaume Lussier [Lus 04] présentait un caractère exploratoire : il s'agissait d'étudier comment mettre en œuvre le principe proposé, et d'évaluer sa pertinence pour révéler des fautes de conception.

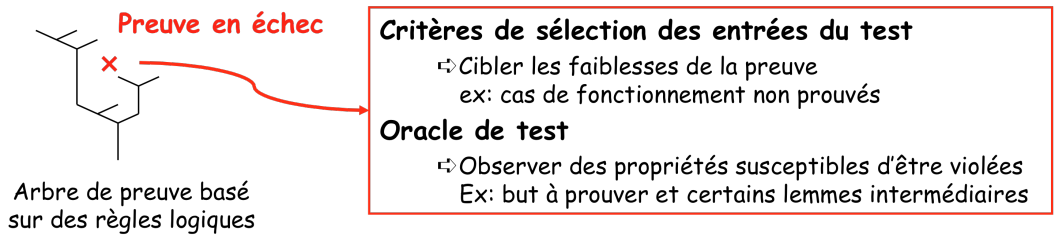


Figure II-a Principe du test guidé par la preuve

En pratique, il faut distinguer le cas des preuves informelles (§II.1.1) et celui des preuves formelles partielles (§II.1.2). Dans le premier cas, les informations utiles au test doivent être extraites d'un discours informel. Nous avons défini une méthode d'analyse pour dégager l'articulation logique d'une démonstration, et pointer sur les parties douteuses. Dans le deuxième cas, tout est formalisé et on peut savoir précisément quelles sont les parties non prouvées. Une question est néanmoins celle du niveau de détail des informations à prendre en compte. En particulier, nous verrons qu'à un niveau de détail trop fin il peut être très problématique d'établir un lien entre une branche non prouvée et des cas de fonctionnement de l'algorithme.

II.1.1. Preuves informelles

Les paragraphes suivants décrivent successivement l'approche de test proposée, puis son évaluation expérimentale sur deux exemples d'algorithmes de tolérance aux fautes incorrects, dont des démonstrations (erronées) sont disponibles dans la littérature.

II.1.1.1. Approche de test proposée

L'approche proposée comporte quatre étapes.

Etape 1 – Analyse Préliminaire. Cette étape démarre par une lecture haut niveau de l'algorithme, ses hypothèses, et les propriétés à assurer. La compréhension acquise doit être suffisante pour permettre la réalisation d'un environnement de test. Un prototype de l'algorithme est implémenté, ainsi qu'un oracle de test permettant d'observer la violation des propriétés requises. A partir des hypothèses, on donne une définition opératoire du domaine d'entrée de test, par exemple sous la forme d'une fonction de génération aléatoire qui engendre ce domaine. A l'issue de cette première étape, des premières expériences de test aléatoire peuvent être réalisées et révéler des fautes à faible coût.

Etape 2 – Restructuration de la Preuve Informelle. Cette étape est au cœur de notre approche. Elle consiste à restructurer le discours informel sous forme d'un arbre de preuve semi-formel basé sur la déduction naturelle ou le calcul des séquents. La restructuration est plus légère qu'une formalisation complète du problème, et ne peut permettre de conclure à la correction de l'algorithme. Néanmoins, elle offre une représentation précise de l'articulation logique de la démonstration, ainsi qu'un support pour en guider l'analyse pas à pas. Elle permet une évaluation rapide de la rigueur de la preuve et l'identification de ses faiblesses.

Prenons un exemple simple extrait d'un des algorithmes que nous avons étudiés. Une partie de la démonstration porte sur une propriété (Théorème 3) à satisfaire sous des hypothèses (Γ) non détaillées ici. Un fragment du discours informel est reproduit ci-dessous :

> "If a processor *p* becomes send-faulty, … Similarly, if *p* just became receive-faulty in the expected broadcast before its slot, … [*argument pour ces deux cas*]
> If a processor *p* becomes receive-faulty in its transition to the next step, but *p* is not the next expected broadcaster, … [*argument pour ce dernier cas*]."

On reconnaît la forme d'une preuve par cas, ce qui s'exprime en calcul des séquents par l'application successive de deux règles d'inférence, cut et $\vee\vdash$. La Figure II-b montre la restructuration sous forme d'un arbre de racine $\Gamma \vdash$ *Théorème 3*. Notre formulation des séquents utilise des notations synthétiques pour les éléments du discours informel (par exemple, "a processor *p* becomes send-faulty" est noté *SFault*). La preuve est décomposée en trois branches : preuve du Théorème 3 dans les deux premiers cas (Branche 1), preuve dans le dernier cas (Branche 2), preuve de la complétude de la décomposition en cas. Cette dernière branche était omise dans le discours informel : les auteurs l'ont probablement considérée

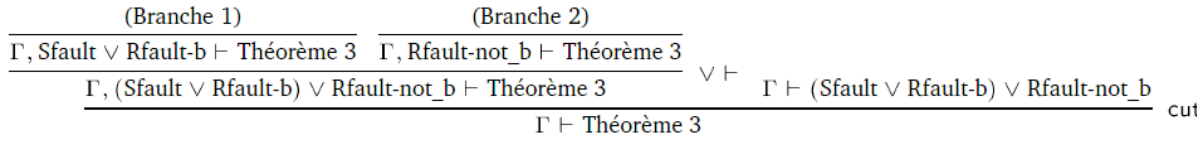


Figure II-b Exemple de preuve par cas en calcul des séquents

comme triviale. En interprétant les étapes de raisonnement en termes d'étapes d'inférence d'un système de déduction, nous avons dû l'expliciter.

La restructuration aide à se poser des questions sur la validité de la preuve, en examinant chaque application d'une règle d'inférence. Au fur et à mesure de la construction de l'arbre, une évaluation subjective est portée sur les feuilles rencontrées. Elle est annotée sur l'arbre :

- `true` pour une feuille que nous jugeons valide.
- `false` pour une feuille que nous jugeons fausse.
- ⊥ pour une feuille que nous jugeons non concluante. Typiquement, des parties de raisonnement non triviales ont été omises.
- ? pour une branche que nous ne parvenons pas à développer. Le discours informel est trop confus pour permettre une interprétation en termes de règles d'inférence.

A la fin de l'analyse, nous obtenons une vue d'ensemble de la preuve informelle qui identifie des parties jugées faibles (correspondant aux feuilles `false`, ⊥, ?).

Etape 3 – Conception et réalisation du test guidé par la preuve. Cette étape exploite les résultats de l'analyse précédente pour déterminer des critères de sélection de test, et renforcer l'oracle de test avec l'observation de lemmes intermédiaires. Notons qu'on ne s'attend pas à une parfaite adéquation entre des étapes de preuve douteuses et des entrées de test conduisant à défaillance. Nous utilisons donc une mise en œuvre par test statistique, les cas à couvrir étant testés plusieurs fois avec des entrées aléatoires différentes. Cette démarche est distincte du test aléatoire "aveugle" de l'Etape 1, puisque le profil de test est ici guidé vers des cas de fonctionnement extraits de la preuve. La thèse de Guillaume Lussier [Lus 04] détaille la démarche proposée, et fournit des exemples appliqués aux algorithmes étudiés. En pratique, la connaissance d'un séquent feuille n'est pas toujours exploitable dans la conception du test. Il faut alors remonter dans l'arbre jusqu'à arriver à un séquent introduisant des propriétés commandables ou observables.

Etape 4 – Retour sur la preuve. La dernière étape exploite les informations collectées lors de la restructuration de la preuve, puis lors du test, pour reprendre les étapes de raisonnement qui avaient été jugées faibles. Lorsque des défaillances de l'algorithme ont été observées, la reprise de la preuve vise à affiner le diagnostic de la faute. L'analyse est guidée par la connaissance des scénarios de défaillance. La non-observation de défaillances est un encouragement à consolider la preuve informelle. La reprise de la preuve s'appuie alors sur les indications fournies par l'observation de lemmes intermédiaires.

II.1.1.2. Evaluation expérimentale

L'approche a été appliquée à deux études de cas, présentées respectivement dans [LW 02] et [LW 04a]. Une vue plus détaillée de ces études de cas pourra être trouvée dans les chapitres 3 et 4 de la thèse de Guillaume Lussier [Lus 04].

Le FT-RMS (*Fault-Tolerant Rate Monotonic Scheduling*) est un algorithme d'ordonnancement de tâches ayant pour but de tolérer des fautes transitoires, par ré-exécution des instances de tâches fautives. Des variantes de cet algorithme ont été implémentées dans des systèmes d'exploitation temps-réel [DMM+ 99] [EKM+ 99]. La définition originelle du FT-RMS dans [GMM 97] contenait deux fautes de conception. L'une d'entre elle a été corrigée dans une version révisée de l'algorithme [GMM+ 98], mais une faute résiduelle a ensuite été trouvée par d'autres auteurs [SS 99]. Dans notre étude, nous avons considéré la preuve informelle de la version révisée telle que publiée dans [GMM+ 98].

La deuxième étude de cas est un protocole d'appartenance de groupe (*Group Membership Protocol*, ou GMP) permettant d'obtenir un accord sur l'identité des processeurs non-défaillants dans une architecture répartie ; les processeurs défaillants sont exclus du groupe. Le GMP étudié est une variante du service de groupe offert par le *Time-Triggered Protocol* (TTP) [KG 94] pour des systèmes embarqués. Cette variante, proposée dans [KLR 97], cherche à minimiser les besoins en bande passante réseau : le coût du protocole est de seulement un bit par trame émise sur le bus. Pour mettre au point l'algorithme et sa preuve informelle, les auteurs se sont appuyés sur l'analyse formelle d'une instance à quatre processeurs, en utilisant des techniques de model checking. Après publication de ces travaux, ils se sont cependant aperçus d'une faute de conception, également trouvée par [CR 99].

Pour ces deux algorithmes, l'étape d'analyse préliminaire n'a pas posé de problème particulier. Le Tableau II-A montre les résultats des tests aléatoires réalisés à cette étape, selon un profil d'entrée construit à partir des hypothèses de chaque algorithme. Dans le cas du FT-RMS, l'oracle de test observe le respect des échéances de tâches. Dans le cas du GMP, la vérification porte sur les trois théorèmes que la preuve est censée établir. Le théorème 3, dont la violation est observée, correspond à une propriété d'auto-diagnostic en temps borné (le processeur défaillant ne doit plus se considérer comme faisant partie du groupe). Le taux de défaillance obtenu (6.15%) montre qu'il n'est pas nécessairement très difficile de révéler une faute ayant échappé aux auteurs de la preuve. De façon générale, on a probablement toujours intérêt à mener de premières expériences de test permettant une vérification "grossière" de l'algorithme à faible coût.

	FT-RMS	**GMP**		
	Violation échéance de tâche	Violation théorème 1	Violation théorème 2	Violation théorème 3
Résultats sur 10^4 cas de test	0.04%	—	—	6.15%

Tableau II-A Résultats du test aléatoire à la fin de l'étape 1

La restructuration du discours informel s'est avérée un support très utile pour identifier des faiblesses dans le raisonnement, et ce pour les deux algorithmes. Notre analyse a permis de mettre en évidence aussi bien des problèmes de haut niveau (glissement sémantique de certaines notions au cours de la preuve, oublis majeurs se répercutant sur tout l'arbre de preuve), que des problèmes circonscrits à quelques étapes de raisonnement (décomposition en cas incomplète, confusion de dates dans un raisonnement sur le temps, raccourci abusif

"similaire au raisonnement précédent"). Au final, notre évaluation globale de la rigueur de la preuve a été très différente dans le cas du FT-RMS et du GMP.

Lors de l'exercice de restructuration, la plupart des feuilles de la preuve du FT-RMS ont reçu une évaluation ⊥ ou ? (dans ce dernier cas, le développement de l'arbre était stoppé par l'obscurité du discours). L'identification de ces faiblesses n'a cependant pas permis de guider le test à l'étape 3. Nos résultats expérimentaux ont en fait montré qu'il n'y avait pas de corrélation entre la couverture des cas fonctionnels extraits de la preuve et l'obtention de défaillances de l'algorithme. Une des décompositions en cas s'est avérée incomplète, mais il n'y avait pas non plus de corrélation entre le cas manquant et les défaillances. Intuitivement, une raison est que tous les cas de preuve focalisent sur des paires de tâches, l'instance fautive à ré-exécuter et une autre tâche qui interfère directement avec elle. Les interférences multiples et indirectes ne sont jamais considérées, alors qu'elles constituent la principale difficulté des analyses d'ordonnançabilité. De fait, toutes les violations d'échéance observées correspondent à des schémas d'interférence complexes, et les informations issues de la preuve ne guident en rien vers leur identification. A ce stade, notre conclusion a été que le principe d'un test guidé par la preuve ne semble pas pertinent lorsque l'étape de restructuration met en évidence un manque de rigueur important.

La preuve informelle du GMP est beaucoup plus aboutie. Nous n'avons pas été bloqués par des parties obscures, et la restructuration a pu être poussée à un niveau de détail fin. Cette étape a nécessité une semaine de travail, à l'issue de laquelle plusieurs problèmes ont été identifiés. Deux branches de la preuve du théorème 3 ont été jugées non concluantes (⊥). D'autres problèmes concernent un lemme intermédiaire – appelé *Conjoint 5* – issu de la preuve du théorème 1, et posé en hypothèse lors des preuves des théorèmes 2 et 3. Dans l'arbre de preuve du conjoint 5, trois branches ont été jugées fausses. Cela ne signifie pas nécessairement que le conjoint 5 soit lui-même faux, mais il doit clairement être considéré comme non prouvé. Nous avons pris en compte ces informations dans la conception du test. L'oracle a été renforcé par l'observation du conjoint 5. Un profil d'entrée fonctionnel a été conçu pour permettre de tester ce lemme, ainsi que l'impact de son éventuelle invalidité sur les différents théorèmes. Ce test global a été complété par des tests ciblant deux classes d'entrée correspondant aux branches non concluantes du théorème 3. Le Tableau II-B montre les résultats obtenus. Dans chaque cas, la deuxième ligne en italique donne les taux de violation observés pour un test plus long, de même taille que le test aléatoire aveugle.

	Taille du test	Violations théorème 1	Violations théorème 2	Violations théorème 3	Violations conjoint 5
Test du conjoint 5	45	—	—	4	—
	10^4	—	—	*7.65%*	—
Test du théorème 3 (classe 1)	10	—	—	—	—
	10^4	—	—	—	—
Test du théorème 3 (classe 2)	10	—	—	2	—
	10^4	—	—	*19%*	—

Tableau II-B Résultats du test guidé par la preuve informelle du GMP

Le résultats suggèrent que le conjoint 5 pourrait être valide malgré ses branches de preuve fausses. Les violations du théorème 3 ne sont en tout cas pas liées à des violations de ce conjoint. Les tests plus ciblés montrent un lien avec la classe 2 d'entrées, qui permet d'élever le taux de défaillance à 19%. La classe 1 ne semble pas pouvoir provoquer de défaillance.

Ces résultats permettent un retour constructif sur la preuve. Nous avons fait l'exercice de reprendre la preuve du conjoint 5, ainsi que celle du théorème 3 pour la première classe d'entrée. Les sous-arbres correspondants ont été retravaillés avec succès, et nous considérons maintenant ces propriétés comme prouvées. En reprenant le théorème 3 pour la deuxième classe d'entrée, nous avons réussi à développer l'arbre de sorte à aboutir à une unique feuille fausse qui caractérise précisément les cas de défaillance.

L'étude du GMP a ainsi montré que l'identification des lacunes d'une preuve pouvait s'avérer utile pour guider le test. Une des lacunes identifiées (un raccourci abusif dans la preuve du théorème 3) a permis de focaliser sur une classe d'entrée pertinente vis-à-vis de la faute. Le retour sur la preuve a alors permis d'expliquer les résultats du test, en pointant sur la faille de raisonnement qui a laissé passé la faute, et en affinant la caractérisation des cas de défaillance. L'exemple infructueux du FT-RMS suggère néanmoins que l'efficacité de notre approche pourrait dépendre de la rigueur de la preuve informelle considérée.

II.1.2. Preuves formelles partielles

Ces travaux ont été étendus au cas des preuves formelles, mais partielles. L'idée est d'éviter la perte complète de l'effort de preuve lorsque celle-ci n'a pas aboutir, le test venant prendre le relais. De plus, comme précédemment, le test peut fournir des contrexemples – ou confirmer la validité probable de lemmes en suspens – pour guider la poursuite du travail.

Par rapport à l'approche précédente, l'étape de restructuration de la preuve n'a plus lieu d'être. Du fait de la formalisation, il n'y a pas d'étapes implicites ou confuses de raisonnement ; l'identification de branches non prouvées ne dépend plus d'une évaluation subjective. Par contre, on peut s'attendre à ce qu'il soit plus difficile d'extraire des informations constructives pour le test. Dans une démonstration informelle, le raisonnement des auteurs n'est jamais très éloigné des cas de fonctionnement de l'algorithme. Dans une preuve formelle, les tactiques utilisées peuvent conduire à des artefacts de preuve qui paraissent sans relation avec les cas de fonctionnement. Notre proposition a été d'étudier le principe du test guidé par la preuve à deux niveaux d'abstraction :

- Le *niveau des lemmes* identifie la structure générale de la preuve en termes des principaux lemmes. L'extraction d'informations pour le test se base alors sur la connaissance des lemmes non prouvés et de leurs relations avec les autres lemmes.
- Le *niveau des séquents* s'intéresse aux détails des arbres de preuve des lemmes non prouvés. On cherche alors à exploiter la connaissance précise des séquents en échec pour guider le test.

L'effort d'analyse au niveau des lemmes est *a priori* bien moindre que celui au niveau des séquents. En contrepartie, les informations extraites pourraient être trop imprécises pour efficacement guider le test.

Pour expérimenter les deux niveaux d'abstraction proposés, le choix d'une étude de cas s'est heurté à une difficulté. S'il était relativement aisé de trouver des exemples de preuves informelles incorrectes dans la littérature, les exemples de preuves formelles partielles sont plus difficiles à obtenir : seuls les succès de preuve sont mis à disposition dans le domaine public. Nous avons alors été amenés à envisager une approche expérimentale d'injection de faute dans une preuve complète existante (§II.1.2.1). Cette approche a ensuite été mise en œuvre sur une étude de cas dans l'environnement de preuve PVS[7] (§II.1.2.2) [LW 04b].

[7] Prototype Verification System, voir [ORS+ 95] et http://pvs.csl.sri.com/.

II.1.2.1. Injection de faute dans une preuve formelle

La Figure II-c schématise l'approche d'injection de faute. Elle part de la spécification d'un algorithme correct, et de sa preuve achevée. La faute introduite consiste, par exemple, en une modification élémentaire (mutation) de la description PVS de l'algorithme ; un certain nombre de lemmes deviennent alors non prouvés, ou mal définis. Il faut donc propager la modification à d'autres parties de la modélisation PVS. L'ampleur de la propagation peut être plus ou moins importante, selon la faute injectée. Dans un premier temps, on cherche à modifier les lemmes et définitions directement impactés par la modification de l'algorithme. Les tactiques de preuves associées à ces lemmes doivent éventuellement être adaptées. Les modifications de lemmes peuvent ensuite nécessiter de revoir la structure générale de la preuve, entraînant encore de nouvelles modifications. Le processus termine lorsque l'on obtient une preuve partielle considérée comme représentative d'une tentative de preuve de l'algorithme modifié. Notons que, en pratique, nous avons également stoppé le processus lorsque nous jugions que l'obtention d'une preuve partielle réaliste nécessitait une modification trop importante de la structure de preuve. La faute n'était alors pas retenue pour expérimentation. Nous avons aussi écarté les fautes trop faciles à révéler par un test aléatoire aveugle : seules les fautes induisant un taux de défaillance inférieur à 5% ont été retenues.

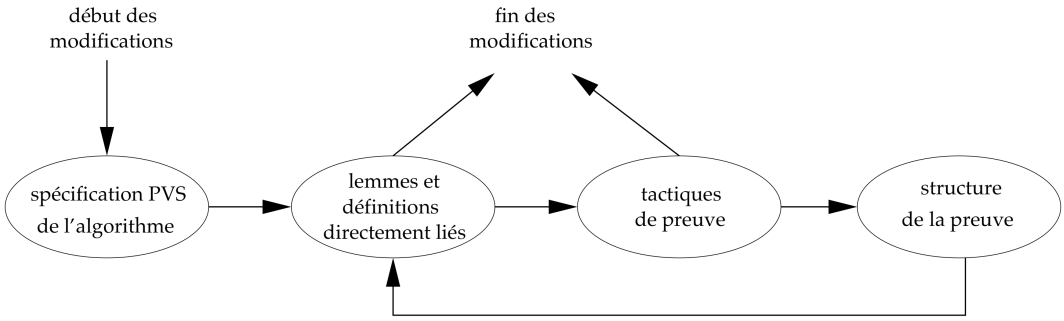


Figure II-c Cycle d'injection d'une faute

II.1.2.2.Application à une étude de cas

L'algorithme étudié est le service de groupe réellement implémenté dans le *Time-Triggered Protocol* (TTP) [KG 94] pour des systèmes critiques. Nous avons déjà mentionné une variante de ce GMP qui a fourni un exemple de preuve informelle erronée. Cet échec a conduit l'un des auteurs de la preuve informelle (J. Rushby) à travailler sur une première modélisation formelle du GMP sous PVS. Il a développé une approche de preuve originale, présentée dans [Rus 00]. Cette approche a ensuite été réutilisée par H. Pfeifer de l'Université de Ulm (Allemagne) pour prouver le protocole GMP finalement implémenté dans le TTP. Au fur et à mesure de l'avancement de la preuve, ce protocole et sa modélisation PVS ont connu un certain nombre d'évolutions [Pfe 00] [PvH 01] [Pfe 03]. Dans le cadre de notre étude, nous nous sommes basés sur la dernière version présentée dans [Pfe 03], pour laquelle nous avons pu obtenir les sources PVS et les scripts de preuve associés. Il s'agit d'un exemple non trivial, avec environ 2500 lignes de source PVS réparties dans 28 théories, et un total de 348 obligations de preuve.

L'approche développée par J. Rushby est de renforcer une propriété de sûreté à prouver en la décomposant en une disjonction de "configurations" (états abstraits) pour lesquelles la propriété pourra facilement être démontrée inductive. Les transitions entre les configurations forment naturellement une représentation sous la forme d'un diagramme qui donne un aperçu du fonctionnement de l'algorithme (Figure II-d).

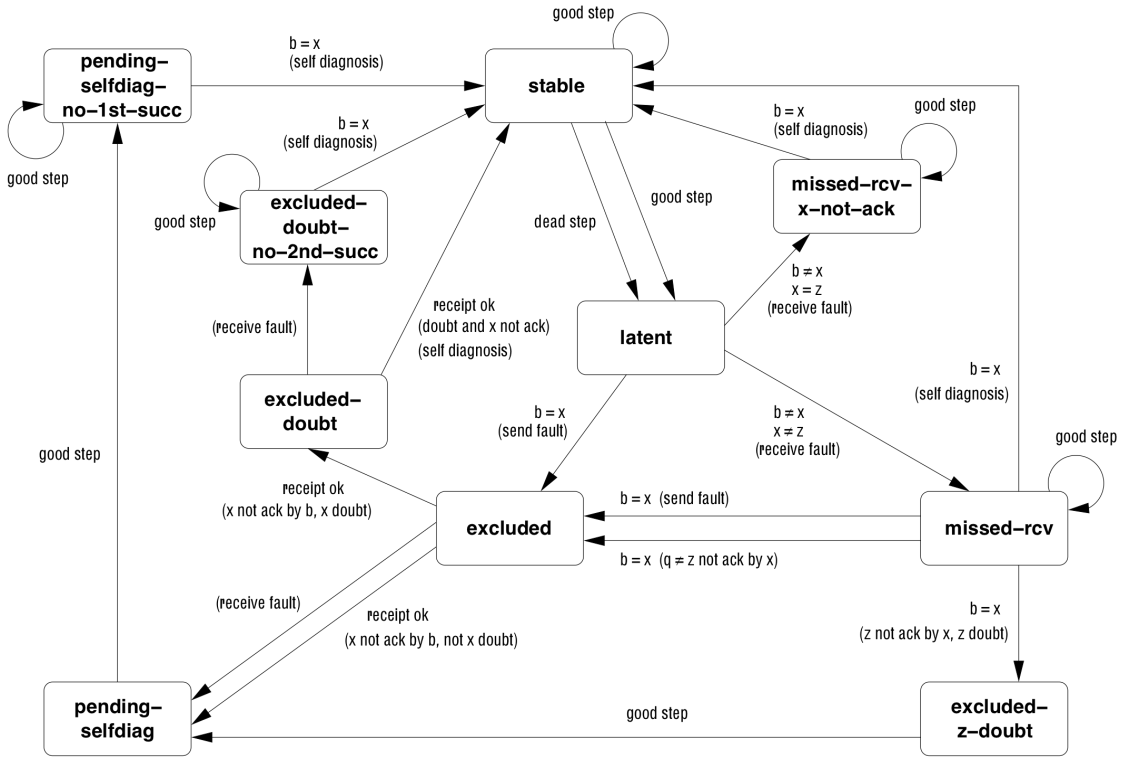


Figure II-d Diagramme des configurations du GMP

La structure générale de la preuve suit la structure de ce diagramme. Pour les propriétés invariantes (validité des vues locales du groupe, accord sur les membres du groupe), on montre que :

- La propriété est vraie dans chaque configuration ;
- Les transitions du diagramme sont correctes (partant d'un état satisfaisant la configuration de départ, on arrive à un état satisfaisant la configuration d'arrivée, sous les hypothèses de franchissement de la transition) ;
- Le diagramme est complet (l'état initial est dans la configuration *stable*, et partant de *stable* le système ne peut jamais arriver à une configuration non définie.

La propriété d'auto-diagnostic en temps borné se prouve en montrant que le système ne peut rester en-dehors de *stable* plus longtemps que le temps autorisé.

Cette structure de preuve, qui se base sur une vue comportementale de l'algorithme, se prête bien à l'extraction d'informations pour le test. Notamment, l'analyse haut niveau des lemmes en échec va toujours permettre une interprétation en termes d'activation de chemins de *stable* à *stable*. Par exemple, si le lemme non prouvé exprime la correction d'une transition, le test va cibler les chemins passant par cette transition. Les profils implémentés cherchent à rendre ces chemins équiprobables. L'analyse au niveau des séquents vise à extraire des informations supplémentaires dans l'arbre preuve du lemme, pour éliminer certains chemins ou restreindre le sous-domaine d'entrée testé pour un chemin.

Quatre fautes ont été injectées. La première, décrite dans le manuscrit de Guillaume Lussier, correspond à une faute réelle survenue lors du développement PVS (la preuve partielle reconstituée n'est cependant pas la preuve partielle réelle, que nous n'avons pas eue à disposition). Les trois autres fautes sont des fautes artificielles de type mutation. Dans chaque cas, il a été possible d'aboutir à une preuve partielle ne comportant qu'un petit nombre de lemmes non prouvés.

	Propriété violée	Nb lemmes en échec	Taux de défaillance : Test aléatoire	Test niveau lemmes	Test niveau séquents
Faute 1	Autodiagnostic	2	0.61%	(1) 1.02% (2) 0.10%	(1) 98.5% (2) 100%
Faute 2	Autodiagnostic	1	1.15%	2.34%	N/A
Faute 3	Validité des vues	1	2.86%	16.58%	86.4%
Faute 4	Autodiagnostic	4	1.36%	(1) 2.38% (2), (3) 3.42% (4) 0%	(1), (2) N/A (3) 80.7%

Tableau II-C Résultats du test guidé par la preuve formelle

Le Tableau II-C montre les résultats expérimentaux. Les taux de défaillance ont été estimés sur 5 jeux de 10^4 séquences de test générées selon chaque profil. Le profil aléatoire considère le domaine d'entrée tel que défini par les axiomes PVS. Les profils guidés ciblent chaque lemme non prouvé. Par exemple, pour la faute 1, le tableau distingue les taux de défaillance selon que le profil cible l'un ou l'autre des deux lemmes en échec. Au niveau des séquents, N/A indique que nous avons échoué à affiner la conception du test (fautes 2 et 4).

L'exemple de la faute 3 montre que l'analyse au niveau des lemmes peut déjà permettre d'identifier un sous-domaine d'entrée avec une densité élevée des cas de défaillance. Les résultats sont décevants pour les autres fautes. L'amélioration par rapport au test aléatoire est notable (ex : taux de défaillance doublé ou triplé) mais reste insuffisante. Notons que les critères retenus *sont* pertinents : il ne peut y avoir de violation de propriété que via des chemins correspondant aux lemmes non prouvés. Mais cette information est encore imprécise vis-à-vis des conditions de défaillance. La génération de larges échantillons de données de test est alors recommandée.

L'analyse au niveau des séquents demande beaucoup plus d'effort que l'analyse précédente, et n'est à entreprendre que si le test aléatoire et le test des lemmes n'ont pas révélé de faute. L'effort investi n'est pas toujours payant. L'établissement d'un lien entre séquent non prouvé et cas de fonctionnement de l'algorithme s'avère en effet problématique : complexité des séquents en nombre de formules, manipulations syntaxiques ne permettant plus de comprendre ce que le séquent cherche à prouver, … Dans nos expériences, il n'a pas toujours été possible d'extraire des informations constructives à partir des arbres de preuve. Néanmoins, lorsque l'analyse a abouti, elle a permis de guider précisément le test.

Le principe d'un test guidé par la preuve demanderait à être expérimenté sur d'autres exemples de preuves formelles partielles – en particulier, des preuves qui ne se baseraient pas sur un modèle tel que le diagramme des configurations. Ces travaux exploratoires ont cependant permis de défricher le sujet, et d'étudier à la fois la pertinence des informations que l'on peut extraire d'une preuve partielle, et les difficultés que cette extraction d'information présente. Pour lever certaines des difficultés rencontrées, la thèse de Guillaume Lussier ouvre des perspectives en termes de modifications automatiques des arbres de preuve. Guillaume Lussier a notamment montré que l'ordre d'application des règles d'inférence a un impact sur la difficulté de l'analyse humaine. Des tactiques de migration de règles pourraient ainsi permettre d'alléger l'effort requis. Toutefois, un effort humain restera toujours nécessaire pour obtenir une compréhension intuitive, en termes de comportement de l'algorithme, de ce qu'un séquent cherche à prouver.

II.2. Analyse de contre-exemples de modèles SCADE

Les travaux qui suivent ont pour cadre la vérification du système de commande de vol, un des systèmes les plus critiques à bord d'un avion. Chez Airbus, le logiciel correspondant fait l'objet d'une conception formelle en SCADE[8], à partir de laquelle 90% du code est automatiquement généré. Plusieurs vérifications formelles sont ensuite réalisées au niveau du code [DSF 06] : preuves à la Hoare pour les parties codées manuellement, analyses basées sur l'interprétation abstraite pour vérifier des propriétés non fonctionnelles sur l'ensemble du code (absence de débordement des calculs, de division par zéro, …). En amont, la conception SCADE est validée par test intensif des modèles dans un environnement de simulation.

Si la pratique opérationnelle est de tester les modèles, la vérification par model checking est étudiée dans un cadre R&D. Nous avons publié un bilan des études sur une dizaine d'années [BVW+ 09]. Notre conclusion a été de recommander une utilisation "légère" (*lightweight*) [Sai 96] du model checking. La vérification formelle est alors appliquée à un petit nombre de fonctions critiques et peut ne considérer qu'un sous-ensemble des comportements possibles. L'objectif est d'aider à la mise au point des modèles SCADE : on s'intéresse principalement à l'obtention de contrexemples, sans nécessairement chercher à poursuivre l'analyse lorsque le model checker échoue à donner un résultat. Dans [BVW+ 09], une expérimentation a montré que ce type d'utilisation pouvait s'avérer efficace pour révéler au plus tôt certaines fautes trouvées lors des essais labos (c'est-à-dire lors des tests des équipements réels couplés à un simulateur de vol sophistiqué). L'analyse des contrexemples obtenus peut néanmoins représenter un effort important, et nos travaux ont visé à assister cette analyse.

La présentation ci-dessous commence par préciser la motivation de nos travaux, en les situant par rapport à d'autres contributions sur l'analyse de contrexemples et sur le test (§II.2.1). L'approche retenue est structurelle. Elle repose sur l'identification de chemins du modèle activés par un contrexemple (§II.2.2 et §II.2.3). Un prototype a été implémenté, dont les fonctionnalités sont illustrées sur une étude de cas (§II.2.4).

II.2.1. Objectifs de l'analyse et travaux connexes

La Figure II-e schématise un processus de vérification itératif. L'artefact vérifié comprend un modèle SCADE connecté à des observateurs synchrones [HLR 93] spécifiant une propriété souhaitée et des hypothèses sur l'environnement du modèle. On s'intéresse au cas où un contrexemple est retourné (partie droite de la figure).

L'ingénieur doit diagnostiquer les raisons de la violation de propriété observée. Pour cela, il va d'abord rejouer le contrexemple sur le modèle, suivre les parties du modèle activées lors de l'exécution et identifier des événements clés ayant contribué à violer la propriété. Une fois que les détails de l'exécution sont bien compris, l'ingénieur doit déterminer si le contrexemple correspond à un scénario avion réaliste. Si tel n'est pas le cas, il faut revoir la formulation des hypothèses sur l'environnement ou même la formulation de la propriété. Si le contrexemple est réaliste, alors une faute de conception vient d'être révélée et il faut corriger le modèle SCADE. Plusieurs itérations vont ainsi être effectuées pour mettre au point les observateurs et éventuellement corriger le modèle SCADE. A chaque itération, l'ingénieur doit obligatoirement apporter une modification avant de lancer une nouvelle analyse formelle : comme la plupart des model checkers, *SCADE Design Verifier* fournit un unique contrexemple, de sorte que relancer l'analyse sur la même version du modèle et de ses observateurs produirait toujours ce contrexemple. Cela ralentit le processus de mise au point.

[1] [8] http://www.esterel-technologies.com/products/scade-suite

Il serait plus efficace d'avoir des retours multiples sur les comportements incorrects avant d'entreprendre une modification. Notons enfin que le processus itératif montré en Figure II-e peut lui-même être répété plusieurs fois : même si la vérification conclut à une satisfaction de la propriété, l'ingénieur peut vouloir aller plus loin en supprimant des hypothèses ou en élargissant le modèle analysé.

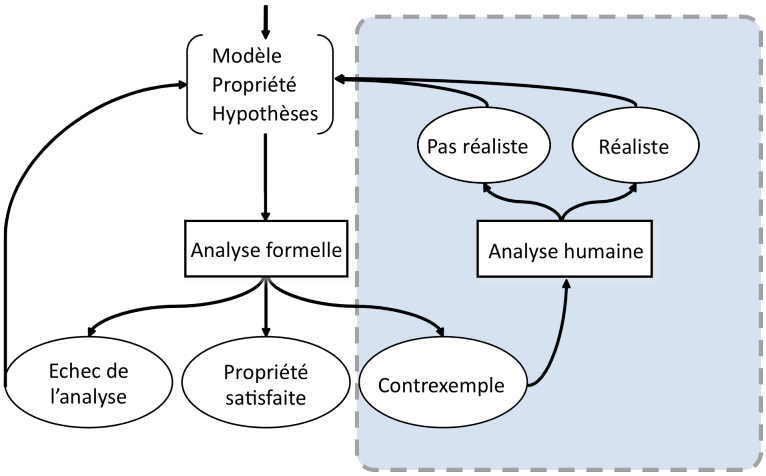


Figure II-e Processus itératif en phase de mise au point de modèles

Au total, l'analyse humaine des contrexemples va nécessiter un effort important qu'il serait souhaitable d'alléger. Décider du caractère réaliste d'un contrexemple relève d'une expertise avionique et paraît difficilement automatisable. Par contre, il devrait être possible d'apporter un support aux ingénieurs pour :

1. Faciliter la compréhension des détails de l'exécution du contrexemple ;
2. Forcer le model checker à chercher plusieurs contrexemples, illustrant des scénarios de violation différents.

Pour satisfaire ces deux objectifs, nous avons choisi de mettre en œuvre une analyse structurelle des chemins d'un modèle SCADE activés par le contre-exemple. Il s'agit d'une part d'extraire l'information pertinente pour expliquer la violation de la propriété observée, et d'autre part de guider le model checker vers l'exploration de comportements différents, c'est-à-dire activant d'autres chemins du modèle. On voit là le lien avec des techniques de test structurel.

L'aide à la compréhension de contrexemples a été un objectif de travaux antérieurs aux nôtres, parmi lesquels [GV 03] et [BNR 03] sont les plus proches dans l'esprit. Comme nous, ces auteurs ont fait le choix d'analyser les contrexemples en termes d'emplacements structurels activés lors de l'exécution. Notre approche se distingue cependant des leurs par deux aspects. Tout d'abord, une de nos contraintes était que l'analyse structurelle soit réalisée entièrement en dehors du model checker. L'avantage est de rester indépendant de l'outil de model checking, et de ne nécessiter aucune visibilité sur la vérification en cours. Un outil tel que *SCADE Design Verifier* est en effet une boîte noire pour ses utilisateurs. L'inconvénient par rapport aux autres travaux est qu'il ne nous est pas possible de récupérer les informations exploitées lors de la construction du contrexemple (par exemple, des ensembles de transitions correctes dans [BNR 03]). Dans notre cas, l'information utile doit être reconstituée *a posteriori*. La vérification par model checking étant déjà très coûteuse en temps, cela signifie que notre analyse doit rester peu complexe et permettre des retours rapides à l'utilisateur. Le

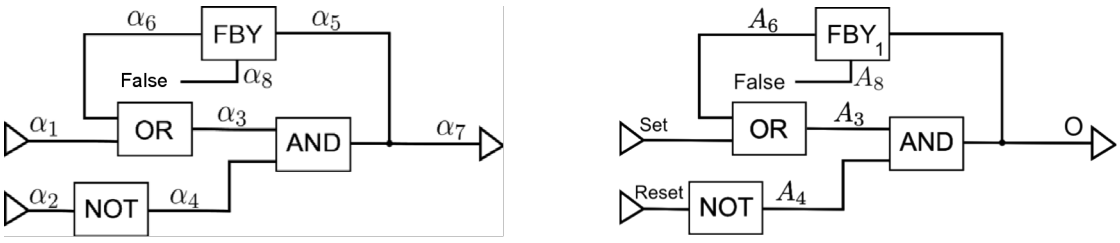
deuxième aspect distinctif de notre approche est que nous traitons des modèles SCADE flots de données (très similaires à des modèles de circuits au niveau des portes logiques), alors que [GV 03] et [BNR 03] se placent dans le cadre du model checking de programmes impératifs. Cela induit des différences importantes sur la façon de relier l'exécution du contrexemple à la structure de l'artefact analysé. En particulier, dans un model synchrone flot de données, tous les opérateurs sont exécutés simultanément à chaque cycle. Les algorithmes que nous avons développés sont en fait à rapprocher de ceux utilisés pour le test structurel de circuits [BA 00] ou pour le test et le débogage de programmes Lustre [GJJ+ 03] [LP 09]. Notamment, [LP 09] a proposé une formalisation de la notion de *chemin actif* que nous avons réutilisée dans nos travaux.

Les paragraphes qui suivent donnent une vue intuitive de notre approche. Les détails de formalisation peuvent être trouvés dans [BVW+ 10] et dans la thèse de Thomas Bochot [Boc 09]. Nous nous sommes concentrés sur les modèles pour lesquels le model checking est le mieux adapté, c'est-à-dire les modèles ne comportant que des variables booléennes. Le traitement des calculs numériques a été mis de côté pour des travaux futurs.

II.2.2. Chemins dans un modèle SCADE

SCADE est un langage graphique basé sur le langage synchrone Lustre [HCR+ 91]. Un modèle SCADE consiste en un réseau d'opérateurs, dont les nœuds sont les opérateurs et les arcs connectent la sortie d'un opérateur à l'entrée d'un autre opérateur. L'exécution est pilotée par une horloge, chaque cycle d'horloge correspondant à une lecture des entrées du modèle et un calcul des sorties. A titre illustratif, la Figure II-f montre la modélisation d'une bascule RS à reset prioritaire. Le réseau comporte des opérateurs booléens (*OR, AND, NOT*) et un opérateur temporel (*Followed-by* ou *FBY*) qui permet d'introduire un retard dans la propagation d'une donnée. Les données sont portées par les arcs. Pour faciliter la discussion, la Figure II-f donne deux étiquetages complémentaires du réseau : (1) avec des identifiants d'arcs α_i, (2) avec des noms de variables. Notons que des arcs différents peuvent porter la même donnée, par exemple les arcs α_5 et α_7 portent tous deux la valeur de la sortie *O* au cycle courant. Pour une variable *V*, on note *V(n)* sa valeur au cycle *n*.

Un chemin complet est une suite d'arcs connectant une valeur d'entrée à une valeur de sortie. Par exemple, le chemin $p1 = < \alpha_1, \alpha_3, \alpha_7 >$ connecte *Set(n)* à *O(n)*. Les opérateurs temporels induisent des chemins avec boucle. Dans l'exemple, l'opérateur *FBY* permet de réinjecter les résultats du calcul au cycle *n-1* dans le calcul au cycle suivant. Ainsi, le chemin $p1' = < \alpha_1, \alpha_3, \alpha_5, \alpha_6, \alpha_3, \alpha_7 >$ connecte *Set(n-1)* à *O(n)*. Plus généralement, un chemin de la forme $< \alpha_1, \alpha_3 > < \alpha_5, \alpha_6, \alpha_3 >^k < \alpha_7 >$ où *k* est le nombre d'itérations de la boucle, permet de propager la valeur *Set(n-k)* pour affecter *O(n)*.



(1) Nœuds et arcs (2) Données associées aux arcs

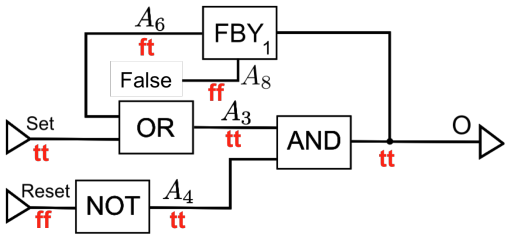
Figure II-f Modélisation d'une bascule RS

L'existence d'un chemin complet indique un canal de propagation *potentiel* pour affecter la sortie. La propagation n'est effective qu'aux cycles *n* où le chemin est actif. Pour déterminer la condition d'activation d'un chemin, nous avons repris les travaux de Lakehal et Parissis [LP 09] sur le test structurel de programmes Lustre. Selon ces travaux, la condition d'activation est une expression Lustre booléenne, construite récursivement en considérant les opérateurs traversés par le chemin. Par exemple, le chemin *p1'* ci-dessus a la condition d'activation suivante :

$$(false \rightarrow pre((Set \vee \neg A6) \wedge (\neg A3 \vee A4))) \quad \wedge \quad (A6 \vee \neg Set) \quad \wedge \quad (\neg A3 \vee A4)$$

où l'opérateur *FBY* traversé par le chemin induit une condition d'activation avec des opérateurs temporels Lustre, l'opérateur d'initialisation (->) et l'opérateur "précédent" (*pre*).

La Figure II-g montre un exemple de scénario d'entrée sur deux cycles : les entrées booléennes *Set* et *Reset* prennent respectivement les séquences de valeurs *true true* et *false false*. L'exécution de ce scénario sur le modèle (par exemple en utilisant le simulateur de l'atelier *SCADE Suite*) affecte des valeurs aux variables auxiliaires et à la variable de sortie, ce qui permet d'évaluer les conditions d'activation des chemins. La Figure II-g indique quels sont les chemins actifs à chaque cycle (dans un modèle flot de données, plusieurs chemins sont actifs simultanément). Les chemins de la forme $< \alpha_8, ...>$ ne sont jamais activés. Intuitivement, cela signifie que l'initialisation à *false* de la bascule ne joue aucun rôle dans ce scénario, et n'affecte ni *O(1)* ni *O(2)*. Si l'on veut chercher à expliquer une valeur prise par la sortie *O*, on doit focaliser l'analyse sur les chemins actifs au cycle correspondant. Cette idée est mise en œuvre par notre méthode d'analyse de contrexemples.



Chemins actifs au cycle 1 : *p1, p2*.
Chemins actifs au cycle 2 : *p1, p1', p2, p2'*

Où : $p1 = < \alpha_1, \alpha_3, \alpha_7 >$ de *Set(n)* à *O(n)*
$p1' = < \alpha_1, \alpha_3, \alpha_5, \alpha_6, \alpha_3, \alpha_7 >$ de *Set(n-1)* à *O(n)*.
$p2 = < \alpha_2, \alpha_4, \alpha_7 >$ de *Reset(n)* à *O(n)*.
$p2' = < \alpha_2, \alpha_4, \alpha_5, \alpha_6, \alpha_3, \alpha_7 >$ de *Reset(n-1)* à *O(n)*.

Figure II-g Chemins activés par l'exécution d'un scénario

II.2.3. Des chemins actifs aux causes d'un contrexemple

Notre analyse porte sur le modèle global constitué du modèle cible et de ses observateurs (Figure II-h). Un contrexemple n'est autre qu'un scénario d'entrée de ce modèle global, tel que la sortie *O* de l'observateur de propriété soit falsifiée au dernier cycle *N* d'exécution. Il s'agit alors d'extraire l'information utile pour expliquer cette falsification. Comme indiqué plus haut, les causes vont être recherchées en considérant les chemins actifs au cycle *N*. Une cause est alors définie comme un sous-ensemble des chemins complets actifs dans le modèle global, tel que les valeurs d'entrée à l'origine de ces chemins puissent reproduire la falsification observée. C'est-à-dire que ces valeurs d'entrées sont suffisantes pour :

- reproduire la valeur *O(N) = false* observée sur le contrexemple,
- assurer que les chemins de la cause soient actifs au cycle *N*, et qu'ils propagent donc bien les valeurs d'entrée jusqu'à *O(N)*.

La formalisation de cette notion de *cause* peut être trouvée dans [BVW+ 10] et [Boc 09], ainsi qu'une discussion de définitions alternatives que nous n'avons pas retenues. Je me borne ici à illustrer la notion de cause sur un exemple, pour en donner l'intuition.

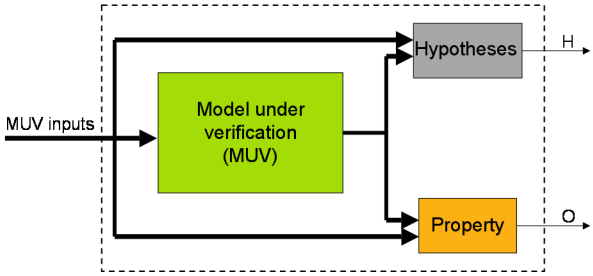


Figure II-h Architecture de vérification

Reprenons le petit exemple de la bascule et du scénario d'entrée montré en Figure II-g. Supposons que l'on cherche la cause de *O(2) = true*. Il y a quatre chemins actifs au cycle *2* : *p1, p1', p2, p2'*.

Selon notre définition, {*p1, p2*} est une cause. Les valeurs d'entrée aux origines de ces chemins sont *Set(2) = true* et *Reset(2) = false*. Intuitivement, lorsque l'on effectue un set sans reset, alors la sortie courante de la bascule sera nécessairement à *true*. De fait, si on considère n'importe quelle variante du scénario avec les mêmes valeurs *Set(2)* et *Reset(2)*, mais des valeurs arbitraires pour *Set(1)*, *Reset(1)* et l'initialisation de la bascule, on obtiendra inéluctablement l'activation de *p1* et *p2* au cycle *2* et un résultat *O(2) = true*.

Une autre cause est {*p1', p2', p2*}. Les valeurs d'entrée correspondantes sont : *Set(1) = true*, *Reset(1) = false* et *Reset(2) = false*. Intuitivement, s'il y a eu un set dans le passé, et aucun reset depuis, alors la sortie courante est *true*.

Enfin, par construction, l'union de deux causes est encore une cause. On a donc une troisième cause {*p1, p2, p1', p2'*}. Celle-ci n'apporte pas de nouvelle information par rapport aux autres. Cela suggère une notion de minimalité basée sur la relation d'inclusion : une cause est minimale si elle n'inclut aucune autre cause. On peut vérifier que les deux premières causes sont bien minimales. Chacune suffit pour expliquer indépendamment la valeur *O(2)* observée, et elles ne contiennent pas de chemins superflus.

Revenons maintenant aux objectifs initiaux de nos travaux, qui étaient de faciliter la compréhension de contrexemples, et de guider la recherche de plusieurs contrexemples différents. Pour répondre au premier objectif, une cause d'un contre-exemple a été définie comme un ensemble de chemins de propagation entre (i) certaines valeurs d'entrée du modèle à certains cycles de la séquence et (ii) la sortie *O* de l'observateur de la propriété au cycle de la violation. L'assistance fournie va alors consister à extraire automatiquement les causes du contrexemple, et de préférence ses causes minimales, pour les montrer à l'utilisateur. Le second objectif est reformulé en termes de recherches de contrexemples exhibant de nouvelles causes minimales. Pour cela, notre proposition est de modifier automatiquement la formulation de l'observateur de propriété pour indiquer model checker de :

- chercher un contrexemple dans lequel de nouveaux chemins sont actifs, ou
- chercher un contrexemple dans lequel certains chemins des causes minimales déjà connues sont inactifs.

La recherche de contrexemples différents peut ainsi s'inspirer de stratégies de test structurel. Dans le cadre de la thèse de Thomas Bochot, une première stratégie cherchant l'activation de nouveaux chemins a été implémentée.

II.2.4. Implémentation et expérimentation sur une étude de cas

Le prototype STANCE (*Structural Analysis of Counter-Examples*) implémente l'analyse proposée. STANCE a été développé dans l'environnement Matlab/Simulink[9] car cet environnement offre une interface de programmation commode pour raisonner sur l'exécution des modèles et l'activation de leur structure interne. Notons que Simulink intègre un model checker (*Simulink Design Verifier*) basé sur la même technologie que celui de SCADE. STANCE représente environ 700 lignes du langage de script Simulink. Une interface graphique permet de visualiser les causes.

Le calcul des causes met en œuvre un algorithme récursif, qui effectue un parcours en arrière à la fois sur la structure du modèle global (de la sortie de l'observateur de propriété à certaines entrées du modèle) et sur le temps (du cycle *N* de la violation à certains cycles antérieurs). Le pire cas est quadratique en *N* x la taille du modèle, lorsque toute la structure doit être parcourue à tous les cycles d'exécution. Nous avons démontré que l'algorithme termine et produit bien des causes du contrexemple. La preuve s'effectue par induction sur l'arbre d'exécution de l'algorithme.

La minimalité des causes n'est malheureusement pas une propriété inductive dans le cas général. Ceci est dû à l'éventuelle présence de chemins reconvergents, connectant une même valeur d'entrée *I(k)* à la sortie *O(N)* de l'observateur de propriété (l'exemple de la bascule RS était sans reconvergence, car il y a un unique chemin entre chaque valeur d'entrée et la sortie). La reconvergence est un problème bien connu dans le domaine du test du matériel et induit une complexité exponentielle des algorithmes de génération de tests [BA 00]. Notre discussion de ce problème a conclu qu'il serait trop coûteux de chercher à le traiter de façon exacte. Rappelons que notre analyse intervient en support au model checking déjà très coûteux en temps, et n'a d'intérêt que si le surcoût est négligeable. Nous avons alors fait le choix de tolérer que les causes calculées ne soient pas minimales dans certains cas ; nous avons cependant démontré qu'elles le seront toujours en l'absence de reconvergence.

STANCE a été utilisé sur une étude de cas fournie par Airbus. Le modèle à vérifier synthétise une décision booléenne, qui s'exprime à l'aide d'opérateurs temporels complexes tels que des bascules, des confirmateurs, et divers types de déclenchement sur front d'entrée. La Figure II-i montre l'architecture de vérification de ce modèle, dans la syntaxe Simulink. On voit que l'observateur de propriété inclut également quelques opérateurs temporels. Notons que le modèle à vérifier comporte des chemins reconvergents.

Un premier appel du model checker par STANCE renvoie un contrexemple de 53 cycles. STANCE analyse alors ce contrexemple en moins d'une seconde et trouve une seule cause ; nous avons pu vérifier que cette cause est minimale. Elle est montrée à l'utilisateur à l'aide d'une interface offrant deux vues complémentaires (Figure II-j) :

- une vue structurelle où les chemins de la cause sont colorés pour mettre en évidence les parties du modèle impliquées,
- une liste d'événements clés qui fournit des informations temporelles relatives aux chemins (valeurs aux origines des chemins, et valeurs intermédiaires importantes pour comprendre le comportement des opérateurs temporels traversés).

Ces deux vues ont été jugées utiles par nos interlocuteurs Airbus pour acquérir une compréhension rapide du déroulement du contrexemple.

[9] http://www.mathworks.com/products/simulink/

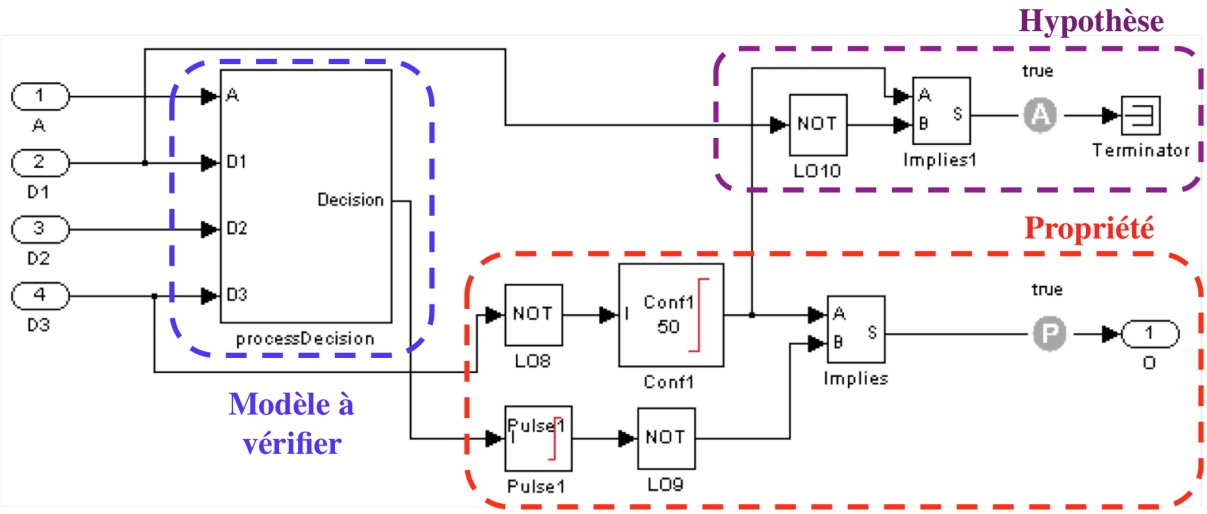


Figure II-i Modèle global de l'étude de cas

STANCE implémente également une stratégie simple de recherche de contrexemples différents. L'outil sélectionne un arc qui n'appartient à aucun chemin des causes connues, et modifie automatiquement l'observateur pour tenter de violer la propriété en passant par cet arc. Appliquée à l'étude de cas, cette stratégie permet au model checker de trouver un nouveau contrexemple de 77 cycles comportant deux causes minimales. La première cause correspond à celle déjà connue, mais la deuxième montre une activation très différente du modèle à vérifier (cf. la Figure II-k, à comparer visuellement à la Figure II-j). Le contrexemple met ainsi en évidence un nouveau type de comportement indésirable.

Renvoyer plusieurs contrexemples devrait permettre une convergence plus rapide vers des observateurs et un modèle corrects. La stratégie implémentée par STANCE n'est clairement qu'une première tentative pour illustrer la faisabilité d'une exploration guidée par des objectifs de couverture structurelle. Pour aller plus loin, d'autres stratégies – et combinaisons de stratégies – devraient être proposées et comparées expérimentalement.

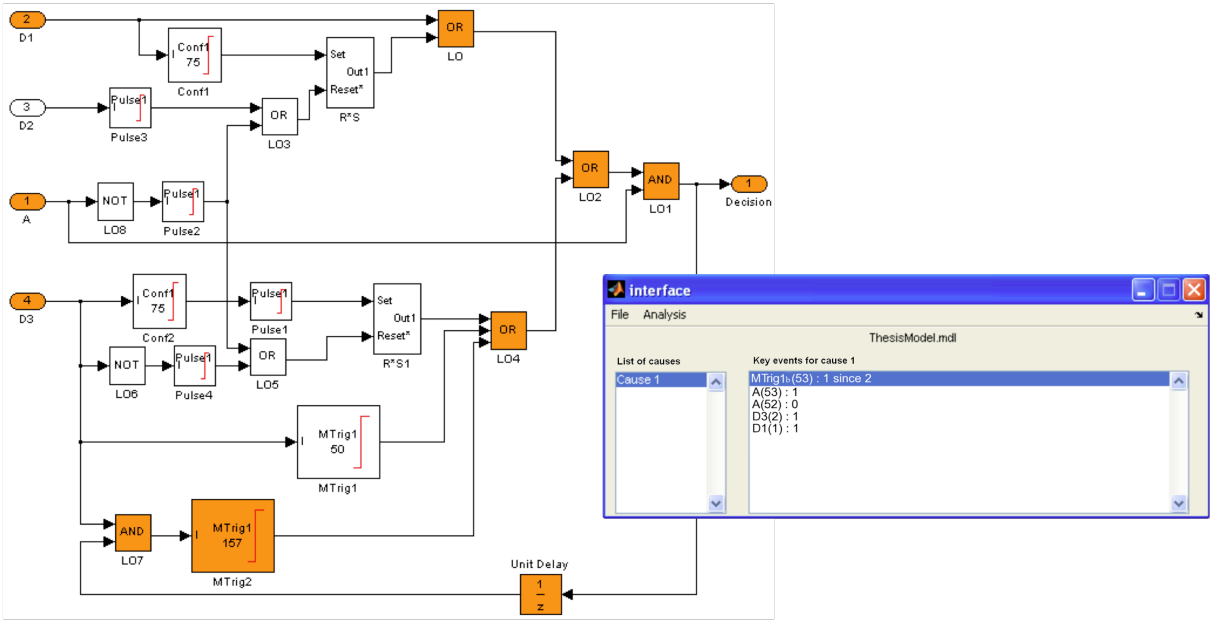


Figure II-j Modèle coloré et liste d'événements retournés par STANCE

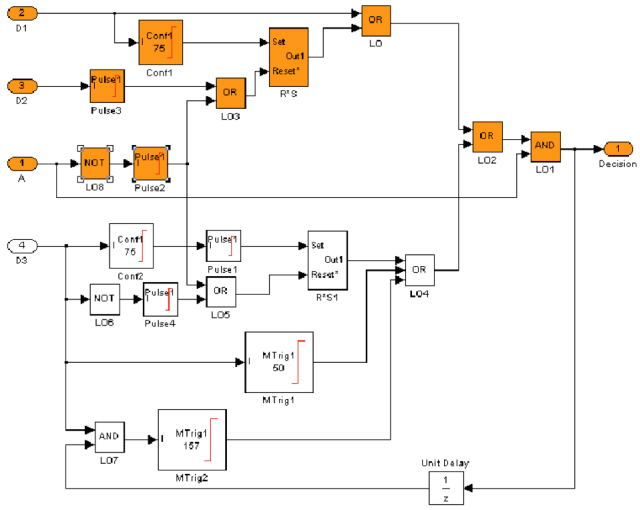


Figure II-k Modèle coloré pour la nouvelle cause minimale

II.3. Conclusion et discussion

Ce chapitre a présenté deux exemples de couplage de techniques de test et de vérification formelle.

Le couplage test et preuve de théorème présentait un caractère fortement exploratoire. Les approches de test fonctionnel s'appuient classiquement sur une analyse du comportement attendu, en faisant des hypothèses sur la manifestation des fautes : on en déduit des classes de valeurs d'entrées ou des fragments de comportement à couvrir durant le test. Nous avons voulu étudier la possibilité d'extraire ce type d'information à partir de la structure d'une preuve partielle. On s'appuie alors sur l'analyse fonctionnelle déjà réalisée par l'auteur de la preuve, et on cherche à compléter la preuve en ciblant ses lacunes. Des premières expériences ont été menées pour évaluer la faisabilité et la pertinence de cette idée. Les résultats obtenus sont mitigés. Pour des preuves avec un petit nombre de lemmes en suspens, il a été possible d'identifier des cas fonctionnels exploitables par le test. Cependant, ces cas fonctionnels sont souvent imparfaitement liés aux cas de défaillance, sauf lorsque l'on a pu analyser la preuve à un niveau de détail très fin (niveau des séquents d'une preuve formelle). L'effort d'analyse est alors très important, voire rédhibitoire pour tout autre que l'auteur de la preuve. Avec le recul, la contribution la plus susceptible de donner lieu à extension pourrait en fait être l'approche de restructuration des preuves informelles – pas seulement à des fins de test, mais en tant que support au développement et à la vérification de preuves en langage naturel. Les perspectives consisteraient à s'intéresser aux travaux portant sur l'analyse de langages naturels contrôlés, en particulier ceux appliqués à la vérification de textes mathématiques [Zin 06] [CFK 09].

Les travaux autour du model checking présentaient un caractère plus appliqué. Ils ont débouché sur un prototype mettant en œuvre l'analyse structurelle de contrexemples. Cette analyse est destinée à aider les ingénieurs en phase de mise au point des modèles SCADE. Elle permet de filtrer l'information utile à la compréhension des contrexemples, ainsi que de guider le model checker vers la recherche de nouveaux contrexemples. Les travaux se sont concentrés sur les modèles booléens, pour lesquels le model checking est le plus adapté. Notons que le prototype accepte en fait tout type de modèle, mais va stopper le parcours d'un chemin lorsqu'il rencontre un opérateur convertissant des flots numériques en un flot booléen (par exemple, un opérateur '<'). La valeur de sortie de l'opérateur est alors portée sur la liste d'événement clés. Une extension facile serait de donner explicitement l'équation synthétisant cette valeur en fonction des entrées numériques du modèle (ex : $X(2) + Y(2) < Z(1)$). Pour un

modèle essentiellement booléen, mais comportant quelques calculs numériques simples, cela pourrait suffire à la compréhension du contrexemple ; la notion de cause ne nécessiterait alors pas de modification majeure. En pratique, les études R&D menées à Airbus montrent que le model checker n'arrive à traiter les calculs numériques que dans la mesure où ils sont peu nombreux et peu complexes [BVW+ 09].

L'utilisation d'objectifs structurels pour chercher des contrexemples différents ouvre de nouvelles perspectives en relation avec des problématiques de test. On pourrait notamment s'intéresser aux stratégies de dépliage structurel mises en œuvre dans l'outil de test GATeL [MA 00], sur des expressions Lustre. Les approches de test concolique [GKS 05] [SMA 05] [WMM+ 05] me paraissent également pertinentes : partant de la trace détaillée d'un chemin d'exécution sous test, elles mettent en œuvre des stratégies symboliques pour chercher à couvrir de nouveaux chemins. Bien que ces approches aient été proposées pour tester des programmes impératifs, leur principe peut être adapté aux chemins dans les modèles flot de données synchrones. Différentes combinaisons de stratégies pourraient ainsi être expérimentées, en incorporant éventuellement des éléments heuristiques.

Chapitre III. Test basé sur des métaheuristiques de recherche : analyse de paysages

Les métaheuristiques de recherche, par exemple le recuit simulé [KGV 83] ou les algorithmes génétiques [Hol 75], sont utilisées pour résoudre des problèmes d'optimisation complexes, pour lesquels on ne dispose pas de méthode exacte efficace. La généralité des métaheuristiques les rend applicables à de nombreuses classes de problèmes, et je vais m'intéresser ici aux applications pour la génération de tests. Le principe est de reformuler un problème de test en un problème d'optimisation par rapport à une fonction objectif, appelée fonction *fitness* (lorsque l'on cherche à la maximiser) ou fonction *coût* (lorsque l'on cherche à la minimiser). L'espace des solutions correspond alors à un espace d'entrées de test, ou de séquences d'entrées. La recherche d'optima dans cet espace nécessite des exécutions du programme sous test, afin d'évaluer l'adéquation de solutions candidates par rapport à l'objectif. Les résultats d'évaluation permettent de guider la génération de nouvelles solutions candidates, selon un processus itératif.

Le test basé sur des métaheuristiques constitue un domaine très actif, et il serait impossible de citer ici tous les travaux pertinents. On peut renvoyer le lecteur à deux articles de synthèse. Le premier, publié en 2004, faisait le point sur l'application des métaheuristiques à des problèmes de test [McM 04]. Le deuxième, plus récent, s'intéresse au cadre plus général du génie logiciel, mais accorde une large place au test avec environ 300 références dans le domaine [HMZ 09]. Ces deux articles de synthèse montrent la diversité des problèmes de test abordés via les métaheuristiques. Si la majorité des travaux cités concerne le test structurel de programmes, en particulier le test des branches, on trouve aussi des applications à d'autres problèmes tels que : le test de mutation, le test de pre/post conditions, le test d'exceptions, l'évaluation de comportements temps-réel, ou encore le test de la sécurité-innocuité de systèmes de contrôle/commande. Nos propres travaux au LAAS ont porté sur cette dernière classe de problèmes (voir aussi des exemples de travaux apparentés dans [SGD 95] et [WB 04]). Il s'agissait de chercher des scénarios de test susceptibles de mettre en défaut la sécurité de systèmes de contrôle/commande, en considérant le couplage du logiciel de contrôle à son environnement physique (dispositifs matériels tels que capteurs et actionneurs, lois physiques du procédé contrôlé). Au-delà du problème de test visé, ces travaux nous ont amené à aborder une question fondamentale, sur laquelle j'ai choisi de mettre l'accent dans ce chapitre : comment caractériser les problèmes de test et le comportement des métaheuristiques appliquées à ces problèmes ? L'enjeu est de pouvoir comprendre les performances observées, et de maîtriser la mise en œuvre des métaheuristiques.

Cet enjeu est important, car une métaheuristique de recherche n'est pas un algorithme prêt à l'emploi. Une métaheuristique correspond plutôt à une stratégie générale, dont on doit régler les paramètres selon le problème à résoudre. Les choix retenus affectent les performances. Il

est typiquement recommandé d'essayer plusieurs implémentations alternatives d'une métaheuristique, voire des implémentations de plusieurs métaheuristiques. Ces différents essais sont largement ad hoc, demandent un effort important, et ne garantissent pas de trouver les "bons" choix pour un problème donné.

Dans la littérature générale sur les métaheuristiques, des approches quantitatives ont émergé pour analyser la dynamique de la recherche d'optimum. Un concept central pour ces approches est celui de *paysage*. Si l'on considère que le coût (ou la fitness) d'une solution candidate définit son altitude, alors la métaphore du paysage suggère des images de pics, de vallées, de plateaux… Intuitivement, le "relief" d'un tel paysage devrait avoir un fort impact sur la dynamique des stratégies utilisées pour l'explorer. Plusieurs mesures ont ainsi été définies pour caractériser les paysages, et chercher à expliquer – ou prédire – les performances des métaheuristiques. Les résultats publiés concernent des problèmes combinatoires classiques en théorie de la complexité, tels que le problème de la satisfiabilité booléenne ou celui du voyageur de commerce. Il nous a paru intéressant de chercher à transférer le principe de ces analyses quantitatives à un problème de test. Nos travaux dans cette direction se sont déroulés au début des années 2000. A notre connaissance, ils ont été parmi les premiers à utiliser des mesures de paysages dans le domaine du test. D'autres travaux contemporains des nôtres se référant à des analyses de paysage sont [BS 03], [MH 04] et [WB 04].

La présentation de notre contribution introduit d'abord le problème de test que nous cherchions à traiter, et les difficultés que nous avons rencontrées dans la mise en œuvre d'une métaheuristique, le recuit simulé (§III.1). Ces difficultés ont motivé notre intérêt pour une approche quantitative. Le paragraphe suivant (§III.2) effectue un tour d'horizon rapide des mesures existantes. Il justifie notre choix d'en retenir deux dans le cadre de nos travaux : le diamètre du paysage, et l'autocorrélation ρ_1. Le paragraphe III.3 décrit notre étude expérimentale de ces mesures et les résultats obtenus. Je montre notamment comment l'exploitation de ces mesures a permis un retour sur notre problème de test initial, pour améliorer le paramétrage du recuit simulé. Enfin, le paragraphe III.4 propose une nouvelle mesure, le Taux de Génération de Meilleures Solutions (TGMS), visant à surveiller la convergence du processus de recherche et à implémenter des critères d'arrêt. Des résultats expérimentaux sont là encore montrés.

Dans toute recherche à caractère empirique, il est important de décrire précisément les configurations expérimentales qui ont conduit aux résultats obtenus. L'exercice de la HDR ne permettant pas de rentrer dans ce niveau de détail, j'invite le lecteur intéressé par les résultats des paragraphes III.3 et III.4 à consulter le manuscrit de thèse de Olfa Abdellatif-Kaddour [Abd 03] (co-encadrée avec Pascale Thévenod-Fosse, LAAS). Pour une vue plus synthétique, l'article [WTA 07] discute également ces résultats, et donne les configurations expérimentales en annexe.

III.1. Du problème de test et des difficultés qui ont motivé nos travaux

La thèse d'Olfa Abdellatif-Kaddour portait sur le test de propriétés de sécurité (*safety*) de systèmes de contrôle/commande. Un système de contrôle/commande fait intervenir un programme de contrôle, en interaction cyclique avec son environnement physique. Nous nous intéressions à des propriétés de haut niveau, relatives à la maîtrise du procédé contrôlé. Pour tester de telles propriétés, la plateforme expérimentale doit typiquement connecter le programme de contrôle à un simulateur du monde physique, pour prendre en compte les lois physiques du procédé, les caractéristiques des dispositifs matériels (capteurs, actionneurs) mis en place, et l'occurrence de fautes dans ces dispositifs. Idéalement, la sélection des tests à exécuter sur cette plateforme devrait favoriser des scénarios "stressants" vis-à-vis de la propriété. Pour cela, nous avons proposé une stratégie incrémentale de construction de scénarios de test, chaque étape explorant les suites possibles des scénarios "dangereux" trouvés à l'étape précédente. L'implémentation d'une étape de la stratégie peut mettre en jeu une technique d'exploration simple (recherche aléatoire uniforme) ou une technique plus sophistiquée guidée par un critère d'optimisation. C'est dans ce cadre que nous nous sommes intéressées aux métaheuristiques de recherche, et plus particulièrement au recuit simulé.

Nous avons appliqué la stratégie incrémentale à une étude de cas, un système de contrôle/commande d'une chaudière à vapeur. Cette étude de cas, présentée ci-après, offre une bonne illustration des difficultés pour régler les paramètres du recuit simulé.

III.1.1. Test d'une chaudière à vapeur

La chaudière à vapeur (*steam boiler*) est une étude de cas ayant suscité de nombreuses contributions [ABL 96]. Nous disposions d'un cahier des charges en anglais, d'une version C du programme de contrôle (4 800 lignes de code commentaires inclus), et d'un simulateur Tcl/Tk de l'environnement physique. Cet environnement comprend un réservoir, des pompes pour alimenter le réservoir en eau, ainsi que des capteurs pour surveiller l'état de chaque pompe (ouverte ou fermée), le niveau d'eau dans le réservoir et la quantité de vapeur sortante.

La défaillance la plus critique est l'explosion de la chaudière, qui survient lorsque le niveau d'eau est en-dehors d'un intervalle de sécurité (c'est-à-dire, le niveau est trop haut ou trop bas par rapport à des seuils prédéfinis). Le programme de contrôle surveille et régule le niveau d'eau de façon à toujours rester dans l'intervalle autorisé. Lorsque des fautes physiques affectent des pompes ou des capteurs, le programme doit identifier les dispositifs défaillants. Il décide alors soit de continuer à opérer en mode dégradé, soit de lancer un arrêt d'urgence.

Notre objectif était de chercher des scénarios de faute conduisant à l'explosion de la chaudière. La Figure III-a montre la plateforme expérimentale utilisée. Le programme de contrôle opère en boucle fermée avec le simulateur. Les tests consistent à perturber le fonctionnement du système, en envoyant des commandes d'injection de faute. Par exemple, la commande `Faute_Vapeur(3)` simule la défaillance du capteur de vapeur à partir du cycle *3* d'exécution. Dans la construction de scénarios, nous considérons comme "dangereuses" les situations où le programme de contrôle base ses décisions sur une vue erronée de son environnement physique. Ceci inclut des situations telles que :

- La non détection d'une défaillance de capteur ou actionneur,
- La détection à tort de la défaillance d'un dispositif fonctionnant correctement,
- Une mauvaise estimation du niveau d'eau (le niveau d'eau dans le réservoir est en-dehors de l'intervalle calculé par le programme).

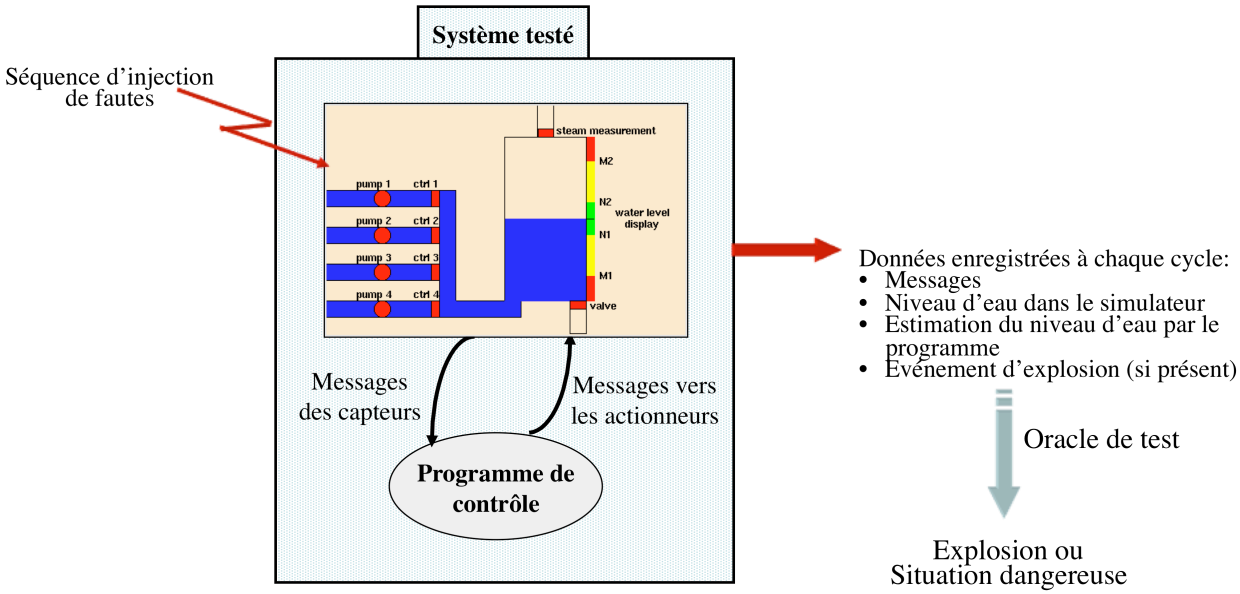


Figure III-a Plateforme de test de la chaudière

L'atteinte d'une situation dangereuse est alors un objectif de test intermédiaire pour l'étape 1 de notre stratégie. Par exemple, les expériences de test à cette étape révèlent que le programme de contrôle est incapable de détecter une défaillance du capteur de vapeur survenant au cycle 3 (à ce cycle, le programme quitte son mode transitoire d'initialisation). Nous retenons le scénario dangereux `Faute_Vapeur(3)` comme préfixe des explorations à effectuer à l'étape 2. Nous obtenons finalement un scénario d'explosion :
`Faute_Vapeur(3); Faute_Pompe2(4); Faute_Capteur_Eau(8).`

Dans [ATW 03a], le principe de la construction incrémentale de scénarios est illustré en prenant l'heuristique d'exploration la plus simple : la recherche aléatoire. Cette heuristique s'est avérée étonnamment efficace pour la chaudière, puisque nous avons pu trouver quatre classes différentes de scénarios d'explosion[10]. On ne peut cependant pas toujours s'attendre à une telle efficacité, d'où notre intérêt pour des techniques d'exploration plus sophistiquées.

Le choix du recuit simulé par rapport à d'autres techniques candidates, comme les algorithmes génétiques, se justifie par les raisons suivantes :

- Les algorithmes génétiques génèrent une population de solutions (c'est-à-dire un ensemble de séquences de test dans notre cas) qu'ils font évoluer sur plusieurs générations. Au contraire, le recuit simulé travaille sur une solution à la fois : il nous a semblé que le nombre total d'exécutions de test serait moins élevé. C'était un critère important pour nous, car l'application de séquences de test au système {programme + simulateur} est très coûteuse en temps (ici, 5s par cycle).
- Les algorithmes génétiques font jouer deux mécanismes d'évolution : la mutation et le croisement. L'efficacité de ce dernier dépend de la capacité à combiner judicieusement les caractéristiques de parents différents, pour produire des descendants de bonne qualité. Nous avons jugé intuitivement plus facile de nous concentrer sur des opérateurs tels que la mutation, qui produisent des "voisins" d'une solution candidate. Le recuit simulé est un exemple typique de métaheuristique basée sur la notion de voisinage.

[10] A notre connaissance, une seule classe de scénarios avait été identifiée dans la littérature. Ces scénarios ne sont pas dus à des *bugs* dans le programme, mais à des fautes de spécification dans le cahier des charges logiciel.

Plusieurs expérimentations du recuit simulé on été réalisées sur notre étude de cas, correspondant à différents choix d'implémentation. J'introduis ces choix ci-dessous, ainsi que les résultats obtenus.

III.1.2. Mise en œuvre du recuit simulé

La Figure III-b donne l'algorithme général du recuit simulé, pour une recherche dans un espace S de solutions. On veut une solution $s \in S$ qui minimise une fonction coût $f : S \rightarrow \mathbb{R}$. Après initialisation avec une première valeur de s, on cherche une nouvelle solution s' dans le voisinage V de la solution courante. Une solution qui améliore le coût ($\delta \leq 0$) est toujours retenue. Un mouvement vers une solution moins bonne ($\delta > 0$) est possible, mais avec une certaine probabilité qui dépend de δ et d'un paramètre t appelé température. La température décroit durant la recherche selon un processus de refroidissement R, réduisant progressivement cette probabilité. On itère jusqu'à ce que la condition d'arrêt de la recherche soit vraie (objectif atteint, nombre maximal d'itérations atteint, …).

```
Sélectionner une solution initiale s
Sélectionner une température initiale t = t0 >0
Initialiser le nombre d'itérations i = 1
TANT QUE (condition d'arrêt = Faux)
        Sélectionner aléatoirement une solution s' ∈ V(s)
        δ = f(s') – f(s)
        SI (δ≤0) ALORS
                s = s'
        SINON
                Générer uniformément x dans [0, 1]
                SI (x < exp(-δ/t)) ALORS s = s' FIN SI
        FIN SI
        i = i+1
        t = R(t,i)
FIN TANT QUE
```

Figure III-b Algorithme du recuit simulé

Dans l'étude de cas de la chaudière, l'espace S est un ensemble de séquences de test, où chaque séquence contient des commandes d'injection de faute datées. La définition précise de S (fenêtre temporelle des injections, préfixe…) dépend de l'étape de test considérée, plusieurs espaces pouvant être explorés à une étape (voir [ATW 03a]). Etant donné S, l'implémentation du recuit simulé requiert de régler les paramètres suivants :

- La fonction coût f – La forme générale du coût est : *SI (explosion ou situation dangereuse) ALORS retourner 0 SINON ...* La difficulté réside dans la détermination des coûts non nuls. Comment évaluer les séquences n'entrainant ni une explosion de la chaudière, ni l'atteinte d'une situation dangereuse ? Nous avons essayé plusieurs possibilités, prenant en compte l'incertitude sur le niveau d'eau (amplitude de l'intervalle calculé par le programme), le nombre de dispositifs défaillants, la distance par rapport aux seuils de sécurité, et des combinaisons pondérées de ces éléments.
- Le voisinage V – Il nous a semblé raisonnable de considérer des séquences voisines différant d'une seule commande d'injection (ajout, suppression, changement de date).

- Le processus de refroidissement *R* – Deux exemples classiques sont un refroidissement géométrique $t_{i+1} = \alpha\, t_i$ (avec α proche de 1.0), et le refroidissement de Lundy & Mees [LM 86] $t_{i+1} = t_i /(1+ \beta\, t_i)$ (avec β proche de zéro). Nous avons expérimenté les deux. Dans chaque cas, le calibrage des valeurs α (resp. β) et t_0 nécessite des exécutions préliminaires avec un échantillon de séquences de test.
- La condition d'arrêt – La recherche est stoppée lorsque l'objectif est atteint (*f(s) = 0*) ou qu'un nombre *MAX_ITER* d'itérations est atteint. Nous avons essayé *MAX_ITER =100* et *200.* Ces nombres d'itérations sont petits par rapport à ce que nous avons pu voir dans la littérature, où on laisse beaucoup plus de temps à l'algorithme pour converger vers un optimum. Dans notre cas, nous sommes très contraints par les temps d'exécution élevés des séquences de test.

Notre expérience est que tous ces réglages requièrent un effort important.

III.1.3. Des premiers résultats (très) décevants

En dépit de l'effort investi, les performances du recuit simulé se sont avérées décevantes pour l'étude de cas de la chaudière. Nous n'avons pas réussi à apporter d'amélioration significative par rapport à la recherche aléatoire. Pour certains des espaces explorés, la forte densité des solutions à coût nul explique ce résultat : aucune technique sophistiquée ne peut alors concurrencer l'efficacité et le faible coût d'une simple recherche aléatoire. Néanmoins, dans des cas moins triviaux, on aurait espéré de meilleures performances du recuit simulé.

Pour illustrer notre propos, prenons l'exemple d'un espace exploré à l'étape 2 de la stratégie de test. Le Tableau III-A compare les résultats de la recherche aléatoire et du recuit simulé sur cet espace. Les expériences impliquent 35 exécutions du processus de recherche selon chaque technique. Une recherche est réussie lorsqu'elle trouve une solution à coût nul. On stoppe alors le processus de recherche à l'itération correspondante. Une recherche infructueuse est stoppée à MAX_ITER = 200. Le Tableau III-A montre à la fois le nombre de succès et le nombre total d'itérations pour les 35 tentatives. Les deux implémentations du recuit simulé ne diffèrent que par la fonction coût utilisée, tous les autres paramètres ont le même réglage.

Les résultats montrent bien l'impact des réglages du recuit simulé sur les performances obtenues. On voit ici *a posteriori* que le coût 2 constitue un plus mauvais choix d'implémentation que le coût 1. Pourtant, les deux formulations de coût nous semblaient également raisonnables. Rien ne laissait prévoir la supériorité du coût 1, et cette supériorité reste intuitivement inexplicable après observation des résultats.

Nos essais pour jouer sur le réglage de la fonction coût restent peu satisfaisants. On voit que la meilleure implémentation du recuit simulé n'a pas plus de réussites que la recherche aléatoire. Si on prend en compte le taux de succès par itération, le recuit simulé peut sembler légèrement plus efficace (25/3778 contre 26/4187), mais cette "amélioration" n'est pas statistiquement significative : un test montre que l'hypothèse d'égalité ne peut être rejetée. Les résultats obtenus ne justifient pas l'effort mis en œuvre pour implémenter le recuit simulé.

Les difficultés que nous avons rencontrées montre que le réglage des paramètres peut être un point dur pour l'utilisation du recuit simulé. Une procédure ad hoc, basée sur des essais successifs et guidée par l'intuition, s'est avérée ici insuffisante. Nous aurions besoin de moyens objectifs pour caractériser les "bons" choix d'implémentation.

	Recherche aléatoire	Recuit simulé coût 1	Recuit simulé coût 2
Recherche réussie	26	25	16
Nombre total d'itérations (pour les 35 tentatives)	4187	3778	4453

Tableau III-A Exemple de résultats expérimentaux décevants

Certains travaux sur les métaheuristiques de recherche permettent d'aborder cette question par une approche quantitative. Plusieurs mesures ont été proposées pour mettre en relation les problèmes d'optimisation et le comportement des heuristiques appliquées à ces problèmes. Nous avons décidé d'étudier le pouvoir prédictif de certaines de ces mesures dans le cadre d'un problème de test.

III.2. Mesures pour l'étude comportementale des métaheuristiques

La thèse de Mériema Belaidouni [Bel 01] donne un bon point d'entrée pour les travaux sur l'étude du comportement des métaheuristiques. Elle propose une classification des mesures selon trois grandes catégories, correspondant à la structure que l'on cherche à caractériser. Les trois structures sont :

- L'*espace de recherche*, défini par un couple (S, f) où S est l'espace des solutions est f une fonction coût associant un nombre réel à chaque point de l'espace ;
- Le *paysage de recherche*, défini par un triplet (S, f, V) où V est un opérateur de voisinage connectant chaque point de S à un ensemble non vide de points voisins ;
- Le *paysage de processus*, défini par un quadruplet (S, f, V, ϕ) où ϕ est un processus de recherche basé sur la notion de voisinage (par exemple, ϕ est une implémentation du recuit simulé).

Le Tableau III-B donne une vue d'ensemble des mesures selon cette classification.

Espace de recherche	• Taille de l'espace • Nombre de solutions optimales • Intervalle de coûts *[fmin, fmax]* • Distribution des coûts (*DOS = density of states*)
Paysage de recherche	• Diamètre D (distance maximale entre deux points) • Variation des coûts à distance fixe : autocorrélation ρ_d et mesures apparentées • Variation des distances à coût fixe, notamment mesures de neutralité • Variation conjointe coût et distance, notamment *FDC (Fitness Distance Correlation)* • Nombre et disposition spatiale des optima locaux et globaux
Paysage de processus	• Distribution des coûts vus par le processus (*PCD = process cost density*) • Nombre et taille des bassins d'attraction

Tableau III-B Caractérisation des structures associées à un problème de recherche

La structure la plus étudiée est le paysage de recherche. Par rapport à l'espace de recherche, elle introduit l'opérateur de voisinage. Cela permet de définir la distance entre deux points *s* et *s'* de *S* comme le nombre minimal d'applications de *V* pour passer de *s* à *s'*. Les différentes mesures du paysage de recherche s'appuient sur cette notion de distance. Certaines sont faciles à évaluer analytiquement (le diamètre *D*), mais la plupart nécessitent des techniques expérimentales qui peuvent être très coûteuses en temps. C'est par exemple le cas des mesures liées à la disposition spatiale des optima, puisqu'il faut d'abord trouver un échantillon représentatif de ces optima.

La structure la plus complète est le paysage de processus. Sa caractérisation est expérimentale et nécessite des exécutions de ϕ. Certaines mesures définies pour l'espace de recherche et le paysage de recherche sont ainsi modifiées pour prendre en compte le comportement du processus. Par exemple, la distribution des coûts vus par le processus (PCD) est l'analogue de la distribution des coûts dans l'espace de recherche (DOS), mais en utilisant ϕ comme technique d'échantillonnage. DOS et PCD ne sont pas identiques, car ϕ introduit un biais dans l'exploration des points de *S*. De même, le nombre et la taille des bassins d'attraction pour ϕ dépendent de la disposition spatiale des optimaux locaux et globaux dans le paysage de recherche, mais le lien n'est pas trivial. En pratique, l'étude des bassins d'attraction s'effectue généralement avec un processus ϕ simple, l'heuristique de descente. Pour des métaheuristiques sophistiquées, un problème récurrent est la variabilité du comportement de ϕ d'une exécution à l'autre. Il peut alors être très difficile d'obtenir des mesures statistiquement significatives. De fait, dans la littérature, on trouve peu de travaux sur le paysage de processus comparé aux travaux sur le paysage de recherche.

Toutes les mesures ne sont pas pertinentes pour nos besoins. Prenons l'exemple de DOS et PCD. Ces mesures sont utiles pour comparer différents réglages, lorsque l'on peut se satisfaire de solutions sub-optimales. Dans ce cas, un espace de recherche sera considéré comme d'autant meilleur que la distribution des coûts a une moyenne faible et une variance élevée : il est alors facile de produire des solutions de coût faible, et les très faibles coûts s'écartant de la moyenne ne sont pas rares. Similairement, on considérera qu'un paysage de processus est meilleur qu'un autre s'il a tendance à générer des coûts plus faibles. Dans notre cas, ces considérations ne s'appliquent pas car seules les solutions à coût nul nous intéressent. Nous voulons atteindre un objectif de test, par exemple l'explosion de la chaudière ou l'atteinte d'une situation dangereuse. Des solutions à coût faible, mais n'atteignant pas l'objectif de test, ne sont pas satisfaisantes (la même remarque s'appliquerait au cas d'un objectif de test structurel, lié à l'activation d'une branche particulière d'un programme).

D'autres mesures seraient pertinentes, mais restent hors de portée du fait du coût prohibitif de leur évaluation expérimentale. Rappelons que nous sommes contraints par le temps d'exécution du système sous test. Au final, nous retenons deux mesures du paysage de recherche : le diamètre et l'autocorrélation.

Le *diamètre D* mesure la distance maximale entre deux points. Intuitivement, un petit diamètre devrait être plus favorable qu'un diamètre élevé, car tout point du paysage est atteignable plus rapidement (par applications successives de *V*). Si le diamètre est élevé, l'atteinte d'une solution optimale peut nécessiter un grand nombre d'itérations, même dans le cas idéal où le processus de recherche sélectionnerait la trajectoire la plus courte depuis le point de départ. Des exemples de calcul de diamètre pour des problèmes combinatoires NP complets sont donnés dans [AZ 00].

L'*autocorrelation* ρ_d a été introduite par [Wei 90] pour caractériser la rugosité d'un paysage. Intuitivement, un paysage lisse est tel que des points voisins ont des coûts similaires, alors qu'un paysage rugueux présentera un profil local très "accidenté". Ce dernier cas est défavorable pour une recherche basée sur le voisinage, et suggère de changer les définitions de *f* ou *V* pour que la recherche soit mieux guidée. L'autocorrélation ρ_d mesure la variation des coûts de points à distance *d*, et le plus souvent on se concentre sur $d = 1$. L'évaluation expérimentale de ρ_1 s'effectue par une marche aléatoire dans le paysage (voir Figure III-c). Une valeur obtenue proche de zéro indique que les coûts de points voisins sont indépendants, une valeur proche de 1.0 est le cas le plus favorable. Bien que l'autocorrélation n'apporte qu'une information limitée sur la structure du paysage de recherche, elle reste une des mesures les plus reconnues. Des exemples de travaux utilisant cette mesure ou des mesures apparentées sont [SS 92, Hor 96, AZ 98, AZ 00].

Sélectionner un point de départ s_0
Mémoriser $f(s_0)$
POUR $i=1$ à $T-1$ FAIRE
    Sélectionner aléatoirement $s_i \in V(s_{i-1})$
    Mémoriser $f(s_i)$
FIN POUR

(1) Algorithme de la marche aléatoire

$$\hat{\rho}_1 = \frac{\sum_{i=0}^{T-2}\left(f(s_i)-\overline{f}\right)\left(f(s_{i+1})-\overline{f}\right)}{\sum_{i=0}^{T-1}\left(f(s_i)-\overline{f}\right)^2}$$

Où T est la taille de l'échantillon, et $\overline{f}$ la moyenne arithmétique de l'échantillon

(2) Estimateur de ρ_1

Figure III-c Evaluation expérimentale de ρ_1 [Hor 96]

III.3. Etude du pouvoir prédictif du diamètre et de l'autocorrélation

Nous avons étudié l'utilisation du diamètre et de l'autocorrélation pour identifier les paysages potentiellement adaptés à la recherche par recuit simulé. Le principe est de jouer sur les définitions de *f* et *V* pour obtenir un réglage tel que :

- Le diamètre *D* soit significativement inférieur au nombre maximal d'itérations autorisées (*MAX_ITER*).
- L'autocorrélation ρ_1 soit élevée.

Le critère du diamètre vise à assurer que la recherche aura une chance d'atteindre une solution optimale dans le temps imparti, quel que soit le point de départ. Le critère de l'autocorrélation vise à sélectionner des réglages de *f* et *V* tels que le principe d'une recherche dans le voisinage soit pertinent.

Notre objectif était de revisiter l'étude de cas de la chaudière pour déterminer si ces deux critères permettaient un réglage plus efficace du recuit simulé. Avant cela, il nous a paru nécessaire d'expérimenter nos critères sur des exemples plus simples de problèmes.

Le premier exemple (§III.3.1) est extrait de la littérature générale sur les métaheuristiques de recherche. Le problème de l'affectation quadratique (QAP) est un problème NP complet sur lequel plusieurs techniques de recherche ont été expérimentées [Con 90, Tai 91, MDC 95, MF 00]. La recherche par recuit simulé est efficace pour ce problème, pour lequel on connaît un opérateur de voisinage bien adapté (*f* est déjà déterminée par le problème, seul *V* reste à choisir). Le QAP fournit donc un exemple de ce qui est considéré comme un "bon" paysage dans la littérature, nous le retenons à titre de comparaison.

Le deuxième exemple (§III.3.2) s'inspire d'un problème de calendrier traité par des travaux sur la génération de test [Tra 00]. Nous avons défini deux instances artificielles (*Cal1* et *Cal2)* d'un problème de recherche dans un espace de dates. Ces instances sont utilisées pour effectuer des expériences contrôlées. Nous faisons varier les réglages de f et V, et nous étudions la correspondance entre (i) les mesures de D et ρ_1, et (ii) l'efficacité du recuit simulé avec ces réglages. Ces expériences nous permettent ainsi d'étudier le pouvoir discriminant de nos critères pour identifier les "bons" et les "mauvais" réglages.

Notre retour sur le test de la chaudière est présenté dans un dernier paragraphe (§III.3.3).

III.3.1. Analyse du paysage de recherche d'une instance de QAP

On trouvera dans [AZ 00] une définition du problème de l'affectation quadratique, ainsi qu'une présentation de l'opérateur de voisinage adapté à ce problème (*2-exchange neighborhood*). Nos expériences ont porté sur l'instance NUG15 du problème, initialement définie dans [NVR 68][11]. Il y a seulement quatre solutions optimales dans un espace de taille 15! (soit environ 1.3 10^{12} points), ce qui rend la recherche aléatoire uniforme inefficace.

Nous avons considéré les deux implémentations du recuit simulé étudiées dans [Con 90] : LM1 est le recuit simulé avec le processus de refroidissement de Lundy & Mees (voir §III.1.2) ; LM2 est une variante de l'algorithme de base du recuit simulé qui s'avère plus efficace sur plusieurs instances de QAP, dont NUG15. Nous prenons *MAX_ITER* = 5000 itérations, pour garder le même ordre de grandeur que [Con 90] (5250 itérations pour l'instance NUG15).

Dans la Figure III-d.1, l'évaluation de l'autocorrélation est faite sur 100 exécutions de la marche aléatoire, avec une longueur *T=1000*.

Nos expériences confirment les performances attendues du recuit simulé, avec une supériorité de LM2 par rapport à LM1 (Figure III-d.2). Alors que la recherche aléatoire ne trouve aucune solution optimale, la recherche par recuit simulé est efficacement guidée par la structure du paysage sous-jacent. L'autocorrélation mesurée sur ce paysage (0.74) donne une idée de ce que l'on peut considérer comme satisfaisant lors des réglages du recuit simulé.

- Diamètre D = 14 (<< 5000)
- Autocorrélation $\hat{\rho}_1$ = 0.74 (écart type 0.01)

(1) Mesures du paysage

	Recherche aléatoire	LM1	LM2
% réussites (1000 tentatives, MAX_ITER = 5000)	0%	8%	11.1%

(2) Performances de la recherche

Figure III-d Evaluation expérimentale sur l'instance NUG15

III.3.2. Analyse des paysages de recherche de deux problèmes de calendrier

Cal 1 et Cal 2 sont deux problèmes artificiels de recherche dans un espace de 401 865 dates. Ils ont respectivement une et deux solutions optimales. Pour chaque problème, nous avons considérés 24 paysages de recherche, correspondant à la combinaison de quatre fonctions coût et six opérateurs de voisinages.

[11] On peut trouver une archive des instances du QAP étudiées dans la littérature sur la page d'Eric Taillard : http://mistic.heig-vd.ch/taillard/

Le coût 1 affecte une valeur de coût aléatoire à chaque point de l'espace. On s'attend à obtenir des paysages inadéquats pour le recuit simulé, quel que soit l'opérateur de voisinage choisi. Le coût 2, ainsi que les voisinages V1 et V2, s'inspirent des travaux de Tracey [Tra 00] sur la génération de test : leur combinaison devrait donner de bons résultats. Les coûts 3 et 4 correspondent à un autre mode de calcul, lié au nombre de jours entre la date candidate et l'optimum le plus proche. Ces coûts devraient bien guider le processus de recherche si on les associe aux opérateurs V3-V6, qui considèrent un voisinage de la forme *[date candidate ± nb jours max]* (avec différentes taille d'intervalles, et différentes façon de traiter les dates aux frontières de l'espace de recherche).

Le Tableau III-C montre le diamètre associé aux différents opérateurs de voisinage. Les valeurs en gras sont celles acceptées par le critère de comparaison avec MAX_ITER = 2000 ; les valeurs en italiques sont les valeurs rejetées par le critère. La même convention (gras = valeur acceptée, italique = rejetée) est utilisée dans le Tableau III-D, pour les valeurs d'autocorrélation mesurées sur les 24 paysages de Cal1. Sans surprise, tous les paysages avec un coût aléatoire sont rejetés. De plus, la mesure de l'autocorrélation nous apprend que V3-V6 ne sont pas adéquats pour le coût 2, et que V1-V2 – initialement conçus pour aller avec le coût 2 – peuvent être associés aux coûts 3 et 4.

Le Tableau III-E montre le pourcentage de recherches réussies pour les paysages de Cal1 acceptés par les deux critères précédents (en gras) ou rejetés par au moins un des critères (en italique). On voit que les 14 paysages rejetés correspondent exactement aux 14 plus mauvais scores du recuit simulé. Ce résultat est encourageant quant à la pertinence des mesures considérées. Leur pouvoir discriminant est cependant limité : les 10 paysages retenus de Cal1 fournissent certes les meilleurs scores, mais ces scores exhibent une forte disparité (de 6.3% à 65.1%) que ne laissaient pas prévoir les valeurs de diamètre et d'autocorrélation. Notons que le plus mauvais score (6.3%) reste significativement plus élevé que celui attendu d'une recherche aléatoire uniforme (0.5%). Le paysage correspondant n'est donc pas inadéquat pour le recuit simulé, même si d'autres réglages permettraient des performances encore meilleures.

Des résultats similaires sont obtenus pour Cal2 : correcte élimination de "mauvais" paysages, limitations pour discriminer les paysages restants.

	V1	V2	V3	V4	V5	V6
Diamètre	**550**	**1100**	*6700*	*13400*	**250**	**500**

Tableau III-C Diamètre des paysages de Cal1 et Cal2

	V1	V2	V3	V4	V5	V6
Coût1	*0 (0)*	*0 (0)*	*0 (0)*	*0 (0)*	*0 (0)*	*0 (0)*
Coût2	**0.82 (0.08)**	**0.95 (0.04)**	*0.22 (0.09)*	*0.22 (0.09*	*0.36 (0.19)*	*0.33 (0.17)*
Coût3	**0.99 (<10^{-2})**	**0.99 (<10^{-2})**	**0.89 (<10^{-2})**	**0.89 (<10^{-2})**	**0.98 (0.02)**	**0.98 (0.01)**
Coût4	**0.99 (<10^{-2})**	**0.99 (<10^{-2})**	**0.99 (<10^{-2})**	**0.99 (<10^{-2})**	**0.99 (<10^{-2})**	**0.99 (<10^{-2})**

Tableau III-D Autocorrélation des paysages de Cal1 : moyenne (écart type) sur 100 marches aléatoires, T= 1000

	V1	V2	V3	V4	V5	V6
Coût1	*0%*	*0%*	*0%*	*0%*	*0.2%*	*0.2%*
Coût2	**6.3%**	**16.1%**	*0%*	*0%*	*1.4%*	*1.4%*
Coût3	**47.6%**	**65.1%**	*2.1%*	*2.1%*	**16.9%**	**16.8%**
Coût4	**60.8%**	**9.5%**	*3.7%*	*3.7%*	**40.3%**	**40.2%**

Tableau III-E Pourcentage de réussites du recuit simulé pour les paysages de Cal1 (1000 tentatives, MAX_ITER = 2000)

III.3.3. Analyse du paysage de recherche de la chaudière

Contrairement aux études de cas précédentes, le test de la chaudière est un problème pour lequel la recherche aléatoire uniforme est plutôt efficace. L'espace de recherche le plus difficile est celui mentionné dans le paragraphe III.1.3, où la recherche aléatoire aboutit 26 fois à une solution à coût nul pour un total de 4187 itérations (voir Tableau III-A). On se concentre ici sur le problème de test correspondant. En ce qui concerne le recuit simulé, nous avions observé que le choix de la fonction coût avait un fort impact sur les performances de la recherche, mais que la meilleure fonction coût trouvée n'apportait pas d'amélioration significative par rapport à la recherche aléatoire.

La taille de l'espace de recherche est de 10 077 695 points, et on laisse *MAX_ITER* = 200. Commençons par caractériser les deux espaces de recherches correspondant respectivement aux coûts 1 et 2 du Tableau III-A :

- $D = 9$, ce qui est acceptable pour *MAX_ITER* = 200.
- L'autocorrélation est estimée sur dix marches aléatoires de longueur $T = 100$. Pour le coût1, on obtient $\hat{\rho}_1 = 0.52$ (écart type 0.17); pour le coût 2, $\hat{\rho}_1 = 0.53$ (écart type 0.13). Ces valeurs ne permettent pas de discriminer les deux fonctions coût candidates. Dans chaque cas, les coûts de solutions voisines ne sont pas décorrélés, mais l'autocorrélation mesurée est plus faible que celles acceptées pour les paysages de QAP, Cal1 et Cal2 (nous avions toujours $\hat{\rho}_1 \geq 0.74$). Ainsi, ces résultats ne révèlent pas une inadéquation flagrante des paysages de recherche, mais suggèrent de chercher une fonction coût avec une autocorrélation plus élevée.

Plutôt que d'imaginer une fonction coût entièrement nouvelle, nous avons repris les fonctions existantes, définies comme des combinaisons pondérées d'éléments de coût. Le calibrage des différents poids avait été effectué par quelques essais avec des valeurs jugées raisonnables. Nous pouvons maintenant reprendre ce calibrage en adoptant une approche plus systématique : nous faisons varier les poids et retenons les valeurs qui donnent l'autocorrélation la plus élevée.

A partir du coût 1, le meilleur calibrage donne $\hat{\rho}_1 = 0.62$ (écart type 0.14) : soit $\text{Coût1}_{\text{bis}}$ la fonction résultante. A partir du coût 2, nous n'avons pu obtenir aucune amélioration. Cela pourrait vouloir dire que la définition générique de cette fonction n'est pas adaptée à l'opérateur de voisinage utilisé. Regardons maintenant si $\text{Coût1}_{\text{bis}}$ permet d'améliorer le recuit simulé. Le Tableau III-F donne les résultats obtenus sur 35 exécutions de la recherche. Les résultats de la recherche aléatoire et du recuit simulé avec les anciens coûts sont rappelés à titre de comparaison. On voit que le paysage avec $\text{Coût1}_{\text{bis}}$ est le plus efficace pour la recherche par recuit simulé. De plus, les performances observées sont maintenant meilleures que celles de la recherche aléatoire : un test de χ^2 conduit à rejeter l'hypothèse d'égalité des taux de réussite par tentative, ainsi que par itération, avec un seuil de risque de 2%. Si l'on

considère notre échec initial pour trouver un réglage adéquat du recuit simulé, ces résultats confirment que l'approche quantitative devrait être bien plus fiable qu'un jugement *ad hoc*.

	Recherche aléatoire	Recuit simulé Coût1	Recuit simulé Coût2	**Recuit simulé $\mathbf{Coût1_{bis}}$**
Recherche réussie	26	25	16	**33**
Nombre total d'itérations (pour les 35 tentatives)	4187	3778	4453	**2573**

Tableau III-F Amélioration du recuit simulé pour l'exemple de la chaudière

III.4. Critères d'arrêt pour la recherche

Jusqu'à présent, nous avons considéré des mesures du paysage de recherche. Ces mesures ne prennent pas en compte la dynamique du *processus* de recherche. Malheureusement, comme expliqué précédemment (§III.2), la caractérisation du paysage de processus est problématique. Outre le coût expérimental des mesures existantes, la variabilité du comportement observé d'une exécution à l'autre rend difficile l'obtention de résultats exploitables.

Pour améliorer la dynamique de la recherche, notre proposition est de mettre en place un suivi en-ligne des exécutions du processus ϕ retenu. Il s'agit donc de caractériser non pas le paysage de processus, mais un parcours dans ce paysage. L'idée est de détecter les exécutions dont la convergence vers un optimum est trop lente par rapport au nombre d'itérations alloué. On stoppe alors ces exécutions pour relancer la recherche sur une autre graine. Notons que pour un paysage de processus donné, cette approche ne diminue pas la difficulté de la recherche. Elle vise seulement à éviter certaines itérations inutiles.

D'autres auteurs ont étudié des critères d'arrêt basés sur la surveillance de la convergence [OVW 98]. Leur approche est cependant différente de la nôtre, car il s'agit pour eux d'analyser la convergence d'une population vers un ensemble d'optima locaux et globaux, dans le cadre d'algorithmes génétiques.

III.4.1. Approche proposée

L'approche se base sur une nouvelle mesure, le Taux de Génération de Meilleures Solutions (TGMS) :

$$TGMS = \frac{nombre\ d'itérations\ améliorant\ le\ coût}{nombre\ total\ d'itérations}$$

Nous nous intéressons à l'évolution du TGMS au cours d'une exécution du recuit simulé. D'après nos observations sur les études de cas, une recherche réussie comporte typiquement trois phases (Figure III-e). La phase 1 caractérise un début de la recherche avec un coût courant facile à améliorer : de meilleures solutions candidates sont produites à un taux élevé. A la fin de la phase 1, des points à coût faible sont atteints, et l'amélioration devient difficile : la décroissance rapide du TGMS (phase 2) indique l'approche de l'optimum qui sera trouvé par cette exécution réussie. La phase 3 caractérise les dernières itérations, qui explorent un petit sous-espace dans le voisinage de l'optimum, jusqu'à le trouver. La Figure III-e est bien sûr très schématique. Dans les exemples réels d'exécution de la recherche, les durées respectives des phases varient, et certaines phases peuvent être absentes. Des exemples de recherches réussies sans phase 1 ou sans phase 3 sont montrées dans la Figure III-f.1.

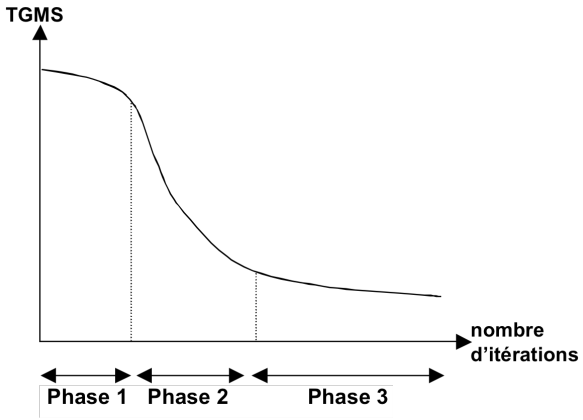
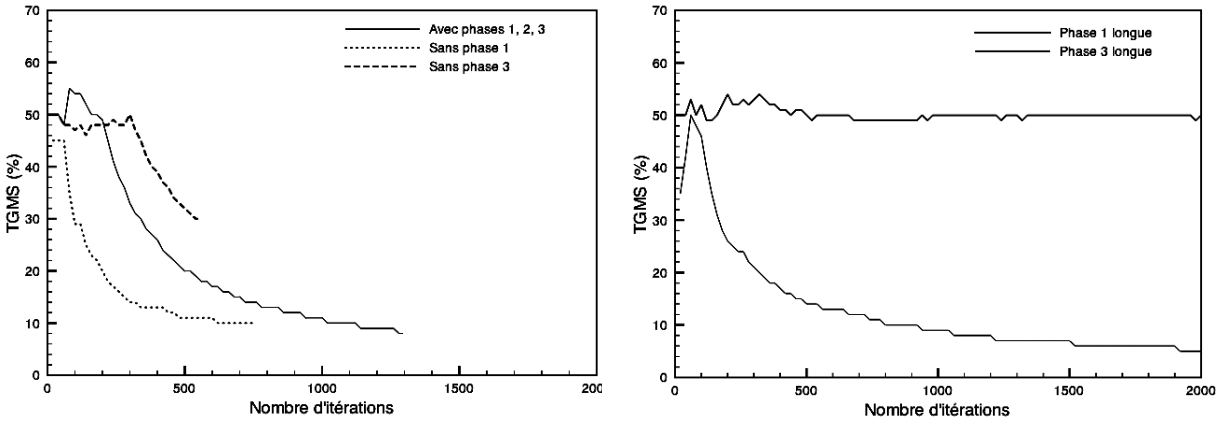


Figure III-e Allure typique de l'évolution du TGMS pour des exécutions réussies



(1) Exemples de recherches réussies **(2) Exemples de recherches en échec**

Figure III-f Exemples d'évolution du TGMS pour des exécutions du recuit simulé (paysage de Cal1 avec coût 4 et V6)

Nous avons observé que les durées des phases peuvent – dans une certaine mesure – aider à identifier les exécutions avec et sans succès. Les exécutions sans succès correspondent typiquement à des durées longues des phases 1 ou 3, où la qualification de "longue" se réfère au nombre maximal d'itérations alloué à la recherche. Des exemples concrets d'exécutions sans succès sont montrés dans la Figure III-f.2.

Une phase 1 longue indique que le processus de recherche améliore continûment le coût. Néanmoins, les itérations n'apportent que de petites améliorations, et à la fin de la recherche le coût courant est encore éloigné de l'optimum. Une phase 3 longue indique une stagnation de la recherche : on n'arrive plus à améliorer le coût, soit parce la recherche est piégée par un optimum local, soit parce que la région explorée forme un plateau sur lequel les mouvements sont neutres. Il serait souhaitable d'arrêter de telles exécutions, et d'utiliser les ressources restantes pour une nouvelle recherche.

Nous proposons deux critères d'arrêt complémentaires, basés sur la surveillance en-ligne du TGMS. Le premier critère (resp. le second) détermine si la phase 1 (resp. la phase 3) est trop longue pour espérer converger vers un optimum avant la fin des *MAX_ITER* itérations. Les évaluations sont effectuées en parallèle, et l'exécution est stoppée dès qu'un des deux critères devient vrai. La forme générique des critères d'arrêt est la suivante :

- phase 1 trop longue – *TGMS* ≥ *Seuil_Haut* pendant *max(α MAX_ITER, D)* itérations.
  La phase 1 est identifiée par une valeur de TGMS excédant un seuil haut. Une durée "trop longue" s'apprécie en fonction du nombre maximal d'itérations : le paramètre α ($0 < \alpha < 1$) permet un ajustage de la durée. De plus, il paraît raisonnable de ne pas stopper la phase 1 avant que le processus ait passé un minimum de temps à explorer le paysage. Dans la formulation proposée, le nombre d'itérations avant arrêt est au moins le diamètre *D* du paysage.
- phase 3 trop longue – *TGMS* ≤ *Seuil_Bas* pendant *max(β MAX_ITER, |V|)* itérations.
  De même que précédemment, la durée de la phase est ajustée au nombre maximal d'itérations, via un paramètre $0 < \beta < 1$. De plus, nous voulons laisser le temps à la recherche d'explorer le voisinage de la solution candidate atteinte à la fin de la phase 2. Ainsi, la phase 3 ne pourra pas être stoppée avant une durée correspondant à la taille |V| du voisinage (nombre maximum de voisins connectés à un point).

Ces critères ont été calibrés expérimentalement, en utilisant d'abord les études de cas QAP, Cal1 et Cal2, puis en confirmant les résultats sur l'étude de cas de la chaudière. En plus d'expérimenter différentes valeurs pour α, β et les seuils, nous avons considéré différents démarrages du suivi en-ligne du TGMS : soit à la première itération, soit après un nombre initial d'itérations. Dans tous les cas, nous avons rejoué les exécutions du recuit simulé qui avaient été effectuées lors de l'analyse des paysages de recherche (voir §III.3), mais cette fois avec les critères d'arrêt. Nous avons ainsi pu déterminer le pourcentage de recherches en échec arrêtées, ainsi que le pourcentage de recherches réussies arrêtées à tort. Nous avons également mesuré l'impact sur les performances globales de la recherche, en termes du taux *nombre de réussites / nombre total d'itérations*. Ce taux agrège les effets des faux positifs (recherches arrêtées à tort) et des faux négatifs (recherches en échec qui ne sont pas arrêtées).

III.4.2. Résultats expérimentaux

En considérant l'ensemble des études de cas, les implémentations des critères d'arrêt qui donnent les meilleurs résultats sont les suivantes :

Implémentation 1	Le suivi en-ligne du TGMS démarre après MAX_ITER/10 itérations Phase 1 trop longue : Seuil_Haut = 50%, $\alpha = 0.1$ Phase 3 trop longue : Seuil_Bas = 10%, $\beta = 0.1$
Implémentation 2	Le suivi en-ligne du TGMS démarre à la première itération Phase 1 trop longue : Seuil_Haut = 50%, $\alpha = 0.25$ Phase 3 trop longue : Seuil_Bas = 10%, $\beta = 0.25$

Les tableaux III-G et III-H donnent des exemples de résultats expérimentaux avec ces critères. Notons que les résultats sur QAP pourraient suggérer une supériorité de l'implémentation 2 : l'amélioration des performances globales est ici du même ordre que celle obtenue avec le processus de recherche LM2 proposé par [Con 90]. Cependant, cette supériorité n'est pas confirmée sur les autres études de cas. En considérant l'ensemble des résultats expérimentaux, nos conclusions sont les suivantes :

- Les deux implémentations arrêtent généralement un pourcentage satisfaisant des recherches en échec ;
- Malheureusement, elles arrêtent également des recherches qui auraient réussi. Ce sont typiquement des recherches qui trouvent l'optimum vers la fin des *MAX_ITER* itérations, de sorte que les performances globales ne sont en fait pas dégradées par un arrêt précoce.

- De façon générale, nous observons que les performances globales de la recherche sont soit améliorées soit inchangées, jamais dégradées.

Puisque nos critères d'arrêt ne dégradent pas les performances de la recherche, et peuvent l'améliorer dans certains cas, notre recommandation est de les utiliser systématiquement. Les implémentations 1 et 2 peuvent être indifféremment choisies. Notons cependant que l'arrêt et la relance de la recherche ne peuvent se substituer à un bon réglage du paysage de recherche sous-jacent (voir les résultats montrés pour la chaudière, Tableau III-H, où l'impact de la fonction coût est le seul significatif). Au mieux, le suivi du TGMS peut éviter de gaspiller l'effort de test quand la convergence est trop lente. L'adéquation – ou l'inadéquation – du paysage de recherche au recuit simulé reste inchangée.

	LM1			**LM2**		
	Sans critère	**Impl. 1 des critères**	**Impl. 2 des critères**	**Sans critère**	**Impl. 1 des critères**	**Impl. 2 des critères**
% recherches en échec stoppées	—	100%	100%	—	100%	100%
% recherches réussies stoppées	—	76.25%	50%	—	60.36%	18.92%
Réussites/itérations	$1.7\ 10^{-5}$	$1.9\ 10^{-5}$	$2.4\ 10^{-5}$	$2.4\ 10^{-5}$	$4.3\ 10^{-5}$	$5.5\ 10^{-5}$

Tableau III-G Critères d'arrêts appliqués au QAP

	Coût 1bis			**Coût 2**		
	Sans critère	**Impl. 1 des critères**	**Impl. 2 des critères**	**Sans critère**	**Impl. 1 des critères**	**Impl. 2 des critères**
nb recherches en échec stoppées	—	1 (sur 2)	1 (sur 2)	—	5 (sur 19)	5 (sur 19)
nb recherches réussies stoppées	—	2 (sur 33)	2 (sur 33)	—	0 (sur 16)	0 (sur 16)
Réussites/itérations	$12.8\ 10^{-3}$	$13.0\ 10^{-3}$	$13.3\ 10^{-3}$	$3.6\ 10^{-3}$	$4.0\ 10^{-3}$	$4.0\ 10^{-3}$

Tableau III-H Critères d'arrêts appliqués à la chaudière

III.5. Conclusion et discussion

Les études empiriques contribuent au développement d'un corpus de connaissances sur l'application de métaheuristiques à des problèmes de test. L'étude présentée dans ce chapitre porte sur l'utilisation de mesures pour guider le paramétrage d'une métaheuristique basée sur la notion de voisinage. Ces mesures sont traditionnellement utilisées dans le cadre de problèmes combinatoires liés à la recherche opérationnelle ou à la théorie de la complexité des algorithmes. Nous avons montré leur intérêt dans le cadre d'un problème de test.

Les deux mesures reprises de la littérature, le diamètre et l'autocorrélation, ont un pouvoir prédictif limité. Elles s'avèrent tout de même utiles pour identifier des paysages inadéquats, et rejaillir sur les définitions de la fonction coût et de l'opérateur de voisinage. Dans le cadre de notre étude, nous avons pu revisiter avec succès l'exemple de la chaudière, et trouver une meilleure fonction coût que celle obtenue par nos essais initiaux. Ce résultat est très encourageant au vu des difficultés que nous avions rencontrées dans le paramétrage du recuit simulé : malgré tous nos efforts, nous avions échoué à trouver une fonction coût adéquate.

La méthodologie suivie pour transférer les deux mesures à notre problème de test a été la suivante. Dans un premier temps, nous avons repris une instance de problème représentative

du cadre d'utilisation classique de ces mesures (ici, une instance de QAP). Outre l'intérêt de retenir cet exemple à titre de comparaison, ceci nous a permis de vérifier que nous comprenions bien la mise en œuvre de la mesure, et que nous pouvions reproduire des résultats cohérents avec ceux publiés dans la littérature. Dans un deuxième temps, nous avons considéré des problèmes artificiels inspirés de problèmes de test, sur lesquels nous pouvions effectuer des expériences contrôlées. L'utilisation de problèmes artificiels simples, dont on peut contrôler les propriétés, est classique dans la littérature générale des métaheuristiques. Un exemple bien connu est constitué des paysages NK [Kau 89], utilisés dans le cadre des algorithmes génétiques. Il est à noter que notre exemple Cal1 a été réutilisé par certains auteurs [LI 08] pour étudier d'autres types de fonction coût, en reprenant l'approche par évaluation du diamètre et de l'autocorrélation. Enfin, ce n'est que dans un dernier temps que nous avons considéré le problème de la chaudière, bénéficiant ainsi des retours obtenus à partir des problèmes plus simples.

Le diamètre et l'autocorrélation caractérisent le paysage de recherche, qui n'inclut pas le processus de recherche. La caractérisation de paysages de processus restant un point dur, nos travaux pour prendre en compte la dynamique de la recherche ont suivi deux directions. La première, que je n'ai pas présentée dans ce chapitre, a consisté à modifier l'algorithme de base du recuit simulé. Après plusieurs essais, nous avons réussi à trouver une variante qui marche bien dans l'exemple de la chaudière [ATW 03b], mais qui n'est pas nécessairement réutilisable pour d'autres exemples : nous avons notamment observé des performances médiocres quand cette variante du recuit simulé est appliquée aux instances de QAP, Cal1 et Cal2. La deuxième direction est celle de la surveillance en-ligne du TGMS, avec arrêt et relance de la recherche lorsqu'un problème de convergence trop lente est détecté. Cette approche se veut moins spécifique que la précédente, est peu coûteuse à mettre en œuvre, et a donné des résultats stables sur l'ensemble des problèmes étudiés. Le gain en performance est cependant modeste, avec un effet neutre dans beaucoup de cas. D'autres études seraient nécessaires pour affiner l'analyse des évolutions typiques du TGMS pour les exécutions avec et sans succès.

Au-delà des résultats sur le diamètre, l'autocorrélation, et le TGMS, nos travaux montrent que la communauté du test peut s'inspirer des études existantes sur le comportement des métaheuristiques. Dans la littérature sur les métaheuristiques, les mesures proposées visent à répondre à deux objectifs long terme, énoncés notamment dans [Bel 01] :

1. Identifier des propriétés quantifiables pour caractériser et différencier les structures de problèmes ;
2. Déduire les implications de ces propriétés sur le comportement et les résultats des méthodes de recherche basées sur les métaheuristiques.

Ces objectifs sont ambitieux. Les résultats les plus aboutis concernent des paysages de recherche dits *élémentaires*, qui satisfont certaines propriétés mathématiques exploitables dans la mise en œuvre des métaheuristiques [RS 02, WSH 08]. Un exemple de paysage élémentaire est celui induit par la version symétrique du problème du voyageur de commerce. Dans la version asymétrique de ce problème, ou dans le cas de QAP, le paysage peut encore être vu comme la superposition d'un petit nombre de composantes élémentaires [RS 02]. Dans le cas général, les paysages sont cependant plus difficiles à caractériser, et il serait intéressant d'identifier les mesures les plus pertinentes pour des problèmes de test.

Nos travaux ont abordé l'analyse de paysages par le biais d'un problème de test spécifique : l'étude de cas de la chaudière, pour laquelle la fonction coût dépend de propriétés applicatives. Il serait profitable d'approfondir l'étude de mesures pour des problèmes de test génériques, comme le test structurel de branches. On peut alors définir des fonctions coût

applicables à toute instance du problème [VBL+ 10], et étudier la mise en œuvre des métaheuristiques dans un cadre réutilisable. Le test des branches a déjà suscité de nombreuses études empiriques et théoriques, en tant que domaine d'application de métaheuristiques (notamment récemment [Arc 09, VBL+ 10, HM 10]). Des liens plus étroits avec les approches quantitatives mentionnés plus haut semblent pertinents.

Chapitre IV. TERMOS : un langage de scénarios pour le test de systèmes mobiles

Les avancées technologiques du monde "sans fil" ont conduit au développement d'applications et services dédiés à l'informatique mobile (*mobile computing*). Celle-ci met en jeu des dispositifs (téléphones et ordinateurs portables, assistants numériques, véhicules intelligents) qui se déplacent dans le monde physique tout en se connectant aux réseaux par des moyens sans fil (IEEE 802.11, Bluetooth, …). Les systèmes ainsi formés sont des systèmes répartis caractérisés par de fréquentes connexions et déconnexions de nœuds, par l'établissement de réseaux ad hoc, et par un comportement fortement dépendant du contexte local environnant. Ces caractéristiques posent de nouveaux défis pour le test, aussi bien d'un point de vue technique que conceptuel.

D'un point de vue technique, la mise en œuvre du test suppose des plateformes expérimentales adaptées. On peut bien sûr chercher à effectuer le test avec des dispositifs réels évoluant dans le monde physique. Par exemple, dans [LNY 04] une application téléphonique est testée à l'aide de personnes se déplaçant dans une zone urbaine ; dans [BKK+ 04] une application de type véhicule-à-véhicule est testée avec trois véhicules sur une route. Cependant, il est évident que ces types de test sont limités. En pratique, une grande partie des tests devra être réalisée avec des moyens de simulation. Les moyens spécifiques aux systèmes mobiles incluent (i) des simulateurs réseau[12], permettant de simuler les communications sans fil et filaires, et (ii) des simulateurs de contexte[13], pour gérer le mouvement des dispositifs dans un environnement virtuel et produire les données contextuelles dépendantes de la localisation. Les simulateurs disponibles sont généralement issus de recherches sur des méthodes d'*évaluation quantitative*, mais peuvent également servir à des fins de *vérification* par test. Quel que soit le but recherché, une question inhérente à la simulation est cependant celle de la représentativité par rapport à un comportement réel. Les travaux de [CSS 02] ont montré que, selon le simulateur réseau utilisé, les résultats obtenus pouvaient diverger tant d'un point de vue qualitatif que quantitatif. Une explication était que les simulateurs implémentent différemment la couche physique, avec des niveaux de détail difficilement comparables. Une autre raison avancée était la possible présence de fautes de programmation affectant le code des simulateurs. [BGK+ 02] a également souligné la nécessité de mieux vérifier la correction du code des simulateurs, dont l'utilisation reste incontournable en pratique.

---

[12] Un exemple très utilisé est ns-2 : http://nsnam.isi.edu/nsnam/index.php/User_Information

[13] Deux exemples sont Ubiwise, construit à partir d'un moteur graphique de jeu vidéo 3D (http://home.comcast.net/~johnjbarton/ubicomp/ur/ubiwise/) et IMPORTANT que nous avons utilisé pour certaines expérimentations décrites dans ce chapitre (http://nile.cise.ufl.edu/important/).

D'un point de vue plus conceptuel, un défi est de déterminer les abstractions adéquates pour modéliser les applications mobiles, et définir des stratégies de test basées sur des modèles. Dans [TYC+ 04], les auteurs ont montré les limitations de modèles structurels basés sur le code applicatif. Le comportement peut dépendre de données contextuelles gérées à un plus bas niveau (intergiciel), de sorte que les conditions intéressantes pour révéler des fautes n'apparaissent pas explicitement dans la structure du code. Des travaux postérieurs ont alors porté sur l'enrichissement des modèles structurels pour expliciter les dépendances vis-à-vis du contexte [LCT 06] [WER 07]. Pour le test fonctionnel, les travaux les plus avancés concernent le test des protocoles en utilisant des modélisations en SDL (*Specification and Description Language*) [CGM+ 04] [NV 05]. Comme SDL n'est pas un formalisme adapté à la spécification de systèmes mobiles, les auteurs doivent introduire des artifices de modélisation, en particulier pour représenter la notion de communication dans le voisinage. De plus, la génération de test nécessite de choisir *a priori* les configurations spatiales testées (paramétrables dans le cas de [CGM+ 04]). Il serait intéressant pour le test fonctionnel de disposer de formalismes dédiés à la mobilité, mais il faut souligner que les recherches en cours n'ont pas encore débouché sur un cadre de modélisation bien établi. Notamment, les algèbres de processus équipées d'une notion explicite de mobilité, comme les *mobile ambients* [CG 00], paraissent plus adaptés à la mobilité logique (*mobile computation*) que physique (*mobile computing*). Dans de tels formalismes, la mobilité est vue comme l'entrée et la sortie de "boîtes" (ou domaines administratifs) organisées hiérarchiquement. Cette vue reste adéquate pour exprimer la mobilité physique entre différents points d'accès à l'infrastructure, mais est insuffisante pour prendre en compte la dynamique de réseaux ad hoc. De façon générale, l'absence d'un cadre de modélisation bien établi va poser problème pour la conception du test, que l'on considère la sélection de test ou l'oracle de test.

Nos travaux se sont intéressés à une modélisation partielle du comportement, focalisée sur des scénarios d'interaction. L'objectif poursuivi se situe dans la lignée de travaux sur le test passif [CMM 09] : plutôt que de chercher à produire quelques cas de test ciblés, on laisse tourner l'application et on vérifie la trace d'exécution obtenue. Dans notre cas, l'exécution utilise des moyens de simulation, en considérant des modèles de mobilité pour générer le mouvement aléatoire des nœuds. La trace de test enregistrée contient à la fois des données relatives à la communication entre les nœuds et des données contextuelles liées aux configurations spatiales successives. L'analyse de la trace consiste alors à rechercher toutes les occurrences de scénarios prédéfinis, soit pour observer la satisfaction de propriétés (scénarios d'exigence) soit pour déterminer la couverture de fragments de comportement (scénarios représentant des objectifs de test).

Ces travaux ont démarré dans la deuxième moitié des années 2000. Ils nous ont d'abord conduit à réfléchir aux spécificités des interactions au sein de systèmes mobiles. Nous en avons déduit des extensions aux langages de scénarios existants (§IV.1), avec notamment une représentation explicite des configurations spatiales des nœuds sous forme de graphes étiquetés. Le langage TERMOS (*Test Requirement language for Mobile Settings*) incorpore ces extensions aux Diagrammes de Séquences UML [OMG 10] pour représenter des scénarios d'exigence ou des objectifs de test (§IV.2). L'analyse des traces met alors en jeu à la fois des algorithmes d'appariement de graphes (§IV.3) et des calculs d'ordres partiels d'événements (§IV.4). Le premier aspect a fait l'objet de la thèse de Minh Duc N'Guyen [NGu 09], co-encadré avec Nicolas Rivière (LAAS). Le deuxième constitue une partie de la thèse de Zoltan Micskei, non encore soutenue, et qui a fait l'objet d'un co-encadrement avec l'Université Technologique de Budapest. Des prototypes ont permis d'illustrer ces différents concepts.

IV.1. Spécificités des scénarios dans un contexte mobile

Les langages de scénarios graphiques, comme les *Message Sequence Charts* [ITU 04] ou les Diagrammes de Séquences UML [OMG 10], permettent de décrire des interactions dans un système distribué. La description peut servir différents buts : capture d'exigences [KSH 07], spécification d'objectifs de test (c'est-à-dire, d'interactions à couvrir par des cas de test) [GHN 93], conception de cas de test [PJ 04] ou rétro-conception de traces d'exécution [BLL 06]. Typiquement, on dessine la ligne de vie de chaque participant à l'interaction, et on fait apparaître les ordres partiels des communications. Les langages offrent différents opérateurs pour décrire des ordres complexes, tels que des opérateurs de choix, d'itération, de parallélisme et de séquencement. Certains introduisent des modalités pour distinguer les comportements potentiels et nécessaires, tels les *Live Sequence Charts* (LSC) [DH 01] ou la version 2 des Diagrammes de Séquences [OMG 10]. Les communications sont généralement point à point, avec un émetteur et un récepteur pour chaque message. La topologie de connexion n'est pas un élément central, et n'est pas supposée changer durant l'interaction.

Dans un système mobile, le mouvement des nœuds conduit à une topologie de connexion intrinsèquement instable. Les liens avec les autres nœuds mobiles et les nœuds fixes d'infrastructure peuvent être établis ou détruits selon la localisation. De plus, des nœuds peuvent apparaître ou disparaître à tout moment, par exemple lorsqu'un utilisateur allume ou éteint un dispositif portable, lorsque le dispositif se met en veille ou que sa batterie est épuisée. Il est donc souhaitable que les scénarios prennent explicitement en compte la configuration spatiale des nœuds et son évolution au cours de l'interaction.

En plus de la communication point à point, un mode de communication naturel pour les système mobiles est la diffusion dans le voisinage. Le nœud diffuse un message radio sans destinataire spécifique. Tout nœud voisin à portée de communication peut écouter ce message et y réagir. Ce type de diffusion est notamment utilisé dans le cadre de services nécessitant la découverte de voisins, par exemple des services de groupe ou de routage de messages dans les réseaux ad hoc.

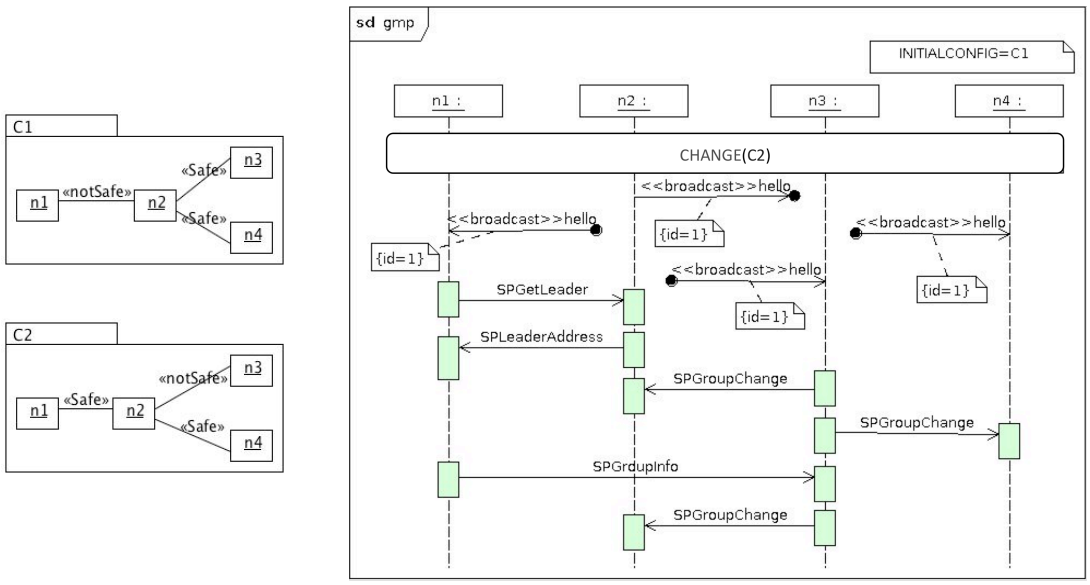
Notre proposition a été d'étendre les langages de scénarios pour faire apparaître les configurations spatiales des nœuds et permettre de représenter la diffusion dans le voisinage. Plus précisément, nous avons considéré trois extensions :

1. Introduction d'une vue spatiale,
2. Représentation d'événements de changement de configuration spatiale,
3. Représentation d'événements de communication par diffusion dans le voisinage.

Ces extensions peuvent être apportées aux différents langages existants. A titre d'exemple, nous considérons ici les Diagrammes de Séquences UML (UML SD) sur lesquels notre langage TERMOS est basé.

IV.1.1. Introduction d'une vue spatiale

Un scénario d'interaction dans un contexte mobile est décrit au moyen de deux vues complémentaires : une vue événementielle (comme dans le cas de scénarios classiques) et une vue spatiale décrivant les topologies de connexion. La Figure IV-a illustre ces deux vues en UML SD. Le scénario montré vient d'une étude de cas que nous avons réalisée [WMN+ 07], un service de groupe (*GMP*) pour des réseaux ad hoc [HJR 04]. Dans ce GMP, les groupes fusionnent ou se scindent en fonction des mouvements des nœuds mobiles. Le service utilise la notion de distance sûre entre paires de nœuds (correspondant à un seuil prédéfini) pour déterminer la composition des groupes.



(1) Vue spatiale **(2) Vue événementielle**

Figure IV-a Scénario de fusion et scission concurrentes pour des groupes de nœuds mobiles

Conceptuellement, la vue spatiale consiste en un ensemble de graphes étiquetés, correspondant aux différentes configurations qui se produisent lors du scénario. Nous les représentons en UML à l'aide de diagrammes d'objets. Le scénario de la Figure IV-a comporte ainsi deux configurations spatiales *C1* et *C2*. Les nœuds des configurations sont décrits par des instances, avec un identifiant (*n1, n2, ...*) et éventuellement des attributs contextuels (non montrés ici). Les labels portés sur les arcs caractérisent la connexion des nœuds. Les différents types de connexion sont représentés par des stéréotypes. Dans l'exemple du GMP, on distingue deux types de connexion pertinents : « *Safe* » lorsque les nœuds sont à distance de sécurité, et « *notSafe* » lorsqu'ils sont à portée de communication mais à une distance supérieure au seuil de sécurité.

IV.1.2. Evénements de changement de configuration

Pour faire le lien entre vues spatiale et événementielle, chaque changement de configuration apparaît explicitement comme un événement global dans la vue événementielle. Dans l'exemple de la Figure IV-a, *C1* est la configuration initiale du scénario, comme indiqué dans une boîte de commentaires apparaissant dans la vue événementielle. Les changements de configuration sont alors représentés par des événements globaux de la forme *CHANGE(nouvelle_config)*. Dans l'exemple du GMP, le changement de *C1* à *C2* correspond à un mouvement du nœud *n2* qui s'approche de *n1* tout en s'éloignant de *n3*.

Dans d'autres scénarios, les changements de configuration peuvent résulter de l'apparition ou de la disparition de nœuds. Comme les Diagrammes de Séquences ne permettent pas de représenter aisément une telle structure dynamique, nous avons convenu que chaque nœud présent dans au moins une configuration du scénario ait une ligne de vie complète dans la vue événementielle. Si un nœud se trouve en fait inactif pendant un fragment du scénario, alors il ne peut participer à aucune interaction de communication dans ce fragment. La vue événementielle indique explicitement la configuration spatiale sous-jacente à tout événement de communication. On peut ainsi vérifier automatiquement que les interactions montrées sont compatibles avec la vue spatiale, et émettre un avertissement dans le cas contraire. Nous verrons ultérieurement un exemple d'outil incluant ce type de vérification (§IV.4.3).

Représenter chaque changement de configuration par un événement global est une abstraction du fait que la topologie des nœuds est une propriété globale du système. Il est important de comprendre qu'un tel événement ne résulte pas d'une synchronisation active des nœuds ; le changement de configuration se produit arbitrairement dans le monde physique. Dans l'exemple du GMP, le changement vers *C2* se produit mais va rester non détecté jusqu'à ce que le nœud *n2* diffuse sa nouvelle localisation par un message *Hello*.

IV.1.3. Diffusion dans le voisinage

Pour représenter la diffusion dans le voisinage, nous utilisons les concepts UML de messages *perdus* et *trouvés*. Un message perdu n'a pas de récepteur explicite. De même, un message trouvé provient d'un émetteur non spécifié. Nous attribuons le stéréotype « *broadcast* » aux messages diffusés aux voisins, pour les distinguer des messages perdus/trouvés "standard". Une diffusion implique un événement d'émission suivi d'un ou plusieurs événements de réception par des nœuds voisins. Une étiquette est attachée au message pour identifier sans ambiguïté le lien de causalité entre l'événement d'émission et chaque réception. Comme précédemment, on peut vérifier que l'interaction montrée est compatible avec la topologie de connexion.

Dans l'exemple de la Figure IV-a, le message *Hello* diffusé par *n2* est reçu par ses voisins *n1*, *n3* et *n4*. Détectant le changement de configuration, *n1* va initier une fusion de groupe alors que *n3* va initier une scission. Les opérations concurrentes de fusion et scission peuvent alors induire différents entrelacements de messages, dont celui représenté dans la figure. Nos tests ont observé que ce scénario spécifique entraîne une défaillance du service de groupe (notamment, les nœuds ont des vues incohérentes de l'appartenance de *n4*) [WMN+ 07]. On voit ici comment les extensions proposées sont adéquates pour décrire ce type de scénario – caractéristique des interactions dans un contexte mobile – où l'on doit prendre en compte à la fois les évolutions de configuration spatiale et les entrelacements des communications.

IV.2. Présentation générale du langage TERMOS

La représentation graphique de scénarios peut intervenir à différentes étapes du processus de développement, allant de la capture des exigences à des étapes de test tardives. Nos travaux se concentrent sur l'utilisation de scénarios pour vérifier des traces d'exécution. L'approche est schématisée dans la Figure IV-b. La plateforme expérimentale doit offrir des facilités pour observer et collecter des données lors de l'exécution de l'application cible. La trace recueillie comporte à la fois des données relatives aux messages échangées par les nœuds du système et des données contextuelles calculées par le simulateur de contexte. Ces données contextuelles doivent notamment permettre de reconstituer les configurations spatiales successives. On cherche alors à vérifier automatiquement si la trace de test exhibe des comportements décrits dans des scénarios. TERMOS (*Test Requirement language for Mobile Settings*) [WMR+ 10] est un langage formel conçu dans ce but. Il intègre les extensions aux diagrammes de séquences présentées précédemment. Des restrictions syntaxiques sont cependant apportées à la description des ordres partiels dans la vue événementielle et la sémantique est choisie pour bien s'adapter à la vérification de traces.

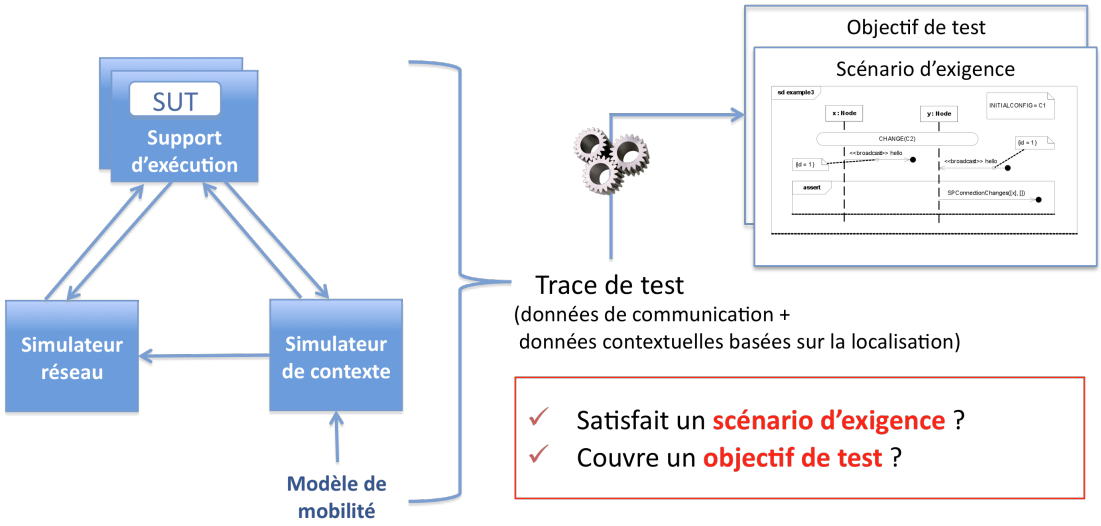
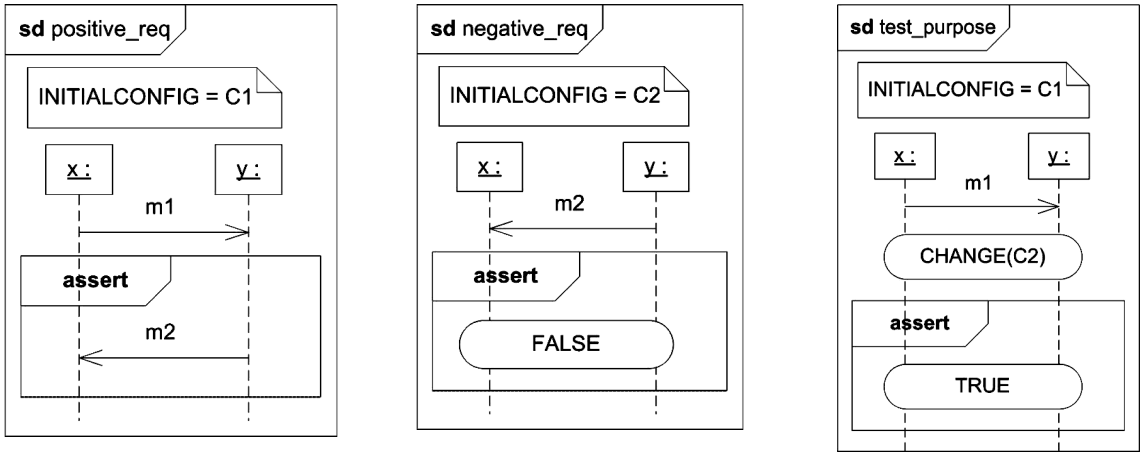


Figure IV-b Motivations du langage TERMOS

TERMOS considère trois classes de scénarios illustrées par la Figure IV-c. Les *exigences positives* décrivent des propriétés invariantes de la forme : chaque fois que *A* se produit, alors *B* doit suivre. Les *exigences négatives* décrivent des comportements interdits qui ne devraient jamais être observés dans la trace. Les *objectifs de test* décrivent des comportements à couvrir pendant le test, c'est-à-dire dont on souhaite observer au moins une occurrence dans la trace.

Un scénario TERMOS se termine toujours par un fragment *assert* (la Figure IV-a n'était donc pas un scénario TERMOS, mais pourrait facilement être convertie en objectif de test). Tout ce qui précède le *assert* représente un comportement potentiel, alors que le contenu du *assert* est nécessaire. Ce contenu caractérise la classe du scénario. Pour une exigence négative, il est réduit à un invariant *FALSE*. Comme on ne peut pas établir *FALSE*, cela signifie que le comportement précédant le *assert* ne doit jamais se produire. Pour un objectif de test, le contenu est réduit à un invariant *TRUE*, trivialement établi dès que le *assert* est atteint. Une exigence positive se termine par un fragment *assert* différent des invariants triviaux *TRUE* et *FALSE*.



(1) Exigence positive **(2) Exigence négative** **(3) Objectif de test**

Figure IV-c Trois classes de scénarios TERMOS (vue événementielle)

Nous interprétons les scénarios TERMOS comme des motifs génériques de comportement, qui peuvent être exhibés par différents sous-ensembles de nœuds, à différents points de la trace. Dans la Figure IV-c, les identifiants de nœud *x* et *y* sont des identifiants *symboliques*. Par exemple, l'exigence positive est interprétée comme suit :

> "Chaque fois qu'une paire de nœuds exhibe la configuration *C1*, et que le nœud jouant le rôle de *x* envoie un message *m1* au nœud jouant le rôle de *y*, alors ce dernier doit répondre par un message *m2*."

A un certain point de l'exécution, on peut avoir deux instances simultanées de *C1* dans le système, l'une avec les nœuds concrets *n1* et *n2* correspondant à *x* et *y*, et l'autre avec *n1* et *n3*. Plus tard dans l'exécution, le nœud concret *n1* peut jouer le rôle de *y* dans encore une autre instance de *C1*.

Conformément à ce qui précède, l'analyse d'une trace de test pour rechercher des occurrences de scénarios comporte deux étapes :

1. Déterminer quels nœuds physiques du système peuvent jouer le rôle des nœuds symboliques apparaissant dans la vue spatiale du scénario ;
2. Pour ces nœuds, analyser l'ordre des événements dans les configurations identifiées.

La première étape traite la vue spatiale et correspond à un problème d'appariement de graphes. Les configurations du scénario fournissent une séquence de graphes motif que l'on cherche à retrouver dans les configurations concrètes d'exécution, reconstituées à partir des données contextuelles. A l'issue de cette étape, on connaît tous les sous-ensembles de nœuds du système qui exhibent la séquence de configurations du scénario, ainsi que les dates des événements de changement de configuration correspondants.

La deuxième étape correspond à une interprétation classique de la vue événementielle en termes d'ordres partiels d'événements. Cette interprétation s'effectue en fonction de la sémantique choisie pour les opérateurs offerts par UML SD, en utilisant les informations fournies par l'appariement de graphes.

On voit ainsi comment la sémantique de TERMOS combine appariement de graphes et calculs d'ordres partiels d'événements. Ces deux aspects sont expliqués dans les paragraphes qui suivent.

IV.3. Sémantique de la vue spatiale

La vue spatiale d'un scénario est interprétée comme une séquence de graphes motifs dont on cherche toutes les occurrences dans les configurations concrètes de la trace. Les graphes motifs peuvent comporter des labels multiples, des variables de labels et des jokers '*', comme illustré par la configuration *C3* de la Figure IV-d. Conceptuellement, chaque nœud de *C3* est étiqueté par un triplet de labels *<l1, l2, l3>*, le premier représentant l'identifiant du nœud et les deux autres représentant des attributs contextuels. Pour les trois nœuds de la figure, ces triplets sont respectivement *<x, v1, *>*, *<y, 1, 2>* et *<z, v1, *>*. Les identifiants symboliques *x*, *y* et *z* sont des exemples de variables de label. Une autre variable, *v1*, indique que les nœuds concrets jouant les rôles de *x* et *z* doivent avoir leur premier attribut à une valeur non spécifiée, mais identique. Un joker indique que l'on peut ignorer la valeur du label correspondant (attribut de nœud, ou type de connexion).

Comparer une séquence de graphes motifs $C_1, \ldots, C_m$ à une séquence de configurations concrètes $CC_1, \ldots, CC_n$ induit un raisonnement à deux niveaux.

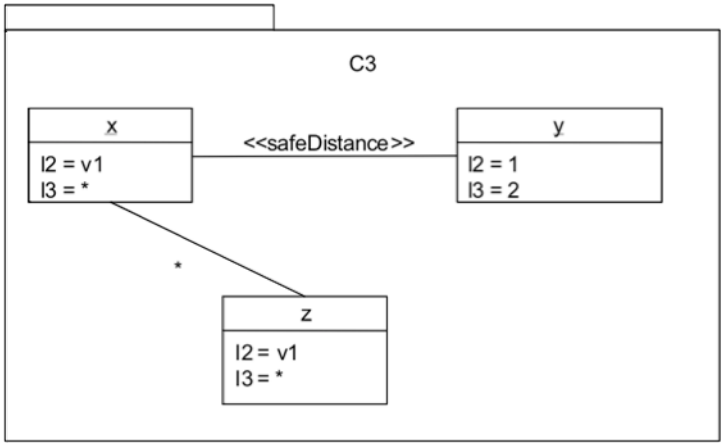


Figure IV-d Configuration spatiale TERMOS avec différents types de labels

Au niveau de base, il faut pouvoir comparer des paires de graphes C_i et CC_j, pour exhiber tous les sous-graphes de *CCj* isormorphes au motif C_i. La recherche d'isomorphismes de sous-graphe est un problème très étudié dans la littérature [Ull 76] [MB 00], y compris pour des motifs comportant des variables de label [Gue 06]. Nous avons choisi d'utiliser un outil existant, développé par des collègues du LAAS dans le cadre de travaux sur la spécification d'architectures dynamiquement reconfigurables [Gue 06] [GD 06]. Cet outil retourne des isomorphismes de sous-graphe de la forme *(f, val)*, où *f* est injection associant un nœud concret à chaque nœud du motif, et *val* est une valuation des variables qui unifie les labels.

Au second niveau, on s'appuie sur la comparaison de paires de graphe pour identifier les occurrences de séquences de motifs. L'appariement de séquences de graphe a été beaucoup moins étudié que l'appariement de paires de graphes (voir par exemple l'article de synthèse [CFS+ 04]), et c'est là que se situe notre contribution à la sémantique de la vue spatiale. L'algorithme que nous avons proposé est, à notre connaissance, original.

IV.3.1. Appariement de séquences de graphes

Chaque occurrence de la séquence motif peut être encodée par une structure de donnée *match* comportant deux champs :

- Un tableau *index* caractérisant la fenêtre temporelle de l'appariement, avec les dates de début et de fin des instances des configurations motif recherchées ;
- Une valuation *val* de toutes les variables apparaissant dans les différents graphes motif successifs.

Les figures IV-e et IV-f donnent l'intuition visuelle de ce qu'est un *match*. Une instance du motif C_1 apparaît comme sous-graphe du système dans la configuration concrète CC_2. Le changement de configuration système $CC_2 \rightarrow CC_3$ n'affecte pas ce sous-graphe, qui reste une instance de C_1. Un autre changement de configuration système conduit ensuite à une instance de C_2, puis à une instance de C_3, jusqu'au changement $CC_7 \rightarrow CC_8$ qui détermine la fin de l'appariement. La fenêtre temporelle définie par le *match* sera utile pour examiner les ordres partiels d'événements (§IV.4). Par exemple, supposons que la vue événementielle du scénario montre l'envoi d'un message *msg* dans la configuration C_3. On sait alors que *msg* doit être recherché dans la sous-trace du système commençant à la date du changement $CC_4 \rightarrow CC_5$ et finissant à la date de $CC_7 \rightarrow CC_8$. De plus, la valuation du *match*, qui détermine notamment les valeurs des identifiants symboliques de nœud, permet de savoir quels nœuds concrets sont supposés être émetteur et récepteur de *msg*.

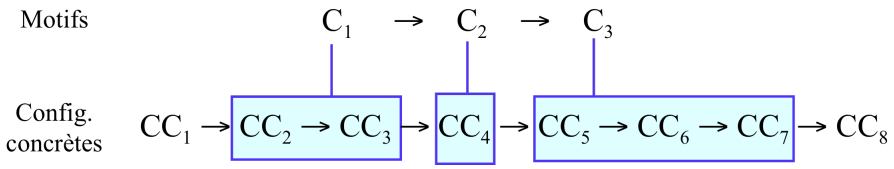


Figure IV-e Identification d'une occurrence de la séquence motif

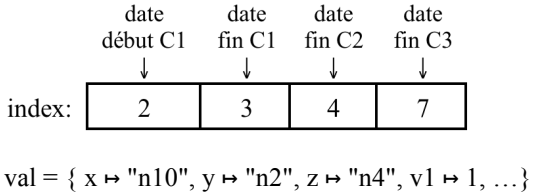


Figure IV-f Encodage dans un *match*

Ce type d'appariement de séquences a été peu traité dans la littérature. Les travaux les plus proches des nôtres concernent le traitement d'images. Dans [SVB 01], les auteurs cherchent des séquences motifs (*pictorial queries*) dans une séquence de graphes extraits d'images vidéo. Une différence avec nous est cependant que leurs motifs ne contiennent pas de variables de labels, et qu'un nœud du motif correspond à au plus un objet dans une image. Dans notre cas, il peut y avoir plusieurs instances d'un motif C_i dans une configuration concrète CC_j, avec différentes valuations possibles des variables (dont les identifiants de nœuds). Par exemple, supposons qu'il y ait plusieurs instances du premier motif C_1. Chaque instance de C_1 peut à son tour amener plusieurs possibilités pour C_2, avec différentes dates de transition $C_1 \rightarrow C_2$ et différentes valuations pour les variables qui sont nouvelles dans C_2 (les variables qui ne sont pas nouvelles doivent garder la même valeur). La clé de la recherche consiste à explorer toutes ces possibilités, en retenant des valuations cohérentes tout au long de la séquence et en identifiant correctement les transitions entre instances de motifs.

Le manuscrit de thèse de Minh Duc N'Guyen [NGu 09] détaille l'algorithme de la recherche séquentielle, et formalise les propriétés attendues d'un *match*. Je donne ici une vue intuitive de ces propriétés.

Propriété 1 : présence des instances des motifs dans les configurations concrètes. Dans l'exemple de la Figure IV-f, on veut évidemment que l'instance de C_1 déterminée par *val* apparaisse effectivement comme sous-graphe de CC_2 et CC_3, que l'instance de C_2 apparaisse comme sous-graphe de CC_4, etc.

Propriété 2 : absence de nœuds interdits dans les configurations concrètes. Cette propriété est relative au traitement des nœuds qui apparaissent ou disparaissent d'un motif à l'autre. La Figure IV-g montre un exemple de séquence motif avec apparition d'un nœud (avec une syntaxe simplifiée des graphes, et en ne considérant que les labels d'identifiant de nœud). L'interprétation de ce motif est que le nouveau nœud *y* n'était présent dans le système pour aucune des configurations concrètes appariées à C_1. Dans la Figure IV-h, *Match1* est incorrect bien qu'il satisfasse la propriété 1 : le nœud concret *n2* n'est pas nouveau. Par contre, *Match2* est correct.

Propriété 3 : maximalité des fenêtres temporelles du match. Dans la Figure IV-h, *Match3* est incorrect car la date de début de l'appariement de C_1 pourrait être en *t*, voire plus tôt si le nœud concret *n1* était déjà présent dans CC_{t-1}. *Match2* est correct, l'appariement ne pourrait pas commencer plus tôt du fait du nœud interdit *n2*.

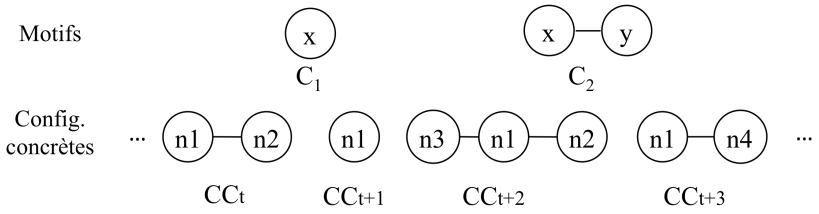


Figure IV-g Motif avec apparition d'un nœud

Match1 (incorrect selon propriété 2)

index :	t	t+1	t+2
val :	{x ↦ "n1", y ↦ "n2"}		

Match2 (correct)

index:	t+1	t+1	t+2
val :	{x ↦ "n1", y ↦ "n2"}		

Match3 (incorrect selon propriété 3)

index :	t+1	t+1	t+2
val :	{x ↦ "n1", y ↦ "n3"}		

Figure IV-h Exemples de *matchs* corrects et incorrects pour le motif précédent

Une conséquence de la propriété 3 est que toutes les dates encodées dans un *match* correspondent à des événements de changement de configuration concrète. Un scénario comportant deux graphes motif successifs identiques n'aura aucun *match*. Le début et la fin de l'appariement, ainsi que les dates de transition $C_i \rightarrow C_{i+1}$, sont toujours déterminées par des événements concrets dans le système testé : changement du type de connexion entre deux nœuds, changement de la valeur de l'attribut d'un nœud, apparition d'un nœud, disparition d'un nœud, ou toute combinaison de ces événements.

Les événements d'apparition de nœud sont les plus complexes à traiter. Dans l'exemple de la Figure IV-g, si le nœud concret *n1* joue le rôle de *x*, la date de fin de C_1 peut être *t+1* (avant l'apparition des nœuds *n2* ou *n3*), *t+2* (avant l'apparition du nœud *n4*) ou plus tard. Selon le nouveau nœud choisi pour jouer le rôle de *y*, qui est donc interdit dans les configurations précédentes, la date de début pour une fenêtre temporelle maximale de C_1 va être différente. De plus chaque choix va nécessiter des explorations spécifiques pour apparier la suite de la séquence.

IV.3.2. Complexité de l'appariement de séquences de graphes

L'appariement de séquences de graphes est un problème fondamentalement très coûteux.

A la base, la recherche d'isomorphismes de sous-graphes (utilisée pour comparer deux graphes) est un problème NP complet. Soit *Nmotif* et *Nconcret* le nombre de nœuds dans chacun des deux graphes comparés. Le pire cas est quand tous les nœuds concrets peuvent jouer le rôle de tous les nœuds du motif, d'où une complexité exponentielle $O(Nconcret^{Nmotif})$. Le cas le plus favorable est quand au plus un nœud concret peut jouer le rôle d'un nœud du motif, auquel cas la complexité est seulement quadratique en *Nmotif* et *Nconcret*.

La recherche de tous les *matchs* va induire de multiples appels à la fonction de comparaison de deux graphes. Supposons que l'on recherche toutes les occurrences d'une séquence de *m* graphes motif dans une séquence de *n* configurations concrètes ($n \geq m$). Pour simplifier la discussion, soit *H* une abstraction du coût de la comparaison de deux graphes. Deux cas peuvent être distingués : (1) la séquence motif ne comporte pas d'apparition de nœud, et (2) elle en comporte.

Dans le premier cas, le coût de la recherche est dominé par le coût des comparaisons entre le premier graphe C_1 et chaque CC_i. Une fois qu'un appariement a été trouvé pour C_1, tous les identifiants de nœuds sont déterminés par la valuation candidate, et il n'y a plus qu'une possibilité pour l'appariement des graphes suivants. La complexité est donc $O(nH)$.

Dans le deuxième cas, chaque apparition de nœud va induire plusieurs possibilités, selon l'identité du nœud concret choisi pour jouer le rôle du nouveau nœud. Lorsque toutes les transitions $C_i \rightarrow C_{i+1}$ sont déterminées par des apparitions de nœuds, une estimation pessimiste donne une complexité $O(n^m H^m)$. Dans le pire cas pour H, on a une complexité exponentielle à la fois dans le nombre m et la taille *Nmotif* des graphes de la séquence motif.

Au vu de cette analyse, il est clair que l'appariement de séquences ne peut pas s'appliquer à grande échelle. Néanmoins, il reste utilisable dans le cadre de nos travaux, pour vérifier des traces expérimentales par rapport à des petits scénarios graphiques.

Les scénarios graphiques s'intéressent typiquement aux interactions d'un faible nombre d'entités. Par exemple, les scénarios que nous avons dérivés de l'étude du service de groupe (GMP) pour des réseaux ad hoc (ex : le scénario de la Figure IV-a) ne comportaient jamais plus de cinq nœuds. Ce nombre d'entités est comparable à ce que nous avons pu voir dans le cadre d'autres travaux exploitant des descriptions de scénarios, et de toute façon la représentation graphique ne pourrait pas s'accommoder d'un grand nombre de lignes de vie. Similairement, on peut s'attendre à ce que le nombre de configurations successives du scénario reste modeste (ex : typiquement de un à trois graphes motifs successifs pour les scénarios du GMP).

Le nombre et la taille des configurations concrètes extraites de la trace sont moins critiques pour la complexité de l'appariement. Plusieurs exemples de travaux expérimentaux sur les protocoles de routage font apparaître des configurations testées impliquant entre 5 et 50 nœuds concrets, avec des durées d'enregistrement de trace allant de 30 secondes à quelques minutes [DS 01] [JMB 01] [CGM+ 04] [MV 04] [HPJ 05]. Nos propres expérimentations montrent que de telles configurations sont à la portée de l'appariement de séquences.

IV.3.3. Prototypage et premières expérimentations

L'algorithme d'appariement de séquences de graphes a été implémenté dans un outil, *GraphSeq* [NWR 10]. Cet outil représente environ 2 000 lignes de code C++, sans compter la fonctionnalité de comparaison de deux graphes (réutilisée de [Gue 06]).

L'implémentation a été validée par des tests intensifs. Nous avons considéré à la fois des cas de test spécifiques, illustrant les points délicats de l'appariement (comme ceux montrés dans la Figure IV-h) et un large échantillon de 900 paires de séquences générées aléatoirement avec les caractéristiques suivantes :

- Taille des séquences : nombre m de graphes motif de 1 à 5 et nombre n de configurations concrètes de 1 à 100.
- Taille des graphes : nombre de nœuds *Nmotif* de 1 à 5 et *Nconcret* de 1 à 25.
- Par construction, la séquence motif apparaît au moins une fois dans la séquence concrète.

Ces tests ont été utiles lors de la mise au point de *GraphSeq*, avec notamment des vérifications de non-régression après modification du code.

Nous avons ensuite mené des expériences pour illustrer le principe de la recherche de configurations spatiales sur des scénarios réels. Nous avons repris notre étude précédente du

GMP pour les réseaux ad hoc [WMN+ 07]. Nous en avons extrait trois exemples de scénarios de défaillance que nous avons considérés comme des objectifs de test. Les traces d'exécution du GMP ont été analysées pour retrouver les séquences motifs, en considérant un paramétrage similaire à celui des expériences déjà menées sur cette étude de cas (Figure IV-i). Sur 100 exécutions, *GraphSeq* a permis de trouver respectivement 75, 84 et 114 *matchs* pour chaque scénario. Notons que tous les *matchs* ne correspondent pas forcément à des occurrences des scénarios, puisqu'ils ne prennent en compte que la vue spatiale, et que l'ordre des événements de communication reste à analyser. Par exemple, pour le premier scénario, il y avait en fait 27 occurrences de la défaillance recherchée.

Dans l'étude expérimentale du GMP, le mouvement des nœuds est géré de façon élémentaire : à chaque nœud est attaché un stub GPS qui délivre des coordonnées physiques compatibles avec un mouvement aléatoire à vitesse constante. Une gestion du mouvement plus sophistiquée nécessite un simulateur de contexte. Pour illustrer ce principe, nous avons connecté *GraphSeq* au simulateur IMPORTANT[14], développé à l'Université de Californie du Sud (USC), aux Etats Unis. Cet outil offre un ensemble de modèles de mobilité paramétrables. Les traces générées sont compatibles avec le simulateur ns-2. Des expérimentations ont été menées avec différents modèles de mobilité et différents motifs recherchés. La Figure IV-j donne un exemple pour le modèle *Freeway* utilisé pour simuler le mouvement de véhicules dans un réseau routier. La trace enregistrée sur 15 minutes correspond à une séquence de 345 configurations concrètes différentes, dans laquelle nous avons trouvé 11 *matchs*.

Paramétrage des exécutions du GMP :

- Nombre de nœuds aléatoire dans [6..15], avec des dates d'apparition aléatoires
- Mouvement aléatoire de chaque nœud à une vitesse de 10 $m.s^{-1}$
- Durée d'enregistrement : 5 minutes

Motifs recherchés :

- Scénario 1 - fusion et scission concurrentes
  Variante de la Figure IV-a, avec apparition du nœud *n1* dans la configuration *C2*
- Scénario 2 - fusions concurrentes
  Motif de deux configurations successives avec les trois mêmes nœuds
- Scénario 3 – deux scissions successives
  Motif de trois configurations successives avec les trois mêmes nœuds

Figure IV-i Analyse des traces de mobilité du GMP

Paramétrage du simulateur :

- Déplacement sur une carte routière avec deux autoroutes à 3 voies
- Nombre de véhicules = 15
- Vitesse maximale = 40 $m.s^{-1}$
- Accélération maximale = 4 $m.s^{-2}$
- Distance minimale entre deux véhicules sur la même voie : 40m
- Durée d'enregistrement : 15 minutes

Motif recherché :

- Scénario de la Figure IV-a

Figure IV-j Analyse d'une trace du simulateur IMPORTANT (modèle Freeway)

[14] http://nile.cise.ufl.edu/important/

IV.4. Sémantique de la vue événementielle

La fenêtre temporelle d'un *match* définit les événements de changement de configuration pertinents pour le scénario. Elle identifie une sous-trace dans laquelle chercher les événements de communication. Tous les paramètres symboliques des configurations successives, avec notamment les identifiants de nœuds, ont des valeurs connues. Il reste à déterminer si les communications observées correspondent au scénario. Cela nécessite d'interpréter la vue événementielle en termes d'ordres partiels d'événement.

La difficulté est qu'il n'existe pas "une" sémantique pour les diagrammes de séquence UML. Le standard OMG [OMG 2010] reste informel et mentionne des points de variation sémantique (différentes interprétations de la notation sont possibles). Pour avoir une idée plus précise des interprétations données, il faut en fait se tourner vers les nombreuses sémantiques formelles qui ont été proposées dans la littérature. Une de nos contributions a été de dresser un état des lieux des sémantiques existantes, de façon à guider notre choix pour TERMOS (§IV.4.1). La sémantique retenue est donnée par la construction d'un automate symbolique, dont les variables dépendent des configurations spatiales (§IV.4.2). L'algorithme correspondant s'inspire de travaux autour d'un autre langage de scénario, les LSC [DH 01] [Klo 03], ainsi que de travaux adaptant les modalités de nécessité/potentialité des LSC aux Diagrammes de Séquences [Kus 06] [HM 08]. Deux prototypes ont été développés pour TERMOS (§IV.4.3).

IV.4.1. Choix d'interprétation et restrictions syntaxiques

Pour dresser l'état des lieux, nous avons retenu un échantillon de 13 sémantiques formelles proposées pour les Diagrammes de Séquences UML depuis la version 2.0. Cet échantillon contient à la fois des travaux de référence cités par la plupart des autres, et des travaux moins cités car représentatifs d'utilisations très spécifiques des scénarios. Notre démarche a été la suivante. Nous avons cherché à identifier systématiquement les points dans lesquels les sémantiques différaient. Ces points indiquent des choix à effectuer, et les différentes solutions proposées sont les options relatives à ces choix. Pour chaque choix, nous avons exhibé des exemples de diagrammes pour lesquels l'interprétation diffère selon l'option retenue, de façon à avoir une vue concrète des conséquences de chaque décision. Ce travail nous a permis d'obtenir une catégorisation claire des choix et options. Nous avons alors pu prendre nos propres décisions pour TERMOS.

Ce travail d'analyse est décrit en détail dans [MW 10]. Les principales décisions sont synthétisées ci-dessous.

Interprétation des interactions de base. Les scénarios TERMOS décrivent des interactions partielles, concernant un sous-ensemble de nœuds et un sous-ensemble de messages entre ces nœuds (la plupart des sémantiques considèrent que l'interaction montrée est complète). Cela signifie que la trace vérifiée peut contenir un préfixe ou un suffixe encadrant cette interaction. De plus, durant l'interaction, d'autres événements peuvent s'entrelacer avec les événements montrés. La trace vérifiée est interprétée comme pouvant relever de trois catégories : valide, invalide et non concluante (quelques sémantiques considèrent une classification en deux catégories seulement, valide/autre ou invalide/autre).

Combinaison de fragments. Les opérateurs de choix, parallélisme, etc. définissent des fragments d'interactions (*CombinedFragment*). Pour combiner ces fragments avec le reste du diagramme, l'interprétation de l'OMG est celle d'un séquencement faible. Dans TERMOS comme dans plusieurs autres sémantiques, l'entrée et la sortie d'un fragment sont considérées

comme des points de synchronisation pour toutes les lignes de vie concernées. Cela limite l'expressivité des diagrammes et facilite la vérification des traces.

Calcul de plusieurs ordres partiels. L'opérateur de choix (*alt*) définit plusieurs ordres partiels pour le diagramme, un pour chaque branche du choix. La synchronisation à l'entrée du *alt* simplifie déjà beaucoup le calcul de ces ordres partiels : toutes les lignes de vies choisissent la même branche au même instant. TERMOS apporte une restriction supplémentaire, en n'autorisant que la forme déterministe du choix (similaire à un *if then else*). De plus, les prédicats de garde ne peuvent faire références qu'aux paramètres de messages déjà reçus ou émis, ou aux variables valuées dans la configuration courante. De la sorte, les entités et messages non représentés dans le scénario ne peuvent pas changer la valeur de vérité des prédicats.

Interprétation des opérateurs *assert*, *neg*, *consider*, *ignore*. Ces opérateurs sont cruciaux pour déterminer la validité des traces, mais notre état des lieux montre que leur interprétation apporte beaucoup de confusion. Aussi leur utilisation est-elle sévèrement restreinte dans TERMOS. Un scénario ne peut avoir qu'un opérateur *assert*, placé à la fin du diagramme, et couvrant toutes les lignes de vie. Le basculement entre comportement potentiel et obligatoire est donc un point de synchronisation global. L'opérateur *neg* n'est pas du tout utilisé, il est remplacé par l'assertion d'un prédicat *FALSE*. *Consider* et *ignore* sont des opérateurs pour modifier l'alphabet de la trace. Dans TERMOS, l'interprétation est que tout message non représenté peut s'entrelacer avec les messages du scénario : l'opérateur *ignore* n'est donc pas nécessaire. Nous utilisons *consider* quand certains messages ne doivent pas apparaître dans la trace durant l'interaction. Enfin, les emboîtements d'opérateurs *assert* et *consider* sont strictement limités : soit l'opérateur est au niveau racine du diagramme, soit on a un emboîtement à exactement un niveau (un *assert* dans un *consider* racine, ou un *consider* dans un *assert* racine). Les emboîtements au sein d'autres opérateurs (ex : un opérateur *par* mettant *consider(msg)* en parallèle avec un fragment où *msg* apparaît, …) sont interdits.

Les options prises pour TERMOS permettent d'encoder les ordres partiels dans une structure à états finie, ce qui est bien adapté à un objectif de vérification.

IV.4.2. Construction de l'automate

L'approche retenue s'inspire étroitement de celle développée par Klose [Klo 03] pour construire l'automate correspondant à un diagramme LSC. On en retrouve les deux grandes étapes de pré-traitement (*preprocessing*) et de dépilage (*unwinding*) de classes d'événements. Les algorithmes correspondants pour TERMOS sont donnés dans un rapport technique [HWE+ 08]. La construction de l'automate est ici illustrée sur l'exemple en Figure IV-k, gardé très simple pour alléger la présentation. Il s'agit d'un scénario fictif où des nœuds fixes d'infrastructure diffusent périodiquement des informations. Tout nœud mobile s'approchant d'un nœud d'infrastructure peut demander des détails supplémentaires, auxquels cas ces détails doivent lui être donnés. Le diagramme ne contient pas d'opérateurs de choix (*alt*) ou de parallélisme (*par*), et définit en fait un ordre total d'événements. Pour des scénarios plus complexes, le rapport technique [HWE+ 08] détaille le traitement associé à chaque opérateur.

Dans l'étape de pré-traitement, le diagramme est parsé pour en extraire les éléments atomiques, et calculer les relations entre eux. Par exemple, les atomes apparaissant sur la ligne de vie du nœud d'infrastructure sont : le début de la ligne de vie, le changement de configuration, l'envoi de *information*, la réception de *getDetail*, l'entrée dans le *assert*, l'envoi de *détails*, la sortie de *assert* et la fin de la ligne de vie. Les atomes sont regroupés en classes indiquant la simultanéité. Par exemple, toutes les lignes de vie entrent dans le *assert* en même temps et les atomes correspondants appartiennent à la même classe. On calcule

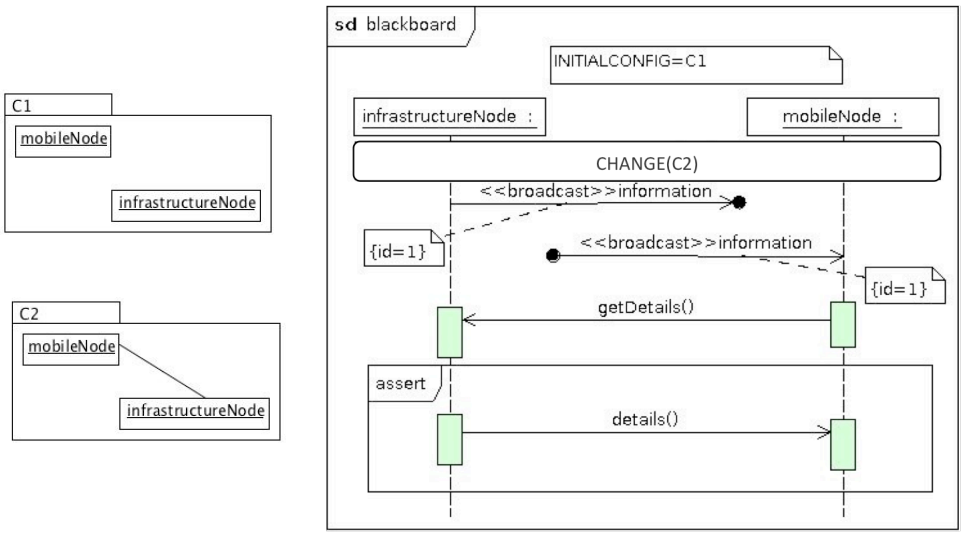


Figure IV-k Exemple de Scénario TERMOS

ensuite les relations de précédence et de conflit entre classes d'atomes. Par exemple, l'envoi de *getDetail* précède sa réception, et ces deux événements précèdent l'entrée dans le *assert*. Les relations de conflits concernent des classes d'atomes incompatibles car situées dans différents opérandes d'un *alt*. Ce type de relation n'est pas illustré par l'exemple de la Figure IV-k.

La deuxième étape consiste à dépiler les classes d'atomes pour construire les ensembles de coupure correspondant aux états de l'automate. Un état de l'automate est un état global du scénario qui représente la progression des différentes lignes de vie. L'algorithme démarre dans un état initial où toutes les lignes de vie sont à leur début. Puis, on utilise les relations de précédence et de conflit pour déterminer les classes d'atomes prêtes à être dépilées, ce qui définit des transitions vers des états successeurs. Chaque transition est étiquetée en fonction de la classe d'atomes dépilée. Le label peut comporter une expression d'événement ou un prédicat. Dans le premier cas, le tirage de la transition est interprété comme la consommation d'un événement de la trace en cours de vérification. Ce tirage peut alors déclencher une action de mise à jour de certaines variables, selon l'unification trouvée entre l'événement symbolique du scénario et l'événement concret de la trace. Dans le deuxième cas, la transition est tirée sans consommation d'événement lorsque l'évaluation du prédicat est vraie. Si l'analyse de la trace arrive dans un état où aucune transition ne peut être tirée, on sort de l'automate avec un verdict qui dépend de la catégorie de l'état courant.

Toutes ces notions sont informellement illustrées sur l'exemple.

L'automate de la Figure IV-l a trois catégories d'états. Les états représentés par un double cercle sont des états d'acceptation triviale : ils correspondent à des traces non concluantes qui n'exhibent pas le comportement potentiel décrit avant le *assert*. Les états avec un cercle simple sont des états de rejet : la trace est invalide. Les états avec un triple cercle sont des états d'acceptation forte : la trace a permis d'atteindre la fin du assert, elle est valide.

Dans la Figure IV-l, on voit que chaque état est étiqueté par un identifiant et par un ensemble de variables dont la valeur est connue dans cet état. Par exemple, l'état initial a l'identifiant *0*. Les seules variables valuées sont celles issues de la configuration spatiale courante C_1 : on connaît l'identité des deux nœuds concrets jouant le rôle de *infrastructureNode* et de *mobileNode*.

Les transitions sortant des états *0* et *1* illustrent la syntaxe des expressions d'événements. La transition *0* → *1* consomme un changement de configuration vers C_2. La boucle *0* → *0* consomme tout événement qui ne correspond pas à un changement de configuration (c'est-à-dire, tout événement de communication). La transition *1* → *2* consomme l'envoi de *information*. L'expression correspondante est un triplet où *!information* dénote l'envoi du message, *infrastructureNode* est le nœud réalisant cet événement, et *$1* est un identifiant symbolique de message. Comme *$1* est une variable libre dans l'état *1*, elle peut correspondre à importe quel identifiant généré par la plateforme de test. Le tirage de la transition met à jour cette variable, qui n'est plus libre dans l'état *2* quand on attend l'événement de réception du message.

Les transitions *5* → *6* et *8* → *9* sont étiquetées par le prédicat *true* et ne consomment pas d'événement de la trace : elles correspondent à l'entrée et la sortie du *assert*. Dans le cas général, un prédicat peut comporter des variables. Elles doivent faire partie de l'ensemble des variables valuées dans l'état de départ de la transition.

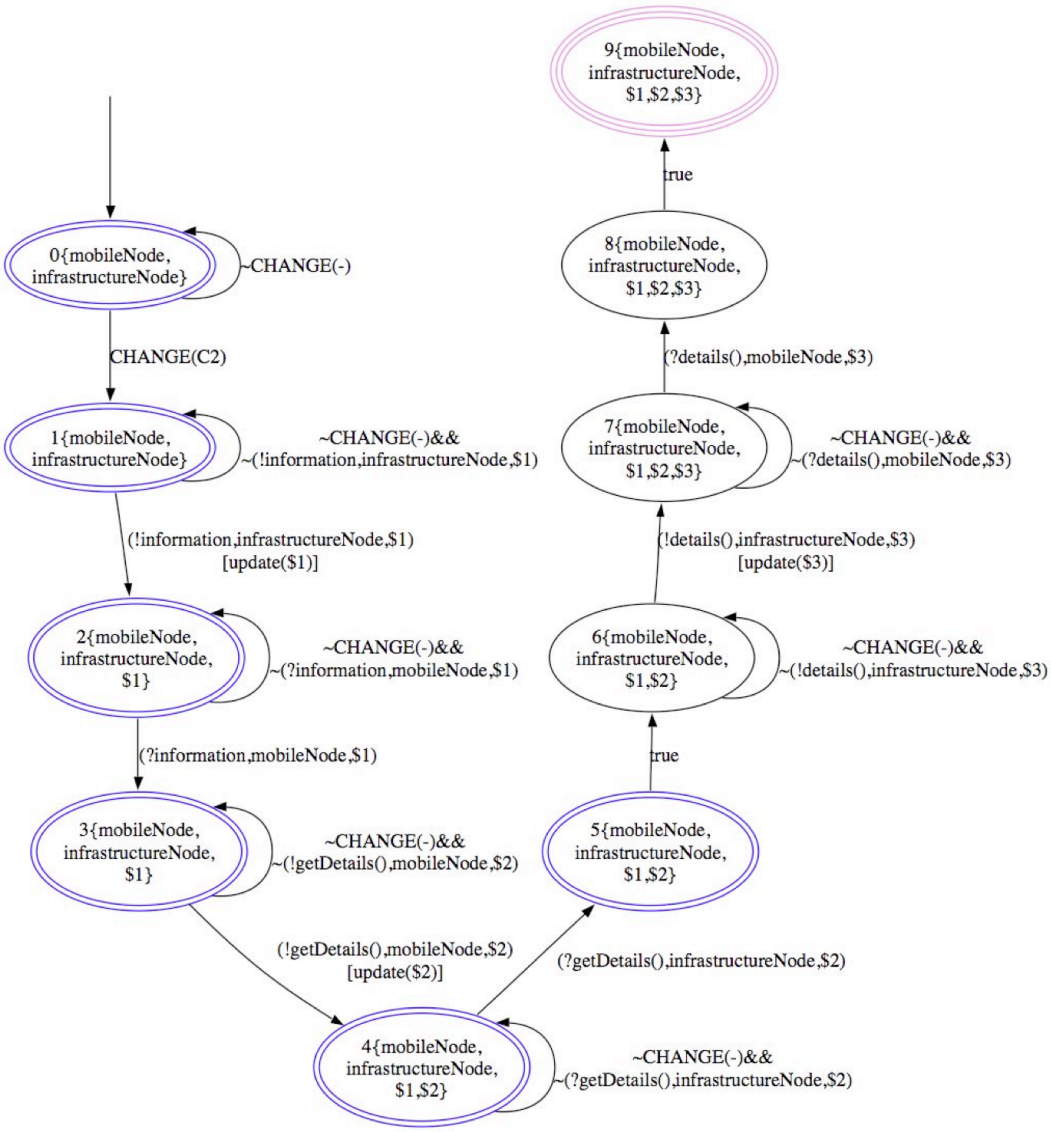


Figure IV-l Automate symbolique pour le scénario précédent

IV.4.3. Prototypage

Deux prototypes ont été développés, implémentant le principe de la construction de l'automate selon différents points de vue.

Le premier prototype, développé au LAAS, s'est focalisé sur la l'étude de la correction des algorithmes. Le but était de nous convaincre que ces algorithmes donnaient bien aux diagrammes l'interprétation souhaitée. Pour cela, le prototype offre une visualisation graphique de l'automate (la Figure IV-l a été produite par cet outil). Cela nous a permis de vérifier manuellement le résultat pour un ensemble de diagrammes illustrant les différents opérateurs du langage. Dans l'étape de pré-traitement, le prototype extrait les atomes directement des diagrammes bruts exportés au format SVG (*Scalar Vector Graphics*). L'avantage était de pouvoir avoir des retours rapides sur les algorithmes, sans se soucier de spécialiser un éditeur ou un parser UML au profil TERMOS.

L'intégration de TERMOS dans la technologie support de UML a été étudiée par le deuxième prototype, issu de l'Université Technologique de Budapest. Le prototype a été développé comme un plug-in Eclipse. Un profil TERMOS a été défini pour les vues spatiales et événementielles ; cependant, il faut noter que l'éditeur de modèles utilisé, UML2Tools[15], ne permet pas de représenter des constructions telles que les messages perdus et trouvés, qui sont essentielles à TERMOS. Le prototype effectue des vérifications syntaxiques, et certaines vérifications sémantiques sur la cohérence des deux vues (il vérifie que l'émetteur et le récepteur d'un message sont connectés dans la configuration courante). L'extraction des atomes s'effectue à partir des éléments UML. L'automate est sauvé sous forme de fichier XML. Il permet de vérifier des traces dans un format textuel simple.

La connexion à *GraphSeq* n'a pas encore été réalisée.

IV.5. Conclusion et discussion

Ce chapitre a présenté une approche utilisant des scénarios graphiques pour tester des systèmes mobiles. Le langage formel TERMOS, basé sur les Diagrammes de Séquences UML, est au cœur de cette approche. Les concepts clés du langage sont les suivants :

- TERMOS incorpore trois extensions que nous avons trouvées utiles pour représenter des interactions dans un contexte mobile. La vue spatiale permet d'abstraire le mouvement des nœuds, ainsi que leurs apparitions et disparitions dynamiques, par une séquence de graphes étiquetés. La vue événementielle contient explicitement les événements de changement de configuration. Les événements de communication incluent la notion de diffusion dans le voisinage.
- TERMOS permet de spécifier des propriétés sur des sous-ensembles de nœuds exhibant des motifs de configuration spatiale prédéfinis. Les propriétés concernent les ordres partiels des événements de communication et de changement de configuration. Elles peuvent prendre trois formes : exigence positive, exigence négative ou objectif de test.
- La sémantique de TERMOS combine appariement de graphes et une sémantique opérationnelle des Diagrammes de Séquences basée sur un automate à états.

Le but de TERMOS est de vérifier des traces d'exécution. Nous n'avons pas encore une chaine à outil complète permettant de le faire, mais les composants majeurs sont là.

[15] http://wiki.eclipse.org/MDT-UML2Tools

L'appariement de séquences de graphes a été implémenté dans l'outil *GraphSeq* [NGu 09] [NWR 10]. La construction de l'automate a été consolidée via deux prototypes [WMR+ 10]. Le développement d'un troisième prototype est en cours au LAAS. Il reprend le principe d'une intégration aux outils UML, mais avec l'éditeur Papyrus[16] plutôt que UML2Tools. L'interfaçage avec *GraphSeq* est également dans nos proches perspectives.

Les algorithmes développés pour traiter la vue événementielle de TERMOS sont de facture classique. De fait, l'interprétation de scénarios graphiques a été très étudiée. Les sémantiques proposées pour la version 2 des Diagrammes de Séquences UML s'appuient sur un solide corpus de travaux sur des notations antérieures, incluant notamment les *Message Sequence Charts* (MSC) et les *Live Sequence Charts* (LSC). L'intérêt des LSC est que cette notation possédait déjà des modalités pour distinguer les comportements potentiels et nécessaires, et que l'introduction d'opérateurs tels que *assert* et *neg* constitue précisément une des évolutions majeures entre les versions 1 et 2 des Diagrammes de Séquences. Nous avons trouvé plusieurs sémantiques des Diagrammes de Séquences revendiquant une filiation avec les LSC [CK 04] [Kus 06] [KW 06] [HM 08] : la sémantique de TERMOS s'inscrit dans cette même filiation. Au-delà des détails des algorithmes développés pour ce langage, une contribution intéressante réside dans la démarche mise en œuvre pour déterminer nos choix d'interprétation. Nous sommes partis d'un échantillon de sémantiques formelles pour établir une catégorisation des choix et des options relatives à chaque choix [MW 10]. On peut espérer que cette catégorisation sera utile à d'autres praticiens UML qui – comme nous – auront à se repérer dans la multitude des sémantiques existantes, pour déterminer ce qui correspond le mieux à leurs besoins.

L'algorithme développé pour comparer des séquences de graphes est, à ma connaissance, une contribution originale. Le cadre classique de l'appariement de graphes est la comparaison de deux graphes, et il y a eu comparativement peu de travaux sur les aspects séquentiels. En particulier, les travaux appliqués à l'analyse de la topologie de réseaux se sont intéressés à des problèmes différents de celui que nous avons traité [BDK+ 07]. Typiquement, ces problèmes s'apparentent à de l'appariement de séquences de caractères, où l'isomorphisme de graphes remplace la notion d'égalité des caractères. Par exemple, on cherche à savoir si "$g_2\ g_3$" apparaît comme sous-séquence de "$g_1\ g_2\ g_3\ g_4$", ou encore on veut connaître la distance entre "$g_1\ g_2\ g_3\ g_4$" et "$g_1\ g_5\ g_3\ g_4$". Ce type d'analyse permet de comparer l'évolution courante du réseau à des historiques connus, pour détecter des anomalies ou reconstituer des informations manquantes. La comparaison de deux séquences $g_1 \ldots g_m$ et $G_1 \ldots G_n$, où les g_i sont des sous-graphes des G_j pendant des fenêtres temporelles variables, a été traitée dans le cadre de l'analyse d'images vidéo [SVB 01]. Cependant, les auteurs n'ont pas considéré la forme générique du problème, où la valuation des variables peut induire plusieurs instances des sous-graphes motif dans les G_j. Les travaux réalisés autour de TERMOS permettent ainsi d'étendre des travaux existants sur l'appariement séquentiel, tout en leur offrant un nouveau domaine d'application : le test de systèmes mobiles.

[16] http://wiki.eclipse.org/MDT/Papyrus

Programme de recherche

Mon programme de recherche pour les années à venir s'articule autour de trois axes principaux.

L'axe *Test de logiciels aspectisés* peut être vu comme relevant de la même philosophie que les travaux présentés en chapitre I de ce mémoire. Il s'agit d'adapter la conception du test à la technologie de développement logicielle utilisée. La technologie considérée est ici la programmation orientée-aspect. Elle est vue comme un moyen permettant d'implémenter des architectures réflexives pour la tolérance aux fautes.

L'axe *Test statistique basé sur des métaheuristiques d'optimisation* témoigne de mon intérêt constant pour les procédés de génération probabilistes et heuristiques (voir par exemple le chapitre III). Certains développements récents [PC 10] ouvrent de nouvelles perspectives quant à l'association de ces procédés. Ces nouvelles perspectives m'amènent à remettre le test statistique au premier plan en tant qu'objet de recherche.

Enfin, l'axe *Des systèmes mobiles aux systèmes ubiquitaires* vise d'une part à poursuivre les travaux présentés dans le chapitre IV de ce mémoire, et d'autre part à faire évoluer mes recherches vers le test de systèmes ubiquitaires à grande échelle.

Test de logiciels aspectisés

La programmation orientée-aspect (*Aspect Oriented Programming*) permet de développer des formes de spécialisation de programme de façon transverse au code applicatif. En d'autres termes, cette technique de programmation permet d'ajouter un ou plusieurs mécanismes particuliers, les aspects, à des programmes sans avoir à modifier leur code. Les points d'attachement (*joint-points*) des aspects sont définis par des expressions régulières. Le processus de composition avec le code applicatif est appelé "tissage".

Bien que cette technique soit conceptuellement attractive, le test de logiciels aspectisés reste problématique. Les travaux pionniers dans le domaine ont notamment considéré :

- les fautes de programmation spécifiques aux langages orientés aspects [BA 06] [AX 08] [FMR 08] ;
- les étapes de test unitaire [LN 05] [AX 06], d'intégration [LFM 09] et de non régression [SG 05] [DMB+ 10] pour tester graduellement le code applicatif et les aspects ;
- les critères de couverture applicables à un logiciel aspectisé [Zha 03] [XX 06] [WG 10] ;
- l'enrichissement de l'oracle de test avec des observations spécifiques au comportement des aspects [ZR 03] [DBG+ 09].

Ce sont ainsi tous les problèmes fondamentaux du test qui doivent être revisités pour s'adapter à la technologie des aspects.

L'angle sous lequel je compte attaquer ces problèmes correspond à un cadre particulier d'utilisation des aspects. Ces derniers sont vus comme un moyen d'introduire des capacités d'adaptation et de tolérance aux fautes dans les systèmes informatiques. Les travaux seront conduits conjointement avec des collègues du groupe TSF du LAAS, qui travaillent depuis plusieurs années sur la conception d'architectures réflexives pour la tolérance aux fautes [RKF+ 03] [Fab 10]. Nous chercherons à définir des patrons de conception pour des mécanismes de base dans de telles architectures, ainsi que des règles permettant leur implémentation via un usage maîtrisé des constructions offertes par les langages orientés aspect. Ce cadre bien défini devrait faciliter la définition de stratégies de test spécialisées. Nous nous intéresserons à la modularité de la vérification (détermination d'étapes de test incrémentales, qui exploitent les vérifications effectuées aux étapes précédentes), à la sélection de tests basée sur des modèles des mécanismes, et aux architectures de test permettant de commander et observer le logiciel aspectisé. De premiers travaux ont déjà démarré sur l'observation d'interférences indésirables lorsque plusieurs mécanismes de tolérance aux fautes sont greffés en un même point du code de base [LFW 11].

Il est à noter que la finalité des mécanismes introduits par les aspects devra être prise en compte dans nos travaux, en ce sens que les tests devront considérer non seulement l'activité fonctionnelle du logiciel cible, mais aussi les fautes à tolérer et les changements auxquels s'adapter.

Test statistique basé sur des métaheuristiques d'optimisation

Ainsi que rappelé en introduction du mémoire, mes travaux de thèse avaient concerné le développement d'une approche probabiliste de génération de test, le test statistique. Cette approche consiste à exécuter un programme avec des entrées aléatoires, le profil de test et le nombre d'entrées à générer étant déterminés à partir de critères de couverture structurelle ou fonctionnelle. La génération probabiliste permet de compenser l'imperfection des critères existants vis-à-vis des fautes recherchées. Un point dur est cependant la détermination du profil de test, pour lequel on ne peut pas toujours obtenir de solution analytique. Dans ce cas, nous avons proposé une approche empirique nécessitant plusieurs itérations en partie manuelles. Le test statistique a été expérimenté avec succès sur des logiciels de contrôle commande provenant de différents domaines d'application : aéronautique, ferroviaire ou nucléaire.

Une autre approche de production d'entrées de test consiste à utiliser des métaheuristiques issues de travaux en recherche opérationnelle, telles que le recuit simulé ou les algorithmes génétiques. Un objectif de test (par exemple, activer une branche particulière du programme) est alors reformulé en un problème d'optimisation que l'heuristique va chercher à résoudre en exhibant des entrées adéquates. Cette approche a suscité beaucoup d'intérêt de la part de la communauté du test du logiciel, et a permis de traiter des objectifs très variés. Le chapitre III a présenté mes propres contributions dans le domaine.

Partant de nos travaux antérieurs sur le test statistique, des chercheurs de l'Université de York (Royaume Uni) ont récemment suggéré une nouvelle application possible des métaheuristiques : la détermination automatique de profils de test [PC 10]. L'objectif de la recherche n'est alors plus d'exhiber des données d'entrée – comme dans le cadre d'une utilisation classique des métaheuristiques – mais d'exhiber une distribution des probabilités sur le domaine d'entrée (le profil de test), à partir de laquelle les entrées vont être générées. Cette mise en œuvre du test statistique me paraît prometteuse. Des premières investigations

communes ont été menées, incluant un exemple industriel traité lors de mes travaux de thèse [PCW 11].

La mise en œuvre du test statistique basée sur des métaheuristiques soulève plusieurs problèmes intéressants. L'un d'eux est comment encoder les profils de test. Les solutions retenues devront notamment permettre de représenter les différents exemples de profils issus de mes travaux, tout en se prêtant aux manipulations requises par la recherche métaheuristique. Un deuxième problème concerne le paramétrage des métaheuristiques, en particulier le choix de la fonction objectif à optimiser et le choix des opérateurs pour parcourir l'espace de recherche. Il est crucial pour l'efficacité du test statistique que la fonction objectif prenne en compte un critère de diversité des données de test : il faut pénaliser les profils dégénérés sélectionnant toujours les mêmes valeurs pour couvrir un sous-domaine. Les opérateurs pour parcourir l'espace de recherche concernent la production de nouveaux profils de test à partir des profils déjà produits : ils impliquent de définir des notions de voisinage et de combinaison de profils. Enfin, un dernier problème est comment injecter des connaissances dans le processus de recherche heuristique, pour mieux le guider. On pourra considérer des informations automatiquement extraites d'analyses statiques du programme à tester, ou des connaissances rentrées manuellement, telles qu'une identification sommaire de classes de valeurs d'intérêt.

Ces travaux devraient permettre de compléter mes contributions passées sur le test statistique du logiciel. En particulier, le fait de disposer d'un cadre automatisé de recherche de profils de test va faciliter la conduite d'expériences contrôlées sur le comportement du test statistique, par exemple pour évaluer la tolérance vis-à-vis de variations dans les distributions de probabilités résultantes, ou pour évaluer l'impact de la sévérité des critères de couverture retenus. Au-delà de l'étude du test statistique, le saut conceptuel consistant à considérer un espace de recherche des *stratégies* de génération, et non plus des *données* à générer, me paraît propre à ouvrir de nouvelles perspectives pour l'utilisation de métaheuristiques dans le cadre du test.

Des systèmes mobiles aux systèmes ubiquitaires

Mes travaux continueront de s'intéresser à la problématique du test dans le cadre de l'informatique mobile sûre de fonctionnement. Le chapitre IV a présenté une approche basée sur des scénarios graphiques qui incluent une représentation explicite des configurations spatiales des nœuds du système. L'utilisation des scénarios pour vérifier des traces de test met alors en jeu à la fois des algorithmes d'appariement de graphes et des algorithmes de calcul d'ordres partiels d'événements.

Dans la continuité de ces travaux, une première perspective concerne l'algorithmique de traitement des scénarios. La combinatoire de l'appariement de graphes est notamment très coûteuse, ce qui limite l'application de l'approche. Il faudra étudier les optimisations possibles en termes de temps de calcul et d'espace mémoire. On pourra envisager la mise en œuvre de pré-traitements, ainsi qu'un entrelacement astucieux des algorithmes d'appariement de graphes et de calcul d'ordres partiels, pour permettre une meilleure efficacité.

Une deuxième perspective est l'enrichissement de la vue spatiale des scénarios, pour une meilleure prise en compte de la dépendance vis-à-vis du contexte environnant. Les nœuds mobiles sont en effet susceptibles de s'adapter à un contexte complexe, dû au niveau d'instrumentation élevé de l'environnement dans lequel ils évoluent. Il faudra proposer une extension des descriptions spatiales existantes qui aille au-delà de la représentation de quelques attributs contextuels simples. Par ailleurs, les scénarios actuels prennent mal en compte le caractère stable ou instable des configurations rencontrées, ce qui nécessiterait par

exemple d'attacher des durées temporelles minimales ou maximales aux configurations, ou encore de redéfinir la granularité des changements de configurations avec différentes trajectoires transitoires possibles entre deux configurations stables.

Les perspectives précédentes se placent dans le cadre de tests réalisés en environnement de simulation, avant déploiement d'applications et services pour des dispositifs mobiles. Elles peuvent être considérées comme insuffisantes vis-à-vis de systèmes, qualifiés d'*ubiquitaires*, caractérisés par l'interconnexion à grande échelle de constituants très hétérogènes (dispositifs mobiles, mais aussi super-calculateurs, fermes de serveurs, myriade de petits systèmes enfouis dans des objets de la vie courante, …). De tels systèmes vont combiner des problématiques liées à la mobilité, à la fusion de données, et à la composition de services développés et maintenus en dehors des applications cible. Il sera illusoire de chercher à reproduire un échantillon représentatif des situations opérationnelles par des tests en laboratoire. Aussi, au-delà de la continuation de mes travaux actuels sur le test de système mobile, mes recherches à plus long terme s'orienteront-elles vers le test (actif) en-ligne. Cette thématique s'inscrit dans le cadre de réflexions en cours dans notre groupe au LAAS, sur des approches proactives pour garantir la persistance de la sûreté de fonctionnement dans les systèmes ubiquitaires.

Le test en-ligne est mentionné comme un des grands défis actuels, dans un article de prospective récent [Ber 07]. Il apparaissait déjà dans [OGP 03] comme une recommandation pour réduire les taux de défaillance des services Internet. Des exemples récents de méthodes proactives de test en vie opérationnelle sont fournis par [BP 05] et [CMK 08]. J'envisage deux rôles pour le test en-ligne : (1) révéler les fautes dont les conditions d'activation sont difficilement reproductibles par des tests en laboratoire, (2) permettre à un composant d'acquérir des informations sur son environnement, par exemple pour guider des reconfigurations adaptatives. Les problèmes soulevés concernent la définition d'une politique pour déterminer quand lancer les tests, la spécialisation de tests génériques à un contexte opérationnel courant, ou encore la minimisation des interférences sur le fonctionnement du système.

Références Bibliographiques

[Abd 03] Abdellatif-Kaddour, O. 2003. Contribution au Test Orienté Propriété pour des Systèmes de Contrôle-Commande : Construction Incrémentale de Scénarios de Test Sélectionnés par la Méthode du Recuit Simulé. Doctorat de l'Institut National Polytechnique de Toulouse.

[ABL 96] Abrial, J-R., Börger, E., and Langmaar, H. (Eds) 1996. *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*. Springer Verlag.

[ABM 98] Ammann, P.E., Black, P.E., Majurski, W. 1998. Using Model Checking to Generate Tests from Specifications. *Proc. 2nd IEEE Int. Conf. on Formal Engineering Methods (ICFEM'98)*, IEEE Computer Society, 46-54.

[Abr 96] Abrial, J.R. 1996. *The B-Book – Assigning Programs to Meanings*. Cambridge University Press.

[AL 81] Anderson, T., Lee, P.A. 1981. *Fault Tolerance: Principles and Practice*, Prentice Hall.

[ALR+ 04] Avizienis, A., Laprie, J-C., Randell, B., Landwehr, C.E. 2004. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. on Dependable and Secure Computing*, 1(1): 11-33.

[AO 08] Ammann, P., Offutt,J. 2008. *Introduction to Software Testing*. Cambridge University Press.

[Arc 09] Arcuri, A. 2009. Theoretical Analysis of Local Search in Software Testing. *Proc. Symp. on Stochastic Algorithms, Foundations and Applications (SAGA)*, 156-168.

[ATW 03a] Abdellatif-Kaddour, O., Thevenod-Fosse, P., and Waeselynck, H. 2003a. Property-Oriented Testing: A Strategy for Exploring Dangerous Scenarios. *Proc. ACM Symposium on Applied Computing (SAC'2003)*, Melbourne, USA, 1128-1134.

[ATW 03b] Abdellatif-Kaddour, O., Thevenod-Fosse, P., and Waeselynck, H. 2003b. An Empirical Investigation of Simulated Annealing Applied to Property-Oriented Testing. *Proc. ACS/IEEE Int. Conf. on Computer Systems and Applications (AICCSA'03)*.

[AW 95] Avritzer, A., Weyuker, E.J. 1995. The Automatic Generation of Load Test Suites and the Assessment of the Resulting Software. *IEEE trans. on Software Engineering*, 21(9): 705-716

[AX 06] Anbalagan, P., Xie, T. 2006. APTE: automated pointcut testing for AspectJ programs. *Proc. 2nd Workshop on Testing Aspect-Oriented Programs (WTAOP '06)*, ACM, 27-32.

[AX 08] Anbalagan, P., Xie, T. 2008. Automated Generation of Pointcut Mutants for Testing Pointcuts in AspectJ Programs. *Proc. 19th Int. Symp. on Software Reliability Engineering (ISSRE'08)*, IEEE Computer Society, 239-248.

[AZ 98] Angel, E., and Zissimopoulos, V. 1998. Autocorrelation Coefficient for the Graph Bipartitioning Problem. *Theoretical Computer Science*, 191: 229-243.

[AZ 00] Angel, E., and Zissimopoulos, V. 2000. On the Classification of NP-Complete Problems in terms of their Correlation Coefficient. *Discrete Applied Mathematics*, 99(1-3): 261-277.

[BA 00] Bushnell, M., Agrawal, V. 2000. *Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits.* Kluwer Academic Publishers.

[BA 06] Bækken, J.S., Alexander, R.T. 2006. A Candidate Fault Model for AspectJ Pointcuts. *Proc. 17th Int. Symp. on Software Reliability Engineering (ISSRE'06)*, IEEE Computer Society, 169-178.

[BB 94] Breuer, P.T., Bowen, J.P. 1994. Towards correct executable semantics for Z. *Proc. 8th Z User Workshop*, Springer/BCS, 185-209.

[BBF+ 99] Behm, P., Benoît, P., Faivre, A., Meynadier, J-M. 1999. Météor: a Successful Application of B in a Large Project. *Proc. World Congress on Formal Methods (FM'99)*, LNCS 1708, Springer Verlag, 369-387.

[BBM 98] Behm, P., Burdy, L., Meynadier, J-M. 1998. Well Defined B. *Proc. 2nd Int. B Conference*, LNCS 1393, 29-45.

[BBP 98] Barbey, S., Buchs, D., Péraire, C. 1998. Modelling the Production Cell Case Study Using the Fusion Method. Technical Report 98/298, EPFL-DI.

[BCM 00] Burton, S., Clark, J., McDermid, J. 2000. Testing, Proof and Automation – An Integrated Approach. *Proc. 1st Int. Workshop on Automated Program Analysis, Testing and Verification*.

[BDK+ 07] Bunke, H., Dickinson, P.J., Kraetzl, M., Wallis, W.D. 2007. *A Graph-Theoretic Approach to Enterprise Network Dynamics*, Series: Progress in Computer Science and Applied Logic (PCS), Vol. 24, Birkhäuser. Voir le chapitre 8 : Matching Sequences of Graphs, 131-143.

[BDM 97] Behm, P., Desforges, P., Mejia, F. 1997. Application de la méthode B dans l'industrie ferroviaire. *Application des techniques formelles au logiciel*, Arago 20, OFTA, 59-87.

[Beh 00] Behnia, S. 2000. Test de modèles formels en B : cadre théorique et critères de couverture. Doctorat de l'Institut National Polytechnique de Toulouse.

[Bel 01] Belaidouni, M. 2001. Métaheuristiques et paysages de recherche. Doctorat de l'Université d'Angers.

[Ber 07] Bertolino, A. 2007. Software Testing Research: Achievements, Challenges, Dreams. *Proc. Future of Software Engineering (FOSE'07)*, IEEE Computer Society, 85-103.

[BFL 02] Briand, L., Feng, J., Labiche, Y. 2002. Using Genetic Algorithms and Coupling Measures to Devise Optimal Integration Test Orders. *Proc. 14th ACM Int. Conf. on Software Engineering and Knowledge Engineering (SEKE)*, 43-50.

[BFS 05] Belinfante, A., Frantzen, L., Schallhart, C. 2005. Tools for Test Case Generation. *Model-Based Testing of Reactive Systems*, LNCS 3472, Springer Verlag, 391-438.

[BGK+ 02] Bhargavan, K., Gunter, C.A., Kim, M., Lee, I., Obradovic, D., Sokolsky, O., Viswanathan, M. 2002. Verisim: Formal Analysis of Network Simulations. *IEEE trans. on Software Engineering*, 28(2): 129-145.

[Bin 96] Binder, R.V. 1996. Testing Object-Oriented Software : A Survey. *Software Testing, Verification & Reliability (STVR)*, 6(3/4): 125-252.

[BKK+ 04] de Bruin, D., Kroon, J., van Klaveren, R., Nelisse, M. 2004. Design and Test of a Cooperative Adaptive Cruise Control System. *Proc. Intelligent Vehicles Symposium*, IEEE, 392-396.

[BLL 06] Briand, L., Labiche, Y., Leduc, J. 2006. Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software. *IEEE Trans. on Software Engineering*, 32(9), 642-663.

[BLL+ 08] Bendisposto, J., Leuschel, M., Ligot, O., Samia, M. 2008. La validation de modèles Event-B avec le plug-in ProB pour RODIN. *Technique et Science Informatiques*, 27(8): 1065-1084.

[BLM+ 04] du Bousquet, L., Ledru, Y., Maury, O., Oriat, C., Lanet, J.-L. 2004. A Case Study in JML-Based Software Validation. *Proc. 19th IEEE Int. Conf. on Automated Software Engineering (ASE'04)*, IEEE Computer Society, 294-297.

[BLS 02] Briand, L.C., Labiche, Y., Sun, H. 2002. Investigating the Use of Analysis Contracts to Support Fault Isolation in Object Oriented Code. *Proc. 2002 ACM SIGSOFT Int. Symp. on Software testing and analysis (ISSTA'02)*, ACM, 70-80.

[BLW 03] Briand, L.C., Labiche, Y., Wang, Y. 2003. An Investigation of Graph-Based Class Integration Test Order Strategies. *IEEE Trans. on Software Engineering*, 29(7): 594-607.

[BNR 03] Ball, T., Naik, M., Rajamani, S.K. 2003. From Symptom to Cause: Localizing Errors in Counterexample Traces. *Proc. 30th SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL 2003)*, ACM, 97-105.

[Boc 09] Bochot, T. 2009. Vérification par Model Checking des commandes de vol : applicabilité industrielle et analyse de contre-exemples. Doctorat de l'Université de Toulouse, délivré par l'Institut Supérieur de l'Aéronautique et de l'Espace.

[BP 05] Bertolino, A., Polini, A. 2005. The Audition Framework for Testing Web Services Interoperability. *Proc. 31st Euromicro Conf. on Software Engineering and Advanced Applications (EUROMICRO-SEAA 2005)*, IEEE Computer Society, 134-142.

[BS 03] Baresel, A., and Sthamer, H. 2003. Evolutionary Testing of Flag Conditions. *Proc. Genetic and Evolutionary Computation Conference (GECCO 2003)*. Springer Verlag, LNCS 2724: 2442-2454.

[BVW+ 09] Bochot, T., Virelizier, P., Waeselynck, H., Wiels, V. 2009. Model Checking Flight Control Systems: the Airbus Experience. *Proc. 31st Int. Conf. on Software Engineering (ICSE Companion 2009)*, IEEE CS Press, 18-27.

[BVW+ 10] Bochot, T., Virelizier, P., Waeselynck, H., Wiels, V. 2010. Paths to property violation: a structural approach for analyzing counter-examples. *Proc. 12th IEEE Int. High Assurance Systems Engineering Symposium (HASE 2010*), IEEE CS Press, 74-83.

[BW 99] Behnia, S., Waeselynck, H. 1999. Test criteria definition for B models. *Proc. World Congress on Formal Methods (FM'99)*, LNCS 1708, Springer Verlag, 509-529.

[CAB+ 94] Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., Jeremaes, P. 1994. *Object-Oriented Development – The Fusion Method*. Object-Oriented Series, Prentice Hall.

[CFK 09] Cramer, M., Fisseni, B., Koepke, P., Kühlwein, D., Schröder, B., Veldman, J. 2009. The Naproche Project Controlled Natural Language Proof Checking of Mathematical Texts. *Proc. Workshop on Controlled Natural Language (CNL 2009)*, LNCS 5972, Springer Verlag, 170-186.

[CFS+ 04] Conte, D., Foggia, P., Sansone, C., Vento, M. 2004. Thirty Years of Graph Matching in Pattern Recognition. *Intl. Journal of Pattern Recognition and Artificial Intelligence*, 18(3), 265-298.

[CG 00] Cardelli, L., Gordon, A.D. 2000. Mobile Ambients. *Theoretical Computer Science*, 240(1), 177-213.

[CGM+ 04] Cavalli, A., Grepet, C., Maag, S., Tortajada, V. 2004. A Validation Model for the DSR Protocol. *Proc. 24th Int. Conf. on Distributed Computing Systems Workshops (ICDCSW'04)*, IEEE, 768-773.

[CK 04] Cavarra, A., Küster-Filipe, J. 2004. Formalizing Liveness-Enriched Sequence Diagrams Using ASMs. In W. Zimmermann & B. Thalheim (Eds) : *Abstract State Machines*, Springer, 62-77.

[CMK 08] Chu, M., Murphy, C., Kaiser, G. 2008. Distributed In Vivo Testing of Software Applications. *Proc. 1st Int. Conf. on Software Testing, Verification, and Validation (ICST'08)*, IEEE Computer Society, 509-512.

[CMM 09] Cavalli, A., Maag, S., Montes de Oca, E. 2009. A Passive Conformance Testing Approach for a MANET Routing Protocol. *Proc. ACM Symp. on Applied Computing (SAC'2009)*, ACM, 207-211.

[Con 90] Connolly, D.T. 1990. An Improved Annealing Scheme for the QAP. *European Journal of Operational research*, 46(1): 93-100.

[CSS 02] Cavin, D., Sasson, Y., Schiper, A. 2002. On the Accuracy of MANET Simulators. *Proc. 2nd ACM Workshop On Principles Of Mobile Computing (POMC'02)*, ACM, 38-43.

[Cuk 97] Cukic, B. 1997. Combining Testing and Correctness Verification in Software Reliability Assessment. *Proc. 2nd High-Assurance Systems Engineering Workshop (HASE '97)*, IEEE Computer Society, 182-187.

[CR 99] Creese, S.J, Roscoe, A.W. 1999. TTP: A case study in combining induction and data independence. Technical Report PRG-TR-1-99, Oxford Univ. Computing Laboratory.

[CR 02] Castanet, R., Rouillard, D. 2002. Generate Certified Test Cases by Combining Theorem Proving and Reachability Analysis. *Proc. IFIP 14th Int. Conf. on Testing Communicating Systems (TestCom 2002)*, Kluwer, 249-266.

[DBG+ 09] Delamare, R., Baudry, B., Ghosh, S., Le Traon, Y. 2009. A Test-Driven Approach to Developing Pointcut Descriptors in AspectJ. *Proc. 2nd Int. Conf. on Software Testing Verification and Validation (ICST 2009)*, IEEE Computer Society, 376-385.

[DEF 03] Dollé, D., Essamé, D., Falampin, J. 2003. B dans le transport ferroviaire – L'expérience de Siemens Transportation Systems. *Technique et Science Informatiques*, 22(1): 11-32.

[DF 93] Dick, J., Faivre, A. 1993. Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. *Proc. Int. Formal Methods Europe Symposium (FME'93)*, 268-284.

[DF 94] Doong, R-K., Frankl, P.G. 1994. The ASTOOT Approach to Testing Object-Oriented Programs. *ACM Trans. on Software Engineering and Methodology (TOSEM)*, 3(2): 101-130.

[DGG 04] Denise, A., Gaudel, M-C., Gouraud, S-D. 2004. A Generic Method for Statistical Testing. *Proc. 15th Int. Symp. on Software Reliability Engineering (ISSRE 2004)*, IEEE Computer Society, 25-34.

[DH 01] Damm, W., Harel, D. 2001. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1), 45-80.

[Dij 76] Dijkstra, E.W. 1976. *A Discipline of Programming*. Prentice Hall.

[DMB+ 10] Delamare, R., Munoz, F., Baudry, B., Le Traon, Y. 2010. Vidock: A Tool for Impact Analysis of Aspect Weaving on Test Cases. *Proc. 22nd IFIP WG 6.1 Int. Conf. on Testing Software and Systems (ICTSS 2010)*, Springer Verlag, LNCS 6435, 250-265.

[DMM+ 99] Dong, L., Melhem, R.G., Mossé, D., Ghosh, S., Heimerdinger, W.L., Larson, A. 1999. Implementation of a Transient-Fault-Tolerance Scheme on DEOS – A Technology

Transfer from an Academic System to an Industrial System. *Proc. 5th Real Time Technology and Applications Symposium (RTAS'99)*, IEEE Computer Society, 56-66.

[DN 84] Duran, J.W., Ntafos, S.C. 1984. An Evaluation of Random Testing. *IEEE Trans. Software Engineering*, 10(4): 438-444.

[DS 01] Devarapalli, V., Sidhu, D. 2001. MZR: A Multicast Protocol for Mobile Ad Hoc Networks. *Proc. IEEE Int. Conf. on Communications (ICC 2001)*, IEEE, 886-891.

[DSF 06] Duprat, S., Souyris, J., Favre-Felix, D. 2006. Formal Verification Workbench for Airbus Avionics Software. *Proc. Embedded Real-Time Software (ERTS 2006)*, http://www.sia.fr/evenement_detail_erts2006_embedded_real_time_actes_365.htm.

[EKM+ 99] Egan, A., Kutz, D., Mikulin, D., Melhem, R.G., Mossé, D. 1999. Fault-Tolerant RT-Mach (FT-RT-Mach) and an Application to Real-Time Train Control. *Software – Practice and Experience*, 29(4): 379-395.

[Fab 10] Fabre, J-C. 2010. Architecting Dependable Systems Using Reflective Computing: Lessons Learnt and Some Challenges. *Architecting Dependable Systems VII*, Springer Verlag, LNCS 6420, 273-296.

[FFJ+ 10] Falcone, Y., Fernandez, J-C., Jéron, T., Marchand, H., Mounier, L. 2010. More Testable Properties. *Proc. 22nd IFIP WG 6.1 Int. Conf. on Testing Software and Systems (ICTSS 2010)*, LNCS 6435, Springer Verlag, 30-46.

[FG 09] Fraser, G., Gargantini, A. 2009. An Evaluation of Model Checkers for Specification Based Test Case Generation. *Proc. 2nd Int. Conf. on Software Testing, Verification, and Validation (ICST 2009)*, IEEE Computer Society, 41-50.

[FLS 08] Fitzgerald, J.S., Larsen, P.G., Sahara, S. 2008. VDMTools: Advances in Support for Formal Modeling in VDM. *SIGPLAN Notices*, 43(2): 3-11.

[FMR 08] Ferrari, F.C., Maldonado, J.C., Rashid, A. 2008. Mutation Testing for Aspect-Oriented Programs. *Proc. 1st Int. Conf. on Software Testing, Verification, and Validation (ICST'08)*, IEEE Computer Society, 52-61.

[Fuc 92] Fuchs, N.E. 1992. Specifications Are (Preferably) Executable. *Software Engineering Journal*, 7(5): 323-334.

[GD 06] Guennoun, K., Drira, K. 2006. Using Graph Grammars for Interaction Style Description – Application to Service-Oriented Architectures. *Int. Journal on Computer Systems Science Engineering*, 21(4), 293-299.

[GH 99] Gargantini, A., Heitmeyer, C. 1999. Using Model Checking to Generate Tests from Requirements Specifications. *Proc. 7th Eur. Software Engineering Conference (ESEC/FSE'99)*, LNCS 1687, Springer Verlag, 146-162.

[GHN 93] Grabowski, J., Hogrefe, D., Nahm, R. 1993. Test Case Generation with Test Purpose Specification by MSCs. *Proc 6th SDL Forum (SDL'93)*, North-Holland, 253–266.

[GJJ+ 03] Gaucher, F., Jahier, E., Jeannet, B., Maraninchi, F. 2003. Automatic State Reaching for Debugging Reactive Programs. *Proc. 5th Int. Workshop on Automated Debugging (AADEBUG'2003)*. http://arxiv.org/html/cs.SE/0309027

[GKS 05] Godefroid, P., Klarlund, N., Sen, K. 2005. DART: Directed Automated Random Testing. *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2005)*, ACM, 213–223.

[GMM 97] Ghosh, S., Melhem, R., Mossé, D. 1997. Fault-Tolerant Rate-Monotonic Scheduling. *Proc. 6th IFIP Conf. on Dependable Computing for Critical Applications (DCCA-97)*, 121-145.

[GMM+ 98] Ghosh, S., Melhem, R., Mossé, D., Sarma, J.S. 1998. Fault-Tolerant Rate-Monotonic Scheduling. *Real-Time Systems*, (15)2: 149-181.

[GPY 02] Groce, A., Peled, D., Yannakakis, M. 2002. Adaptive Model Checking. *Proc. 8th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002)*, LNCS 2280, Springer Verlag, 357-370.

[Gue 06] Guennoun, K. 2006. Architectures Dynamiques dans le Contexte des Applications à Base des Composants et Orientés Services. Doctorat de l'Université Toulouse III.

[GV 03] Groce, A., Visser, W. 2003. What Went Wrong: Explaining Counterexamples. *Proc. of 10th Int. Conf. on Model Checking Software (SPIN'03)*, Springer Verlag, 121-136.

[HBB+ 09] Hierons, R.M., Bogdanov, K., Bowen, J.P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Luettgen, G., Simons, A.J.H., Vilkomir, S., Woodward, M.R., Zedan, H. 2009. Using formal specifications to support testing. *ACM Computing Surveys*, 41(2).

[HCR+ 91] Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D. 1991. The Synchronous Dataflow Programming Language Lustre. *Proceedings of the IEEE*, 79(9): 1305-1320.

[HDW 04] Heimdahl, M.P.E., Devaraj, G., Weber, R.J. 2004. Specification Test Coverage Adequacy Criteria = Specification Test Generation Inadequacy Criteria? *Proc. 8th IEEE Int. Symp. on High Assurance Systems Engineering (HASE 04)*, IEEE Computer Society, 178-186.

[Hie 97] Hierons, R.M. 1997. Testing from a Z Specification. *Software Testing, Verification & Reliability*, 7(1): 19-33.

[HJ 89] Hayes, I., Jones, C.B. 1989. Specifications are not (necessarily) executable. *Software Engineering Journal*, 4(6): 330-338.

[HJR 04] Huang, Q., Julien, C., Roman, G. 2004. Relying on Safe Distance to Achieve Strong Partitionable Group Membership in Ad Hoc Networks. *IEEE Trans. on Mobile Computing*, 3(2), 192-205.

[HKC 95] Hong, H.S., Kwon, Y.R., Cha, S.D. 1995. Testing of Object-Oriented Programs Based on Finite State Machines. *Proc. 2nd Asia Pacific Software Engineering (ASPEC'95)*, 234-241.

[HLR 93] Halbwachs, N., Lagnier, F., Raymond, P. 1993. Synchronous Observers and the Verification of Reactive Systems. *Proc. 3rd Int. Conf. on Algebraic Methodology and Software Technology (AMAST'93)*, Springer Verlag, 83-96.

[HM 08] Harel, D., Maoz, S. 2008. Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams. *Software and Systems Modeling*, 7(2), 237-253.

[HM 10] Harman, M., and McMinn, P. 2010. A Theoretical and Empirical Study of Search Based Testing: Local, Global and Hybrid Search. *IEEE Trans. on Software Engineering*, 36(2): 226-247.

[HMF 92] Harrold, M-J., McGregor, J., and Fitzpatrick, K. 1992. Incremental Testing of Object-Oriented Programs. *Proc. 14th IEEE Int. Conf. on Software Engineering (ICSE-14)*, 68-80.

[HMR 04] Hamon, G., de Moura, L., Rushby, J.M. 2004. Generating Efficient Test Sets with a Model Checker. *Proc. 2nd Int. Conf. on Software Engineering and Formal Methods (SEFM 2004)*, IEEE Computer Society, 261-270.

[HMZ 09] Harman, M., Mansouri, S.A., and Zhang, Y. 2009. Search Based Software Engineering: A Comprehensive Analysis and Review of Trends Techniques and Applications. TR-09-03, King's College London, UK.

[Hol 75] Holland, J. 1975. *Adaptation in Natural and Artificial Systems*. University of Michigan Press.

[Hor 96] Hordijk, W. 1996. A measure of landscapes. *Evolutionary Computation*, 4(4): 335-360.

[HP 95] Hörcher, H.M., Peleska, J. 1995. Using Formal Specifications to Support Software Testing. *Software Quality Journal*, 4(4): 309-327.

[HPJ 05] Hu, Y-C., Perrig, A., Johnson, D.B. 2005. Ariadne: A Secure On-Demand Routing Protocol for Ad Hoc Networks. *Wireless Networks*, 11(1-2), 21-38.

[HR 94] Harrold, M-J., and Rothermel, G., 1994. Performing Data Flow Testing on Classes. *Proc. 2nd ACM SIGSOFT Symp. on the Foundations of Software Engineering*, 154-163.

[HSR 05] Hashim, N.L., Schmidt, H.W., Ramakrishnan, S. 2005. Test Order for Class-based Integration Testing of Java Applications. *Proc. 5th Int. Conf. on Quality Software (QSIC '05)*, 11-18.

[HSS 02] Hayashi, S., Sumitomo, R., Shii, K. 2002. Towards the animation of proofs – testing proofs by examples. *Theoretical Computer Science*, 272(1-2): 177-195.

[HT 90] Hamlet, R., Taylor, R. 1990. Partition Testing Does Not Inspire Confidence. *IEEE Trans. Software Engineering*, 16(12): 1402-1411.

[HWE+ 08] Huszerl, G., Waeselynck, H., Egel, Z., Kovi, A., Micskei, Z., N'Guyen, M.D., Pinter, G., Rivière, N. 2008. Refined Design and Testing Framework, Methodology and Application results. Hidenets D5.3 Deliverable, Project IST- FP6-STREP- 26979. http://www.hidenets.aau.dk/Public+Deliverables

[ITU 04] International Telecommunication Union. 2004. ITU-T Recommendation Z.120: Message Sequence Chart.

[JE 94] Jorgensen, P.C., Erickson, C. 1994. Object-oriented Integration Testing. *CACM*, 37(9): 30-38.

[JJL+ 99] Jéron, T., Jézéquel, J-M., Le Traon, Y., Morel, P. 1999. Efficient Strategies for Integration and Regression Testing of OO Systems. *Proc. 10th Int. Symp. on Software Reliability Engineering (ISSRE'99)*, 260-269.

[JMB 01] Johnson, D.B., Maltz, D.A., Broch, J. 2001. DSR: The Dynamic Source Routing Protocol for Multi-Hop Wireless Ad Hoc Networks. Chapitre 5 de *Ad Hoc Networking*, Addison-Wesley, 139-172.

[Jon 90] Jones, C.B. 1990. *Systematic Software Development using VDM (2nd Edition).* International Series in Computer Science, Prentice Hall.

[Kau 89] Kauffman, S. A. 1989. Adaptation on Rugged Fitness Landscapes. *Lectures in the Sciences of Complexity*, Addison-Wesley, 527-618.

[KG 94] Kopetz, H., Grünsteidl, G. 1994. TTP – A Protocol for Fault-Tolerant Real-Time Systems. *IEEE Computer*, 27(1): 14-23.

[KGV 83] Kirkpatrick, S., Gellat, C.D., and Vecchi, M.P. 1983. Optimization by Simulated Annealing. *Science*, 220(4598):671-680.

[KGH+ 95] Kung, D.C., Gao, J., Hsia, P., Lin, J., Toyoshima, Y. 1995. Class Firewall, Test Order, and Regression Testing of Object-Oriented Programs. *Journal of Object Oriented Programming*, 8(2): 51-65.

[Klo 03] Klose, J. 2003. Live Sequence Charts: a Graphical Formalism for the Specification of Communication Behavior. PhD thesis, Universität Oldenburg, Allemagne.

[KLR 97] Katz, S., Lincoln, P., Rushby, J. 1997. Low-Overhead Time-Triggered Group Membership. *Proc. 11th Int. Workshop on Distributed Algorithms (WDAG '97)*, LNCS 1320, Springer-Verlag, 155-169.

[KSH 07] Kugler, H., Stern, M.J., Hubbard, E.J.A. 2007. Testing Scenario-Based Models. *Proc. 10th Int. Conf. on Fundamental Approaches to Software Engineering (FASE'07),* LNCS 4422, Springer Verlag, 306-320.

[Kus 06] Küster-Filipe, J. 2006. Modelling Concurrent Interactions. *Theoretical Computer Science*, 351(2), 203–220.

[KW 06] Knapp, A., Wuttke, J. 2006. Model Checking of UML 2.0 Interactions. *Proc. MoDELS Workshops 2006*, LNCS 4364, Springer, 42-51.

[Lab 00] Labiche, Y. 2000. Contribution au test des logiciels orientés-objet : ordre de test, modèles et critères associés. Doctorat de l'Institut National Polytechnique de Toulouse.

[LAC 05] Ladani, B.T., Alcalde, B., Cavalli, A.R. 2005. Passive Testing – A Constrained Invariant Checking Approach. *Proc. 17th IFIP TC6/WG 6.1 Int. Conf. on Testing of Communicating Systems (TestCom 2005)*, LNCS 3502, Springer Verlag, 9-22.

[LBJ 06] Le Traon, Y., Baudry, B., Jézéquel, J-M. 2006. Design by Contract to Improve Software Vigilance. *IEEE Trans. Software Engineering*, 32(8): 571-586.

[LCT 06] Lu, H., Chan, W.K., Tse, T.H. 2006. Testing Context-Aware Middleware-Centric Programs: a Data Flow Approach and an RFID-Based Experimentation. *Proc. 14th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (SIGSOFT'06 / FSE-14)*, ACM, 242-252.

[LFM 09] Lemos, O.A.L., Franchin, I.G., Masiero, P.C. 2009. Integration testing of Object-Oriented and Aspect-Oriented programs: A structural pairwise approach for Java. *Science of Computer Programming*, 74(10), 861-878.

[LFW 11] Lauret, J., Fabre, J-C., Waeselynck, H. 2011. Detecting Interferences in Aspect Oriented Programs. *Proc. 13th European Workshop on Dependable Computing (EWDC'11)*, ACM, 93-98.

[LI 08] Lefticaru, R., and Ipate, F. 2008. A Comparative Landscape Analysis of Fitness Functions for Search-Based Testing. *Proc. 10th Int. Symp. on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2008)*, IEEE, 201-208.

[LL 95] Lewerens, C., Linder, T. (eds), 1995. *Formal Development of Reactive Systems – Case Study Production Cell*. LNCS 891, Springer-Verlag.

[LM 86] Lundy, M., and Mees, A.I.. 1986. Convergence of an Annealing Algorithm. Mathematical Programming, 34(1): 111-124.

[LN 05] Lopes, C.V., Ngo, T. 2005. Unit Testing Aspectual Behavior. *Proc. 1st Workshop on Testing Aspect-Oriented Programs (WTAOP '05)*.

[LNY 04] Leung, K.R.P.H., Ng, J.K-Y., Yeung, W.L. 2004. Embedded Program Testing in Untestable Mobile Environment: an Experience of Trustworthiness Approach. *Proc. 11th Asia-Pacific Software Engineering Conference (APSEC 04)*, IEEE, 430-437.

[LP 09] Lakehal, A., Parissis, I. 2009. Structural Coverage Criteria for LUSTRE/SCADE Programs. *Software Testing, Verification & Reliability*, 19(2): 133-154.

[LPU 04] Legeard, B., Peureux, F., Utting, M. 2004. Controlling Test Case Explosion in Test Generation from B Formal Models. *Software Testing, Verification & Reliability*, 14(2): 81-103.

[LTW+ 00] Labiche, Y., Thévenod-Fosse, P., Waeselynck, H., Durand, M-H. 2000. Testing levels for object-oriented software. *Proc. 22nd Int. Conf. on Software Engineering (ICSE'2000)*, ACM Press, 136-145.

[Lus 04] Lussier, G. 2004. Test guidé par la preuve : application à la vérification d'algorithmes de tolérance aux fautes. Doctorat de l'Institut National des Sciences Appliquées de Toulouse.

[LW 02] Lussier, G. Waeselynck, H. 2002. Informal Proof Analysis Towards Testing Enhancement. *Proc. 13th Int. Symp. on Software Reliability Engineering (ISSRE'2002)*, IEEE CS Press, 27-38.

[LW 04a] Lussier, G. Waeselynck, H. 2004a. Proof-Guided Testing: an Experimental Study. *Proc. 28th Ann. Int. Computer Software and Applications Conf. (COMPSAC'2004)*, IEEE CS Press, 528-533.

[LW 04b] Lussier, G. Waeselynck, H. 2004b. Deriving Test Sets from Partial Proofs. Guillaume Lussier, Hélène Waeselynck. *Proc. 15th Int. Symp. on Software Reliability Engineering (ISSRE'2004)*, IEEE CS Press, 14-24.

[MA 00] Marre, B., Arnould, A. 2000. Test Sequences Generation from LUSTRE Descriptions: GATeL. *Proc. 15th IEEE Int. Conf. on Automated Software Engineering (ASE 2000)*, IEEE Computer Society Press, 229-237.

[MB 00] Messmer, B., Bunke, H. 2000. Efficient Subgraph Isomorphism Detection: a Decomposition Approach. *IEEE Trans. on Knowledge and Data Engineering*, 12(2), 307-323.

[MCL 03] Malloy, B.A., Clarke, P.J., Lloyd, E.L. 2003. A Parameterized Cost Model to Order Classes for Class-based Testing of C++ Applications. *Proc. 14th Int. Symp. on Software Reliability Engineering (ISSRE 2003)*, 353-364.

[McM 04] McMinn, P. 2004. Search-Based Software Test Data Generation: A Survey. *Software Testing, Verification & Reliability*, 14(2): 105-156.

[MDC 95] Maniezzo, V., Dorigo, M., and Colorni, A. 1995. Algodesk: an Experimental Comparison of Eight Evolutionary Heuristics Applied to the Quadratic Assignment Problem. *European Journal of Operational research*, 81(1): 188-204.

[Mey 93] Meyer, B. 1993. Systematic Concurrent Object-Oriented Programming. *CACM*, 36(9): 56-80.

[MH 04] McMinn, P., and Holcombe, M. 2004. Hybridizing Evolutionary Testing with the Chaining Approach. *Proc. Genetic and Evolutionary Computation Conference (GECCO 2004)*. Springer Verlag, LNCS 3103, 1363-1374.

[MF 00] Merz, P., and Freisleben, B. 2000. Fitness Lanscape Analysis and Memetic Algorithms for the Quadratic Assignment Problem. *IEEE Trans. on Evolutionary Computation*, 4(4): 337-352.

[ML 05] Mao, C., Lu, Y. 2005. AICTO: An Improved Algorithm for Planning Inter-class Test Order. *Proc. 5th Int. Conf. on Computer and Information Technology (CIT'05)*, 927-931.

[MV 04] Medidi, S.R., Vik, K-H. 2004. Quality of Service-Aware Source-Initiated Ad-Hoc Routing. *Proc. 1st Ann. IEEE Comm. Soc. Conf. on Sensor and Ad Hoc Communications and Networks (SECON 2004)*, IEEE, 108-117.

[MW 10] Micskei, Z., Waeselynck, H. 2010. The Many Meanings of UML 2 Sequence Diagrams: a Survey. A paraître dans *Software and Systems Modeling*. Version en-ligne depuis 2010 : http://www.springerlink.com/content/6716hk1844h16694/.

[NGu 09] N'Guyen, M.D. 2009. Méthodologie de test de systèmes mobiles : une approche basée sur les scénarios. Doctorat de l'Université de Toulouse, délivré par l'Université Paul Sabatier.

[NMO 09] Nienaltowski, P., Meyer, B., Ostroff, J.S. 2009. Contracts for Concurrency. *Formal Aspects of Computing*, 21(4): 305-318.

[Nta 88] Ntafos, S.C. 1988. A Comparison of Some Structural Testing Strategies. *IEEE Trans. on Software Engineering*, 14(6): 868-874.

[NV 05] Noudem, F.N., Viho, C. 2005. Modeling, Verifying and Testing the Mobility Management in the Mobile IPv6 Protocol. *Proc. 8th Int. Conf. on Telecommunications (ConTEL 2005)*, Vol.2, IEEE, 619-626.

[NVR 68] Nugent, C.E., Vollman, T.E., Ruml, J. 1968. An Experimental Comparison of Techniques for the Assignment of Facilities to Locations. *Operations Research*, 16: 150-173.

[NWR 10] N'Guyen, M.D., Waeselynck, H. Rivière, N. 2010. GraphSeq: a Graph Matching Tool for the Extraction of Mobility Patterns. *Proc. 3rd IEEE Int. Conf. on Software Testing, Verification and Validation (ICST 2010)*, IEEE CS Press, 195-204.

[OGP 03] Oppenheimer, D., Ganapathi, A., Patterson, D.A. 2003. Why Do Internet Services Fail, and What Can Be Done About it? *Proc. 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)*.

[OMG 10] Object Management Group. 2010. OMG Unified Modeling Language™ (OMG UML), Superstructure. http://www.omg.org/spec/UML/2.3/Superstructure/PDF/.

[ORS+ 95] Owre, S, Rushby, J., Shankar, N., von Henke, F. 1995. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Trans. on Software Engineering*, 21(2): 107–125.

[OVW 98] O'Sullivan, M., Vössner, S., Wegener, J. 1998. Testing Temporal Correctness of Real-Time Systems – a New Approach using Genetic Algorithms and Cluster Analysis. *Proc. 6th Eur. Conf. on Software Testing, Analysis & Review (EuroSTAR 1998)*.

[PBC 93] Parrish, A.S., Borie, R.B., Cordes, D.W. 1993. Automated Flow Graph-Based Testing of Object-Oriented Software Modules. *Journal of Systems and Software*, 23(2) : 95-109.

[PC 10] Poulding, S.M., Clark, J.A. 2010. Efficient Software Verification: Statistical Testing Using Automated Search. *IEEE Trans. Software Engineering*, 36(6): 763-777.

[PCW 11] Poulding, S.M., Clark, J.A., Waeselynck, H. 2011. A Principled Evaluation of the Effect of Directed Mutation on Search-Based Statistical Testing. *Proc. 4th Int. Workshop on Search-Based Software Testing (SBST 2011)*, IEEE.

[Pel 03] Peled, D. 2003. Model Checking and Testing Combined. *Proc. 30th Int. Coll. on Automata, Languages and Programming (ICALP 2003)*, LNCS 2719, Springer Verlag, 47-63.

[Pfe 00] Pfeifer, H. 2000. Formal Verification of the TTP Group Membership Algorithm. *Proc. IFIP Int. Conf. on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XIII) and Protocol Specification, Testing and Verification (PSTV XX)*, Kluwer Academic Publishers, 3-18.

[Pfe 03] Pfeifer, H. 2003. Formal Analysis of Fault-Tolerant Algorithms in the Time-Triggered Architecture. Doctorat de l'Université de Ulm (Allemagne).

[PG 07] Petit, M., Gotlieb, A. 2007. Uniform Selection of Feasible Paths as a Stochastic Constraint Problem. *Proc. 7th Int. Conf. on Quality Software (QSIC 2007)*, IEEE Computer Society, 280-285.

[PJ 04] Pickin, S., Jézéquel, J-M. 2004. Using UML Sequence Diagrams as the Basis for a Formal Test Description Language. *Proc. 4th Int. Conf. on Integrated Formal Methods (IFM2004)*, LNCS 2999, Springer, 481-500.

[PK 90] Perry, D.E., Kaiser, G.E. 1990. Adequate Testing and Object-Oriented Programming. *Journal of Object-Oriented Programming*, 2(5): 13-19.

[PvH 01] Pfeifer, H., von Henke, F. 2001. Formal Analysis for Dependability Properties: the Time-Triggered Architecture Example. *Proc. 8th IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA 2001)*, 343-352.

[RFR 08] Robert, T., Fabre, J-C., Roy, M. 2008. On-line Monitoring of Real Time Applications for Early Error Detection. *Proc. 14th IEEE Pacific Rim Int. Symp. on Dependable Computing (PRDC 2008)*, IEEE CS, 24-31.

[RKF+ 03] Ruiz, J-C., Killijian, M-O., Fabre, J-C., Thévenod-Fosse, P. 2003. Reflective Fault-Tolerant Systems: From Experience to Challenges. *IEEE Trans. on Computers*, 52(2), 237-254.

[Rou 99] Rouzaud, Y. 1999. Interpreting the B-Method in the Refinement Calculus. *Proc. World Congress on Formal Methods (FM'99)*, LNCS 1708, Springer Verlag, 411-430.

[RS 02] Reidys, C.M., and Stadler, P.F., 2002. Combinatorial Landscapes. *SIAM Review*, 44: 3-54.

[Rus 00] Rushby, J. 2000. Verification Diagrams Revisited: Disjunctive Invariants for Easy Verification. *Proc. Computer-Aided Verification (CAV'00)*, LNCS 1855, Springer Verlag, 508-520.

[Rus 02] Rusu, V. 2002. Verification Using Test Generation Techniques. *Proc. Int. Symp. on Formal Methods Europe (FME'02)*, LNCS 2391, Springer Verlag, 252-271.

[Sai 96] Saiedian, H. (Ed.) 1996. An Invitation to Formal Methods. *IEEE Computer*, 29(4).

[SG 05] Stoerzer, M., Graf, J. 2005. Using Pointcut Delta Analysis to Support Evolution of Aspect-Oriented Software. *Proc. 21st IEEE Int. Conf. on Software Maintenance (ICSM'05)*, IEEE Computer Society, 653-656.

[SGD 95] Schultz, A.C., Grefenstette, J.J., and De Jong, K.A. 1995. Learning to Break Things: Adaptative Testing of Intelligent Controllers. *Handbook on Evolutionary Computation*, chapter G3.5. IOP Publishing Ltd. and Oxford University Press.

[SMA 05] Sen, K., Marinov, D., Agha, G. 2005. CUTE: a Concolic Unit Testing Engine for C. *Proc. 10th European Software Engineering Conference (ESEC/FSE)*, ACM, 263–272.

[Spi 94] Spivey, M. 1994. *La notation Z*. Collection Méthodologies du logiciel, Masson, 1994. Traduit de l'anglais par Michel Lemoine.

[SS 92] Stadler, P.F., and Schnabl, W. 1992. The Landscape of the Traveling Salesman Problem. *Physics Letters*, A 161(4): 337-344.

[SS 99] Sinha, P., Suri, N. 1999. Identification of Test Cases Using a Formal Approach. *Proc. 29th Ann. Int. Symp. on Fault-Tolerant Computing (FTCS-29)*, IEEE Computer Society, 314-321.

[SVB 01] Shearer, K., Venkatesh, S., Bunke, H. 2001. Video Sequence Matching via Decision Tree Path Following. *Pattern Recognition Letters*, 22(5), Elsevier, 479-492.

[Tai 91] Taillard, E.D. 1991. Robust Tabu Search for the Quadratic Assignment Problem. *Parallel Computing*, 17(4&5): 443-455.

[TD 99] Tai, K.-C., Daniels, F.J. 1999. Interclass Test Order for Object-Oriented Software. *Journal of Object-Oriented Programming*, 12(4): 18-25.

[TR 93] Turner, C.D., Robson, D.J. 1993. The State-Based Testing of Object-Oriented Programs. *Proc. IEEE Conf. on Software Maintenance*, 302-310.

[Tra 00] Tracey, N. 2000. A Search-Based Automated Test Data Generation Framework for Safety-Critial Software. Doctorat de University of York, UK.

[TW 93] Thévenod-Fosse, P., Waeselynck, H. 1993. STATEMATE Applied to Statistical Software Testing. *ACM SIGSOFT Software Engineering Notes*, 18(3): 99-109.

[TYC+ 04] Tse, T.H., Yau, S.S., Chan, W.K., Lu, H., Chen, T.Y. 2004. Testing Context-Sensitive Middleware-Based Software Applications. *Proc. 28th Annual Int. Computer Software and Applications Conference (COMPSAC 2004)*, vol. 1, IEEE CS Press, 458-465.

[Ull 76] Ullmann, J.R. 1976. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1), 31-52.

[Utt 01] Utting, M. 2001. Data Structures for Z Testing Tools. Working paper series, University of Waikato, Department of Computer Science, No. 01/4/2001.

[Vaa 91] Vaandrager, F.W. 1991. On the Relationship Between Process Algebra and Input/output Automata. *Proc. 6th Annual Symp. on Logic in Computer Science*, 387-398.

[VBL 97] Van Aertryck, L., Benveniste, M., Le Métayer, D. 1997. CASTING: A formally based software test generation method. *Proc. 1st Int. Conf. on Formal Engineering Methods (ICFEM'97)*, 101-110.

[VBL+ 10] Vos, T.E.J., Baars, A.I., Lindlar, F.F., Kruse, P.M., Windisch, A., and Wegener, J. 2010. Industrial Scaled Automated Structural Testing with the Evolutionary Testing Tool. *Proc. 3rd Int. Conf. on Software Testing, Verification and Validation (ICST 2010)*, IEEE, 175-184.

[Wae 93] Waeselynck, H. 1993. Vérification de logiciels critiques par le test statistique. Doctorat de l'Institut National Polytechnique de Toulouse.

[WAM+ 06] Waeselynck, H., Arlat, J., Majzik, I., Micskei, Z. 2006. Robustness Testing. Rapport LAAS no. 06815.

[WB 95] Waeselynck, H., Boulanger, J-L. 1995. The Role of Testing in the B Formal Development Process. *Proc. 6th Int. Symp. on Software Reliability Engineering (ISSRE'95)*, IEEE CS Press, 58-67.

[WB 98] Waeselynck, H., Behnia, B. 1998. B Model Animation for External Verification. *Proc. 2nd Int. Conf. on Formal Engineering Methods* (ICFEM'98), 36-45.

[WB 04] Wegener, J., and Buehler, O. 2004. Evaluation of Different Fitness Functions for the Evolutionary Testing of an Autonomous Parking System. *Proc. Genetic and Evolutionary Computation Conference (GECCO-2004)*. Springer Verlag, LNCS 3103, 1400-1412.

[WE 02] Win, T., Ernst, M. 2002. Verifying distributed algorithms via dynamic analysis and theorem proving. Technical Report 841, MIT Laboratory for Computer Science.

[Wei 90] Weinberger, E. 1990. Correlated and Uncorrelated Landscapes and How to Tell the Difference. *Biological Cybernetics*, 63: 325-336.

[WER 07] Wang, Z., Elbaum, S., Rosenblum, D.S. 2007. Automated Generation of Context-Aware Tests. *Proc. 29th Int. Conf. on Software Engineering (ICSE'07)*, IEEE CS Press, 406-415.

[WG 10] Wedyan, F., Ghosh, S. 2010. A Dataflow Testing Approach for Aspect-Oriented Programs. *Proc. 12th Int. Symp. on High-Assurance Systems Engineering (HASE'2010)*, IEEE Computer Society, 64-73.

[WJ 91] Weyuker, E.J., Jeng, B. 1991. Analyzing Partition Testing Strategies. *IEEE Trans. Software Engineering*, 17(7): 703-711.

[WLB 00] Wahls, T., Leavens, G.T., Baker, A.L. 2000. Executing Formal Specifications with Concurrent Constraint Programming. *Automated Software Engineering*, 7(4): 315-343.

[WMM+ 05] Williams, N., Marre, B., Mouy, P., Roger, M. 2005. PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis. *Proc. 5th European Dependable Computing Conference (EDCC-5)*, Springer, 281–292.

[WMN+ 07] Waeselynck, H., Micskei, Z., N'Guyen, M.D., Rivière, N. 2007. Mobile Systems from a Validation Perspective: a Case study", *Proc. 6th IEEE Int. Symp. on Parallel and Distributed Computing (ISPDC'07)*, IEEE CS Press, 85-92.

[WMR+ 10] Waeselynck, H., Micskei, Z., Rivière, N., Hamvas, A., Nitu, I. 2010. TERMOS: a Formal Language for Scenarios in Mobile Computing Systems. *Proc. 7th Int. ICST Conference on Mobile and Ubiquitous Systems (MobiQuitous 2010)*. Volume LNICST à paraître, Springer.

[WSH 08] Whitley, D., Sutton, A.M., and Howe, A.E. 2008. Understanding elementary landscapes. *Proc. Genetic and Evolutionary Computation Conference (GECCO-2008)*, ACM, 585-592.

[WT 99] Waeselynck, H., Thévenod-Fosse, P. 1999. A Case Study in Statistical Testing of Reusable Concurrent Objects. *Proc. 3rd Eur. Dependable Computing Conf. (EDDC-3)*, LNCS 1667, Springer Verlag, 401-418.

[WTA 07] Waeselynck, H., Thévenod-Fosse, P., Abdellatif-Kaddour, O. 2007. Simulated Annealing Applied to Test Generation: Landscape Characterization and Stopping Criteria. *Empirical Software Engineering*, 12(1): 35-63.

[XX 06] Xu, D., Xu, W. 2006. State-Based Incremental Testing of Aspect-Oriented Programs. *Proc. 5th Int. Conf. on Aspect-Oriented Software Development (AOSD 2006)*, ACM, 180-189.

[Zha 03] Zhao, J. 2003. Data-Flow-Based Unit Testing of Aspect-Oriented Programs. *Proc. 27th Ann. Int. Conf. on Computer Software and Applications (COMPSAC '03)*, IEEE Computer Society, 188-197.

[Zin 06] Zinn, C. 2006. Supporting the Formal Verification of Mathematical Texts. *Journal of Applied Logic*, 4(4): 592-621.

[ZR 03] Zhao, J., Rinard, M. 2003. Pipa: A Behavioral Interface Specification Language for Aspect. *Proc. 6th Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2003)*, Springer Verlag, LNCS 2621,150-165.

[ZR 07] Zhang, W., Ryder, B.G. 2007. Discovering Accurate Interclass Test Dependences. *Proc. 7th ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE'07)*, 55-62.

Liste des figures

Liste des tableaux

Table des matières

Elimination des fautes : contribution au test du logiciel

Les travaux résumés dans ce mémoire ont pour cadre la sûreté de fonctionnement des systèmes informatiques. Ils portent sur l'élimination des fautes, en s'intéressant plus particulièrement au test du logiciel. Les contributions sont regroupées en quatre chapitres.
Le premier chapitre rassemble des travaux pour adapter la conception du test aux technologies de développement logicielles. Deux technologies sont considérées : la technologie orientée-objet et la méthode formelle B. Le deuxième chapitre porte sur des associations test et vérification formelle. Il s'agit selon les cas de consolider la vérification d'algorithmes partiellement prouvés, ou de faciliter l'analyse de contrexemples retournés par un model checker. Le troisième chapitre traite de la génération de test par des procédés métaheuristiques, en prenant l'exemple du recuit simulé. L'accent est mis sur l'utilisation de mesures pour guider le paramétrage de la métaheuristique. Enfin, le quatrième chapitre aborde le test de systèmes mobiles. Les traces d'exécutions sont vérifiées par rapport à un ensemble de propriétés décrites par des scénarios graphiques, en combinant des algorithmes d'appariement de graphes et de calcul d'ordres partiels d'événements.

Mots-clés : élimination des fautes, test du logiciel, vérification formelle, procédés métaheuristiques, systèmes mobiles.

Fault removal: a contribution to software testing

The research summarized in this report focuses on the dependability of computer systems. It adresses fault removal by means of software testing. The contributions are grouped into four chapters.
Chapter I presents work seeking to adapt the test design to software development technologies. It considers both object-oriented development and the formal B method. Chapter II investigates ways of coupling testing and formal verification. The aim may be to consolidate the verification of partially proved algorithms, or to aid in the analysis of counterexamples obtained from a model checker. Chapter III concerns search-based test generation, taking the example of simulated annealing search. It focuses on a measurement approach to tune the parameters of the metaheuristics. Finally, Chapter IV tackles testing of mobile computing systems. Test traces are checked with respect to properties described in graphical scenarios. The checking involves both graph matching and event order analysis.

Keywords: fault removal, software testing, formal verification, search-based testing, mobile computing systems.