Méthodes et outils pour le test logiciel

Ioannis Parissis

Université Joseph Fourier - Grenoble 1

Document de présentation de travaux

présenté par

Ioannis PARISSIS

en vue de l'obtention de

l'Habilitation à Diriger des Recherches

Discipline Informatique

Méthodes et outils pour le test logiciel

Jury :

Richard Castanet	Professeur à l'ENSEIRB - Bordeaux	**Rapporteur**
Claude Jard	Professeur à l'ENS Cachan - Rennes	**Rapporteur**
Pascale Le Gall	Professeur à l'Université d'Evry	**Rapporteur**
Antonia Bertolino	Directrice de Recherche CNR - ISTI, Pise	**Examinateur**
Nicolas Halbwachs	Directeur de Recherche CNRS - VERIMAG	**Examinateur**
Farid Ouabdesselam	Professeur Université Joseph Fourier	**Examinateur**

Avant propos

Ce document retrace de manière synthétique mes travaux de recherche depuis septembre 1999, date depuis laquelle j'occupe un poste de Maître de Conférences à l'université Joseph Fourier (Grenoble 1) et mène des recherches au sein de l'équipe VASCO du laboratoire LIG. Ces travaux portent sur le test des logiciels et sont étroitement liés aux recherches effectuées de 1993 à 1997, période pendant laquelle j'ai été thésard, puis chercheur contractuel dans cette même équipe.

Le test logiciel est une discipline qui embrasse l'ensemble du cycle de développement. En effet, dès l'analyse des besoins d'une application, apparaissent des exigences et des propriétés à tester dont la caractérisation et l'identification font partie des compétences des testeurs. Il en est de même pour la spécification du produit, la conception de son architecture technique ou la programmation. Si cette caractéristique rend le domaine du test extrêmement intéressant et riche, elle rend également nécessaire une connaissance raisonnable de l'ensemble des domaines du génie logiciel. Mes activités d'enseignant, et plus particulièrement ma forte implication aux enseignements de génie logiciel du DESS Génie Informatique, ont considérablement contribué à l'acquisition de cette connaissance.

Le test logiciel est également une discipline qui se veut étroitement liée et utile à la pratique professionnelle. La prise en compte des problématiques issues de cette dernière s'est faite, entre autres, au moyen de nombreuses collaborations industrielles qui ont ponctué mes travaux, en particulier dans le cadre de projets nationaux (RNRT, RNTL).

Guide de lecture

Le document est structuré en trois chapitres. Le premier chapitre présente de manière synthétique l'ensemble de mes activités de recherche en les situant dans leur contexte scientifique et humain et en traçant leurs principales perspectives.

Les deux chapitres suivants couvrent avec plus de détails les deux thématiques principales de mes recherches, le test de programmes synchrones et la validation d'applications interactives.

Table des matières

1 Résumé synthétique des travaux effectués **1**

1.1 Historique et contexte scientifique 1

1.2 Recherches effectuées 3

1.2.1 Motivations et orientations 3

1.2.2 Test de programmes synchrones 4

1.2.2.1 Contexte & problématique 4

1.2.2.2 Travaux réalisés et contributions 5

1.2.2.3 Sélection de références 6

1.2.2.4 Perspectives 6

1.2.3 Test d'applications interactives et multimodales 7

1.2.3.1 Contexte & problématique 7

1.2.3.2 Travaux réalisés et contributions 8

1.2.3.3 Sélection de références 10

1.2.3.4 Perspectives 10

1.2.4 Test structurel de programmes impératifs 10

1.2.4.1 Contexte & problématique 10

1.2.4.2 Travaux réalisés et contributions 11

1.2.4.3 Sélection de références 12

1.2.4.4 Perspectives 12

1.2.5 Autres recherches 13

2 Test de programmes synchrones **15**

2.1 L'approche synchrone et le langage Lustre 15

2.1.1 Programmation synchrone 15

2.1.2 Le langage Lustre 16

2.1.3 Exemple 17

2.1.4 Vérification de programmes Lustre 18

2.1.4.1 Opérateurs temporels Lustre 18

2.1.4.2 Propriétés de sûreté 19

2.1.4.3 Spécification de l'environnement 19

2.1.4.4 Preuve formelle 20
2.2 Test de programmes spécifiés en Lustre 21
2.3 Automatisation des tests fonctionnels : Lutess 22
2.3.1 Analyse du problème 22
2.3.2 Principes de Lutess 22
2.3.3 Génération de test avec Lutess 23
2.3.3.1 Cas des spécifications booléennes 24
2.3.3.2 Cas des spécifications numériques - utilisation de la programmation par contraintes 29
2.4 Evaluation de la qualité des tests 32
2.4.1 Analyse du problème 32
2.4.2 Critères structurels pour des programmes Lustre 33
2.4.2.1 Définition de critères de couverture pour Lustre 35
2.4.3 Critères pour le test boîte noire 37
2.5 Perspectives 39

3 Test d'applications interactives et multimodales 41
3.1 Validation d'applications interactives 41
3.2 Utilisation de l'approche synchrone pour le test d'applications interactives 44
3.2.1 Analyse du problème 44
3.2.2 Modélisation synchrone de l'interaction 45
3.2.3 Type de propriétés validées 46
3.2.4 Adéquation des techniques de test de Lutess 47
3.2.5 Utilisation d'arbres de tâches 48
3.2.6 Vers une prise en compte de la multimodalité 52
3.2.6.1 Modalités et multimodalité 52
3.2.6.2 Validation d'applications multimodales 53
3.3 Perspectives 54

Bibliographie 57

Chapitre 1

Résumé synthétique des travaux effectués

1.1 Historique et contexte scientifique

Mes activités de recherche s'inscrivent en grande partie dans le domaine des applications réactives synchrones. Au milieu des années 80, la programmation synchrone [11] apparaît comme une proposition crédible dans le domaine des applications temps-réel critiques. Des langages de programmation spécifiques sont proposés (Esterel [20], Signal [54], Lustre [38]), permettant de modéliser et de programmer ces systèmes de manière élégante et sûre. La communauté de recherche Grenobloise a fortement contribué au développement de cette thématique, notamment en proposant le langage Lustre [21]. Conçu au Laboratoire de Génie Informatique (LGI) de l'IMAG en 1986 par P. Caspi et N. Halbwachs, Lustre est un langage déclaratif flot de données, inspiré de Lucid [7].

Les premiers travaux de recherche sur Lustre portent sur la définition formelle de sa sémantique et sur sa compilation [13] [83]. Plusieurs travaux ont également eu lieu sur la vérification formelle de Lustre [32, 82], qui, tirant profit de l'expressivité du langage [79] et des techniques de compilation ou de réduction d'automates finis [17], proposent un principe de preuve simple et efficace par *model-checking*. En effet, il suffit d'intégrer dans le programme Lustre à valider ses propriétés présumées et de s'assurer qu'elles sont vraies dans tout état accessible de ce nouveau programme [40].

En 1993, année de mes débuts dans la recherche, il n'y a pas de travaux significatifs sur le test de programmes Lustre. C'est sous la direction de Farid Ouabdesselam que je commence à étudier les possibilités de tester des programmes

synchrones. Cette même année, le laboratoire Verimag[1] est créé et presque tous les chercheurs menant des activités sur l'approche synchrone y sont intégrés. Mais en ce qui me concerne, je poursuis avec F. Ouabdesselam mes travaux au sein du LGI (et plus tard du laboratoire LSR) et en 1996 je soutiens ma thèse intitulée "Test de logiciels synchrones spécifiés en Lustre" [75], dont une des contributions principales est l'outil de test fonctionnel Lutess [27].

A la même époque, le test de programmes Lustre commence à susciter un certain intérêt dans la communauté scientifique française. En 1994 est soutenue au LAAS la thèse de Christine Mazuet [62], suggérant d'utiliser l'automate engendré par le compilateur Lustre comme modèle du programme et d'avoir recours au test statistique pour assurer la couverture de l'automate. En 1998 voit le jour l'outil Lurette [84], développé à Verimag. Lurette reprend l'essentiel des principes de génération de tests de Lutess en y ajoutant le moyen de résoudre des systèmes d'équations numériques. L'outil Gatel [59, 60] est proposé environ à la même époque. Il repose sur une interprétation des constructions du langage Lustre sous la forme de contraintes sur des variables booléennes ou des variables à valeurs dans des intervalles d'entiers. La génération des données de test consiste à résoudre ces contraintes à l'aide de la programmation logique par contraintes.

Les activités de recherche sur le test synchrone se sont poursuivis au laboratoire LSR après ma thèse, en particulier dans le cadre d'une forte collaboration avec France Telecom (Consultation Thématique Informelle CNET, 1996-1999) portant sur l'application de l'approche à la détection d'interactions dans les services téléphoniques, ainsi qu'au sein du projet RNRT VALISERV (2000-2003). Dans le cadre de ces travaux, auxquels j'ai partiellement participé, se sont déroulés les thèses de Lydie du Bousquet [29] et Nicolas Zuanon [105] qui ont à la fois exploré les moyens de valider les services téléphoniques et proposé de nouvelles techniques de génération de tests pour Lutess.

En octobre 1997 je quitte "définitivement" la recherche. C'est en tant qu'ingénieur à France Telecom que j'apprends, en été 1998, la désignation de Lutess comme meilleur outil pour la détection d'interactions de services au concours de la conférence FIW'98 [36] [28] et suis avec une certaine nostalgie les travaux de mon ancienne équipe.

Un concours de circonstances (certainement imprévu et très probablement heureux) fait que je suis recruté en 1999 à un poste d'enseignant-chercheur à l'université Joseph Fourier et rejoins de nouveau le laboratoire LSR.

[1] Plus précisément, Verimag est, à l'origine, une "unité mixte", terme qui à l'époque dénote une structure constituée de chercheurs d'une institution publique et des ingénieurs d'une entreprise, en l'occurrence Verilog. Quelques années plus tard, ce même sigle désignera le laboratoire actuel.

1.2 Recherches effectuées

1.2.1 Motivations et orientations

Les travaux de recherche que j'ai menés se structurent selon deux grands axes. Le premier axe constitue le prolongement et l'approfondissement des travaux sur le test de logiciels synchrones, avec des objectifs d'amélioration technique et méthodologique de l'approche Lutess et de valorisation des résultats auprès d'utilisateurs du domaine des applications critiques ainsi qu'une volonté de confronter l'approche aux réalités industrielles.

Le deuxième axe reflète le souhait de construire une compétence forte sur le test logiciel en élargissant le spectre des recherches, soit en abordant d'autres domaines d'application soit en étudiant d'autres approches. Des recherches sur le test des applications interactives s'inscrivent dans cette démarche. Elles sont à l'origine d'une longue collaboration, toujours active, avec l'équipe IIHM (Ingénierie de l'Interaction Homme-Machine) du laboratoire LIG et nous ont permis d'explorer un nouveau domaine d'application avec des besoins considérables en validation ainsi que d'ouvrir un champ d'investigation aux perspectives nombreuses et intéressantes.

Par ailleurs, nos avons mené des travaux sur le test structurel de programmes impératifs qui nous ont permis de nous intéresser à la génération automatique de données de test à partir de l'analyse statique du code ainsi que sur l'utilisation de la programmation par contraintes (PLC) pour le test. Une conséquence indirecte de ces travaux est l'extension de l'outil Lutess au moyen de la PLC.

Dans cette même volonté de renforcement du thème du test logiciel, des problématiques nouvelles font partie de mes intérêts de recherche depuis peu, et sont présentées ici en tant que perspectives :

- Le test de services Web est un thème développé récemment dans le cadre de ma collaboration, toujours active, avec l'équipe d'Antonia Bertolino (ISTI-CNR, Pise).
- Une collaboration naissante porte sur la traçabilité et le test dans la conception des systèmes d'information et implique les équipes Adele (J-M. Favre) et SIGMA (D. Rieu) du LIG ainsi que la société luxembourgeoise OneTree Technologies (D. Avrilionis).

Dans la suite du paragraphe, je présente brièvement les thèmes explorés en prenant soin de préciser le contexte dans lequel ils se sont déroulés (projets, thèses...), les contributions apportées et les perspectives ouvertes. Deux de ces thèmes, le test des programmes synchrones et la validation d'applications interactives, correspondent aux travaux les plus conséquents en termes de temps et d'effort consacrés et sont plus longuement développés dans les chapitres suivants.

1.2.2 Test de programmes synchrones

1.2.2.1 Contexte & problématique

Aux premiers travaux de l'équipe sur le test des logiciels synchrones [75] ont succédé des recherches sur l'application de l'approche à la détection d'interactions entre services téléphoniques (collaboration France Telecom). Ces recherches ont montré, d'une part, que l'utilisation de Lutess peut être bénéfique dans un cadre méthodologique qui est celui de la validation de spécifications de haut niveau. Ce premier résultat a, entre autres, permis à l'équipe de construire une compétence dans le domaine des services téléphoniques et de disposer d'une base conséquente de spécifications ainsi que d'un savoir-faire dans la définition des propriétés à valider.

D'autre part, ces recherches ont contribué à faire évoluer les techniques proposées dans Lutess quelques années auparavant. En effet, Lutess adopte une démarche de test en "boîte noire" et propose la génération aléatoire de données de test, contrainte par les propriétés invariantes temporelles de l'environnement du programme. Ce type de génération automatique permet d'éviter la production de séquences d'entrées invalides et sans signification pour le test. Mais le simple test aléatoire produit des séquences non forcément réalistes et, en tous cas, non représentatives des comportements habituels. Ceci est un des défauts bien connus du test aléatoire, accentué ici par la nécessité de générer des séquences, souvent longues, d'entrées.

A partir de ce constat, plusieurs propositions d'amélioration ont été formulées. Certaines suggèrent de donner la possibilité aux ingénieurs d'influer sur les données générées de manière simple et efficace, par le biais de l'introduction de directives de génération. La première d'entre elles consistait en la proposition de spécification de profils opérationnels [71], idée approfondie et évaluée dans la thèse de Lydie du Bousquet [29]. La seconde, formulée par Nicolas Zuanon [105], utilise la spécification de scénarios abstraits, écrits par l'utilisateur, et correspondant à des classes de situations dans lesquelles il souhaite tester le programme.

Une proposition formulée lors des premiers travaux sur Lutess [72] (inspirée de travaux sur le test de spécifications algébriques [14]) restait en grande partie inexploitée. Elle consistait en un guidage automatique de la génération des tests à partir des propriétés que le programme doit vérifier. D'après cette technique, le générateur de test pourrait privilégier la production de données plus aptes à amener le programme à violer ces propriétés.

Outre les aspects de génération de tests, l'évaluation de la qualité des tests produits et l'arrêt du test, question fondamentale dans une démarche de validation, n'était pas suffisamment étudiée.

Enfin, l'impossibilité de traiter des spécifications à variables numériques limitait le spectre d'application de Lutess et les perspectives d'évaluation de l'approche sur des études de cas réelles.

1.2.2.2 Travaux réalisés et contributions

Le premier travail que nous avons entrepris dans cet axe a été l'exploration du guidage de la génération par les propriétés de sûreté, dans le cadre de la thèse de Jérôme Vassy[2] [93]. Plusieurs stratégies heuristiques de génération de tests ont été proposées, correspondant à diverses possibilités d'exploration du modèle formel obtenu par composition d'un modèle de l'environnement du programme et des propriétés. La récente expérience de l'équipe dans le domaine des services téléphoniques nous a fourni plusieurs exemples de spécifications qui ont servi à l'évaluation des techniques de génération proposées. Dans le cadre de la thèse de Besnik Seljimi[3] et du projet RNTL DANOCOPS, nous avons amélioré la technique de génération de tests guidée par les propriétés de sûreté, en introduisant des *hypothèses* sur le programme. Ces hypothèses peuvent correspondre soit à des propriétés considérées comme prouvées, soit à des connaissances que l'ingénieur a sur le programme et qu'il souhaite prendre en compte pendant le test. Dans ce même cadre, nous avons étendu Lutess de sorte à rendre possible la prise en compte de spécifications incluant des variables numériques. Nous avons utilisé pour cela la programmation par contraintes pour exprimer l'ensemble des modèles et algorithmes de génération de Lutess [90]. Grâce à la souplesse de la programmation par contraintes, les profils opérationnels peuvent être spécifiés de manière beaucoup plus puissante, des probabilités d'occurrence pouvant être affectées à des expressions booléennes et non seulement aux simples variables d'entrée [88]. De plus, pour la première fois, toutes les directives de guidage disponibles dans Lutess peuvent être utilisées simultanément [89].

On peut noter des similitudes avec d'autres approches de test de systèmes de transitions, comme celle proposée dans [44]. En présence d'une implémentation I, d'une spécification S et d'un *objectif de test* O, cette approche s'efforce à générer des tests satisfaisant O et étant conformes à S. Ces traces sont ensuite comparées à celle produites par l'implémentation I afin de vérifier leur conformité. La difficulté réside en l'écriture des objectifs de test. Cette difficulté est à comparer à celle, dans Lutess, de produire le bon profil opérationnel ou bien de paramétrer le guidage par les propriétés de sûreté et les hypothèses.

En parallèle avec ces investigations sur la génération des tests, nous nous sommes intéressés au problème de l'évaluation de la qualité des tests produits. Sur le modèle utilisé pour la génération de tests, nous avons défini des mesures

[2] Thèse co-encadrée avec Farid Ouabdesselam, soutenue en octobre 2005.
[3] Thèse co-encadrée avec Laurent Trilling, en cours

de couverture à partir de la notion d'*état suspect* [74], mesures qui présentent l'avantage de pouvoir être utilisées dans le cadre d'un test en boîte noire. Mais leur utilité est limitée étant donné qu'elles ne tiennent pas compte de la manière dont l'application sous test est réalisée. On s'est ensuite placé dans un cadre méthodologique précis, en considérant que les programmes à tester étaient écrits en Lustre/SCADE (contrairement à Lutess qui procède à un test en boîte noire, sans hypothèse sur le code du programme). En repartant d'idées exprimées dans [75], nous avons proposé une approche d'évaluation fondée sur la mesure de la couverture du programme Lustre au moyen de plusieurs critères. Ces derniers constituent une hiérarchie permettant d'adapter la mesure de couverture au niveau de qualité exigée ou aux ressources disponibles. Ce travail, mené dans le cadre de la thèse d'Abdesselam Lakehal[4] [52] s'est en partie appuyé sur une collaboration avec Airbus, l'ONERA et le CEA [16] et a donné lieu au développement d'un outil de mesure de couverture, Lustructu [49], évalué sur une étude de cas réelle [51].

1.2.2.3 Sélection de références

- Articles sur la génération de tests et Lutess : [72, 27, 73, 88, 90].
- Articles sur l'évaluation de la qualité et la couverture : [74, 50, 49].
- Thèses co-encadrées : [93, 52].

1.2.2.4 Perspectives

Aujourd'hui, à l'issue de ces travaux, nous disposons d'un ensemble d'outils qui peuvent contribuer à l'automatisation du test de programmes synchrones de manière significative. Mais leur utilisation effective exige leur intégration dans une méthodologie de test précise dont l'élaboration ne peut s'envisager qu'en collaboration avec des utilisateurs réels. Les collaborations que nous avons eues avec l'industrie ont permis de faire la promotion de l'approche de manière prospective dans des secteurs où la programmation synchrone n'était pas utilisée. Le projet ANR Technologies Logicielles SIESTA (démarrage début 2008), dont le LIG assure la coordination, nous donnera l'occasion de confronter nos travaux aux besoins réels d'un domaine où l'approche synchrone et largement utilisée. Dans ce projet, nous prévoyons l'évaluation de Lutess sur des études de cas fournies par nos partenaires industriels (Airbus, Astrium ST, Turbomeca, Hispano-Suiza) et une poursuite des recherches sur les critères d'évaluation de la qualité.

[4] Thèse co-encadrée avec Farid Ouabdesselam, soutenue en septembre 2006.

1.2.3 Test d'applications interactives et multimodales

1.2.3.1 Contexte & problématique

La validation et la vérification des systèmes interactifs est un thème de recherche qui connaît une évolution considérable ainsi qu'une prise d'importance croissante ces dernières années. En effet, ces applications deviennent les compagnons quotidiens d'un nombre grandissant d'utilisateurs de systèmes commerciaux et de leur bon fonctionnement dépendent des intérêts économiques considérables. De plus, il est fréquent que des processus potentiellement dangereux soient contrôlés par des systèmes interactifs ce qui leur confère un aspect critique.

Un des attributs de qualité essentiels des systèmes interactifs est l'*utilisabilité*, terme désignant un ensemble de critères qu'une application doit satisfaire pour être facilement adoptée et correctement utilisée par un être humain. Ces critères ne sont pas facilement formalisables et, dans tous les cas, pas formalisés d'une manière qui fait consensus dans la communauté de recherche du domaine. On peut citer, par exemple, l'*honnêteté,* définie comme la capacité du système à rendre observable son état et à engendrer une interprétation correcte de l'utilisateur : la mesure de la satisfaction de ce critère nécessite la prise en compte du point de vue de l'utilisateur humain, qui peut varier d'un individu à un autre.

On constate ainsi que les systèmes interactifs doivent être évalués et validés de manière à tenir compte de l'appréciation, très peu contrainte, de l'utilisateur humain. Toute une panoplie d'approches d'évaluation est donc basée sur l'analyse de l'application par des experts, par exemple suite à des expérimentations donnant lieu à une récolte d'éléments sur le terrain. Cette particularité est à l'origine de l'hétérogénéité des cultures et des notations entre l'usager ou ses représentants (centrés sur l'interaction) et les réalisateurs des systèmes rendant difficile de vérifier que les besoins exprimés par les premiers ont bien été pris en compte par les seconds.

Plusieurs tentatives d'introduire des notations, modèles ou méthodes formelles ont eu lieu ces dernières années. Elles consistent, en général, à introduire dans le processus de conception des systèmes interactifs des approches ayant fait leurs preuves dans d'autres domaines. L'utilisation de LOTOS [76], de réseaux de Petri [9] ou de la méthode B [102, 104] en sont des exemples représentatifs. En imposant la construction d'un modèle formel de l'application, elles sont en mesure de vérifier certaines propriétés fonctionnelles, voire d'utilisabilité. L'utilisation de l'approche synchrone et du langage Lustre pour la vérification d'applications interactives a également été étudiée dans cette optique : à partir de la modélisation du système interactif en Lustre, des preuves de validité de certaines propriétés peuvent être opérées par model-checking [85, 25, 91]. Mais

l'adoption de ces approches par la communauté semble difficile à envisager dans un avenir proche, du fait de la nécessité de spécifier la totalité de l'application - opération s'avérant très souvent hors d'atteinte pour les ingénieurs - ainsi qu'à cause de la complexité des preuves mises en oeuvre.

Les limites de complexité que ces études ont fait apparaître sont similaires à ceux qui étaient à l'origine de nos travaux sur le test synchrone. Par ailleurs, l'utilisation de Lustre à des fins de modélisation d'interfaces ayant été menée avec un certain succès, l'étude d'une approche de test s'inspirant de nos travaux sur Lutess semblait une opportunité intéressante. Notre proximité avec l'équipe IIHM et notre volonté réciproque d'explorer le sujet a été le déclencheur final du début de ces recherches, toujours actives après 5 ans de collaboration.

Un aspect important de ces travaux réside dans notre volonté de prendre en compte la multimodalité, c'est à dire l'utilisation de plusieurs modalités (clavier, souris, parole, geste) pour l'interaction. Cette caractéristique, déjà présente dans des applications de réalité virtuelle et pressentie comme une voie majeure dans l'évolution des applications interactives, augmente la complexité des systèmes du fait de la nécessité d'être en mesure d'interpréter correctement plusieurs événements issus de modalités différentes, parfois redondants, liés temporellement et oeuvrant à l'accomplissement d'une même tâche.

La problématique de nos recherches peut se résumer en un ensemble de questions, dont les plus importantes sont :

- Peut-on modéliser l'interaction de manière synchrone ?
- Quel type de propriétés peut-on espérer tester ?
- Quelles techniques de génération de tests sont les plus appropriées ?
- Quel effort cela demande aux testeurs ? Comment le réduire, le cas échéant ?
- Comment prendre en compte la multimodalité ?

1.2.3.2 Travaux réalisés et contributions

Les questions posées ci-dessus ont fait l'objet, majoritairement, de notre participation au projet RNRT VERBATIM (2004-2006), de la thèse de Laya Madani[5] [58] ainsi que des projets de Master Recherche de Sylvain Lemoine et Frédéric Jourde [6] et ont donné lieu à une collaboration entre plusieurs membres des équipes IIHM et VASCO [7].

- Peut-on modéliser l'interaction de manière synchrone ?
  En théorie, il est possible de modéliser un système asynchrone de manière synchrone et inversement. En pratique cela demande d'interposer entre

[5] Thèse co-encadrée avec Farid Ouabdesselam, soutenue en octobre 2007
[6] Projets co-encadrés avec Laurence Nigay en 2004 et 2006
[7] Laurence Nigay, Sophie Dupuy-Chessa, Jullien Bouchet, Catherine Oriat, Lydie du Bousquet

l'application interactive et le générateur de tests synchrone un traducteur, chargé de la synchronisation. Les principes de la construction de ce traducteur ont été présentés dans [58] et évalués sur des études de cas.

- Quel type de propriétés peut-on espérer tester ?
  Cette question se divise en deux parties. La première concerne les propriétés dont nous pouvons évaluer automatiquement la satisfaction en les incluant dans un oracle de test automatique. En partant de l'hypothèse que Lustre est utilisé, on peut exprimer des propriétés invariantes de type sûreté, qui s'expriment dans une logique temporelle du passé. Des propriétés fonctionnelles visant à vérifier qu'une séquence d'entrées données provoque une réaction correcte du système peuvent être ainsi exprimées. Ceci est beaucoup plus aléatoire pour les propriétés d'utilisabilité, certaines étant des propriétés de vivacité, d'autres n'étant pas facilement formalisables.
  La deuxième partie est de nature méthodologique et porte sur le type de propriétés qui peuvent guider la génération de tests, ce qui est traité par la question suivante.
- Quelles techniques de génération de tests sont les plus appropriées ?
  L'utilisation de profils opérationnels, technique présente dans Lutess, a été expérimentée avec succès sur plusieurs études de cas [18, 56]. D'autres techniques de génération pourraient être également utilisées (notamment le guidage par les propriétés).
- Quel effort cela demande aux testeurs ? Comment le réduire, le cas échéant ?
  Il semble clair qu'un langage comme Lustre (étendu pour Lutess), ne correspond pas aux modèles ou notations habituellement utilisés par la communauté IHM. De ce fait, l'effort demandé aux testeurs peut être un obstacle rendant l'utilisation de l'approche prohibitive. Nous avons ainsi porté notre intérêt sur des notations plus habituelles du domaine et avons plus particulièrement étudié l'utilisation d'*arbres de tâches*. Nous avons montré que ces derniers peuvent être traduits en des modèles comparables à ceux manipulés par Lutess [57], en particulier en les enrichissant de spécifications de profils opérationnels.
- Comment prendre en compte la multimodalité ?
  Dans le cas des systèmes multimodaux, les événements échangés sont issus de plusieurs modalités qui peuvent être combinées. L'objectif de la validation dans ce cadre, en plus des points mentionnés ci-dessus pour le cas général des systèmes interactifs, est de s'assurer que la manière dont le système interprète ces événements est correcte. Ceci a plusieurs impacts sur la validation. D'une part, le niveau d'abstraction auquel sont considérés les événements d'entrée est différent et prend en compte les modalités.

D'autre part, des propriétés spécifiques doivent être vérifiées, comme, par exemple, les propriétés de type CARE (Complémentarité, Assignation, Redondance, Equivalence) et le mécanisme de *fusion* mis en oeuvre.
Notre travail a consisté à étudier comment cette prise en compte peut être effectuée dans le cadre de l'utilisation de Lutess, notamment pour la génération de tests.

1.2.3.3 Sélection de références

- Articles : [56, 18, 57].
- Thèse co-encadrée : [58].

1.2.3.4 Perspectives

L'utilisation des arbres de tâches, en particulier, comme des modèles de base pour le test semble être une idée intéressante dont le développement ouvre plusieurs perspectives de travail. En effet, les arbres de tâches doivent être enrichis de sorte à pouvoir spécifier plusieurs types de guidage, tout en restant des modèles intuitifs. Cela concerne, en particulier, la prise en compte de la multimodalité. Cette perspective présente également l'intérêt, mais aussi la difficulté, de s'appuyer sur un modèle défini très tôt dans le cycle de développement (analyse de besoins), ce qui nécessite d'étudier la manière de l'affiner afin qu'il soit exploitable pour le test de l'implémentation concrète.

1.2.4 Test structurel de programmes impératifs

1.2.4.1 Contexte & problématique

La mesure de la couverture structurelle reste aujourd'hui le moyen d'évaluation de la qualité des tests le plus utilisé dans l'industrie. Les critères les plus fréquemment retenus sont la couverture de toutes les instructions ou bien celle de toutes les branches du programme [10].

Du point de vue méthodologique, nous pouvons distinguer deux manières d'envisager un processus de test incluant une évaluation de la couverture structurelle :

- Les tests sont écrits indépendamment du code, à partir d'objectifs fonctionnels, et la couverture du code est calculée après leur exécution. Si des éléments du code (e.g. instructions ou branches) restent à couvrir, d'autres tests doivent être exécutés ; autrement, on peut arrêter le test.
- Les objectifs de couverture sont pris en compte lors de la génération de tests, de sorte que la couverture atteinte soit connue avant l'exécution.

Bien entendu, les deux approches peuvent être combinées : rien n'empêche un testeur de satisfaire un objectif fonctionnel tout en veillant à couvrir une partie du code donnée.

Quelle que soit la démarche adoptée, la génération de tests visant la couverture structurelle est une tâche que tôt ou tard sera effectuée, le plus souvent de manière manuelle. Automatiser partiellement cette génération est une préoccupation des chercheurs depuis des longues années [47, 45] (l'automatisation totale se heurtant à des limites théoriques d'indécidabilité). En particulier, une approche basée sur l'utilisation de la programmation par contraintes a été proposée il y a quelques années au sein de l'outil INKA [34]. Selon cette approche, le programme est traduit en un modèle constitué d'un ensemble de contraintes, dont la résolution permet le calcul de données de tests permettant d'activer un élément particulier du code.

Les limites de complexité de cette approche sont cependant nombreuses, dont, par exemple, la prise en compte des tableaux ou bien le traitement de programmes comportant des pointeurs. Un autre problème est celui de la complexité des calculs induite par la présence d'appels de fonctions ou procédures dans le code, dont l'analyse détaillée est impossible en pratique. Habituellement, ces appels sont manuellement remplacés par des "bouchons", programmes élémentaires retournant le plus souvent de valeurs constantes arbitraires. Or, on note que les bouchons utilisés dans le test unitaire sont par nature imprécis. Ils sont construits par le testeur à partir d'une spécification, souvent incomplète, de l'entité bouchonnée. L'impact principal de cette imprécision se manifeste par la sélection de données de test qui peuvent être erronées. Le comportement de l'entité sous test obtenu par exécution de ces données n'est pas celui qui aurait lieu en présence de la fonction appelée et non de son bouchon. De ce fait, certains composants de l'entité sous test (appelante) sont atteints alors qu'ils ne le seraient pas sans l'utilisation du bouchon. Inversement, il peut arriver que l'utilisation d'un bouchon en dehors des comportements initialement prévus rende artificiellement des composants de l'entité appelante impossibles à atteindre.

1.2.4.2 Travaux réalisés et contributions

Ce dernier problème a été abordé dans le cadre de notre participation au projet RNTL INKA et, plus particulièrement, dans le cadre de la thèse de Karim-Cyril Griche[8] [46].

Dans ce travail nous avons étudié des techniques permettant la génération automatique de données de test structurel unitaire réalistes en présence d'appels d'autres entités du logiciel. Une prise en considération de ces appels au niveau

[8] Thèse co-encadrée avec Farid Ouabdesselam, soutenue en juillet 2005.

du test unitaire doit être réaliste de manière à obtenir un ensemble de données de test qui représente au mieux la couverture de l'entité dans le logiciel mais rester suffisamment simple pour permettre l'utilisation des critères de couverture structurels. Nous avons ainsi proposé de modéliser les fonctions appelées par un ensemble d'approximations, puis de construire les bouchons à partir de ce modèle. Le bouchon est ainsi constitué d'une hiérarchie d'approximations de plus en plus complexes dont chacune représente une classe de comportements de l'entité originale. Ce "bouchonnage en couches", inspiré des travaux sur le "slicing" de programmes [96] a le double avantage de pouvoir simuler l'ensemble des comportements possibles de la fonction originale et d'éviter d'avoir à considérer tous ces comportements en même temps. En effet, les comportements les plus simples à analyser sont placés en tête du bouchon et sont traités en premier quand un appel vers la fonction bouchonnée sera rencontré. Lorsque ces comportements simples ne sont pas suffisants pour générer une donnée nécessaire à atteindre un composant de la fonction appelante, on augmente le réalisme de l'approximation en avançant dans la hiérarchie. Une simplification des bouchons ainsi construits peut être effectuée en tenant compte de l'environnement d'appel (contexte dans lequel est utilisé le bouchon) ainsi que des objectifs de la génération (contrainte sur le résultat attendu du bouchon).

1.2.4.3 Sélection de références

- Articles : [35].
- Thèse co-encadrée : [46].

1.2.4.4 Perspectives

Ce premier travail dans le domaine du test structurel de programmes impératifs nous a permis d'effectuer certaines propositions prometteuses et de définir des orientations pour leur amélioration. Une telle amélioration concerne la solution retenue pour la construction des bouchons qui repose sur des approximations obtenues en s'inspirant de la technique du "slicing". Ces dernières, malgré les optimisations effectuées par la prise en compte de l'environnement d'appel ou des objectifs de génération, restent parfois complexes à utiliser dans un processus de génération automatique de données de test. Une des raisons de cette complexité est l'existence d'appels en cascade de plusieurs fonctions, limitée de manière arbitraire dans notre approche. La prise en compte de cette complexité en utilisant, en particulier, des techniques de slicing interprocédural [41] est une des solutions que nous envisageons.

1.2.5 Autres recherches

Le test de Web Services est un domaine auquel je me suis intéressé pendant mon séjour au laboratoire ISTI-CNR (Pise, équipe d'Antonia Bertolino) et qui fait actuellement l'objet d'un travail commun. Les Web Services sont une application très connue de "Service Oriented Architecture" (SOA), une approche émergente de conception d'applications réparties. Cette approche favorise l'utilisation (et la réutilisation) de composants développés indépendamment ainsi que l'intégration de technologies hétérogènes et constitue, de ce fait, un paradigme de conception peu coûteux et très évolutif. Assurer la correction d'une application SOA construite à partir de la composition d'autres services, dont certains sont découverts dynamiquement, est un défi pour la communauté des chercheurs.

Les Web Services sont des applications développées utilisant des standards adoptés pour le Web, tels que WSDL, UDDI, HTTP, SOAP etc. Si on se réfère à un processus de développement traditionnel, la vérification et la validation de Web Services composites correspond à l'étape du test d'intégration. Mais l'implémentation des Web services composants n'est en général pas connue au moment de la conception des services composites. Seules leurs interfaces doivent être spécifiées (au moyen du langage standard WSDL). Ainsi, le test ne peut s'effectuer que sous des hypothèses sur le comportement de ces composants.

Ces dernières années, plusieurs travaux de recherche ont porté sur la validation de Web services. En règle générale, ils s'appuient sur des spécifications comportementales complémentaires des composants et de la composition (souvent des machines d'états finis). Mais la seule validation du comportement ne peut pas garantir la validité de propriétés exprimées sur les données. La modélisation des exigences sur le flux des données (entrées, sorties) peut être le point de départ d'une approche de validation palliant ce manque que nous explorons actuellement [15].

Enfin, la traçabilité des besoins dans une approche de développement de type MDE (Model Driven Engineering) de Systèmes d'Information et ses implications sur les modèles utilisés ou construits pour le test est un nouveau sujet de recherche que nous abordons dans le cadre de la thèse de Thi-Thu-Minh Nguyen en collaboration avec les équipes ADELE (Jean-Marie Favre) et SIGMA (Dominique Rieu). Ce travail s'inscrit dans une coopération avec la société Luxembourgeoise OneTree Technologies (créée par Denis Avrilionis, ancien chercheur du laboratoire LSR) et bénéficie du soutien du ministère de recherche du Luxembourg.

Chapitre 2

Test de programmes synchrones

2.1 L'approche synchrone et le langage Lustre

2.1.1 Programmation synchrone

L'approche synchrone [11] connaît un essor remarquable depuis environ vingt ans tant sur le plan de la recherche, que sur le plan des applications industrielles qui sont aujourd'hui nombreuses dans les domaines dits «critiques», tels que les transports (ferroviaires ou aériens) ou l'énergie. Ces domaines ont des besoins importants en méthodes, outils et techniques de développement de logiciels fiables et constituent naturellement un champ d'application privilégié des méthodes formelles.

Un programme est dit *synchrone*, s'il vérifie l'hypothèse de synchronisme qui stipule que le calcul des sorties du programme à partir de ses entrées est instantané. En supposant que le temps est divisé en des instants discrets définis par une horloge globale, un programme synchrone, à un instant t, lit ses entrées i_t et calcule ses sorties o_t. L'hypothèse synchrone assure que le calcul et l'émission de o_t est fait instantanément, au même instant t (figure 2.1).

En pratique, on considère qu'un logiciel a un comportement synchrone s'il réagit à son environnement avant toute évolution de ce dernier. Ainsi, si à l'instant t le logiciel reçoit les entrées i_t depuis son environnement externe, il émet les sorties o_t avant qu'une nouvelle entrée i_{t+1} soit disponible. Cette propriété permet de s'abstraire des problèmes de temporalité, très importants quand il s'agit de réaliser des *systèmes réactifs*.

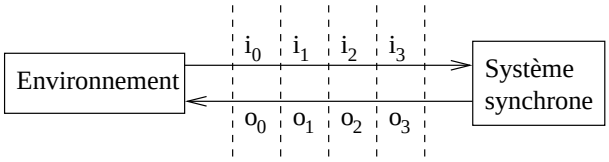

FIG. 2.1 – Un programme synchrone

2.1.2 Le langage Lustre

Lustre [37, 21] est un langage destiné à la spécification et la programmation de systèmes synchrones. Il peut être considéré à la fois comme une logique temporelle du passé [80] et comme un langage de programmation. C'est un langage flot de données : un flot est une séquence de valeurs couplée à une horloge qui indique à quel moment certaines valeurs apparaissent.

Un programme Lustre est structuré en noeuds. Un noeud Lustre est constitué d'un ensemble d'équations qui définissent ses variables de sortie comme des fonctions des variables d'entrée et des variables locales (figure 2.2). Il n'y a pas d'ordre entre les équations et chaque variable définit un flot.

```
node Programme(<entrées>) returns (<sorties>)
var
    <variables locales>
let
    <sorties>=f(<entrées>,<variables locales>) ;
tel
```

FIG. 2.2 – Structure syntaxique d'un programme Lustre

Une expression Lustre contient des constantes, des variables, des opérateurs logiques, arithmétiques ainsi que deux opérateurs spécifiques : "pre" et "->". L'opérateur *précédent* (noté *pre*) donne accès à la dernière valeur qu'une expression vient de prendre (au top d'horloge précédent). L'opérateur *suivi de* (noté ->), est utilisé pour désigner la valeur initiale (à $t = 0$) des expressions :

- Si E est une expression définissant le flot $(e_0, e_1, ..., e_n, ...)$, $preE$ va dénoter le flot $(nil, e_0, e_1, ..., e_{n-1}, ...)$ où nil est une valeur indéfinie. En d'autres termes, $pre\,E$ retourne, à un instant t, la valeur de l'expression E au moment $t-1$.
- Si E et F sont des expressions dénotant, respectivement, les séquences $(e_0e_1, e_2, ..., e_n, ...)$ et $(f_0, f_1, f_2, ..., f_n, ...)$, l'expression $E->F$ définit le flot $(e_0, f_1, f_2, ..., f_n, ...)$.

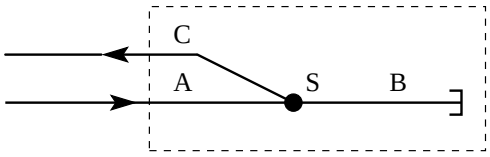

FIG. 2.3 – Une section U-tour de tramways

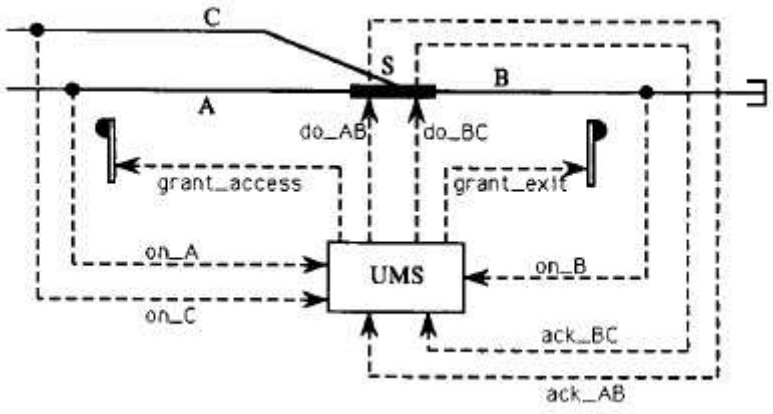

FIG. 2.4 – Le système UMS et son environnement

2.1.3 Exemple d'un programme Lustre

Nous reprenons ici un exemple connu [40] de système réactif contrôlant un aiguillage de tramways.

Dans chaque terminus de ligne de tramway, il y a une section spéciale ("U-turn") permettant aux trams de passer d'une voie à l'autre et de répartir dans le sens inverse (voir fig 2.3). Cette section est composée de trois voies A, B, C et un aiguillage S. Les trams passant de A à C doivent d'abord attendre que l'aiguillage connecte A avec B avant de transiter sur B et attendre que S connecte B avec C avant de répartir vers C.

Le système qui contrôle cette section est appelé *UMS (U-turn section Management System)* (voir fig 2.4). Ce système reçoit en entrée les signaux suivants :

- **ack_AB** et **ack_BC** qui indique si l'aiguillage connecte actuellement A avec B ou B avec C.
- **on_A**, **on_B**, **on_C** issus de trois capteurs, un pour chaque voie de la section, actifs s'il y a un tram sur la voie correspondante.

Les signaux de sortie pour ce système sont :

- **do_AB** et **do_BC** qui sont des requêtes vers l'aiguillage lui demandant de connecter A avec B ou B avec C.
- **grant_access** et **grant_exit** qui sont des autorisations de circulation

```
node UMS (on_A, on_B, on_C, ack_AB, ack_BC : bool)
         returns (grant_access, grant_exit, do_AB, do_BC : bool) ;
var empty_section, only_on_B : bool ;
let
   grant_access = empty_section and ack_AB ;
   grant_exit = only_on_B and ack_BC ;
   do_AB = not ack_AB and empty_section ;
   do_BC = not ack_BC and only_on_B ;
   empty_section = not (on_A or on_B or on_C) ;
   only_on_B = on_B and not (on_A or on_C) ;
tel
```

FIG. 2.5 – Le programme Lustre pour le système UMS

pour les trams (feux bicolores).
Une implémentation de ce système en Lustre est présentée dans la figure 2.5.

2.1.4 Vérification de programmes Lustre

2.1.4.1 Opérateurs temporels Lustre

Le langage Lustre peut être considéré comme une logique temporelle du passé, et peut de ce fait servir pour exprimer des propriétés invariantes décrivant des comportements attendus du système à vérifier ou des assertions sur son environnement d'exécution. Considérons, par exemple, le programme Lustre suivant :

```
node never (A : bool) returns (never_A : bool) ;
let
   never_A = not A -> (not A and pre (never_A)) ;
tel
```

Ce programme reçoit une entrée booléenne et a une unique sortie booléenne. A chaque instant, la sortie est vraie si seulement si l'entrée n'a jamais été vraie depuis le début de l'exécution du programme. Par exemple, le programme produit la séquence de sortie (*true, true, true, false, false*) en réponse à la séquence d'entrée (*false, false, false, true, false*).

D'autres opérateurs temporels peuvent être ainsi définis, dont (A, B et C sont des expressions booléennes Lustre quelconques)[39] :

- *always_from_to (A, B, C)* est vrai si A a toujours été vrai entre les deux derniers instants où B et C ont pris la valeur *vrai*.
- *once_from_to (A, B, C)* est vrai si A a été vrai au moins une fois entre les deux derniers instants où B et C ont pris la valeur *vrai*.
- *once_since (A, B)* est vrai si la valeur de A a été *vrai* au moins une fois depuis le dernier instant où B a été vrai.

– *always_since (A, B)* est vrai si la valeur de A a toujours été *vrai* depuis le dernier instant où B a été vrai.
– *after (A)* est vrai si la valeur de A a été *vrai* au moins une fois.
– *implies (A, B)* représente la valeur logique de l'implication : $A \Rightarrow B$.

2.1.4.2 Propriétés de sûreté

Les propriétés de sûreté garantissent l'absence de comportements dangereux du système. Dans le cas d'UMS, l'objectif est d'éviter deux types d'accidents :
– Une collision, si plusieurs trams sont autorisés à entrer simultanément dans la section critique.
– Un déraillement, si l'aiguillage est mal positionné.

Les propriétés invariantes suivantes garantissent l'absence de tels événements [39] :
– Le tram peut accéder à la section seulement si cette dernière est vide :

```
no_collision = implies(grant_access, empty_section)
```

– Il ne faut pas demander à l'aiguillage deux connexions à la fois :

```
exclusive_req = not(do_AB and do_BC)
```

– L'aiguillage doit toujours connecter A et B entre l'instant où le tram est autorisé d'accéder à la section et l'instant où il arrive sur la voie B :

```
no_derail_AB = always_from_to (ack_AB, grant_access, only_on_B)
```

– L'aiguillage doit toujours connecter B avec C entre l'instant où le tram est autorisé à quitter la section et l'instant où il la quitte :

```
no_derail_BC = always_from_to (ack_BC, grant_exit, empty_section)
```

2.1.4.3 Spécification de l'environnement

Dans le processus de vérification, il est important de fournir une description du comportement de l'environnement du système afin que seules les réactions du programme correspondant à des situations valides soient vérifiées. En Lustre, cette description est écrite à l'aide du mécanisme d'*assertions*, propriétés supposées toujours vraies. Dans l'exemple UMS, de telles propriétés sont :
– L'aiguillage ne peut pas connecter A avec B et B avec C en même temps :

```
not (ack_AB and ack_BC)
```

– L'aiguillage reste stable dans sa position sauf si un signal lui demandant de changer d'état lui parvient :

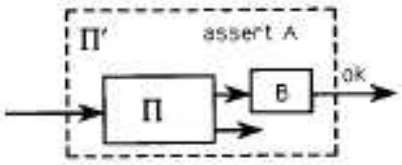

FIG. 2.6 – construire un programme de vérification

```
always_from_to (ack_AB, ack_AB, do_BC)
always_from_to (ack_BC, ack_BC, do_AB)
```

– Initialement, il n'y a pas de trams dans la section :

```
empty_section->true
```

– Les trams obéissent aux feux de circulation :

```
true->implies(edge(not empty_section), pre grant_access)
true->implies(edge(on_C), pre grant_exit)
```

– Lorsque le tram quitte A, il est en B. Lorsque le tram quitte B, il est soit en A ou en C :

```
implies(edge(not on_A), on_B)
implies(edge(not on_B), on_A or on_C)
```

2.1.4.4 Preuve formelle

Étant donné un programme Π, des propriétés de sûreté exprimées par une expression booléenne B et une assertion A, un nouveau programme Π′ peut être construit en mettant ensemble Π, B et l'assertion A comme indiqué dans la figure 2.6. Ainsi, pour vérifier les propriétés du programme Π, il suffit de vérifier que la seule sortie du programme Π′ est toujours vraie pendant toute exécution du programme qui satisfait l'assertion A.

La vérification formelle est réalisée en utilisant la technique de *model-checking* sur une machine d'état finis correspondant à une abstraction booléenne du programme [40] dont tous les états doivent être examinés. Ce principe de preuve formelle se heurte fréquemment au problème de l'explosion du nombre d'états. C'est cette limitation qui a initialement motivé nos travaux sur le test.

2.2 Test de programmes spécifiés en Lustre

Les différents travaux que nous avons conduits sur le test des logiciels synchrones nous ont amené à nous poser la question du positionnement méthodologique de cette activité dans un cycle de développement et à nous intéresser à plusieurs cas de figure :

- Lors du *test unitaire d'un composant*, on peut envisager de *générer automatiquement des tests* à partir de spécifications définies de la même manière que pour la preuve formelle. L'objectif, dans ce cas, est de fournir un moyen de vérification complémentaire à la preuve, moins précis que cette dernière, mais moins exposé aux limitations de complexité.
- Lors de la *spécification d'un composant*, l'animation des spécifications peut aider le concepteur à corriger et à affiner les spécifications avant la réalisation effective du composant.
- Toujours dans le cadre du test unitaire, et quelle que soit la manière dont les données de tests sont produites (fournies par un testeur humain ou bien générées automatiquement), il est nécessaire de pouvoir évaluer la qualité de ces dernières et de décider de l'arrêt du test.

Ces considérations, plutôt générales, peuvent être déclinées de manière différente en fonction du domaine d'application et du cycle de développement considérés qui peuvent être contraints par des normes précises (c'est le cas de DO-178B, dans le domaine de l'avionique) ou, tout simplement, par des processus métier établis. Le domaine d'application considéré a également des répercussions sur les objectifs de l'activité de test et l'importance des moyens mis en oeuvre.

Nos travaux se sont inscrits, à leurs débuts, dans le premier cas de figure, en adoptant notamment la même problématique que les travaux sur la preuve. Nous avons ensuite eu l'occasion de nous intéresser au deuxième cas de figure, dans le domaine des télécommunications, l'objectif étant la validation de spécifications de haut niveau. Cette particularité a permis de mettre l'accent sur la capacité des tests produits à détecter des défauts et sur les moyens fournis à l'utilisateur d'intervenir dans le processus de génération. Les travaux menés dans ce contexte s'intéressent, donc, à la génération de tests et sont présentés en détail dans le paragraphe 2.3.

Le troisième cas de figure nous a intéressé dans le contexte particulier du domaine de l'avionique, l'objectif étant d'évaluer la qualité de tests par rapport à un programme donné écrit en Lustre. Les travaux relatifs à ce thème sont présenté dans le paragraphe 2.4.

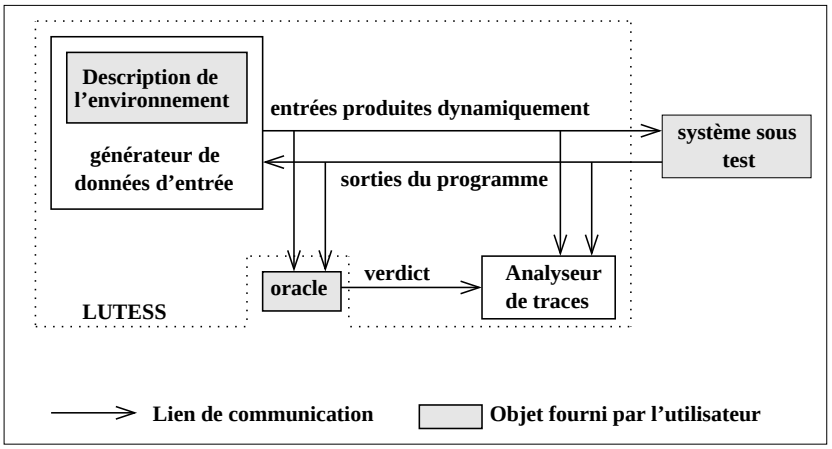

FIG. 2.7 – Architecture générale de Lutess

2.3 Automatisation des tests fonctionnels : Lutess

2.3.1 Analyse du problème

Comme il a été dit plus haut, les travaux sur le test des logiciels synchrones ayant abouti au développement de l'environnement Lutess [75, 27] avaient initialement pour motivation la volonté d'apporter un moyen de vérification complémentaire à la preuve formelle par « model-checking ». Les principes de spécification mis en oeuvre sont donc semblables à cette dernière. Mais contrairement à une preuve complètement automatique, dont le résultat est binaire, le test ne peut qu'être à l'origine d'une confiance relative à la correction du logiciel. Cette confiance sera d'autant plus grande que les tests exécutés seront de nature à détecter des défauts. Le défi principal de nos recherches sur le test est donc d'automatiser le processus de génération de sorte à renforcer la capacité des données de tests à détecter des défauts, soit en *exploitant automatiquement les spécifications du programme*, soit *en donnant à l'utilisateur la possibilité d'intervenir* en guidant la génération.

2.3.2 Principes de Lutess

L'outil Lutess, conçu pour répondre à cette problématique, requiert trois composants pour son fonctionnement (figure 2.7) :

- Le **système sous test** est un programme exécutable synchrone à entrées-sorties booléennes.
- L'**oracle de test** est un programme exécutable synchrone qui observe les

```
node Oracle_UMS(on_A, on_B, on_C, ack_AB, ack_BC,
   grant_access, grant_exit, do_AB, do_BC : bool) returns (ok : bool)
var empty_section, only_on_B,
    no_collision, exclusive_req, no_derail_AB, no_derail_BC : bool ;
let
   ok =  no_collision and exclusive_req and no_derail_AB  and no_derail_BC ;
   no_collision = implies(grant_access, empty_section) ;
   exclusive_req = not(do_AB and do_BC) ;
   no_derail_AB = always_from_to_(ack_AB, grant_access, only_on_B) ;
   no_derail_BC = always_from_to_(ack_BC, grant_exit, empty_section) ;
   empty_section = not (on_A or on_B or on_C) ;
   only_on_B = on_B and not (on_A or on_C) ;
tel
```

FIG. 2.8 – Un oracle pour UMS

entrées et sorties du système sous test et fournit à chaque pas de test un verdict. La figure 2.8 montre un exemple d'oracle pour le système UMS.

– La **description de l'environnement** est fournie sous la forme d'un noeud spécial (*testnode*). Son rôle est, d'une part, d'exclure les comportements irréalistes de l'environnement (opérateur *environment*) qui ne doivent pas être pris en compte pendant la génération de données de test et, d'autre part, de guider la génération de tests au moyen de directives ajoutée par le testeur. Les entrées de ce noeud sont les sorties du programme sous test et ses sorties sont les entrées du programme sous test (fig 2.9). La figure 2.10 montre une version simple de l'environnement du système UMS.

```
testnode Environnement(<sorties du logiciel>)
   returns (<entrées du logiciel>)
var
    <variables locales>
let
   environment(C1,...,Cn) ;
   <directives de génération>
   <définition des variables locales>
tel
```

FIG. 2.9 – Structure syntaxique d'un noeud de test

2.3.3 Génération de test avec Lutess

Nous présentons les différentes techniques de génération de test en deux temps, qui correspondent aux deux grands étapes des travaux sur Lutess :

– Dans un premier temps, on se limite à des spécifications à entrées et sorties booléennes. C'est dans ce cadre simple que la totalité des modèles et techniques de génération associées sont définis et présentés.
– Ensuite, on explique brièvement comment sont prises en compte les spé-

```
testnode Env_UMS(grant_access, grant_exit, do_AB, do_BC : bool)
   returns (on_A, on_B, on_C, ack_AB, ack_BC : bool)
var empty_section, only_on_B : bool ;
let
   environment(not (ack_AB and ack_BC),
               always_from_to_(ack_AB, ack_AB, do_BC),
               always_from_to_(ack_BC, ack_BC, do_AB),
               empty_section->true,
               true->implies(edge(not empty_section), pre grant_access),
               true->implies(edge(on_C), pre grant_exit),
               implies(edge(not on_A), on_B),
               implies(edge(not on_B), on_A or on_C)
             ) ;
   empty_section = not (on_A or on_B or on_C) ;
   only_on_B = on_B and not (on_A or on_C) ;
tel
```

FIG. 2.10 – Spécification de l'environnement d'UMS

cifications numériques, du point de vue théorique et pratique, notamment à l'aide de la programmation par contraintes.

2.3.3.1 Cas des spécifications booléennes

Nous illustrons les différentes techniques de génération de test sur un exemple simple qui est celui d'un contrôleur de climatiseur, dont la signature est donnée dans la figure 2.11. L'entrée booléenne *Bouton* est vraie à chaque fois que l'utilisateur appuie sur le bouton marche/arrêt ; les entrées *Tinf*, *Tok*, *Tsup* sont vraies si la température ambiante est respectivement inférieure, égale ou supérieure à celle fixée par l'utilisateur.

```
node Climatiseur(Bouton,Tinf,Tok,Tsup)
returns(En_marche,Froid,Inactif,Chaud : bool)
```

FIG. 2.11 – Signature Lustre du programme de contrôle de climatiseur

Un oracle pour ce programme pourrait être celui de la figure 2.12 : il se contente de vérifier la fonction "chauffage" du climatiseur.

```
node Oracle(Bouton,Tinf,Tok,Tsup,
    En_marche,Froid,Inactif,Chaud : bool) returns (ok : bool)
let
    ok = implies(En_marche and Tinf, Chaud) ;
tel
```

FIG. 2.12 – Exemple d'oracle

La figure 2.13 est un exemple de spécification de l'environnement pour ce climatiseur. Il stipule que la température ne peut pas passer directement de *Tinf* (froid) à *Tsup* (chaud) sans passer par *Tok*.

```
testnode Env_clim(En_marche, Froid, Inactif, Chaud : bool)
    returns (Bouton, Tinf, Tok, Tsup : bool)
let
    environment(#(Tinf,Tok,Tsup) and
                true -> implies(Tsup, not(pre Tinf))) ;
tel
```

FIG. 2.13 – Exemple de spécification d'environnement

Un noeud de test (*testnode*) est transformé en une machine d'états finis M_{env} = (S_{env}, $s_{init_{env}}$, V_i, V_o, env, $trans_{env}$) où :

- S_{env} est l'ensemble des états de l'environnement (ensemble de valeurs des variables d'état sve) ;
- $s_{init_{env}}$ est l'état initial de l'environnement ;
- i et o étant les vecteurs des paramètres d'entrée et de sortie du logiciel sous test, V_i et V_o sont leurs ensembles de valeurs associés ;
- La fonction env : $S_{env} \times V_i \rightarrow \{false, true\}$ définit à tout moment t l'ensemble des valeurs du vecteur d'entrée i conformes à la spécification de l'environnement (i.e. ensemble de valeurs rendant vraie la fonction env dans l'état où se trouve l'environnement à l'instant t) ;
- La fonction de transition $trans_{env} : S_{env} \times V_i \times V_o \rightarrow S_{env}$ calcule l'état de l'environnement après chaque échange d'entrées-sorties avec le logiciel synchrone.

Ainsi, au noeud de la figure 2.13 correspond la machine d'états finis M_{env} :

- $i = \{Bouton, Tinf, Tok, Tsup\}$
- $o = \{En\_marche, Froid, Inactif, Chaud\}$
- $s_{env} = \{sv_0, sv_1\}$, $s_{init_{env}} = \{(sv_0, sv_1) | sv_0 = true\}$
- $trans_{env} : sv_0 = false, sv_1 = Tinf$
- $env : sv_0 \vee not(sv_0) \wedge (not(Tsup) \vee not(sv_1))$

Simulation aléatoire

Le coeur du générateur de tests consiste en une animation des spécifications de l'environnement (c.à.d un parcours de la machine M_{env}). La sélection d'une donnée de test consiste à chaque instant t à (1) déterminer l'ensemble de valuations possibles V_i du vecteur d'entrée i_t qui dans l'état courant sve_t respectent les contraintes d'environnement : $env(sve_t, i_t) = true$, (2) choisir un élément i_t dans l'ensemble V_i et (3) après avoir envoyé les entrées i_t au logiciel et récupéré les sorties o_t, calculer l'état suivant de M_{env} (au moyen de la fonction de transition $trans_{env}$).

Le pas (1) nécessite la représentation de l'ensemble de valeurs valides de V_i retournées pas la fonction env. Cette dernière étant une fonction booléenne, elle peut être représentée efficacement par un Graphe de Décision Binaire (*BDD* -

Binary Decision Diagram). Cette représentation facilite le choix des valeurs du vecteur d'entrée i dans le pas (2). Soit sel_{env} la fonction effectuant ce choix. En étiquetant de manière adéquate le BDD associé à *env*, on garantit l'*équiprobabilité de tirage* par sel_{env} pour chacun des vecteurs d'entrée [75].

Il est facile de constater que l'équiprobabilité de tirage à chaque pas de test n'est en rien synonyme d'une *équiprobabilité des séquences produites* : en effet, il est fort peu probable que l'on obtienne des séquences telles que $Bouton = faux$ pour plusieurs pas consécutifs, la probabilité à chaque pas pour que cette valeur apparaisse étant *1/2*. Il est par contre fort probable que, dans les séquences produites, la valeur de *Bouton* change quasiment à chaque pas. Cela signifie que le climatiseur va rester très peu allumé et il sera difficile de tester son bon fonctionnement. Cette observation montre qu'il est nécessaire de disposer des moyens de guider la génération de tests de sorte à garantir une meilleure couverture des comportements possibles et, de ce fait, améliorer la capacité de détection de défauts des tests produits. Nous avons envisagé ce guidage de deux manières :

- Complètement automatisé : en se basant sur les propriétés de sûreté du programme, on peut identifier automatiquement des données menant vers des états de violation probable [93, 90].
- Basé sur une spécification supplémentaire de l'utilisateur fournie sous forme de profils opérationnels [71, 26].

Génération guidée par les propriétés de sûreté & des hypothèses

Le principe du test guidé par les propriétés de sûreté [72, 74] est de choisir un vecteur d'entrées tel que la valeur de vérité de la propriété de sûreté dépende des sorties calculées par le programme. Par exemple, si i est une entrée et o une sortie d'un programme devant satisfaire la propriété $i \Rightarrow o$, l'entrée $i = vrai$ doit être générée (sinon, la propriété est vraie quelle que soit la valeur de o). De même, pour la propriété $true \rightarrow \mathbf{pre}\ i \Rightarrow o$ il faut engendrer l'entrée $i = vrai$ à l'instant courant pour que la propriété soit violable à l'instant suivant. D'une manière générale, ce type de guidage consiste à engendrer à un instant t des entrées qui peuvent mener le logiciel dans une situation où la propriété peut être violée à un instant $t + k$, k étant le nombre maximum d'instants à considérer.

Il est nécessaire d'étendre M_{env} défini plus haut afin d'y intégrer les propriétés de sûreté. Les propriétés de sûreté peuvent être représentées par un automate d'états finis $M_{prop} = (S_{prop}, s_{init_{prop}}, V_i, V_o, prop, trans_{prop})$ où la fonction $prop : S_{prop} \times V_i \times V_o \rightarrow \{false, true\}$ définit à tout moment t la valeur de vérité de la propriété.

Étant donnée la machine $M_{env} = (S_{env}, s_{init_{env}}, V_i, V_o, env, trans_{env})$ asso-

ciée à l'environnement, la machine $M_{Ps} = (S_{Ps}, s_{init_{Ps}}, V_i, V_o, pert, env, trans_{Ps})$ permettant un guidage par les propriétés de sûreté peut être formellement définie :

- $S_{Ps} = S_{env} \times S_{prop}$ est l'ensemble des états ;
- $s_{init_{Ps}} = (s_{init_{env}}, s_{init_{prop}})$ est l'état initial ;
- V_i et V_o sont les ensembles de valeurs des entrées i et des sorties o ;
- $trans_{Ps} : S_{Ps} \times V_i \times V_o \rightarrow S_{Ps}$ est la nouvelle fonction de transition, définie par :
  $trans_{Ps}((s_{env}, s_{prop}), i, o) = (trans_{env}(s_{env}, i, o), trans_{prop}(s_{prop}, i, o))$.

La fonction $pert : S_{Ps} \times V_i \rightarrow \{false, true\}$ définit les vecteurs d'entrées dits *pertinents* : $pert(s_{Ps}, i) = \exists o \in V_o,\ (s_{Ps} = (s_{env}, s_{prop})) \wedge$
$env(s_{env}, i) \wedge (\neg prop(s_{prop}, i, o) \vee (\exists i' \in V_i,\ pert(trans_{Ps}(s_{Ps}, i, o), i')))$.

La différence principale avec la simulation d'environnement réside dans le calcul de l'ensemble des vecteurs pertinents. Ces vecteurs respectent les invariants d'environnement et sont susceptibles de violer la propriété de sûreté sur un chemin de longueur $1..k$. Dans [93], trois différentes stratégies de calcul sont proposées.

- La *stratégie d'union* considère tous les chemins de longueur $1..k$ qui mènent le logiciel à une situation susceptible de violer la propriété de sûreté (autrement dit *état suspect*, cf. paragraphe 2.4.3).
- La *stratégie d'intersection* ne considère que les vecteurs d'entrée qui font partie de tous les chemins de longueur $1..k$ qui mènent à un état suspect.
- La *stratégie paresseuse* choisit le vecteur qui mène le plus rapidement à un état suspect.

Ainsi, ce mode de génération nécessite la définition de deux paramètres pour maîtriser la complexité de la génération : la stratégie sélectionnée et la longueur k de chemins explorés.

La syntaxe utilisée (opérateur *safety*) est présentée dans la figure 2.14.

```
testnode Env_clim(En_marche, Froid, Inactif, Chaud : bool)
    returns (Bouton, Tinf, Tok, Tsup : bool)
let
    environment(#(Tinf,Tok,Tsup) and
                true -> implies(Tsup, not(pre Tinf))) ;
    safety( implies(En_marche and Tinf, Chaud) ) ;
    hypothesis( true -> Bouton = En_marche<>pre(En_marche) ) ;
tel
```

FIG. 2.14 – Guidage par propriétés & hypothèses

Dans cette même figure, on remarque la présence de l'opérateur *hypothesis*. En effet, le guidage par les propriétés de sûreté tel qu'il est défini plus haut présente un inconvénient : le programme sous test étant considéré comme une boîte

noire, plusieurs des vecteurs considérés comme pertinents peuvent, en réalité, ne pas l'être. En effet, la définition de la pertinence s'appuie sur une séquence d'entrées et de sorties qui doivent être produites avant d'atteindre un état suspect. Or, ne disposant pas de description du programme sous test (principe du test en boîte noire) nous sommes obligés de considérer que ce dernier peut produire beaucoup plus de valeurs de sortie qu'il n'en est réellement capable. Une solution à ce problème est l'introduction d'*hypothèses sur le programme* [90]. Ces hypothèses peuvent :

- être issues d'une analyse du programme, dans le cas où le code source ou une spécification sont disponibles,
- être issues de l'analyse des résultats de tests préalablement effectués (qui ont pu montrer la validité de certaines propriétés)
- ou bien être le simple fruit de l'expérience ou de l'intuition du concepteur des tests.

Quelle que soit leur origine, les hypothèses peuvent fournir une spécification partielle du programme sous test et leur exploitation peut améliorer la pertinence des vecteurs de test engendrés. En effet, au moment de la sélection de ces vecteurs, le générateur peut tenir compte de ces hypothèses et exclure ceux qui, bien qu'initialement considérés comme pertinents, donnent lieu à une réaction du programme rendant une violation des propriétés impossible.

Formellement, de façon similaire aux machines définies précédemment, on définit une hypothèse par un automate d'états finis M_{hyp} =(S_{hyp}, $s_{init_{hyp}}$, V_i, V_o, hyp, $trans_{hyp}$). Une machine permettant la simulation guidée par les propriétés de sûreté est défini par la machine M =(S, s_{init}, V_i, V_o, val, $trans$) où :

- $S = S_{Ps} \times S_{hyp}$ est l'ensemble des états ;
- $s_{init} = (s_{init_{Ps}}, s_{init_{hyp}})$ est l'état initial ;
- V_i et V_o sont les ensembles de valeurs de vecteurs d'entrées i et de sorties o ;
- $trans : S \times V_i \times V_o \rightarrow S$ est la nouvelle fonction de transition. Elle est définie par :
  $trans((s_{Ps}, s_{hyp}), i, o) = (trans_{Ps}(s_{Ps}, i, o), trans_{hyp}(s_{hyp}, i, o))$
- La fonction $val : S_{Ps} \times V_i \rightarrow \{false, true\}$ définit les vecteurs d'entrées valides. Ce sont les vecteurs *pertinents* qui respectent les hypothèses sur le programme
  $val(s, i) = (s = (s_{Ps}, s_{hyp})) \wedge pert(s_{Ps}, i) \wedge hyp(s_{hyp}, i)$.

L'utilisation du guidage par les propriétés et les hypothèses permet une recherche de violations de propriétés de sûreté pendant l'exécution. Il s'agit d'un principe proche de la preuve, en ce sens qu'un modèle impliquant la spécification de l'environnement, des propriétés de sûreté et une spécification du programme est exploré. Plus le nombre d'hypothèses introduites sera grand, plus ce modèle

sera proche de celui manipulé lors de la preuve. De même, plus la longueur k de chemins de recherche d'états suspects sera grande, plus la complexité de la génération sera proche de celle de la preuve. Ce type de guidage permet donc, en théorie, de réaliser de manière paramétrable la vérification du programme en mettant en oeuvre une recherche dynamique de défaillances de complexité et efficacité croissante.

Utilisation de profils opérationnels

L'utilisation de profils opérationnels [65] définis par l'utilisateur est une méthode couramment utilisée pour guider les tests. Dans notre cas, elle consiste à modifier les probabilités de tirage des valeurs des entrées de sorte à favoriser l'apparition de comportements plus variés, proches d'un profil d'usage. Sans rentrer dans les détails théoriques de la modélisation sous forme de chaînes de Markov [99], nous insistons ici sur le besoin pour un outil de génération d'offrir au testeur le moyen de définir des probabilités de tirage [71].

Le principe de génération de données est le même que pour la simulation aléatoire, à l'exception de la fonction sel_{env} qui n'effectue pas un tirage équiprobable mais tient compte du profil spécifié.

2.3.3.2 Cas des spécifications numériques - utilisation de la programmation par contraintes

La prise en compte de spécifications numériques dans le processus de génération de tests a été réalisée dans le cadre de la thèse de Besnik Seljimi et du projet RNTL DANOCOPS. Bien que les modèles et algorithmes utilisés pour la génération de test restent inchangés, de nouveaux problèmes à la fois théoriques et pratiques apparaissent dès lors que les calculs numériques entrent en jeu. Les difficultés pratiques ont été grandement résolues grâce à l'utilisation de la programmation par contraintes[1] qui permet une réalisation élégante et efficace de

[1] **Principes de programmation par contraintes**. Formellement, un problème de satisfaction de contraintes peut être représenté par un triplet (X, D, C) où :
- $X = \{X_1, X_2, ..., X_n\}$ est un ensemble de variables
- $D = \{D_1, D_2, ..., D_n\}$ est un ensemble de domaines non vides. Chaque domaine D_i représente l'ensemble de valeurs que peut prendre la variable X_i.
- $C = \{C_1, C_2, ..., C_m\}$ est un ensemble de contraintes. Chaque contrainte C_i définit une relation sur un sous-ensemble de variables qui spécifie les combinaisons de valeurs permises pour ce sous-ensemble.

L'instanciation d'une partie ou de l'ensemble des variables $\{X_i = v_i, X_j = v_j, ...\}$ définit un état du problème. Une instanciation qui satisfait l'ensemble des contraintes est dite consistante. Lorsque l'ensemble des variables est instanciée sans violer les contraintes du système, cette instanciation représente une solution du système de contraintes. Lors de la résolution d'un système de contraintes, des algorithmes de propagation sont mis en oeuvre dont le but est de déduire les valeurs qui ne peuvent pas constituer une solution (cette opération est appelée *filtrage*). Le domaine obtenu après le filtrage est appelée *domaine réduit*. Cependant le filtrage ne permet pas toujours de déterminer complètement les solutions du système. L'*énumération* consiste à faire des choix arbitraires puis recommencer le filtrage. Si un choix mène à une

l'ensemble des techniques de génération.

Nous ne discutons dans la suite de ce paragraphe que les conséquences les plus importantes de cette extension de Lutess. A cet effet, nous considérons une version modifiée du contrôleur du climatiseur. Dans cette version les entrées du programme sont la valeur de la température ambiante *Tamb* et la valeur de la température fixée par l'utilisateur *Tutil*, toutes deux de type entier. Le programme va alors calculer la température *Tsort* de l'air sortant du climatiseur (fig. 2.15).

```
node Clim(Bouton : bool ; Tamb, Tutil : int)
returns (En_marche : bool ; Tsort : int) ;
```

FIG. 2.15 – Signature Lustre du climatiseur numérique

La figure 2.16 présente une nouvelle spécification de l'environnement du climatiseur qui impose que les températures *Tamb* et *Tutil* soient comprises dans des intervalles raisonnables et que la température ne change pas de plus d'un degré entre deux instants successifs.

```
testnode Env_clim(En_marche : bool ; Tsort : int)
    returns(Bouton : bool ; Tamb, Tutil : int) ;
var dT : int ;
let
    dT = 0 -> Tamb - pre Tamb ;
    environment( dT >= -1 and dT <= 1
        and Tamb > -20 and Tamb < 60
        and Tutil > 10 and Tutil < 40 ) ;
tel
```

FIG. 2.16 – Description de l'environnement du climatiseur

Modélisation

La nécessité de connaître la différence entre la température ambiante *Tamb* et sa valeur à l'instant précédent *pre Tamb* introduit une variable d'état. L'état de la machine d'états finis correspondant à cet exemple est constitué de deux variables : sv_0 pour distinguer l'état initial et sv_1 pour mémoriser la valeur de *Tamb*.

- Variables d'état : $sve = \{sv_0, sv_1\}$
- État initial : $sv_0 = true$
- Fonction de transition : $sv'_0 = false \wedge sv'_1 = Tamb$
- Contraintes d'environnement :
  - $(sv_0 \wedge dT = 0) \vee (\neg sv_0 \wedge dT = Tamb - sv_1)$
  - $dT \geq -1 \wedge dT \leq 1 \wedge T_{amb} \geq -20 \wedge T_{amb} \leq 60 \wedge T_{util} \geq 10 \wedge T_{util} \leq 60$

contradiction, il est remplacé par un autre et ainsi de suite.

Simulation aléatoire

Pour générer des données de test qui respectent la spécification de l'environnement, il faut déterminer, à chaque instant t, l'ensemble sve_t des valeurs possibles pour les variables d'entrée i_t du programme, telles que $env(sve_t, i_t) = true$ et ensuite choisir un élément dans cet ensemble. Cette fonction est modélisée comme un problème de satisfaction de contraintes. Il s'agit de spécifier un système de contraintes (X, D, C) qui associe aux variables $X = sve_t \cup i_t$ un ensemble de contraintes C, tel que toutes les valeurs de i_t qui satisfont l'ensemble des contraintes représentent un vecteur d'entrée valide. La simulation aléatoire consiste à affecter aléatoirement aux variables des valeurs dans leur domaine réduit, jusqu'à obtenir une affectation complète qui vérifie l'ensemble des contraintes.

Notons que, contrairement à la version booléenne de Lutess, l'énumération aléatoire ne garantit pas l'équiprobabilité entre les différentes solutions possibles (l'ensemble des vecteurs d'entrée conformes à l'environnement), même si aucune des solutions n'est exclue. Cependant, on peut s'interroger sur la pertinence d'un tirage équiprobable quand les domaines des variables sont de taille très différente. Considérons, par exemple, la contrainte $b \in (0..1) \wedge x \in (0..2^{32} - 1) \wedge (b \Rightarrow x = 0)$. Dans une sélection équiprobable entre toutes les solutions, la probabilité de tirer une solution telle que $b = 1, x = 0$ est quasi nulle ($\frac{1}{2^{32}+1}$), alors que cette solution a manifestement un intérêt particulier. Par ailleurs, une sélection aléatoire dans le domaine de chaque variable peut donner une distribution différente selon l'ordre des variables selon lequel cette sélection est effectuée. Ainsi, si on choisit d'abord la variable x, on obtient une probabilité de $\frac{1}{2^{33}}$ d'avoir la solution $b = 1, x = 0$ alors que si on choisit d'abord la variable b, la probabilité d'avoir cette même solution est de $\frac{1}{2}$. Instancier d'abord les variables ayant le plus petit domaine réduit, ce qui, en particulier revient à considérer les booléens avant les variables entières, pourrait permettre d'avoir une distribution plus judicieuse du point de vue du test.

Profils opérationnels

L'utilisation de la programmation par contraintes permet d'étendre le guidage par les profils opérationnels aux contraintes numériques en permettant l'affectation de probabilités non seulement aux variables d'entrée mais à toute expression définie dans le noeud de test. Ainsi, l'utilisateur peut affecter des probabilités conditionnelles à des expressions, en utilisant la notation *prob(C,E,P)* où :

- C est une condition portant sur les valeurs passées des variables.
- E est une expression Lustre de type booléen.

– P est une constante réelle dans l'intervalle (0.0..1.0).

L'expression *prob(C,E,P)* signifie : si *C=true* alors la probabilité que *E=true* est *P*. Un exemple de spécification de profil opérationnel est présenté dans la figure 2.17. Selon ce profil, l'utilisateur a une forte (faible) probabilité d'appuyer sur le bouton marche/arrêt quand le climatiseur est arrêté (en marche). De plus, quand l'air issu du climatiseur est plus chaud que l'air ambiant, la température de ce dernier a de fortes chances d'augmenter.

```
testnode Env_clim(En_marche :bool ; Tout : int)
    returns(Bouton : bool ; Tamb, Tutil : int ;) ;
var dT : int ;
let
    dT = 0 -> Tamb - pre T ;
    environment( dT >= -1 and dT <= 1
        and Tamb > -20 and Tamb < 60
        and Tutil > 10 and Tutil < 40 ) ;
     prob(pre En_marche, Bouton, 0.1) ;
    prob(not(pre En_marche), Bouton, 0.9) ;
    prob(pre En_marche and pre Tamb < Tout, Tamb >= pre Tamb, 0.9) ;
tel
```

FIG. 2.17 – Description de l'environnement du climatiseur avec des probabilités

Pour prendre en compte les probabilités, on traduit l'ensemble des conditions et des expressions en contraintes. Si on a une liste de définitions de probabilités $\text{prob}(C_1, E_1, P_1)$, . . . , $\text{prob}(C_n, E_n, P_n)$, on tire aléatoirement n valeurs réelles $x_1, \ldots, x_n$ dans l'intervalle (0.0..1.0). Pour chaque i on pose la contrainte $C_i \Rightarrow (x_i \leq P_i \Leftrightarrow E_i)$. Mais il est possible que l'ensemble des contraintes ainsi exprimées ne soient pas satisfaisables. Dans ce cas, aucune entrée n'est possible (il y a une incohérence dans la spécification des probabilités). Face à ce problème, nous avons deux choix méthodologiques :

– Considérer que l'utilisateur exprime ces probabilités comme un moyen de guider la génération, sans se soucier de leur distribution effective et de leur cohérence. Dans ce cas on peut envisager la possibilité de continuer la génération en essayant de satisfaire un maximum de contraintes à chaque pas, mais pas forcément toutes.
– Considérer, au contraire, que la génération doit s'arrêter et que l'utilisateur doit rectifier la spécification de probabilités.

Cette question de cohérence d'un profil opérationnel reste une question ouverte.

2.4 Evaluation de la qualité des tests

2.4.1 Analyse du problème

L'évaluation de la qualité des tests est une nécessité dans tout processus de développement. En effet, la décision d'arrêter de tester un programme et

de considérer qu'il est correct est fondée sur la conviction des testeurs que les tests exécutés ont été suffisants pour détecter tout défaut. Cette conviction est le plus souvent laissée à l'appréciation des testeurs, même si, parfois, des obligations plus formelles sont imposés à ces derniers, comme dans le cas de la norme DO-178B (qui impose que chaque besoin fonctionnel soit testé, sans pour autant préciser la manière de le tester). Dans ce cas, la mesure automatisée de la qualité des tests est impossible. Mais il est courant, et impératif dans des secteurs de l'industrie où la qualité des logiciels est un facteur critique, que l'on se base sur des critères précis : on parle de "critères d'adéquation" [97].

La notion de critère a donné lieu a plusieurs études fondamentales depuis une trentaine d'années [33]. Selon [33], un critère est fiable si tous les tests conformes à ce critère détectent les mêmes défauts. Il sera, de plus, dit valide si tout défaut du logiciel peut être détecté par au moins un test conforme au critère. Mais en pratique, nous n'avons aucun moyen de montrer qu'un critère est fiable ou valide ou de concevoir un critère possédant ces deux qualités (cela nécessiterait, en effet, la connaissance de tous les défauts du logiciel). Néanmoins, ces deux attributs expriment bien les propriétés qu'on souhaite voir vérifiées par tout critère.

La définition de critères d'adéquation, à défaut d'être théoriquement garante de la détection de tous les défauts, doit fournir à l'utilisateur des éléments quantifiés lui permettant de comparer deux ensembles de test ou bien de juger de la progression obtenue après plusieurs campagnes de test.

Dans le cas des logiciels synchrones spécifiés en Lustre, on peut se poser la question de l'évaluation de la qualité des tests, et donc de la définition de critères d'adéquation, dans deux cas de figure :

- Des critères d'adéquation peuvent être définis sur le code du programme, si ce dernier est à la disposition des concepteurs des tests. Dans le contexte de nos travaux, présentés dans le paragraphe 2.4.2, seul le cas où le programme est écrit en Lustre nous intéresse.
- Dans un contexte de test en boîte noire, les critères doivent être exclusivement définis à partir des spécifications sans tenir compte du code du programme (par définition inconnu). Une proposition de tels critères est présentée dans le paragraphe 2.4.3.

2.4.2 Critères structurels pour des programmes Lustre

Quelques propositions de critères ont été faites dans le passé pour le cas de Lustre. L'approche proposée par [61] adopte comme modèle du programme l'automate d'états finis produit par le compilateur. Sur cet automate, des critères de couverture classiques tels que la couverture des états ou la couverture des

transitions, ou même celle des séquences de transitions, peuvent être appliqués. L'outil MTC de l'environnement SCADE Suite considère le graphe de contrôle du programme C généré. Ce modèle présente l'intérêt pratique de permettre d'utiliser les nombreux outils de test disponibles pour le langage C.

Ces deux modèles présentent l'inconvénient commun de travailler sur d'autres modèles que le programme original (qui dépendent, de plus, des options de compilation utilisés). Dans nos travaux, nous nous sommes orientés vers l'étude de critères définis sur les programmes Lustre, partant du constat que la notion de couverture est certainement mieux perçue par un utilisateur si elle porte sur le modèle qu'il a lu même construit, c'est à dire le programme Lustre lui-même. C'est ce principe qui a présidé la définition de la plupart des critères de couverture conçus pour d'autres types de programmes.

En effet, plusieurs critères ont été définis dans la littérature portant sur des représentations diverses d'un programme, dont la plus courante est le graphe de flot de contrôle (GFC), graphe orienté dans lequel les conditions sont représentés par des noeuds et les séquences d'instructions contiguës par des arcs (également appelés branches). Un chemin de ce graphe est une séquence de branches reliant le noeud d'entrée du graphe (début du programme) au noeud de fin (fin du programme).

Les graphes de flot de contrôle peuvent être utilisés à des fins d'analyse statique [48, 63] ainsi que pour la définition de critères de couverture [98]. Le critère le plus fort est la couverture des chemins et nécessite l'exécution de tous les chemins du programme, dont le nombre est potentiellement infini, ce qui rend cet objectif impossible à atteindre dans le cas général. De nombreux critères, moins exigeants, ont été proposés : la couverture des instructions ou bien des branches en sont les exemples les plus connus et, sans doute, les plus utilisés dans l'industrie.

Une famille de critères intéressante est celle définie à l'aide de constructions appelées LCSAJ (Linear Code Sequence And Jump) [100]. Un LCSAJ est une séquence finie d'instructions qui se termine par un saut conditionnel ou inconditionnel. Les LCSAJ peuvent être composés, de sorte à obtenir des séquences d'exécution de longueur arbitrairement longue. Ainsi, des critères de couverture ont été définis, nommés *Test Effectiveness Ratio* (TER_i) : TER_0 correspond à la couverture des instructions, TER_1 à la couverture des branches, TER_2 à la couverture des LCSAJ et TER_3, TER_4 ... TER_{n+1} à la couverture des sous-chemins constitués de *2, 3 ... n* LSCAJ. Notons qu'un test satisfaisant TER_{i+1} satisfait forcément TER_i (on dit que TER_{i+1} est plus fort que TER_i).

D'autres critères ont été proposés [22, 81] fondés sur l'analyse du flot de données. Dans ces critères, la couverture est définie en s'appuyant sur l'analyse des sous chemins reliant les définitions des variables à leurs utilisations potentielles.

Toute occurrence d'une variable est considérée soit comme une *définition* (affectation) soit comme une *utilisation* (la variable est référencée). L'utilisation d'une variable figure soit dans un prédicat dont l'évaluation détermine le chemin d'exécution (*utilisation-prédicat*), soit dans un calcul d'une autre variable ou comme valeur de sortie (*utilisation-calcul*). Les critères associés [81, 31] sont nombreux. La couverture de toutes les définitions (all-defs) est le plus simple de ces critères : il nécessite l'exécution d'au moins un chemin $p \in \mathcal{P}$ tel que p contient un sous chemin dans lequel la définition de x atteint d'autres occurrences-utilisation de x et ce pour toute variable x. Le critère le plus fort de cette famille est la couverture de tous les chemins définitions-utilisations (all-du-paths) selon lequel pour toutes les occurrences-définitions d'une variable x et pour tous les chemins q à travers lesquels une utilisation de x est atteignable, un chemin $p \in \mathcal{P}$ doit être exécuté tel que q est un sous-chemin de p.

On peut également noter les critères de couverture d'expressions booléennes définis dans [94, 95], dont la couverture de toutes les décisions (DC) - équivalente à la couverture de toutes les branches. Basés sur une décomposition d'une décision en conditions atomiques, appelées *conditions* [94] ou *clauses* [70], d'autres critères affinant le critère de couverture des décisions ont été proposés, comme la couverture des conditions (CC) la couverture des décisions (DC) la couverture des Conditions/Décisions (DCC), la couverture des conditions multiples (MCC) ou bien la couverture des décisions et conditions modifiées (MC/DC).

2.4.2.1 Définition de critères de couverture pour Lustre

Motivations

Intuitivement, un "bon" critère de couverture correspond à un compromis "acceptable" entre l'efficacité en terme de détection de défauts et le coût nécessaire à sa satisfaction. Idéalement, l'efficacité du critère devrait s'adapter au budget disponible pour le test. En nous basant sur cette intuition, nous nous sommes orientés vers la définition d'une famille de critères semblables à ceux fondés sur les LCSAJ, présentés ci-dessus. En effet, la définition de critères ordonnés par une relation d'inclusion et de complexité paramétrable par la longueur des chemins considérés, permet de les adapter facilement à la complexité d'un programme et aux ressources disponibles pour le test. La définition de ces critères, qui a fait l'objet de la thèse d'Abdesselam Lakehal [52], s'appuie sur le réseau d'opérateurs, représentation courante des programmes Lustre.

Le programme Lustre de la figure 2.18 décrit le fonctionnement d'une bascule. Son réseau d'opérateurs est représenté dans la figure 2.19.

Un *chemin* $p = (e_1, e_2, ..., e_n)$ est une séquence finie $e_1, e_2, ..., e_n$ d'arcs consécutifs. La longueur de p est le nombre n $(n > 1)$ d'arcs de p.

```
node latch (A, E, RCL, CLR : bool) returns (S : bool)
var
   _L1, _L2, _L3, _L4, _L5, _L6, _L7, _L8 : bool;
   C1, C2, C3, C4 : bool;
let
   S = _L6 and E;
  _L6 = not (_L1);
  _L1 = C4 ->_L3;
  C4 = false;
  _L3 = if _L2 then C3 else _L4;
  _L2 = RCL or _L7;
  _L7 = not A;
  C3 = false;
  _L4 = if CLR then C2 else _L5;
  C2 = true;
  _L5 = C1->_L8;
  _L8 = pre (_L1);
  C1 = true;
tel;
```

FIG. 2.18 – Bascule en Lustre

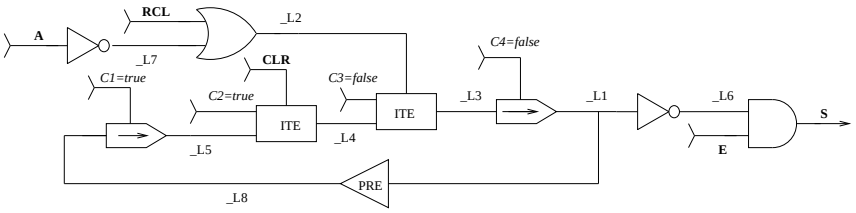

FIG. 2.19 – Réseau d'opérateurs du noeud latch

Le tableau 2.1 donne quelques exemples de chemins du programme latch.

A chaque chemin est associée une *condition d'activation*. Quand elle est vraie, tout changement en entrée du chemin provoque un changement de valeur de sa sortie. Le détail du calcul des conditions d'activation est donnée dans [52].

Le tableau 2.2 illustre cette notion sur le chemin `(A,_L7,_L2,_L3,_L1,_L6,S)` du noeud latch.

Nous avons défini trois familles de critères, chacune étant paramétrée par la longueur des chemins.

- La *couverture de base (BC_n)* est obtenue si tous les chemins du réseau de longueur inférieure à n ont été activés au moins une fois. Il est évident que la satisfaction de BC_n implique celle de BC_m pour tout $n > m$.
  Par exemple, la satisfaction de **BC7** pour l'exemple de la bascule est obtenu par la séquence `(A,E,RCL,CLR) = {(1,1,0,0),(0,1,1,0)}`.
- La *couverture des conditions élémentaires* est obtenue si tous les chemins

#	**Chemin**	**longueur**
1	(E,S)	2
2	(RCL,_L2,_L3,_L1,_L6,S)	6
3	(CLR,_L4,_L3,_L1,_L8,_L5,_L4,_L3,_L1,_L6,S)	11

TAB. 2.1 – Exemples de chemins

Chemin	(A,_L7,_L2,_L3,_L1,_L6,S)
Condition d'Activation	_L7 or not(RCL) and false->true and (not(_L6) or E)
Vecteur d'entrées	(A,E,RCL,CLR) = {(1,1,0,0),(0,1,1,0)}

TAB. 2.2 – Condition d'activation

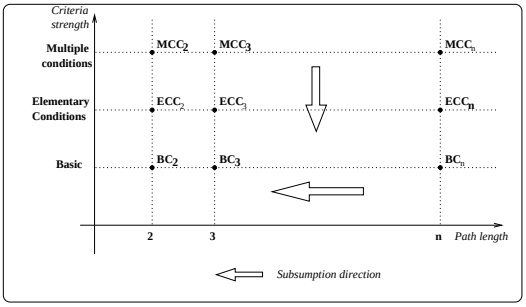

FIG. 2.20 – Relation d'inclusion entre critères

sont activés avec deux valeurs d'entrée différentes (vrai et faux pour les entrées booléennes). Elle renforce la couverture de base en contrôlant les variations des sorties en fonction des entrées.
Sur l'exemple de la bascule, **ECC7** est satisfait par la séquence `(A,E,RCL,CLR) = {(1,1,0,0), (0,1,1,0), (1,1,0,0), (1,1,0,1), (0,1,0,1), (0,1,1,1), (0,1,0,1), (1,0,0,0)}`.

– Enfin, la *couverture des conditions multiples* étend cette exigence à tous les arcs du réseau en imposant que deux valeurs différentes soient prises par chacun d'entre eux.
La satisfaction de **MCC7** dans l'exemple de la bascule est satisfait par la séquence `(A,E,RCL,CLR) = {(1,1,0,0), (0,1,1,0), (1,1,0,0), (1,1,0,1), (0,1,0,1), (0,1,1,1), (0,1,0,1), (1,0,0,0)}`.

Pour une même longueur de chemin, une relation d'inclusion est définie entre les trois familles de critères. Ces inclusions sont résumées dans la figure 2.20.

LUSTRUCTU [49] est un outil de mesure de couverture de programmes Lustre, dont le fonctionnement est présenté dans la figure 2.21. Il analyse le réseau d'opérateurs et calcule les chemins nécessaires à la satisfaction d'un critère donné ainsi que leurs conditions d'activation. Le résultat est un "noeud de couverture", programme Lustre calculant la satisfaction des conditions. Ce dernier est exécuté avec un jeu de test pour évaluer le taux de chemins couverts.

2.4.3 Critères pour le test boîte noire

Dans le cas classique d'utilisation de Lutess, on considère le logiciel comme une boîte noire, c'est à dire que nous ne disposons d'aucun modèle de fautes du

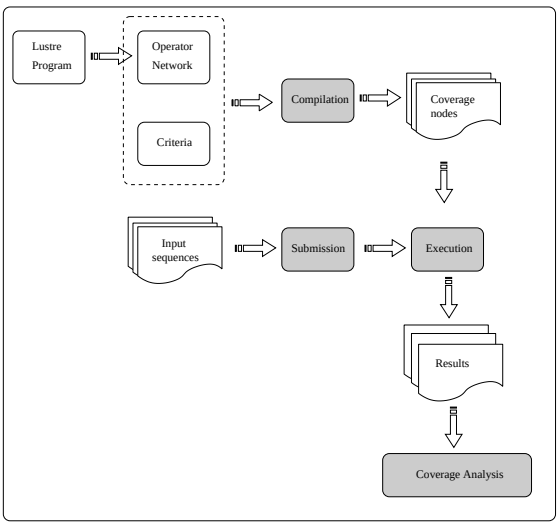

FIG. 2.21 – Fonctionnement de Lustructu

logiciel, nous ne faisons pas d'hypothèse sur les comportements de ce dernier et nous ne disposons pas de son code. Une manière directe de mesurer la qualité du test serait de compter le nombre de violations des propriétés de sûreté obtenues. Mais cette solution a le défaut d'être dépendante de l'implantation du logiciel ; en particulier, dans le cas où aucune violation n'est détectée, il n'est pas possible de savoir si ce résultat est dû à de mauvais cas de test ou à une implantation correcte vis-à-vis de la propriété testée.

Nous avons proposé [74, 93] d'utiliser la notion d'*état suspect* comme base de définition de critères d'évaluation. Intuitivement, un état est dit suspect s'il est possible de générer, dans cet état, un vecteur d'entrées qui met le logiciel en situation de fournir une réponse invalidant les propriétés de sûreté. Ainsi, un état suspect permet de caractériser formellement une situation critique. En se référant à la définition donnée au paragraphe 2.3.3.1 de la machine $M_{Ps} = (S_{Ps}, s_{init_{Ps}}, V_i, V_o, pert, env, trans_{Ps})$ permettant un guidage par les propriétés de sûreté, un état $s \in S_{Ps}$ est dit suspect ssi

$$\exists i \in V_i, \exists o \in V_o \; tel \; que \; \neg Ps(s,i,o) \; et \; (s,i) \in env$$

À partir de la notion d'état suspect, nous avons défini trois mesures :

1. Le *nombre total d'états suspects atteints* exprime le nombre de fois où le logiciel a été confronté à une situation critique.
2. Le *nombre total d'états suspects distincts atteints* exprime le nombre de situations critiques différentes auxquelles le logiciel a été confronté. Il est possible de compléter cette mesure en précisant la proportion de chacune

d'entre elles par rapport au nombre total de situations critiques atteintes.

3. Le *nombre d'états suspects atteints par une séquence A et non atteints par une séquence B* permet de comparer les séquences de test générées.

Par analogie avec le test structurel où la confiance dans le logiciel augmente avec le (ou les) taux de couverture retenu(s), notre confiance dans le logiciel augmente avec le nombre de situations critiques atteintes. Cependant, nous ne pouvons pas être sûrs d'avoir testé toutes les situations critiques ; au mieux nous pouvons estimer le nombre maximum d'états suspects (et donc de situations critiques) sans avoir la garantie qu'ils soient tous accessibles.

2.5 Perspectives

Plusieurs perspectives sont actuellement à l'étude et seront en grande partie traitées dans le cadre du projet ANR-Technologies logicielles SIESTA (1/2008-12/2010). Ces perspectives sont de nature méthodologique et technique.

Du point de vue méthodologique, le principal enjeu est de vérifier l'adéquation aux besoins de futurs utilisateurs de l'ensemble des techniques de génération de test et d'évaluation proposées. Une évaluation sur des études des cas fournis par nos partenaires industriels en collaboration avec d'autres laboratoires, permettra de progresser sur ce point.

Du point de vue technique, nos travaux sur l'évaluation des tests connaissent aujourd'hui certaines limitations que nous souhaitons traiter :

- La présence de plusieurs horloges n'est pas prise en compte dans l'évaluation de la couverture.
- La récente introduction d'automates synchrones dans Lustre/SCADE est également une source d'évolution pour la définition des critères et la manière de les utiliser.
- Enfin, la complexité de l'évaluation de la couverture (nombre de chemins) peut devenir prohibitive pour des noeuds Lustre de taille importante (ou incluant des appels à d'autres noeuds). De critères adaptés à l'intégration de plusieurs noeuds doivent, donc, être étudiés.

Un dernier sujet de réflexion concerne la vérification de la cohérence dans la spécification de profils opérationnels.

Chapitre 3

Test d'applications interactives et multimodales

3.1 Validation d'applications interactives

Les applications interactives sont aujourd'hui omniprésentes dans notre vie quotidienne, que ce soit lors de l'utilisation de systèmes commerciaux ou bien de logiciels embarqués sur les nombreux dispositifs portables et communiquants dont nous dépendons de manière croissante et leur bon fonctionnement est un facteur essentiel de l'économie. Leur utilisation est également fréquente dans des secteurs critiques (transports, aviation ...) ce qui accentue les exigences de qualité qui pèsent sur elles.

La multimodalité (capacité à interagir au moyen de plusieurs modalités, telles que le geste, la voix ...) contribue également à la complexification de ces systèmes, ce qui se traduit par une augmentation de la taille et de la complexité de leur code et une augmentation du risque d'introduction de fautes pendant le développement. En conséquence, la validation et la vérification des systèmes interactifs est une activité d'une haute importance et un domaine de recherche actif, qui connaît une évolution considérable ces dernières années.

Les exigences attendues d'un système interactif s'expriment souvent en termes d'*utilisabilité*, qualité qui caractérise sa capacité à permettre à l'utilisateur d'atteindre ses objectifs avec efficacité, en tout confort et sécurité. Elle est principalement déterminée par la **souplesse** et la **robustesse** de l'interaction [3]. La souplesse représente le degré des possibilités de choix offertes et regroupe plusieurs propriétés dont l'*atteignabilité* (possibilité de naviguer dans l'ensemble des états observables du système), l'*interaction multifilaire* (réalisation de plusieurs tâches de manière entrelacée) ou la *multiplicité du rendu* (plusieurs représenta-

tions d'un même concept). La robustesse a pour objectif de prévenir les erreurs et d'augmenter les chances de succès de l'utilisateur. Elle comprend des propriétés comme l'*observabilité* (capacité du système à rendre perceptible par l'utilisateur l'état pertinent du système), l'*insistance* (capacité du système à forcer la perception de son état), l'*honnêteté* (capacité du système à rendre observable son état sous une forme qui engendre une interprétation correcte de la part de l'utilisateur), la *curabilité* (capacité pour l'utilisateur de corriger une situation non désirée) ou la *prévisibilité* (capacité de prévoir, pour un état donné, l'effet d'une action).

On peut remarquer que ces propriétés ne sont pas facilement formalisables (voire pas formalisables du tout), entre autres parce qu'elles comportent une part d'appréciation subjective (par exemple, "l'état pertinent" du système dépend de la perception de chaque utilisateur). Leur évaluation sur une application donnée nécessite, donc, l'intervention d'utilisateurs humains. Dans [42] les techniques d'évaluation et de validation de systèmes interactifs sont classées en cinq catégories :

- **Evaluation expérimentale** : analyse des enregistrements d'utilisateurs réels.
- **Inspection** : examen statique de l'application par rapport à un ensemble de critères.
- **Interrogatoire/entretien** : réalisation d'enquêtes auprès des utilisateurs.
- **Analyse de modèles** : utilisation de modèles de l'utilisateur et/ou de l'interface pour générer des prédictions d'utilisabilité.
- **Simulation** : utilisation de modèles de l'utilisateur et/ou de l'interface pour imiter le comportement interactif des utilisateurs avec le système.

On note que l'existence de modèles, parfois formels, dans les deux dernières catégories rend possible le recours à la validation automatique, tandis que les trois premières comportent des techniques difficilement automatisables.

Si l'on se réfère au cycle de développement de systèmes interactifs de la figure 3.1 [8], trois étapes sont particulièrement concernées par la validation :

- L'analyse des besoins, où des modèles de l'utilisateur ou de l'interface sont spécifiés.
- L'étape de conception, où des méthodes formelles peuvent être utilisées pour spécifier l'application ou son interface et vérifier des propriétés.
- L'étape de test et validation, où les modèles et spécifications précédentes sont utilisés.

Des méthodes de vérification automatique ont été proposées dans le passé, notamment s'appuyant sur des modèles développés dans l'étape de conception. Parmi elles, on peut citer une approche basée sur LOTOS [76] qui s'appuie sur le concept d'interacteur pour structurer la description des interfaces. L'interface

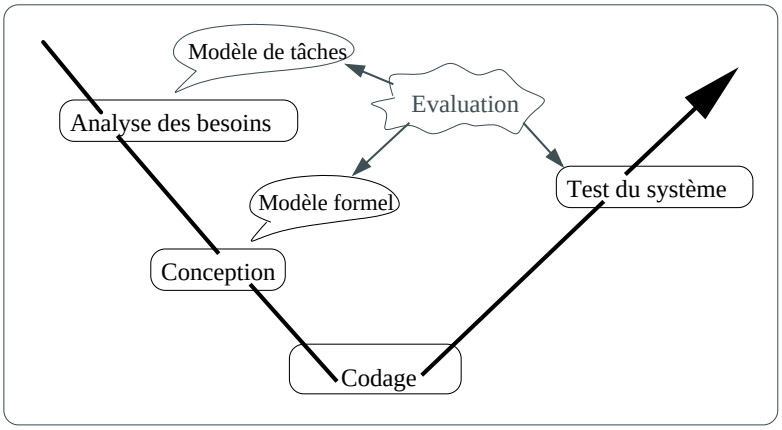

FIG. 3.1 – Processus de développement des systèmes interactifs

est une composition d'interacteurs qui fonctionnent en parallèle et échangent des événements. Le modèle LOTOS sert à vérifier des propriétés comme l'atteignabilité, la réactivité, la conformité [77] exprimées dans la logique ACTL (Action-Based Temporal Logic). Ce modèle reflète bien l'architecture de l'interface mais pas le comportement dynamique de l'interaction. C'est pourquoi, il a été proposé d'exprimer cette dynamique dans la notation CTT (ConcurrTaskTree) [64], qui peut être automatiquement traduite en LOTOS.

Selon l'approche ICO (« Interactive Cooperative Objects ») [9] une spécification formelle est réalisée dans un langage orienté objet où le comportement de l'application est décrit par un réseau de Petri de haut niveau. Cette spécification peut être exécutée, testée et vérifiée en utilisant les outils relatifs aux réseaux de Petri.

Dans [102, 104], il est proposé d'utiliser la méthode B [4] et le modèle événementiel pour valider les tâches utilisateur à la phase de conception et spécification à partir d'un modèle de tâches CTT.

L'utilisation du langage Lustre et des outils de vérification associés a été également proposée pour la spécification et la validation d'interfaces à fenêtres. [85, 25, 91]. L'interface est modélisée par un réseau d'interacteurs, automates qui réagissent à des actions d'entrée en modifiant leur état et en générant des événements de sortie. Le modèle Lustre de l'interface est utilisé pour la vérification de propriétés par model-checking [24].

Enfin, l'utilisation de modèles de Markov a été proposée pour tester l'utilisabilité d'Interfaces Homme Machine à la phase de conception [92]. L'approche s'applique aux interfaces modélisables au moyen de machines d'états finis, l'utilisabilité étant mesurée par le nombre d'étapes nécessaires à l'utilisateur pour réaliser des tâches.

Enfin, on peut citer la méthode proposée dans [43] dont le principe est de générer l'ensemble complet de séquences d'interaction à partir de l'analyse de modèles de tâches (version simplifiée de HTA [2] et MAD [87], sans récursivité ni boucles).

3.2 Utilisation de l'approche synchrone pour le test d'applications interactives

3.2.1 Analyse du problème

A l'issue de l'état de l'art présenté au paragraphe précédent, bref mais représentatif des recherches du domaine, on constate que la plupart des approches de vérification proposées nécessitent une spécification complète de l'application interactive, ce qui est en général trop fastidieux pour être réaliste. Par ailleurs, peu de travaux spécifiquement dédiés au test automatique ne s'appuyant pas sur une telle spécification ont eu lieu.

L'approche de génération automatique de tests présentée dans le chapitre 2 et intégrée dans Lutess a été utilisé avec succès pour la recherche d'interactions entre services téléphoniques (victoire au concours FIW'98 [36]). Cette expérience a montré que Lutess pourrait être utilisé en dehors du cadre strictement synchrone et que la richesse des modes de génération qu'il intègre était un facteur déterminant pour la détection d'anomalies dans les applications testées.

Pour ces raisons, nous avons souhaité aborder l'étude du test d'applications interactives en nous intéressant dans un premier temps à l'utilisation de Lutess. La problématique abordée par ce travail exploratoire peut être résumée en cinq questions :

1. Comment réaliser une modélisation synchrone de l'interaction ?
2. A quel type de propriétés est adapté ce mode de validation ?
3. Les techniques de test de Lutess sont-elles adéquates ?
4. En supposant que les réponses aux trois premières questions font apparaître un intérêt de l'approche, nous devons nous interroger sur sa compatibilité avec la culture des concepteurs d'applications interactives. Cela nous amène à étudier des moyens d'adapter l'approche à des modèles couramment utilisés dans le domaine.
5. Enfin, la prise en compte de la multimodalité est le dernier aspect que nous étudions ainsi que les défis que cette dernière pose en termes de validation.

Les réponses apportées sont présentées dans la suite du chapitre. Elles sont le résultat du travail effectué en partie dans le cadre du projet RNRT VERBATIM et de la thèse de Laya Madani [58].

FIG. 3.2 – Gauche : un utilisateur de Memo, équipé d'un casque semi-transparent.
Droite : une vue à travers le casque semi-transparent. L'utilisateur mobile est devant le bâtiment de l'UFR IMAG et peut voir deux notes digitales.

Nous illustrons ces réponses sur un exemple d'application, Memo [19], un système interactif multimodal qui permet d'annoter des localisations physiques avec des « post-it » virtuels. Les post-it digitaux peuvent être ensuite lus, portés ou supprimés par d'autres utilisateurs mobiles. L'utilisateur de Memo de la figure 3.2 est équipé d'un casque semi-transparent qui permet la fusion de données (les notes digitales) avec l'environnement réel. De plus, un GPS et un magnétomètre permettent au système de calculer la localisation et l'orientation de l'utilisateur. Ce dernier peut effectuer trois tâches : (1) Changer l'orientation et la localisation de l'utilisateur mobile, (2) manipuler (récupérer, placer ou supprimer) une note, (3) quitter le système.

3.2.2 Modélisation synchrone de l'interaction

Selon [12], il est possible de construire une trace synchrone à partir d'une trace asynchrone, si on affecte à chaque signal s une horloge (variable) hs qui est toujours présente. A chaque cycle de l'horloge globale, on examine l'existence de chaque signal : si le signal s existe, alors l'horloge hs sera vraie. Ce principe permet de faire communiquer simplement Lutess avec une application interactive, à condition qu'un traducteur soit introduit prenant en charge la désynchronisation des événements d'entrée et la synchronisation des événements de sortie.

Un autre aspect important est que, dans la cadre de l'utilisation de Lutess, les types des données échangées entre le générateur de test et l'application interactive sont limités (booléens, numériques) ce qui impose un codage des entrées et de sorties. Dans l'exemple de Memo, les entrées sont codées comme suit :

1. *Localisation* est un vecteur booléen indiquant le mouvement de l'utilisateur

selon les directions x, y et z. A un instant donné, *Localisation[plus]= true* signifie que l'abscisse de l'utilisateur augmente d'une valeur fixe prédéfinie.

2. *Orientation* est un vecteur booléen indiquant les changements d'orientation de l'utilisateur selon trois angles d'orientation : yaw, pitch et roll. *Orientation[pitchplus]= true* signifie que l'utilisateur baisse la tête d'un angle fixe prédéfini.
3. *cmdget*, *cmdset* et *cmdremove* sont des variables booléennes correspondant aux commandes "get", "set" ou "remove". A un instant, *cmdget= true* indique que l'utilisateur envoie la commande "get" au système.

L'application Memo produit les cinq sorties booléennes suivantes :

1. *memoDisplayed*, vrai quand au moins une note est affichée sur le viseur.
2. *memoCarried*, vrai quand l'utilisateur porte une note.
3. *memoTaken*, vrai si l'utilisateur vient de récupérer une note.
4. *memoSet*, vrai si l'utilisateur vient de poser une note.
5. *memoRemoved*, vrai si l'utilisateur vient de supprimer une note.

Le coût de la création du traducteur peut varier en fonction de l'application. Dans le cas de Memo, ce coût est faible, parce que les événements en entrée et en sortie sont de type booléen ou entier (ou peuvent être facilement codés en tant que tels). Mais on peut imaginer que ce codage ne soit pas toujours facile. En effet, si pour Memo les aspects "contrôle" dominent l'application qui manipule peu de données, d'autres applications sont essentiellement axées sur la manipulation de données (par exemple un annuaire électronique), ce qui rend cette modélisation beaucoup plus difficile.

3.2.3 Type de propriétés validées

On peut s'interroger sur le type de propriétés que l'on peut intégrer dans un oracle automatique. Ces dernières doivent être évaluables à partir du résultat de l'exécution du programme, ce qui exclut la prise en compte de propriétés de type "vivacité" (certaines propriétés d'utilisabilité sont de tel type, comme l'atteignabilité, permettant de garantir à l'utilisateur qu'un ensemble de tâches lui sont accessibles à tout instant).

Ainsi, on peut automatiser l'évaluation de propriétés fonctionnelles exprimant le fait qu'une séquence d'événements produits par l'utilisateur donne lieu à la production d'une séquence appropriée d'événements de sortie. Dans la mesure où ces propriétés peuvent s'exprimer dans une logique temporelle du passé, leur écriture en tant que formules Lustre est donc possible.

Dans le cas de Memo, les exemples de propriétés ci-dessous expriment que les notes sont récupérées, placées ou supprimées uniquement par des commandes adéquates :

- Après avoir vu une note et avant de la récupérer, l'utilisateur doit faire une commande "get" à un instant où une note est affichée (plus précisément, à un instant où une note est suffisamment proche de l'utilisateur pour être manipulée).

```
once_from_to(cmdget and pre memoDisplayed,
             memoDisplayed,memoTaken)
```

- Entre l'instant où l'utilisateur voit ou porte une note et l'instant où une note est supprimée, il doit faire une commande "remove".

```
once_from_to(cmdremove and (pre memoDisplayed or pre memoCarried),
             memoDisplayed or memoCarried, memoRemoved)
```

On peut exprimer de la même manière que l'état de Memo ne change que suite à l'arrivée d'événements d'entrée adéquats [58].

Les mêmes contraintes sont valables pour des propriétés d'utilisabilité. Cependant, comme nous l'avons déjà mentionné, le manque de formalisation dans leur définition nous incite à nous orienter vers l'étude des moyens d'écrire des scénarios concourant à leur validation plutôt qu'à chercher à tout prix leur expression formelle. Par exemple, pour tester l'atteignabilité, on peut spécifier un scénario décrivant les actions permettant de naviguer entre les différents états de l'application (dans Memo, l'action *"set"* change l'état de *"not memoCarried"* à *"memoCarried"*) puis observer son déroulement. Ce constat nous amène naturellement à la question suivante : l'adéquation des techniques de tests de Lutess.

3.2.4 Adéquation des techniques de test de Lutess

Rappelons que la génération de tests s'appuie sur un ensemble d'invariants contraignant l'environnement de l'application (donc, l'utilisateur) ainsi que sur des directives de guidage. Dans le cas de Memo, le comportement de l'utilisateur est contraint par les invariants suivants :

1. L'utilisateur ne peut pas bouger sur un axe dans les deux directions en même temps (de même, l'utilisateur ne peut pas tourner autour d'un axe dans les deux directions à la fois) :

```
not (Localisation[xminus] and Localisation[xplus])
not (Localisation[yminus] and Localisation[yplus])
not (Localisation[zminus] and Localisation[zplus])
```

```
not (Orientation[yawminus] and Orientation[yawplus])
not (Orientation[pitchminus] and Orientation[pitchplus])
not (Orientation[rollminus] and Orientation[rollplus])
```

2. L'utilisateur ne peut pas envoyer plus d'une commande à un instant (ceci est physiquement impossible) :

```
AtMostOne (cmdget, cmdset, cmdremove)
```

Nous avons déjà mentionné au chapitre 2 l'insuffisance de la simulation aléatoire de ces invariants qui produit des comportements dans lesquels chaque événement a la même probabilité d'occurrence à un instant donné. Cela signifie, par exemple, que *Localisation[xminus]* est généré autant de fois que *Localisation[xplus]* et, en conséquence, la position et l'orientation de l'utilisateur changeraient très peu. Si nous souhaitons que ce dernier explore le terrain, la génération de données doit être guidée par un profil opérationnel. Ainsi, si l'on souhaite que l'utilisateur tourne plus souvent la tête vers la droite, on pourrait écrire :

```
proba((Orientation[yawminus], 0.80),
      (Orientation[yawplus], 0.01),
      (Orientation[pitchminus], 0.01),
      (Orientation[pitchplus], 0.01),
      (Orientation[rollminus], 0.01),
      (Orientation[rollplus, 0.01))
```

On peut également spécifier qu'une commande "get" a une forte probabilité d'être générée quand l'application affiche une note sur le viseur :

```
proba ((cmdget, 0.8, pre memoDisplayed))
```

En théorie, il est donc possible d'exprimer des scénarios explorant l'ensemble du modèle du comportement de l'utilisateur avec différents profils pour ce dernier (utilisateur expert, utilisateur naïf ....). Mais ce modèle synchrone écrit en Lustre et enrichi de profils opérationnels n'est pas facile à construire pour les concepteurs d'applications interactives, non familiers de ce langage. Une manière de palier à cet inconvénient est de s'appuyer sur des modèles qui sont plus habituels dans le processus de développement des telles applications.

3.2.5 Utilisation d'arbres de tâches

Parmi les modèles les plus connus dans le domaine de la conception d'applications interactives, on trouve les *arbres de tâches*. Ces derniers sont souvent utilisés dès l'analyse des besoins pour décrire l'interaction entre l'application et l'utilisateur. Ils sont constitués de tâches composées de sous-tâches liées par

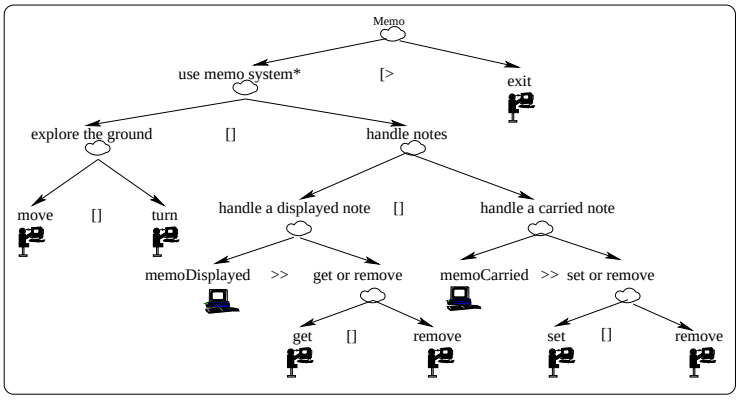

FIG. 3.3 – Exemple d'arbre de tâches CTT

des opérateurs temporels [1]. Plusieurs modèles de tâches ont été proposés : JSD (Jackson System Development) [55], HTA (Hierarchical Task Analysis) [2], MAD (Méthode Analytique de Description de tâches) [86], UAN (User Action Notation) ou bien CTT (Concur Task Tree) [64, 78]. Parmi ces modèles, nous avons retenu CTT, une notation bien connue, intégrée dans un éditeur graphique qui permet de créer et sauvegarder un modèle en plusieurs formats. La figure 3.3 montre un exemple d'arbre de tâches dans la notation CTT correspondant à l'application Memo. On y distingue trois des opérateurs disponibles en CTT :

1. **Activation (T1 >> T2)** : La tâche T2 est activée par l'exécution de la T1.
2. **Choix (T1 [] T2)** : Une seule de deux tâches est exécutée, T1 ou T2.
3. **Désactivation (T1 [> T2)** : T1 est désactivée lorsque la première action de T2 est effectuée.

Cet arbre montre que l'utilisateur peut utiliser l'application itérativement (opérateur d'itération *) et peut être interrompu (opérateur de désactivation [>) par la tâche "exit". La tâche "use memo system" est un choix entre les trois tâches suivantes : découvrir le terrain ("explore the ground"), manipuler une note visible ("handle a displayed note") ou manipuler une note portée ("handle a carried note"). Si le système affiche une note ("memoDisplayed"), l'utilisateur peut (opérateur d'activation >>) récupérer ou supprimer cette note. Si l'utilisateur est en train de porter une note ("memoCarried"), l'utilisateur peut la placer ou la supprimer.

On distingue quatre types de tâches :

1. **Tâche usager** : activité cognitive interne sans interaction avec le système, comme la réflexion pour résoudre un problème, la planification, la lecture d'un message, etc.

2. **Tâche application** : réalisation par le système, comme l'affichage du résultat d'une requête, la production d'alerte, etc.

3. **Tâche interactive** : action usager avec un feedback immédiat par le système, comme l'édition d'un document.

4. **Tâche abstraite** : tâche composée d'autres sous-tâches.

Nous avons étudié le moyen de traduire automatiquement un arbre de tâches CTT vers un modèle similaire à celui utilisé par Lutess pour simuler l'environnement (cf. paragraphe 2.3.3). Pour cela nous faisons certaines observations et hypothèses de modélisation.

Du point de vue de la modélisation de l'interaction, les tâches usager ne sont pas intéressantes, car elles ne correspondent à aucune interaction avec le système (ni action ni réaction).

Quant aux tâches application, nous les assimilons à des machines d'états élémentaires à deux états, dont la seule transition consiste à passer de l'état initial à l'état final en émettant une sortie (cela correspond à une vision "boîte noire" de l'application).

La principale hypothèse concerne les tâches interactives. Nous supposons qu'une tâche interactive est assimilée à une action élémentaire suivie d'une réaction. Elle est modélisée par une machine d'états finis à E/S avec une seule transition étiquetée avec cette action et sa réaction à partir de l'état initial vers l'état final. Cette modélisation doit être fournie en même temps que l'arbre des tâches.

Enfin, une tâche abstraite est composée d'autres sous-tâches liées par des opérateurs de CTT. Une machine à E/S est également associée à une tâche abstraite, et est le résultat de la composition des machines à E/S de ses sous-tâches, définie en détail dans [58].

Dans l'arbre de tâches de Memo, la tâche abstraite *"Memo"* est définie à l'aide de six tâches interactives : *"get"*, *"set"*, *"remove"*, *"move"*, *"turn"*, *"exit"* et deux tâches application : *"memoDisplayed"*, *"memoCarried"*. Ces tâches sont modélisées par les machines à E/S illustrées par la figure 3.4. La machine à E/S correspondant à l'arbre de tâches de Memo de la figure 3.3 est illustrée par la figure 3.5.

Dans [58] nous avons également étudié la possibilité de décorer les arbres de tâches avec des profils opérationnels et d'obtenir des automates probabilistes associés. La manière de définir une sémantique pour les arbres de tâches dans ce cadre est proche des travaux sur la définition de versions probabilistes de LOTOS [69].

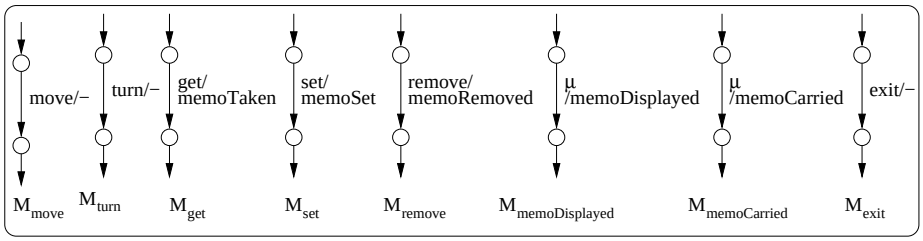

FIG. 3.4 – Les machines à E/S pour les tâches :
"move", "turn", "get", "set", "remove", "memoDisplayed", "memoCarried" et "exit"

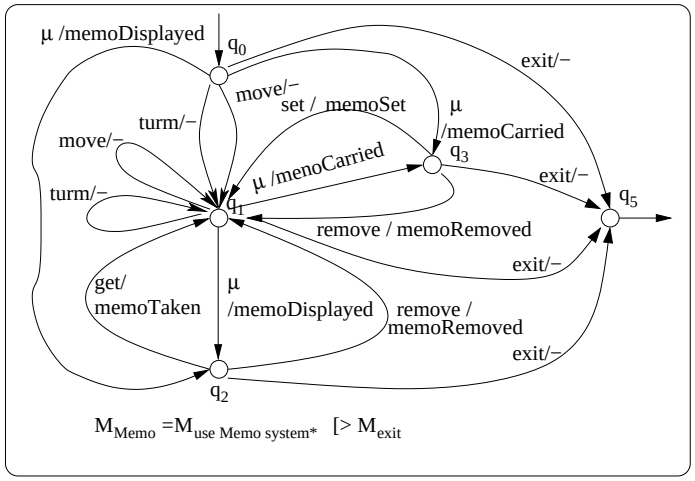

FIG. 3.5 – La machine à E/S pour l'arbre de tâches de Memo

3.2.6 Vers une prise en compte de la multimodalité

3.2.6.1 Modalités et multimodalité

D'après [68][67], une modalité est un couple constitué d'un dispositif physique d et d'un langage d'interaction L : <d,L>. Un dispositif physique est un objet du système qui acquiert (dispositif d'entrée) ou distribue (dispositif de sortie) de l'information. Des exemples de dispositifs sont le clavier, la souris, le microphone ou l'écran. Le langage d'interaction se définit par un ensemble d'expressions bien définies (i.e., un ensemble conventionnel de symboles) qui porte des sens.

Nous distinguons deux classes de modalités : les modalités actives et les modalités passives. Une modalité d'entrée est active quand l'utilisateur doit réaliser une action explicite avec un dispositif en vue de spécifier une commande au système comme la parole <microphone, langage pseudo-naturel> ou les interfaces graphiques manipulables <souris, la manipulation directe>. Elle est passive quand le dispositif associé ne requiert pas d'action explicite comme la capture par un GPS de la localisation d'un utilisateur.

Un système interactif multimodal [67] dispose d'au moins deux modalités pour un sens donné (entrée ou sortie) comme la parole et une interface manipulable par la souris. Les modalités peuvent être indépendantes ou combinées [66], si une fusion entre les différents types de données associées à ces modalités a lieu. Elles peuvent être utilisées de manière séquentielle ou parallèle [23, 53] dans une fenêtre temporelle (intervalle de temps) TW.

Dans l'exemple de l'application Memo, il y a cinq modalités d'entrée. Trois modalités actives sont utilisées par l'utilisateur pour manipuler les notes - (souris, commandes de bouton), (microphone, commandes vocales) et (clavier, commandes de clavier). Deux modalités passives sont utilisées pour déterminer la localisation et l'orientation de l'utilisateur : (magnétomètre, trois angles d'orientation en radians) et (capteur de localisation GPS, localisation).

Les propriétés CARE [23] définissent quatre manières d'utiliser des modalités : la Complémentarité, l'Assignation, la Redondance et l'Equivalence.

- Une modalité M est dite assignée à un ensemble T de tâches, si chaque tâche $t \in T$ ne peut être provoquée que par la modalité M.
- Les modalités d'un ensemble M sont équivalentes pour la tâche t, si t peut être provoquée par l'une quelconque des modalités de M.
- Les modalités d'un ensemble M sont complémentaires pour la tâche t, si pour exécuter t toutes ces modalités doivent être utilisées simultanément ou de manière séquentielle restreinte par une fenêtre temporelle.
- Les modalités d'un ensemble M sont redondantes pour la tâche t, si pour exécuter t la même information doit être émise par chacune de ces moda-

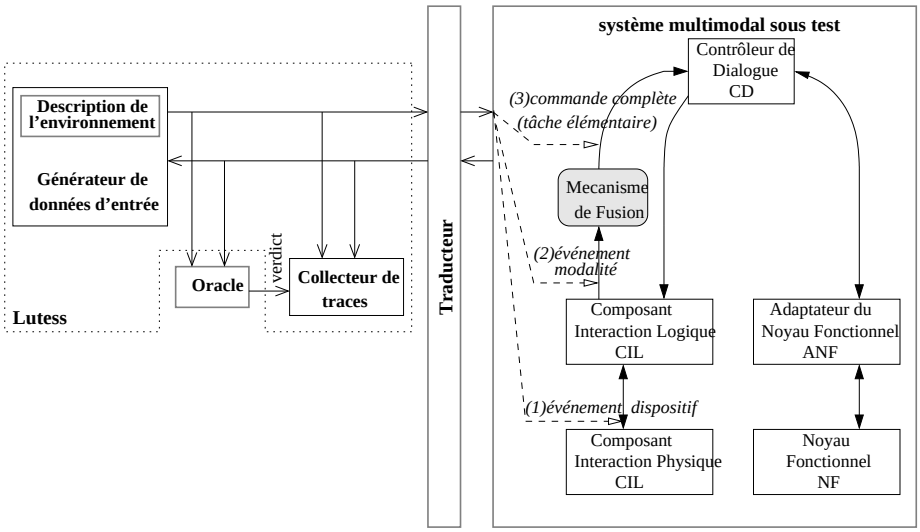

FIG. 3.6 – Connexion Lutess-système multimodal organisé selon le modèle PAC-Amodeus

lités simultanément ou de manière séquentielle restreinte par une fenêtre temporelle.

3.2.6.2 Validation d'applications multimodales

Parmi les approches formelles proposées pour la validation de systèmes interactifs, certaines proposent des extensions leur permettant de prendre en compte la multimodalité. C'est en particulier le cas de l'approche ICO qui a été étendue [6] par l'ajout du temps, ainsi que par des mécanismes de communication et de structuration, ou bien de la méthode B [103, 101]. Une méthode formelle de modélisation et de vérification de systèmes interactifs multimodaux est également proposée dans [5, 30]. Cette approche se concentre sur les modalités d'entrée et leur fusion et consiste à définir un modèle formel pour le système interactif multimodal et les propriétés.

Dans le cas de Lutess, la prise en compte de la multimodalité nécessite, d'une part, l'adaptation du niveau d'abstraction des événements d'entrée-sortie échangés et, d'autre part, la définition de moyens de tester les aspects liés à la multimodalité.

Nous nous sommes appuyés sur le modèle PAC-Amodeus [67], selon lequel un système multimodal est composé de cinq composants principaux et dispose d'un Mécanisme de Fusion qui effectue la fusion de modalités (figure 3.6). Le Noyau Fonctionnel représente le fonctionnement conceptuel du système tandis que l'Adaptateur de Noyau Fonctionnel est l'intermédiaire entre le Contrôleur de Dialogue et les concepts implémentés dans le Noyau Fonctionnel. Le Contrô-

leur de Dialogue prend en compte toute la gestion du dialogue et est responsable de la succession de tâches, contrôle l'enchaînement des états liés à l'interaction et réalise la dynamique de l'interface. Le Composant Interaction Logique est l'intermédiaire entre les événements logiques et les événements de niveau dispositif. Les événements logiques issues de Composant Interaction Logique vers le Mécanisme de Fusion sont des événements de niveau modalité. Finalement, le Composant Interaction Physique fournit l'interaction physique avec l'utilisateur et dépend de dispositifs physiques.

Les événements échangés pendant le test peuvent se situer à trois niveaux d'abstraction :

1. Au niveau du Composant Interaction Physique (événements de niveau dispositif, tels que les clics souris).
2. Au niveau Composant Interaction Logique (événements correspondant au langage des modalités).
3. Au niveau du Mécanisme de Fusion (événements correspondant aux commandes complètes indépendantes des modalités).

Le niveau d'abstraction retenu pour le test est logiquement celui du Composant Interaction Logique, seul qui permet de représenter les modalités.

En ce qui concerne la génération des données de test, nous avons montré qu'une utilisation judicieuse de profils opérationnels peut permettre de générer des événements dans la même fenêtre temporelle et de tester ainsi l'utilisation synergique de modalités (en cas de Redondance ou Complémentarité).

3.3 Perspectives

Qu'il s'agisse ou non d'application multimodales, les principales perspectives de ces travaux concernent la notation (ou langage) qui doit être adoptée pour la modélisation des tests. L'utilisation d'arbres de tâches semble une approche prometteuse [57].

La multimodalité n'étant pas prévue dans ce type de modèles, sa prise en compte dans la modélisation est un de nos sujets de réflexion actuels. Une piste intéressante semble être le couplage entre un modèle de tâches et des modèles spécifiques à la multimodalité, tels qu'ils ont été intégrés dans l'éditeur ICARE développée au sein de l'équipe IIHM [19].

Une autre question ouverte, spécifique à la validation d'applications multimodales, est celle de la prise en compte du temps. En effet, la multimodalité est liée à la notion de fenêtre temporelle, intervalle de temps pendant lequel l'occurrence de deux événements peut donner lieu à une fusion de modalités. Si

l'utilisation de profils opérationnels permet de produire des événements synergiques dans la même fenêtre, cela ne se fait que de manière approximative. Il est en particulier impossible, avec nos formalismes, de générer deux événements séparés d'une durée de temps prédéfinie. De même, la prise en compte dans un oracle de test de la vérification de la fusion nécessite la possibilité d'exprimer le temps.

Bibliographie

[1] Dittmar A. More precise descriptions of temporal relations within task models. In *DS-VIS*, pages 151–168, 2000.

[2] Shepherd A. Analysis and training in information technology tasks. In D. Diaper, Ed. Task Analysis for Human-Computer Interaction, 1989.

[3] Gregory D. Abowd, Joëlle Coutaz, and Laurence Nigay. Structuring the space of interactive system properties. In James A. Larson and Claus Unger, editors, *Engineering for Human-Computer Interaction*, volume A-18 of *IFIP Transactions*, pages 113–129. North-Holland, 1992.

[4] J.-R. Abrial. *The B-book : assigning programs to meanings.* Cambridge University Press, New York, NY, USA, 1996.

[5] Yamine Aït Ameur and Nadjet Kamel. A generic formal specification of fusion of modalities in a multimodal hci. In René Jacquart, editor, *IFIP Congress Topical Sessions*, pages 415–420. Kluwer, 2004.

[6] Philippe Palanque Amélie Schyn, David Navarre and Luciana Porcher Nedel. Description formelle d'une technique d'interaction multimodale dans une application de réalité virtuelle immersive. In *IHM*, Caen, France, November 2003.

[7] E.A. Ashcroft and W.W. Wadge. *LUCID, the data-flow programming language.* Academic Press, 1985.

[8] Sandrine Balbo. *Evaluation ergonomique des interfaces utilisateur : Un pas vers l'automatisation.* PhD thesis, Université Joseph Fourrier, Grenoble, France, Septembre 1994.

[9] Rémi Bastide, Philippe A. Palanque, Duc-Hoa Le, and Jaime Munoz. Integrating rendering specifications into a formalism for the design of interactive systems. In *DSV-IS*, pages 171–190, 1998.

[10] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, 1983.

[11] A. Benveniste and G. Berry. The Synchronous Approach to Reactive and Real-Time Systems. *Proceedings of the IEEE*, 79(9) :1270–1282, 1991.

[12] Albert Benveniste, Benoît Caillaud, and Paul Le Guernic. From synchrony to asynchrony. In *CONCUR*, pages 162–177, 1999.

[13] J-L. Bergerand. LUSTRE : Un langage déclaratif pour le temps réel. Technical report, INPG, Grenoble, France, 1986.

[14] G. Bernot, M-C. Gaudel, and B. Marre. Software testing based on formal specifications : a theory and a tool. *Software Engineering Journal*, 6 :387–405, 1991.

[15] A. Bertolino, E. Marchetti, and I. Parissis. Perspectives on data flow-based validation of web services compositions. In *Workshop on the Role of Software Architecture for Testing and Analysis*, Boston, USA, July 2007.

[16] B. Blanc, G. Durrieu, A. Lakehal, O. Laurent, B. Marre, I. Parissis, C. Seguin, and V. Wiels. Automated functional test case generation from data flow specifications using structural coverage criteria. In *3rd European Congress on Embedded Real Time Software (ERTS2006)*, Toulouse, France, January 2006.

[17] A. Bouajjani, J.C. Fernandez, and N. Halbwachs. Minimal model generation. In *Workshop on Computer-Aided Verification*, Rutgers (N.J.), 1990.

[18] J. Bouchet, L. Madani, L. Nigay, C. Oriat, and I. Parissis. Formal testing of multimodal interactive systems. In *DSV-IS Design, Specification and Validation of Interactive Systems*, Salamanca, Spain, March 2007.

[19] J. Bouchet and L. Nigay. Icare : a component-based approach for the design and development of multimodal interfaces. In *CHI Extended Abstracts*, pages 1325–1328, 2004.

[20] F. Boussinot and R. De Simone. The Esterel language. *Proceedings of the IEEE*, 79(9) :1293–1304, 1991.

[21] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre : A declarative language for programming synchronous systems. In *POPL*, pages 178–188, 1987.

[22] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Trans. Software Eng.*, 15(11) :1318–1332, 1989.

[23] Joëlle Coutaz, Laurence Nigay, Daniel Salber, Ann Blandford, Jon May, and Richard M. Young. Four easy pieces for assessing the usability of multimodal interaction : the care properties. In *INTERACT*, pages 115–120. Chapman & Hall, 1995.

[24] Bruno d'Ausbourg. Using model checking for the automatic validation of user interface systems. In *DSV-IS*, pages 242–260, 1998.

[25] Bruno d'Ausbourg. Contribution à la définition de systèmes de confiance et de systèmes utilisables. HDR, Avril 2001.

[26] L. du Bousquet. Test fonctionnel statistique de systèmes spécifiés en Lustre. Thèse, Université Joseph Fourier, Grenoble, France, septembre 1999.

[27] L. du Bousquet, F. Ouabdesselam, I. Parissis, J.-L. Richier, and N. Zuanon. Lutess : a testing environment for synchronous software. In *Tool Support for System Specification, Development and Verification*, pages 48–61. Advances in Computer Science. Springer Verlag, June 1998.

[28] L. du Bousquet and N. Zuanon. Feature interaction detection contest : Lutess testing tool. technical report PFL, IMAG - LSR, Grenoble, France, 1998.

[29] Lydie Du Bousquet. *Test fonctionnel statistique de systémes spécifié en Lustre.* PhD thesis, Grenoble, France, Septembre 1999.

[30] Nadjet Kamel et Yamine Ait-Ameur. Modèle formel général pour le traitement d'interactions multimodales. In *IHM*, pages 219–222, Namur, Belgique, 2004. ACM.

[31] Phyllis G. Frankl and Elaine J. Weyuker. An applicable family of data flow testing criteria. *IEEE Trans. Software Eng.*, 14(10) :1483–1498, 1988.

[32] A-C. Glory. Vérification de propriétés de programmes flots de données synchrones. Technical report, Université Joseph Fourier, Grenoble, France, 1989.

[33] J. Goodenough and S. Gerhart. Toward a Theory of Test Data Selection. *IEEE Transactions on Software Engineering*, pages 156–173, 1975.

[34] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 53–62, 1998.

[35] K-C. Griche and I. Parissis. Automatic control flow based generation of stubs for structural testing. In *IASTED Conf. on Software Engineering*, pages 339–344, Innsbruck, Austria, February 17-19, 2004, 2004.

[36] N. Griffeth, R. Blumenthal, J.-C. Gregoire, and O. Tadashi. Feature interaction detection contest. In *Feature Interactions in Telecomunications and Software Systems*, pages 327 – 359. IOS Press, September 1998.

[37] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9) :1305–1320, 1991.

[38] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Data Flow Programming Language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320, 1991.

[39] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language lustre. *IEEE Trans. Software Eng.*, 18(9) :785–793, 1992.

[40] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language lustre. *IEEE Trans. Software Eng.*, 18(9) :785–793, 1992.

[41] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23, pages 35–46, Atlanta, GA, June 1988.

[42] Melody Y. Ivory and Marti A Hearst. The state of the art in automating usability evaluation of user interfaces. *ACM Comput. Surv.*, 33(4) :470–516, 2001.

[43] Francis Jambon, Patrick Girard, and Yohann Boisdron. Dialogue validation from task analysis. In David J. Duke and Angel R. Puerta, editors, *DSV-IS*, pages 205–224. Springer, 1999.

[44] Claude Jard and Thierry Jéron. Tgv : theory, principles and algorithms. *STTT*, 7(4) :297–315, 2005.

[45] Robert Jasper, Mike Brennan, Keith Williamson, Bill Currier, and David Zimmerman. Test data generation and feasible path analysis. In *ISSTA '94 : Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 95–107, New York, NY, USA, 1994. ACM Press.

[46] Griche Karim-Cyril. *Génération automatique de bouchons pour le test structurel basée sur l'analyse du flot de contrôle*. PhD thesis, Université Joseph Fourier, Grenoble, France, july 2005.

[47] J.C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7) :385–394, 1979.

[48] S. Rao Kosaraju. Analysis of structured programs. *J. Comput. Syst. Sci.*, 9(3) :232–255, 1974.

[49] A. Lakehal and I. Parissis. Lustructu : A tool for the automatic coverage assessment of lustre programs. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, pages 301–310, 2005.

[50] A. Lakehal and I. Parissis. Structural test coverage criteria for lustre programs. In *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems : a satellite event of the ESEC/FSE'05*, pages 35–43, Lisbon, Portugal, September 2005.

[51] A. Lakehal and I. Parissis. Automated measure of structural coverage for lustre/scade programs : a case study. In *Workshop on Automated Software Testing (AST 2007) - satellite event of ICSE 2007*, Minnesota, USA, May 2007.

[52] Abdesselam Lakehal. *Critères de couverture structurelle pour les programmes Lustre*. PhD thesis, Université Joseph Fourier, Grenoble, France, Septembre 2006.

[53] Joëlle Coutaz Laurence Nigay, Francis Jambon. Formal specification of multimodality. in CHI'95 Workshop on Formal Specification of User Interfaces, May 1995.

[54] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming Real-Time Applications with SIGNAL. *Proceedings of the IEEE*, 79(9) :1321–1336, 1991.

[55] Long J.B. Lim, K.Y. and N. Silcock. Instanciation of task analysis in a structured method for user interface design. In *Proceeding's of the 11th Interdisciplinary Workshop on Informatics and Psychology : Task Analysis in Human-Computer Interaction*, Scharding, June 1992.

[56] L. Madani, C. Oriat, I. Parissis, J. Bouchet, and L. Nigay. Synchronous testing of multimodal systems : An operational profile-based approach. In *16th International Symposium on Software Reliability Engineering (ISSRE 2005)*, Chicago, Illinois, USA.

[57] L. Madani and I. Parissis. Vers la génération automatique de tests à partir dárbres de tâches. In *Atelier Approches Formelles dans l'Assistance au Développement de Logiciels*, Namur, Belgique, May 2007.

[58] Laya Madani. *Utilisation de la programmation synchrone pour la spécification et la validation des systèmes interactifs*. PhD thesis, Université Joseph Fourier, Grenoble, France, Octobre 2007.

[59] Bruno Marre and Agnès Arnould. Test sequences generation from lustre descriptions : Gatel. In *Automated Software Engineering*, pages 229–237, 2000.

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

[61] C. Mazuet. Stratégies de Test pour des Programmes Synchrones - Application au Langage Lustre. Thesis, Institut National Polytechnique de Toulouse, Toulouse, France, december 1994.

[62] C. Mazuet. Stratégies de Test pour des Programmes Synchrones - Application au Langage Lustre. Technical report, Institut National Polytechnique de Toulouse, Toulouse, France, 1994.

[63] Thomas J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4) :308–320, 1976.

[64] Giulio Mori, Fabio Paternò, and Carmen Santoro. Ctte : Support for developing and analyzing task models for interactive system design. *IEEE Trans. Software Eng.*, 28(8) :797–813, 2002.

[65] J. Musa. Operational Profiles in Software-Reliability Engineering. *IEEE Software*, pages 14–32, 1993.

[66] Laurence Nigay and Joëlle Coutaz. A design space for multimodal systems : concurrent processing and data fusion. In A. Henderson E. Hollnagel T. White S. Ashlung, K. Mullet, editor, *INTERCHI*, pages 172–178, Amsterdam, avril 1993. ACM New York.

[67] Laurence Nigay and Joëlle Coutaz. A generic platform for addressing the multimodal challenge. In *CHI*, pages 98–105. ACM Press, 1995.

[68] Laurence Nigay and Joëlle Coutaz. *Espaces conceptuels pour l'interaction multimédia et multimodale*, volume 15, N 9 of *TSI, numéro spécial Multimédia et Collecticiel*, pages 1195–1225. AFCET and Hermes, 1996.

[69] Manuel Núñez and David de Frutos-Escrig. Testing semantics for probabilistic lotos. In *FORTE, Proceedings of the IFIP TC6 Eighth International Conference on Formal Description Techniques, Montreal, Canada, October 1995*, pages 367–382, 1995.

[70] A. Jefferson Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *ICECCS*, pages 119–, 1999.

[71] F. Ouabdesselam and I. Parissis. Constructing Operational Profiles for Synchronous Critical Software. In *6th International Symposium on Software Reliability Engineering*, pages 286–293, Toulouse, France, 1995.

[72] I. Parissis and F. Ouabdesselam. Specification-based Testing of Synchronous Software. In *ACM SIGSOFT Fourth Symposium on the Foundations of Software Engineering*, pages 127–134, San Francisco, USA, 1996.

[73] I. Parissis and J. Vassy. Strategies for automated specification-based testing of synchronous software. In *16th International Conference on Automated Software Engineering*, San Diego, USA, November 2001. IEEE.

[74] I. Parissis and J. Vassy. Thoroughness of specification-based testing of synchronous programs. In *Proceedings of the 14th IEEE International Symposium on Software Reliability Engineering*, pages 191–202, November 2003.

[75] Ioannis Parissis. *Test de logiciels synchrones spécifiés en Lustre.* PhD thesis, Université Joseph Fourier, Grenoble, France, Septembre 1996.

[76] F. Paternò; and G. Faconti. On the use of lotos to describe graphical interaction. In *HCI'92 : Proceedings of the conference on People and computers VII*, pages 155–173, New York, NY, USA, 1993. Cambridge University Press.

[77] Fabio Paternò. A formal approach to the evaluation of interactive systems. *SIGCHI Bull.*, 26(2) :69–73, 1994.

[78] Conte E. Mancini C. Mori G. Paternò F., Ballardin G. Specification of notation of task modelling methodology. Technical report, CNUCE-C.N.R., Pisa, Italy, January 2000.

[79] D. Pilaud and N. Halbwachs. From a synchronous declarative language to a temporal logic dealing with multiform time. In *Symposium on Formal Techniques in Real Time and Fault Tolerant Systems*, Warwick, 1988. Springer Verlag.

[80] Daniel Pilaud and Nicolas Halbwachs. From a synchronous declarative language to a temporal logic dealing with multiform time. In Mathai Joseph, editor, *FTRTFT*, volume 331 of *Lecture Notes in Computer Science*, pages 99–110. Springer, 1988.

[81] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Software Eng.*, 11(4) :367–375, 1985.

[82] C. Ratel. Définition et réalisation d'un outil de vérification formelle de programmes Lustre : Le système Lesar. Technical report, Université Joseph Fourier, Grenoble, France, 1992.

[83] P. Raymond. Compilation efficace d'un langage déclaratif synchrone : le générateur de code LUSTRE-V3. Technical report, Institut National Polytechnique de Grenoble, Grenoble, France, 1991.

[84] Pascal Raymond, Xavier Nicollin, Nicolas Halbwachs, and Daniel Weber. Automatic testing of reactive systems. In *IEEE Real-Time Systems Symposium*, pages 200–209, 1998.

[85] Pierre Roché. *Modélisation et Vérification d'Interface Homme-Machine.* PhD thesis, ENSAE/CERT, Toulouse, France, 1998.

[86] D. Scapin and C. Pierret-Goldbreich. Towards a method for task description : Mad. In L. Berlinguet and D. Berthelette, editors, *Proceedings*

of Work with Display Units (WWU '89), pages 27–34. North-Holland : Elsevier Science, 1989.

[87] D.L. Scapin and C. Pierret-Golbreich. Towards a method for task description : Mad. In *Work with display units 89*, North-Holland, 1990. Elsevier Science.

[88] B. Seljimi and I. Parissis. Using clp to automatically generate test sequences for synchronous programs with numeric inputs and outputs. In *17th International Symposium on Software Reliability Engineering (ISSRE 2006)*, pages 105–116, Raleigh, North Carolina, USA, 2006.

[89] B. Seljimi and I. Parissis. Automatic Generation of Test Data Generators for Synchronous Programs : Lutess V2. In *Workshop on Domain-Specific Approaches to Software Test Automation (DoSTA) - Satellite workshop of ESEC/FSE 2007*, Dubrovnik, Croatia, 2007.

[90] B. Seljimi and I. Parissis. Test de logiciels synchrones avec Lutess : apports de la programmation par contraintes. In *Atelier Approches Formelles dans l'Assistance au Développement de Logiciels*, Namur, Belgique, May 2007.

[91] Irina Sessitskaia. *Apport des techniques d'abstraction pour la vérification des interfaces Homme-Machine*. PhD thesis, ONERA, Toulouse, France, 2002.

[92] Harold Thimbleby, Paul Cairns, and Matt Jones. Usability analysis with markov models. *ACM Trans. Comput.-Hum. Interact.*, 8(2) :99–132, 2001.

[93] Jérôme Vassy. *Génération automatique de cas de test guidée par les propriétés de sûreté*. PhD thesis, Université Joseph Fourier, Grenoble, France, october 2004.

[94] Sergiy A. Vilkomir and Jonathan P. Bowen. Formalization of software testing criteria using the z notation. In *COMPSAC*, pages 351–356, 2001.

[95] Sergiy A. Vilkomir and Jonathan P. Bowen. Reinforced condition/decision coverage (rc/dc) : A new criterion for software testing. In *ZB*, pages 291–308, 2002.

[96] Mark Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4) :352–357, 1984.

[97] E. Weyuker. The Evaluation of Program-based Software Test Data Adequacy Criteria. *Communications of the ACM*, pages 668–675, 1988.

[98] L. White and E. Cohen. A Domain Strategy for Computer Program Testing. *IEEE Transactions on Software Engineering*, pages 247–257, May 1980.

[99] J. Whittaker. Markov chain techniques for software testing and reliability analysis. Thesis, University of Tenessee, May 1992.

[100] M. Woodward, D. Hedley, and M. Hennell. Experience with path analysis and testing of programs. *IEEE Transactions on Software Engineering*, SE-6(3) :278–286, May 1980.

[101] Jean-Marc Mota Mickael Baron Yamine Ait-Ameur, Idir Ait-Sadoune. Validation et vérification formelles de systèmes interactifs multi-modaux fondées sur la preuve. In *IHM*, pages 123–130, Montreal, Canada, 2006. ACM.

[102] Mickael Baron Yamine Ait-Ameur. Bridging the gap between formal and experimental validation approaches in hci systems design : use of the event b proof technique. In *ISOLA*, 2004.

[103] Mickael Baron Yamine Ait-Ameur, Idir Ait-Sadoune. Eétude et comparaison de scénarios de développements formels d'interfaces multi-modales fondés sur la preuve et le raffinement. In *MOSIM - 6ème Conférence Francophone de Modélisation et Simulation. Modélisation, Optimisation et Simulation des Systèmes Défis et Opportunités*, Rabat, Maroc, Avril 2006.

[104] Nadjet Kamel Yamine Ait-Ameur, Mickael Baron. Encoding a process algebra using the event b method. application to the validation of user interfaces. In *ISOLA*, 2005.

[105] N. Zuanon. Test de spécifications de services de télécommunication. Thèse, Université Joseph Fourier, Grenoble, France, juin 2000.