UNE APPROCHE POUR LA GENERATION AUTOMATIQUE DE TESTS DE ROBUSTESSE

Cyril Pachon,
Laboratoire Vérimag, Centre Equation, 2, Avenue de Vignate 38610 Gières
Cyril.pachon@imag.fr

Résumé: La production et l'exécution de séquences de tests restent une approche incontournable pour la validation d'un logiciel. Pour cela, et afin de réduire les coûts, il est intéressant de générer automatiquement les séquences de tests. La plupart des techniques et outils existants sont axés sur le test de conformité d'un logiciel par rapport à sa spécification. Nous proposons une approche axée sur le test de robustesse. Elle a pour but de garantir un comportement acceptable du logiciel en dehors de ses conditions nominales (spécification) vis à vis d'une propriété donnée. C'est à dire sa capacité à résister à différents aléas provenant son de environnement ou de défaillances internes.

Mots Clés: Spécification mutée, Modèle de fautes, Test de robustesse, Vérification et validation de modèles

Summary: The production and the execution of tests suites are a crucial activity for software validation. For that, and to reduce the costs, it is interesting to produce automatically the tests suites. The particular problem of testing if an implementation is "correct" with respect to its specification is referred to as conformance testing. We propose here an alternative approach focussing on robustness testing. It consists in checking the ability of an implementation to satisfy some given properties in spite of internal failures or within a non nominal environment.

Key words: Mutated Specification, faults Model , Robustness testing, Checking and validation of models.

Une approche pour la génération automatique de tests de robustesse

1. INTRODUCTION

La complexité des systèmes réactifs [1] nécessite l'évolution et le développement des méthodes et des outils adéquats pour leur validation. Un dysfonctionnement de programmes critiques, même temporaire, peut avoir de lourdes conséquences économiques, voire attenter à la sécurité des individus. Leur validation avant leur mise en place est essentielle et doit tenter de garantir la correction de l'ensemble des comportements possibles du programme.

La production et l'exécution de séquences de tests restent une approche incontournable dans le processus de validation d'un logiciel [Tretmans 1992,Phalippou 1994]. Pour améliorer la pertinence de la phase de test, et réduire ainsi son coût, il est intéressant en pratique de pouvoir générer automatiquement les séquences de tests à partir d'une description initiale d'un programme (soit le programme source, soit une spécification de plus haut niveau). Divers critères sont alors envisageables pour sélectionner ces séquences de tests : syntaxiques (la couverture du flot de données, ou du flot de contrôle) ou sémantique (le test d'une fonctionnalité précise). Cette dernière approche a donné lieu à de nombreux outils comme Lurette [Raymond 1998], TestComposer de *Object*GEODE [Groz 1999] ,TGV (Test Generation using Verification technology) [Fernandez 1997], TorX [Berlinfante 1999], TVeda [Phalippou 1994bis], etc.

Les techniques existantes de génération automatique de tests sont essentiellement axées sur la *conformité* [Phalippou 1994bis , Tretmans 1992, Morel 2000] : le test a pour objectif de trouver des séquences d'exécution incorrectes par rapport à un comportement nominal [2] du programme. Toutefois, il est souvent nécessaire de considérer également le comportement du logiciel en dehors de ses conditions nominales. Nous parlons alors de la *robustesse* d'un système, c'est à dire sa capacité à résister à différents aléas dus aussi bien à son environnement d'exécution (conditions aux limites, malveillances, etc) qu'à des défaillances internes de certains de ses composants. Si le test de robustesse est assez bien étudié et mis en œuvre dans le domaine du matériel, très peu de travaux existent à notre connaissance dans le cas du logiciel [Arlat 2002, Koopman 1999]. Dans ce papier, nous proposons une approche pour la génération de tests de robustesse en nous inspirant des techniques issues du test de conformité.

Dans un premier temps, en section 2., nous présentons les différents modèles utilisés dans la suite. Puis, en section 3., nous proposons une définition formelle du test de robustesse. Nous décrivons ensuite, en section 4., comment s'effectue la génération automatique des cas de tests. Et enfin avant de conclure, nous présentons, en section teurs le verdict est NONROBUST.

5., un exemple.

2. MODELES

IOLTS : Systèmes de transitions étiquetées avec entrées/sorties : Un IOLTS (Input-Output Labelled Transition Systems) est un des modèles utilisés pour décrire le comportement d'un système réactif. C'est un raffinement des LTS (Labelled Transition Systems) dans lesquels les entrées et les

1Par systèmes réactifs, nous définissons tout composant logiciel en interaction permanente avec son environnement

2Nous entendons par nominal toutes les conditions idéales d'utilisation décrites par une spécification initiale

sorties sont distinguées. Cette distinction est nécessaire du fait de la nature asymétrique de l'activité du test. Ainsi, nous considérons un alphabet fini d'actions A, partitionné en deux ensembles: *les actions d'entrées* A_I et *les actions de sorties* A_O. Notons que les actions internes sont étiquetées par un symbole τ, ($\tau \in A$). τ représente des actions non observables par rapport à l'environnement du système (le testeur) tandis que les autres actions A_I et A_O sont visibles par rapport ce même environnement.

Définition 2.1 *Nous considérons un IOLTS* $M = (Q_M, A_M, T_M, q^{init}_M)$ *où* Q_M *est l'ensemble fini des états,* q^{init}_M *l'état initial,* A_M *l'ensemble fini des actions avec* $A_M \subseteq A$, *et* $T_M \subseteq Q_M \times A_M \cup \{\tau\} \times Q_M$ *la relation de transition.*

Notations : Pour un ensemble fini X, nous notons X^* (resp $X^\omega = X \to N$) l'ensemble des séquences finies (resp infinies) sur X. Pour un IOLTS M, si $\rho \in Q_M^\omega$, nous notons $inf(\rho) = \{q \mid \forall\, n\ \exists\, m\ ,\ m \geq n$ et $\rho(m) = q\}$, $inf(\rho)$ dénote l'ensemble des états de Q_M qui se produit infiniment souvent dans ρ. Soit $\sigma \in A_M^*$, $\alpha, \alpha_i \in A_M$, $p, q, p_i \in Q_M$. $p \xrightarrow{\tau^*\alpha}_M q$ si et seulement si $\exists\, p_0, p_1 \cdots p_n.\ P = p_0 \xrightarrow{\tau}_M p_1 \xrightarrow{\tau}_M \ldots p_n \xrightarrow{\tau}_M q$, $p \overset{\sigma}{\Rightarrow}_M q$ si et seulement si $\exists\, \alpha_0, \alpha_1 \cdots \alpha_n \in A_M,\ p_0, \cdots, p_n \in Q_M$, $\sigma = \alpha_1, \alpha_2 \ldots \alpha_{n-1}$, $p_0 = p$, $p_i \xrightarrow{\tau^*\alpha_i}_M p_{i+1}$, $i < n$ et $q_n = q$. $A(q) = \{\alpha \mid \exists\ q'$ *et* $q \xrightarrow{\tau^*\alpha}_M q'\}$ est l'ensemble des actions immédiates après q, $I(q) = A(q) \cap A_I$ est l'ensemble des entrées après q, et $O(q) = A(q) \cap A_O$ est l'ensemble des sorties après q. Le langage fini accepté par M, noté $L(M)$ est : $L(M) = \{\ \sigma \mid \exists q \in Q_M$ et $q_M^{init} \overset{\sigma}{\Rightarrow}_M q\ \}$. Notons par $p -/\!\!\rightarrow$, si une action a est non présente dans le domaine de sortie d'un état p.

Modèle de spécification : La description des comportements d'un système complexe directement sous la forme d'un IOLTS n'est pas très lisible. En pratique, nous utilisons plutôt des langages spécialisés comme SDL [ITU-T, 1992], Lotos [Logrippo, 1991, Salona, 1993], etc. Ces langages ont une sémantique opérationnelle en termes de IOLTS.

Modèle d'implémentation :
L'implémentation réelle sous test (implementation under test : *IUT*) est une boîte noire. Seules les interactions avec l'environnement sont observables. Le comportement de l'implémentation est supposé modélisé par un IOLTS où les entrées (resp. les sorties) sont les sorties de l'environnement (resp. les entrées).

Architecture de tests : L'architecture de tests détermine une seconde partition de l'ensemble d'action A en actions contrôlables A_c (délivrées par le testeur) et observables A_u (observées par le testeur): $A = A_u \cup A_c$.

Observabilité et Contrôlabilité : Un IOLTS est *déterministe* si et seulement si il existe une seule transition sortante étiquetée par une même action : $\forall \alpha.\ p \overset{\alpha}{\Rightarrow}_M p' \wedge p \overset{\alpha}{\Rightarrow}_M p'' \Rightarrow p' = p''$. Nous supposerons dans la suite que le IOLTS associé à la spécification est déterministe (il suffit de le déterminiser dans le cas contraire) et sans action interne.
Un IOLTS satisfait la condition de contrôlabilité si et seulement si pour chaque état, une sortie est possible, si et seulement si il y a exactement une transition sortante. Formellement, si $|X|$ dénote la cardinalité de l'ensemble X, alors : $\forall p.|I(p)|=0 \vee (|I(p)|=1 \wedge I(p)=A(p))$.

Blocage et Divergence: Un blocage (ou absence d'actions) n'est pas forcément une erreur. Il peut être de type deadlock (plus d'évolution du système), livelock (divergence

d'actions internes) ou exister dans la spécification initiale. Les blocages sont explicités par une action δ sur le IOLTS de la spécification. Pour ce faire, nous remplaçons chaque transition de blocage (p, $-/\not\delta$) ou (p,τ^*,p) par (p,δ,p) [Tretmans 1996].

Cas de tests : Le type de test auquel nous nous intéressons ici est de type boîte noire à travers des interfaces appelées *PCO* (point de contrôle et d'observation). Un cas de test *CT* est un IOLTS contrôlable et déterministe. Chaque trace est étiquetée par un verdict pour détecter, à l'exécution, la conformité ou la non-conformité de l'IUT vis-à-vis de la spécification [Tretmans, 1996].

3. TEST DE ROBUSTESSE

L'IEEE définit la robustesse comme "le degré selon lequel un système, ou un composant, peut fonctionner en présence d'entrées invalides ou de conditions environnementales stressantes". Nous adaptons ici cette définition et proposons la définition informelle suivante:

3.1 Présentation

Définition informelle 3.1
Un système/composant logiciel est dit robuste vis à vis d'une propriété si et seulement si celle-ci est préservée lors de toutes exécutions, y compris en dehors des conditions nominales. Plus précisément, le système garde un fonctionnement acceptable en présence d'entrées invalides, de dysfonctionnements internes, de conditions de stress, de pannes, etc.

A partir de cette définition informelle, nous proposons une définition du test de robustesse basée sur trois éléments:

- **S** est une spécification exprimant les comportements de l'IUT dans l'environnement nominal. S est décrit par un IOLTS déterministe.
- **MF** est un modèle de fautes décrivant l'ensemble des fautes et événements inopportuns pouvant survenir en entrée et en sortie de l'IUT pendant une exécution dans un environnement "réel" (non nominal). Cet ensemble peut inclure les pannes de l'IUT, les pertes lors de communication entre composants, les entrées non prévues et survenues de façon spontanées lors d'une exécution, etc. En particulier MF permet d'étendre ou de restreindre certains comportements décrits par S. Une représentation exacte de cet ensemble dépend du langage de spécification considéré. Il peut par exemple être aussi donné sous forme d'annotations ou de mutations de S. Le formalisme d'un tel modèle reste indépendant du logiciel. Il contient toutes les caractéristiques de l'environnement d'implémentation. Si le modèle de fautes est donné seules les références utiles au logiciel seront considérées. Si ce modèle est induit automatiquement à partir des références du logiciel il sera alors gardé dans sa totalité.
- ***P*** est une propriété dite de *robustesse*. Cette propriété doit être vérifiée par l'IUT lorsque celle-ci évolue dans un environnement "réel", spécifié par MF. Par exemple une propriété peut exprimer : " l'IUT peut toujours revenir dans un état de fonctionnement acceptable lorsqu'il y a eu une erreur ", ou "l'IUT peut toujours préserver un invariant", etc. Il existe de nombreux formalismes pour exprimer une propriété *P*, sur les séquences finies ou infinies. Ces formalismes se rangent dans deux catégories : les logiques temporelles et les automates. Nous supposons que la propriété *P* peut être par exemple exprimée en une logique temporelle linéaire.

Une différence essentielle avec le test de conformité est donc que nous cherchons à établir une relation entre l'IUT et une propriété externe P. La spécification S et le modèle de fautes MF ne servant ici que de guide pour la génération du test. Par ailleurs, le test généré peut avoir des comportements

qui n'apparaissent pas nécessairement dans la spécification.

Informellement, tester la robustesse de l'IUT consiste à exécuter des traces permettant de valider ou d'invalider la propriété P donnée. Ces traces d'exécution sont extraites de la spécification S, en tenant compte du modèle de fautes MF. Nous cherchons donc à extraire des séquences ne vérifiant pas P, et à tester leur exécutabilité sur l'IUT.

Finalement, nous supposons qu'il y a cohérence entre ces différents éléments et l'architecture de test. La propriété de robustesse doit être vérifiable par un testeur extérieur. La description de l'environnement par le modèle de fautes, doit demeurer "générable" pendant le temps d'exécution du test.

3.2 Propriété de robustesse

Pour trouver les séquences d'exécution d'une spécification en respectant la propriété *P*, il est souvent plus facile de construire un automate observateur reconnaissant les séquences de $\neg P$ (et donc de caractériser la non robustesse).
Pour envisager les cas où l'IUT ne respecte pas la spécification, l'observateur (*OBS*) doit être complet afin d'accepter les événements observables non spécifiés de l'IUT. Selon la propriété P considérée, la non robustesse peut être caractérisée soit par des séquences d'exécution finies soit par des séquences d'exécution infinies, par exemple:

- Montrer qu'une implémentation ne viole pas un invariant est exprimé par une séquence finie.
- Montrer qu'une implémentation ne retourne jamais dans une situation d'exécution normale après une situation d'erreur est exprimée par une séquence infinie.

Pour cette dernière description nous utilisons les automates de Rabin [Rabin 1969] reconnaissant des séquences infinies.

Définition 3.2 *Un automate de Rabin R_a est un couple (S,P) ou $S = (Q_s, A_s, T_s, q^{init}_s)$ est un IOLTS et $P=\langle (L_1,U_1), (L_2,U_2),\ldots,(L_k,U_k) \rangle$ est un ensemble de couples $(L_i,U_i) \subseteq \{Q_s \times Q_s\}$ pour $i \in \{1,2,\ldots,k\}$.*
Le langage accepté par R_a, noté $L(R_a)$ est :
$L(R_a) = \{ \sigma | \sigma \in A_s^{\omega} \; \exists k \; \exists \rho \in Q_s^{\omega} \;\; \forall i$ *et* $\rho_i \; \sigma_i, \rightarrow_s \rho_{i+1} \; \rho_0 = q^{init}_s$ **inf**$(\rho) \cap L_k \neq \varnothing$ *et* **inf**$(\rho) \cap U_k = \varnothing\}$.

Dans le cadre du test, il est nécessaire de se restreindre à des séquences finies. Pour cela, nous paramétrons la longueur des séquences d'exécution en associant à chaque ensemble L_i (resp U_i) un compteur cl_i (resp cu_i). Nous introduisons ainsi les automates de *Rabin Paramétrés bornés*.

Définition 3.3 Un automate de Rabin Paramétré borné par compteurs AR_a est un couple (R_a,C) où R_a est un automate de Rabin et $C = \{(cl_1,cu_1),\ldots,(cl_p,cu_p)\}$ avec $(cl_1,cu_1) \in N \times N$ un ensemble de couple de compteur. Une séquence d'exécution $\sigma \in A_s^*$ est acceptée par AR_a si et seulement si $\exists k' | \sigma \cap [L]_{k'}| \geq cl_{k'}$ et $|\sigma \cap U_{k'}| \leq cu_{k'}$.
Nous donnons par exemple la négation d'une propriété informelle "Le système peut toujours revenir dans un mode nominal après être entré dans un mode dégradé" par l'automate de Rabin de la figure 1:

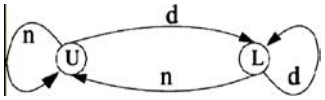

FIG. 1: Automate de Rabin d'une séquence infinie

L'action "d" (respectivement "n") représente l'accès au mode dégradé (respectivement l'accès au mode nominal). L'ensemble des séquences non robustes est exprimé par $(n+d)^*d^{\omega}$. Cet ensemble correspond aux séquences reconnues par l'automate (figure

1). Pour respecter la définition, il faut enfin associer aux états de L et de U les compteurs respectifs avec cu < cl.
Nous décrivons une propriété de sécurité "Toutes actions doivent être différentes de l'action d" par un automate de Rabin Paramétré borné et paramétré par compteur. Le fait de rester sur un état sans action possible est un blocage prévu par la spécification (noté par le symbole δ sur la figure 2). Nous complétons notre observateur par la possibilité d'accepter toutes les actions de l'IUT non prévues par la spécification (noté par le symbole * sur la figure 2). Cette propriété s'écrit enfin:

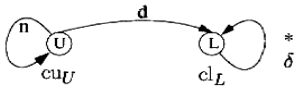

FIG. 2: Automate de Rabin Paramétré borné d'une propriété de sécurité

3.3 Test de robustesse

Dans notre approche le critère de robustesse est exprimé par une propriété *P* : une implémentation à tester (IUT) est considérée robuste si et seulement si elle satisfait P en présence de fautes suivant un comportement de la spécification. $\neg P$ est donc décrit comme un observateur (OBS).
Formellement, un cas de test de robustesse est un IOLTS CT, contrôlable, déterministe. Le verdict associé à une séquence σ (trace de CT) est :

- NONROBUST si σ est une séquence de l'observateur (OBS) exécutable sur l'IUT : $\sigma \in L(IUT) \cap L(OBS) = NONROBUST$.
- ROBUST si σ est une séquence guidée par la spécification mutée mais non inclus par l'observateur : $\sigma \in L(IUT) \cap L(S) - L(OBS) = ROBUST$.

4. GENERATION DE CAS DE TESTS

4.1 Principe

La technique de génération de cas de tests se décrit de la façon suivante: Premièrement, nous calculons le comportement de la spécification S, enrichie du modèle de fautes MF. Cette technique est intéressante dans le cas où l'IUT ne satisfait pas la propriété. Si le comportement enrichi contient des séquences d'exécution incorrectes, alors ces séquences sont candidates pour devenir des cas de tests. Pour effectuer cette génération, nous proposons l'architecture suivante, figure 3:

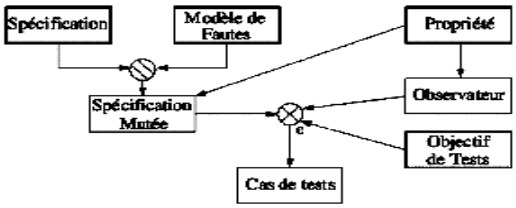

FIG. 3: Architecture de la génération de cas de test

Génération d'une spécification mutée S_m constituée par composition entre la spécification nominale S et le modèle de fautes MF.

Génération d'un observateur OBS, exprimant la propriété de non robustesse $\neg P$. L'objectif de l'observateur a deux rôles, d'abord de vérifier la non robustesse de S_m puis de marquer les séquences d'exécution incorrectes respectant $\neg P$.

Synthèse des cas de tests (à partir du produit synchrone entre S_m et OBS): Les séquences d'exécution incorrectes sont extraites de S_m. Elles deviennent des cas de tests exécutables produisant un verdict de robustesse.

Le processus ci-dessus est entièrement automatisable. La spécification mutée est obtenue par transformation syntaxique. La génération de cas de tests à partir de la spécification mutée, de l'observateur et éventuellement de l'objectif de tests est basée sur l'utilisation d'algorithme en profondeur d'abord comme dans l'outil TGV.

4.2 Composition entre Modèle de fautes et spécification

Les annotations du modèle de fautes vont pouvoir faire muter le code de la spécification. Le résultat est un IOLTS (S_m) augmenté ou diminué en terme de comportements par rapport à la spécification nominale.

4.3 Produit synchrone étendu

L'objectif de ce produit synchrone est de marquer les séquences d'exécution de S. Cependant, le produit a besoin d'être étendu. D'une part, pour ne pas bloquer les exécutions de l'IUT, tant qu'il y a vérification de la robustesse. Et d'autre part, pour prendre en compte les séquences d'exécution non définies de l'IUT par la spécification mutée. En d'autre termes, le résultat de ce produit définira un observateur pour l'IUT. Le produit synchrone étendu est défini de la manière suivante: $S_{\otimes e} = S_{Sm} \otimes_e S_{obs} = (Q_{\otimes e}, A_{\otimes e}, R_{\otimes e}, q^0_{\otimes e})$ où $q^0_{\otimes e} = (q^0_{Sm}, q^0_{obs})$ est l'état initial, $Q_{\otimes e}$ est le sous ensemble de Q_{Sm} x Q_{obs} des états accessibles depuis $q^0_{\otimes e}$ par la relation de transition $R_{\otimes e}$ définie comme suit:

$(p_{Sm}, p_{obs}) \xrightarrow{a} (q_{Sm}, q_{obs})$ si et seulement si une des trois conditions suivantes est vérifiée :

$$p_{Sm} \xrightarrow{a} q_{Sm} \text{ et } p_{obs} \xrightarrow{a} q_{obs} \quad (1)$$

$$p_{Sm} \xrightarrow{a} q_{Sm} \text{ et } p_{obs} \xrightarrow{-/a} \text{ et } a \in A_I \quad (2)$$

$$p_{Sm} \xrightarrow{-/a} \text{ et } p_{obs} \xrightarrow{a} q_{obs} \text{ et } a \in A_o \quad (3)$$

4.4 Construction d'un graphe de test à partir du produit synchrone

Principe : Pour calculer les séquences d'exécution vérifiant *P*, nous effectuons une recherche des composantes fortement connexes sur le produit synchrone (CFC algorithme [Tarjan 1972]). Nous enrichissons cette recherche par l'étiquetage des racines des CFC trouvées. Ce marquage distingue les composantes contenant des états de U et/ou de L ou aucun des deux. Ce marquage se fait de la manière suivant : La CFC est marquée L si elle comporte au moins un état de L. Elle est marquée U si elle ne contient pas d'état de L et contient au moins un état de U. Elle est marquée rien s'il n'y a ni état de L ni état de U.

Nettoyage du graphe : Seules les composantes terminales menant à des CFC terminales étiquetées L sont acceptées par l'observateur. Notre algorithme supprime itérativement toutes les feuilles non étiquetées L ainsi que les transitions entrantes dans ces feuilles jusqu'à n'avoir que des séquences qui vérifie $\neg P$.

4.5 Sélection des cas de tests et verdicts

La sélection est faite parmi les séquences restantes. Dès l'extraction d'une séquence, les verdicts sont mis en fonction des états de L et U. A l'exécution les compteurs vérifient des valeurs fixées aux départs. Le verdict est donc ROBUST si un compteur de U ne dépasse pas une valeur alors que celui de L dépasse une valeur. S'il y a non respect de ces compteurs le verdict est NONROBUST.

5. EXEMPLES

Présentation : L'exemple présenté ici est une modélisation d'ouverture de porte. L'automate est réduit à ouvrir puis refermer la porte dès réception d'un ordre d'ouverture. L'IOLTS de la spécification comporte trois états et quatre actions (réception d'un ordre *REQOPEN ou OTHER* émission de commande *OPEN ou CLOSE*). Le modèle de fautes est réduit à la perte éventuelle des messages *other* transmis et à la possibilité de recevoir plusieurs fois le message ReqOpen. La propriété *P* est : Après toute ouverture, il y a une fermeture. La propriété $\neg P$ est décrite par un automate de Rabin Paramétré réduit à deux états (un état de L et un d'état U), les

compteurs associés aux états respectant $cu_1 << cl_2$.

Mutation de la spécification : Les annotations du modèles de fautes vont ici supprimer certains comportements de la spécification nominale. En effet, là où nous pensions que le message other arriverait toujours, il n'est plus retenu. La mutation a eu comme effet de supprimer cette action sur la spécification. De plus nous rajoutons la possibilité d'avoir un message ReqOpen sans produire le message Open. Concrètement la mutation augmente le comportement de la spécification initiale.

Produit synchrone : Le produit donne tous les comportements possibles.

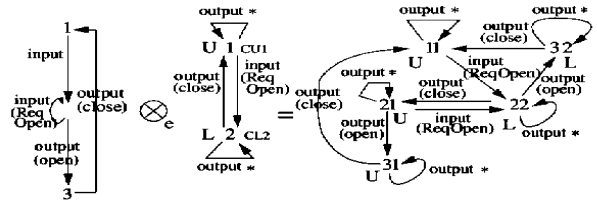

FIG. 4: produit synchrone étendu

Graphe et nettoyage du graphe : Du produit, nous appliquons l'algorithme de recherche des composantes fortement connexes, nous marquons les racines puis nous supprimons toutes parties ne concernant pas la propriété par l'algorithme de nettoyage. Sur cet exemple toutes les séquences sont conservées car le graphe est fortement connexe et comporte des états de L.

Sélection des séquences de tests : Du graphe restant, les séquences sélectionnées sont celles comportant des états de L. Un verdict est associé à chaque état. Par exemple une séquence possible est en débutant de 11 : $(\text{output}^\omega \wedge (\text{input ReqOpen})) \wedge (\text{output}^\omega \vee ((\text{output close}) \wedge (\text{output}^\omega \vee (\text{output open})) \wedge (\text{output}^\omega \vee (\text{output close})))$

6. CONCLUSION

Nous avons proposé une approche formelle pour générer automatiquement des tests de robustesse. Elle est basée sur les notions de mutation de code pour exprimer la dégradation (par modèle de fautes) que peut subir une implémentation dans un environnement non nominal. Un test est caractérisé par une propriété de robustesse qui doit être préservée afin de garantir un comportement acceptable lors de toutes exécutions de l'implémentation. Enfin, un prototype est en cours de réalisation.

Remerciement : Je tiens à remercier toutes les personnes des laboratoires IRISA, LAAS, LaBri, LRI et Vérimag participant du groupe de travail de l'Action Spécifique sur le test de robustesse.

Bibliographie

[Arlat, 2002]Arlat J., Fabre J.-C., Rodriguez M. et Salles F.. – Dependability of COTS microkernel-based systems. *IEEE Trans. on Computers*, vol. 51 (2002)

[Belinfante, 1999]Belinfante A., Feenstra J., de Vries R., Tretmans J., Goga N., Feijs L., Mauw S. et Heerink L.. – Formal Test Automation : a Simple Experiment. *In : 12th International Workshop on Testing of Communicating Systems*. – G. Csopaki et S. Dibuz et K. Tarnay (1999).

[Fernandez, 1997]Fernandez J.-C., Jard C., Jéron T. et Viho C. – An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology. *Science of Computer Programming*, vol. 29, pp. 123–146 (1997).

[Groz, 1999]Groz R., Jéron T. et Kerbrat A. – Automated test generation from SDL specifications. *In : SDL'99 The Next Millenium, 9th SDL Forum, Montréal, Québec*, éd. par Dssouli R., von Bochmann G. et Lahav Y., pp. 135–152. – Elsevier (1999).

[ITU-T, 1992] ITU-T. – ITU-T: Specification and Description Language (SDL), ITU-T Recommendation Z.100, International Telecommunication Union, Geneva (1992).

[Koopman, 1999]Koopman P. et DeVale J.. – Comparing the robustness of posix operating systems. *In : 29th International Symposium on Fault-Tolerant Computing Systems*. – Madison (WI) (1999).

[Logrippo, 1991]Logrippo L., Faci M. et Haj-Hussein M. – An introduction to LOTOS: learning by examples. *Computer Networks and ISDN Systems*, vol. 23, n-.25em.2ex_ 5, pp. 325–342 (1991).

[Morel, 2000]Morel P. – *Une algorithmique efficace pour la génération automatique de tests de conformité*. – Thèse de PhD, UFR IFSIC/ lab IRISA (2000).

[Phalippou, 1994]Phalippou M. – *Relations d'Implantation et Hypothèses de Test sur les Automates à Entrées et Sorties*. – Thèse de PhD, Bordeaux, (1994).

[Phalippou, 1994bis]Phalippou (M.). – Test Sequence Generation using Estelle or SDL Structure Information. *In : Proceedings of FORTE/PSTV(Berne ,Switzerland)*, éd. par et S. Leue (D. H.), pp. 405–420 (1994).

[Rabin, 1969]Rabin M. – Decidability of second-order theories and automata on infinite trees. *Trans. of Amer. Math. Soc.*, vol. 141, pp. 1–35 (1969).

[Raymond, 1998]Raymond P., Weber D), Nicollin X. et Halbwachs N. – Automatic testing of reactive systems. *In : 19th IEEE Real-Time Systems Symposium*. – Madrid, Spain (1998).

[Salona, 1993]Salona A., Vives J. et Gomez S. – An introduction to LOTOS (1993).

[Tarjan]Tarjan R. – Depth first search and linear graph algorithms. *SIAM Journal on computing*, pp. 146–160, (1972).

[Tretmans, 1992]Tretmans J. – *A Formal Approach to Conformance Testing*. – Thèse de PhD, Univ of Twente, Enschede, Th Netherlands (1992).

[Tretmans, 1996]Tretmans J. – Test generation with inputs, outputs, and repetitive quiescence. *Software–Concepts and Tools*, vol. 17, pp. 103–120 (1996).