Génération automatique de tests à partir de patrons de propriétés

Frédéric Dadeau

Laboratoire d'Informatique de l'université de Franche-Comté
INRIA projet CASSIS
16, route de Gray – F-25030 Besançon cedex
email : frederic.dadeau@lifc.univ-fcomte.fr

RÉSUMÉ. *Cet article propose une technique originale de génération de tests, à partir d'un modèle formel d'une application, écrit sous la forme d'une machine abstraite B, et d'une propriété définie suivant certains patrons. Les patrons sont des structures paramétrées génériques permettant d'exprimer une propriété formelle se basant sur des prédicats d'états et des événements du système. La technique présentée se base sur la production automatique de scénarios de tests qui sont issus de la propriété en elle-même et d'un besoin de test, ce dernier décrivant informellement une intention de test d'un ingénieur validation. En fonction du patron de propriété considérée et de son instanciation concrète, un ou plusieurs besoins de tests peuvent s'appliquer. Les scénarios de tests produits sont exprimés sous la forme d'expressions régulières décrivant des enchaînements d'opérations amenant à des états pertinents du système. Un mécanisme d'animation symbolique du modèle est utilisé pour déplier les scénarios et instancier les tests, notamment les paramètres des opérations, jusqu'ici abstraits. Ceci permet de produire des cas de tests abstraits prêts à être concrétisés pour le système sous test. Nous présentons l'application de ces principes à travers une étude de cas issue du milieu industriel.*

MOTS-CLÉS : *Patrons de propriétés, scénarios, animation symbolique, stratégies*

1. Introduction et motivations

La génération de tests à partir de modèles fonctionnels [BEI 95], ou test boîte noire, est un concept consistant à utiliser un modèle formel, décrivant le comportement du système, à la fois pour générer les cas de test et pour calculer l'oracle, permettant d'établir le verdict du test dans l'objectif d'indiquer si le système est, ou non, conforme au modèle considéré. Le calcul des cas de tests se fait suivant diverses techniques visant à couvrir le modèle considéré, en fonction de son type. Par exemple, un modèle présenté sous la forme d'une machine à états finis étendue (Extended Finite State Machine – EFSM) se voit traditionnellement appliqué différents algorithmes de parcours des transitions du modèle dans l'objectif de produire des enchaînements d'opérations. C'est le cas de l'outil SpecExplorer [CAM 05]. Dans le cas de modèles comportementaux du système, où les opérations sont décrites sous la forme de substitutions, comme en B [ABR 96], ou par l'intermédiaire de modèles au format pré/postconditions, comme en UML/OCL, des techniques de couverture structurelle du modèle peuvent être appliquées. C'est le cas de l'outil Leirios Test Generator (LTG) [JAF 07] qui calcule des séquences de tests à partir de machines abstraites B.

Dans ce dernier cas, les cibles de tests sont dépendantes de l'information contenue dans les opérations, et se limitent à décrire des ensembles d'états (en l'occurrence pour LTG, les états satisfaisant l'activation des opérations considérées). Ceci diminue la pertinence des cas de tests, et peut amener à éviter des cas potentiellements intéressants, liés soit à l'expérience de l'ingénieur validation, ou à une propriété spécifique du système que l'on souhaite chercher à illustrer ou à mettre en défaut sur le système. Pour pallier ce problème, des solutions à base de scénarios sont considérées. Un exemple est l'outil de test combinatoire TOBIAS [LED 04] qui déplie des ex-

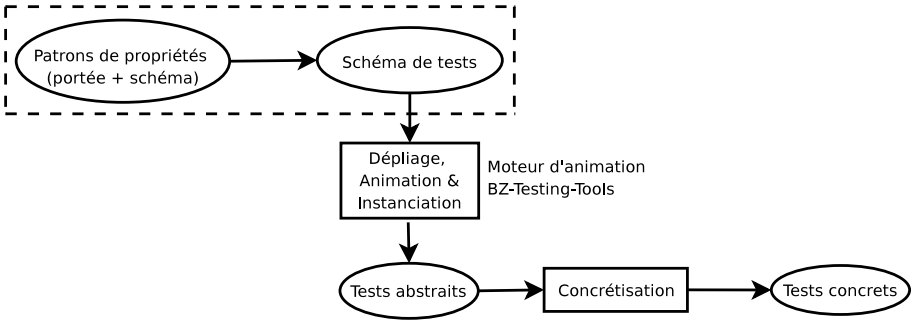

Figure 1. *Démarche de notre approche*

pressions régulières décrivant des combinaisons d'opérations avec leurs paramètres. Reprenant ce principe, nous avons proposé dans [JUL 08, DAD 08] une technique similaire basée sur l'animation symbolique d'un modèle formel écrit sous la forme d'une machine abstraite B.

Malheureusement, même si l'écriture des scénarios est simplifiée grâce à l'assistance fournie par un langage dédié, elle reste manuelle et c'est donc à l'ingénieur validation de concevoir ses scénarios. Or, il apparaît que les scénarios proposés sont souvent supposés exercer une propriété bien spécifique du système et qu'il serait donc possible de les produire automatiquement à partir d'un langage de propriétés adéquat. Le choix de ce langage nous est suggéré par Dwyer dans [DWY 99] qui présente une étude selon laquelle 92% des propriétés exprimées sur un système peuvent être catégorisées suivant un nombre fini et relativement restreint de *patrons*. Ces patrons sont définis dans l'article comme étant la combinaison d'une *portée*, décrivant sur quelle partie de l'exécution du système la propriété s'applique, et d'un *schéma* décrivant la propriété en elle-même.

Les travaux présentés dans cet article s'intéressent donc de faire le lien entre les patrons de propriétés proposés par Dwyer et les scénarios de tests. Cette approche est une extension des travaux précedemment présentés dans [BOU 06] où nous avions utilisé des propriétés de sûreté décrites en Java Temporal Pattern Language [TRE 02] (inspiré des travaux de Dwyer) pour produire des cas de tests par calcul automatique de séquences de test. Nous proposons ici de généraliser l'approche précédente en produisant des tests permettant d'illustrer les cas nominaux de la propriété, ainsi que des tests permettant d'éprouver la robustesse du système. La démarche globale est présentée en Figure 1. Partant de patrons de propriétés (composés chacun d'une portée et d'un schéma de propriété, les scénarios obtenus sont ensuite dépliés et joués sur un moteur d'animation symbolique (ici le moteur de l'outil BZ-Testing-Tools [BOU 04]) qui produit les tests abstraits instanciés. Ceux-ci seront ensuite traduits, lors de la phase dite de concrétisation, dans le langage cible du système sous test considéré.

Cet article s'articule de la manière suivante. Les parties 2 et 3 présentent les deux points de départ de ces travaux, respectivement le test à partir de scénarios et les patrons de propriétés. Nous présentons ensuite en partie 4 la contribution principale de ce papier, à savoir les stratégies de génération de scénarios à partir de propriétés. Une expérimentation est décrite dans la partie 5. Pour finir, la partie 6 présente et compare notre approche avec les travaux similaires, avant de conclure et de donner quelques perspectives dans la partie 7.

2. Principes du Scenario-Based Testing

Le test à partir de scénarios ou *Scenario-Based Testing* est un concept suivant lequel l'ingénieur validation décrit lui-même des scénarios d'utilisation du système, définissant ainsi ses cas de tests.

Dans le cadre du test de systèmes informatiques, il s'agit de décrire un enchaînement d'actions qui exercent une fonctionnalité particulière du système.

L'outil TOBIAS [LED 04] permet de décrire des scénarios de tests à travers des *schémas*, exprimés à l'aide d'expressions régulières décrivant des enchaînements d'appels d'opérations et la combinaison de leurs paramètres. Améliorant ce principe, nous avons proposé de nous appuyer sur l'animation symbolique de modèles formels du système pour nous affranchir de demander à l'ingénieur validation de fournir les paramètres des opérations. Ceci permet de se focaliser uniquement sur la description d'enchaînements d'opérations ponctuées éventuellement de points de passages, des états intermédiaires, qui guident les étapes d'un scénario. Un moteur de résolution de contraintes est ensuite en charge de calculer la faisabilité de la séquence d'opérations ainsi obtenue et d'instancier les paramètres des opérations en conséquence.

L'un des avantages du test à partir de scénarios que nous proposons est qu'il permet de faciliter la production de cas de tests en considérant des valeurs symboliques. Par ailleurs, l'utilisateur peut forcer l'animation à passer par certains états, définis par un prédicat, qui peut permettre de contraindre des valeurs de variables d'états. Un autre avantage est qu'il permet de faciliter la traçabilité des tests obtenus, en considérant que chaque scénario répond à une exigence précise. Nous souhaitons faciliter la conception de scénarios en permettant à l'ingénieur validation d'exprimer des propriétés de plus haut niveau qui permettront ensuite de générer les scénarios.

Nous décrivons dans cette partie les notions utiles à la compréhension du reste de l'article.

2.1. *Définition de la notion de "comportements"*

Le moteur d'animation Prolog de BZ-Testing-Tools [BOU 04] que nous utilisons s'appuie sur une décomposition des opérations des machines abstraites B en *comportements*. Chaque comportement se définit comme un prédicat et une substitution donnant l'évolution des variables d'états et l'instanciation des paramètres de retour de l'opération. Ces comportements sont calculés à partir des chemins du graphe de flot de contrôle de l'opération B considérée.

Exemple 1 (Extraction de comportements) Considérons l'opération B suivante, chargée de l'authentification du porteur d'une carte de paiement via la vérification d'un code PIN. Cette opération prend en paramètre un code qui est comparé au code assigné à la carte. Si le code proposé est incorrect, le nombre d'essais est décrémenté. S'il atteint 0, la carte est bloquée.

```
sw ← checkPin(p) ≙
   IF p ∈ 0..9999 ∧ pin ≠ -1 THEN
      IF p = pin THEN isHoldAuth := true || tries := 3 || sw := ok
      ELSE isHoldAuth := false || tries := tries - 1 ||
         IF tries = 1 THEN mode := blocked || sw := card_blocked
         ELSE sw := wrong_pin
         END
      END
   ELSE sw := wrong_conditions
   END
```

Cette opération contient quatre comportements, qui sont les suivants (où $\Longrightarrow$ est utilisé pour séparer la condition d'activation de la substitution) :

cpt_1 : p ∈ 0..9999 ∧ pin ≠ -1 ∧ p = pin $\Longrightarrow$ isHoldAuth := true ∧ sw := ok

cpt_2 : p $\in$ 0..9999 $\wedge$ pin $\neq$ -1 $\wedge$ p $\neq$ pin $\wedge$ tries = 1 $\Longrightarrow$ mode := blocked $\|$ sw := card_blocked

cpt_3 : p $\in$ 0..9999 $\wedge$ pin $\neq$ -1 $\wedge$ p $\neq$ pin $\wedge$ tries > 1 $\Longrightarrow$ sw := wrong_pin

cpt_4 : (p $\notin$ 0..9999 $\vee$ pin = -1) $\Longrightarrow$ sw := wrong_conditions

2.2. *Langage de scénarios*

Nous présentons ici le langage de scénarios que nous utilisons pour décrire les scénarios, introduit dans [JUL 08]. Celui-ci se présente sous la forme d'expressions régulières qui sont ensuite dépliées et jouées par un animateur symbolique. Le langage de description de scénarios se présente en 3 couches :

1) la couche *séquence*, qui se base sur des expressions régulières et permet de définir les scénarios de tests sous la forme de séquences d'opérations (répétées ou alternées) qui peuvent éventuellement mener à des états ;

2) la couche *modèle*, qui décrit les appels d'opérations au niveau du modèle et constitue l'interface entre le modèle et les scénarios ;

3) la couche *directive*, qui permet de piloter finement les tests qui seront générés, en spécifiant des critères de couverture de comportements qui seront utilisés par le moteur de génération des tests.

La version du langage présentée ici diffère de celle donnée précédemment, introduisant de nouvelles de pilotage au niveau de la couche directive.

2.2.1. *Syntaxe des couches séquence et modèle*

Pour la couche séquence, décrite en Figure 2, la règle SEQ décrit une séquence d'appels d'opérations comme une expression régulière. Un pas dans la séquence est soit un appel d'opération, dénoté par OP1 (décrit en Figure 3, ou un appel d'opération amenant à l'état satisfaisant un prédicat d'état, dénoté par SEQ $\rightsquigarrow$(SP). Ce dernier élément représente une amélioration par rapport aux langages de description de scénarios usuels puisqu'il permet de définir la cible d'une séquence d'opérations sans nécessairement avoir à énumérer les opérations composant la séquence. Les séquences peuvent être composées de la concaténation de deux séquences ("."), la répétition d'une séquence (REPEAT), ou le choix entre deux séquences (CHOICE). En pratique, nous utilisons des opérateurs de répétition bornés : 0 ou 1 fois, exactement n fois, entre n et m fois (n éventuellement égal à 0).

Pour la couche modèle, également décrite en Figure 2, la règle SP décrit un prédicat d'état, tandis que OP est utilisé pour décrire les appels d'opération qui peuvent être (i) un nom d'opération, (ii) le mot-clé $OP qui signifie "n'importe quelle opération", ou (iii) $OP\{OPLIST} signifiant "n'importe quelle opération exceptée celles de OPLIST".

SEQ	: :=	OP1 | "(" SEQ ")"
	|	SEQ "." SEQ
	|	SEQ REPEAT ALL_or_ONE
	|	SEQ CHOICE SEQ
	|	SEQ " $\rightsquigarrow$ (" SP ")"
REPEAT	: :=	"?" | $\underline{n}$ | $\underline{n}$..$\underline{m}$

OP	: :=	operation_name
	|	"$OP"
	|	"$OP \ {" OPLIST "}"
OPLIST	: :=	operation_name
	|	operation_name "," OPLIST
SP	: :=	state_predicate

Figure 2. *Règles syntaxiques pour les couches séquence (à gauche) et modèle (à droite)*

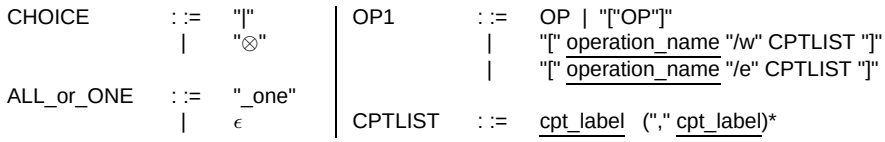

Figure 3. *Règles syntaxiques de la couche directive de génération de tests*

2.2.2. *Syntaxe couche directive de génération de tests.*

Elle permet de specifier le pilotage pour l'étape de génération de tests. Nous proposons trois genres de directives qui visent à réduire le dépliage de l'arbre d'exécution liée à l'expression régulière et/ou la recherche des instanciations d'un scénario de test. Cette partie du langage est donnée en Figure 3.

La règle CHOICE introduit deux opérateurs dénotés $|$ et $\otimes$, pour la couverture des branches d'un choix. Par exemple, si S_1 et S_2 sont deux séquences, $\mathtt{S_1} \mid \mathtt{S_2}$ spécifie que le générateur de tests doit produire des tests couvrant la séquence S_1 et d'autres tests couvrant la séquence S_2, tandis que $\mathtt{S_1} \otimes \mathtt{S_2}$ spécifie que le générateur de tests doit produire des tests couvrant soit la séquence S_1 soit la séquence S_2.

La règle ALL_or_ONE permet de préciser si toutes les solutions de la répétition de séquence seront sélectionnées (ϵ – option par défaut) ou si une seule le sera (_one).

La règle OP1 indique au générateur de tests qu'il doit couvrir un des comportements de l'opération OP. C'est l'option par défaut. L'ingénieur validation peut aussi demander la couverture de tous les comportements de l'opération en l'entourant de crochets. Deux variantes plus fines permettent de sélectionner les comportements qui pourront être appliqués, soit en précisant la liste des comportements autorisés (/w) soit celle des comportements refusés (/e). Les comportements ainsi explicités sont donnés sous la forme de libellés qui doivent être mis en correspondance avec des comportements issus des opérations du système. A titre d'illustration, par la suite, nous utiliserons la construction */w {prog}* qui aura pour objectif de ne sélectionner que les comportements libellés par *prog*, c'est-à-dire les comportements qui font progresser le système en modifiant les variables d'état.

Exemple 2 (Exemple de scénario) Reprenons l'opération `checkPin` de l'exemple précédent. Un schéma exprimant l'appel à cette opération jusqu'au blocage de la carte, et ce, quelque soit le nombre d'essais encore autorisé, s'exprime par (`checkPin`$^{0..3}$ _one) $\rightsquigarrow$ (`mode=blocked`).

2.3. *Dépliage et instanciation des scénarios*

Les scénarios sont dépliés de manière à obtenir l'ensemble des enchaînements d'opérations décrits. En pratique, chaque scénario est traduit en un fichier Prolog, directement interprété par le moteur d'animation de BZ-Testing-Tools. Chaque solution donne un cas de test instancié. Le mécanisme interne de backtracking de Prolog est utilisé pour itérer sur les différentes solutions. L'ensemble final de test est obtenu une fois toutes les solutions épuisées. Le mécanisme d'instanciation utilisé durant ce processus vise à calculer les valeurs des paramètres des opérations employées dans le cas de test, de manière à ce que la séquence soit "faisable" (cf. notion de *feasibility* en B [ABR 96]), notamment vis-à-vis des valeurs attendues pour les variables données dans les prédicats d'états intermédiaires.

Exemple 3 (Dépliage et instanciation de scénarios) Le scénario précédent (`checkPin`$^{0..3}$ $\_one$) $\rightsquigarrow$ (`mode=blocked`) produira par dépliage combinatoire 4 séquences :

(i) $\epsilon \rightsquigarrow$ (`mode=blocked`)

(ii) `checkPin`(X_1) $\rightsquigarrow$ (`mode=blocked`)

(iii) `checkPin`(X_1) . `checkPin`(X_2) $\rightsquigarrow$ (`mode=blocked`)

(iv) `checkPin`(X_1) . `checkPin`(X_2) . `checkPin`(X_3) $\rightsquigarrow$ (`mode=blocked`)

où ϵ désigne la séquence vide et X_1, X_2, X_3 sont des variables qui devront être instanciées par la suite. Supposons que l'état courant du système nous donne `tries=2` (nombre d'essais restants) et `pin=1234`. La séquence (i) ne peut pas être satisfaite, (ii) ne permet pas de bloquer la carte après un seul échec d'authentification, et donc, seules les séquences (iii) et (iv) sont faisables. De par le pilotage demandé, une seule de ces deux solutions sera conservée. Par défaut, ce sera toujours la première, ici, (iii).

Une instanciation des paramètres X_1, X_2 est alors demandée au solveur pour la séquence (iii). Celle-ci réalise l'appel au comportement cpt_2 de `checkPin` suivi du comportement cpt_3 de cette même opération. Les contraintes associées aux variables représentant les paramètres sont ainsi $X_1 \neq 1234$ et $X_2 \neq 1234$. Une instanciation basique produira ainsi : $X_1 = X_2 = 0$, donnant une séquence instanciée ayant pour finalité de bloquer la carte.

3. Présentation des patrons de propriétés

Dans l'article [DWY 99], les auteurs Dwyer, Avrunin et Corbett ont voulu faciliter l'écriture des propriétés en définissant des modèles types auxquels ils associent une formule de logique temporelle LTL, CTL ou une expression régulière quantifiée (QRE). Avec cette approche, il suffit de réussir à traduire la propriété textuelle dans un des modèles pour avoir la traduction en formule. Ces patrons ont été confrontés à une étude de 555 spécifications collectées dans différents domaines. La conclusion de cette étude montre que 92% des propriétés contenues dans ces spécifications pouvaient être exprimées à l'aide de ces patrons de propriétés. Nous présentons dans un premier temps la syntaxe de ces patrons de propriété ainsi que leur sémantique. Puis, nous précisons les patrons utilisés dans notre approche ainsi que certains éléments complémentaires pour leur mise en pratique dans le cadre de la génération de tests.

3.1. *Syntaxe et sémantique des patrons de propriété*

Les patrons de propriété se décomposent en deux parties distinctes : la *portée* et le *schéma de propriété*. La portée définit l'étendue l'exécution du programme sur laquelle le schéma de propriété s'applique. Cinq portées sont présentées :

– **globally** désigne l'intégralité de l'exécution du programme ;

– **before** (R) désigne l'exécution du programme jusqu'à ce que R ait lieu.

– **after** (Q) désigne l'exécution du programme après que Q ait eu lieu.

– **between** Q **and** R représente n'importe quelle partie de l'exécution entre Q et R.

– **after** Q **until** R se comporte comme **between** Q **and** R, mais la partie désignée de l'exécution continue même si R n'a pas lieu.

R et Q désignent soit le passage par un état satisfaisant un prédicat (dans le formalisme du programme ou du modèle considéré), soit le déclenchement d'un événement (à définir vis-à-vis du programme ou du modèle considéré). Dans les descriptions informelles ci-dessus, la condition "S a lieu" doit se lire comme "un état dans lequel le prédicat est vrai est atteint" si S désigne un prédi-

cat, ou, "un événement parmi une liste d'événements donnée se déclenche", si S est un événement. Ces notions sont également reprises dans l'expression des schémas de propriété.

Ces derniers sont au nombre de huit, et permettent de couvrir les principales constructions généralement employées dans l'écriture de propriétés.

– **absence** R. Un état ou un événement R n'a pas lieu pour la portée considérée.

– **existence** R. Un état ou un événement R doit avoir lieu pour la portée considérée.

– **bounded existence** R, k (existence bornée). Un état ou un événement R doit avoir lieu k fois pour la portée considérée.

– **universally** R (universalité). Un état ou un événement R a lieu tout au long de la portée considérée.

– P **precedes** Q. Un état ou un événement Q doit toujours être précédé d'un état ou d'un événement P pour la portée considérée.

– Q **response** P. Un état ou un événement P doit toujours être suivi par un état ou un événement Q pour la portée considérée.

Ces deux derniers schémas font l'objet de variantes, tendant à les généraliser, par l'introduction de la notion de chaînage entre deux listes d'états ou d'événements. Ces variantes sont nommées **Chain Precedence** et **Chain Response** dont les significations sont évidentes.

Dans notre approche, nous avons considéré les événements pouvant survenir comme étant des appels d'opérations consistant en l'activation d'un comportement spécifique. Pour désigner les comportements, nous utilisons le même mécanisme que celui présenté en partie 2.2.2, consistant à associer un libellé à un ou plusieurs comportements. Ces libellés sont notés en indice des opérations auxquels ils sont associés.

3.2. *Exemples issus de l'étude de cas*

Nous considérons une étude de cas issue du monde de la carte à puce. Il s'agit d'un modèle B de l'application Demoney [MAR 01], une spécification de porte-monnaie électronique développée par Trusted Labs. Même si cette application a été développée à des fins de recherche, elle représente une étude de cas représentative et suffisamment complexe d'un application de type carte à puce. Ce porte-monnaie gère le solde de la carte, des plafonds pour le solde et la valeur à débiter, ainsi que deux codes PIN, l'un authentifiant le porteur, l'autre permettant d'authentifier la banque émettrice de la carte.

3.2.1. *Description de l'exemple*

Comme toutes les applications de type carte à puce, Demoney présente une notion de cycle de vie qui débute par une phase de *personnalisation* (perso), durant laquelle différents paramètres de la carte peuvent être fixés à l'aide de la commande `PUT_DATA`. Une fois tous les paramètres renseignés, la personnalisation est validée par l'intermédiaire de la commande `STORE_DATA`. La carte passe ainsi en phase d'*utilisation* (use). Durant celle-ci, l'utilisateur peut démarrer une transaction de crédit ou de débit, par la commande `INIT_TRANSACTION`, et valider cette transaction, par la commande `COMMIT_TRANSACTION`. Si l'utilisateur souhaite créditer le porte-monnaie, il doit passer par une phase d'authentification par code pin, par la commande `VERIFY_PIN`. Lorsque l'utilisateur échoue trois tentatives d'authentification consécutives, la carte passe dans l'état *bloqué* (blocked). Pour la débloquer, la banque doit s'authentifier (commande `VERIFY_PIN`) puis changer le code PIN du porteur à l'aide de la commande `PIN_CHANGE_UNBLOCK`. Si la banque ne parvient

pas à s'authentifier, au bout de quatre essais, la carte est définitivement *perdue* (dead), i.e. plus aucune commande ne peut être exécutée avec succès.

3.2.2. *Propriétés de l'exemple*

De nombreuses propriétés s'appliquent sur cet exemple. Certaines peuvent s'exprimer directement par des invariants, comme par exemple *"le solde du porte-monnaie ne doit jamais être négatif."*. D'autres propriétés, faisant notamment intervenir les possibilités ou les interdictions d'appels d'opérations sont moins évidentes.

Nous donnons ici quelques propriétés informelles et leur expression dans la syntaxe des patrons de propriétés. Dans ces propriétés, on retrouve les noms des différentes commandes, ainsi que des éléments issus de la modélisation, dans les prédicats d'état. Dans ceux-ci, la variable `mode` représente le cycle de vie de la carte (`perso` $\rightarrow$ `use` $\leftrightarrows$ `blocked` $\rightarrow$ `dead`). Les comportements de succès d'exécution d'une opération sont désignés par le libellé *ok*. Pour l'opération `VERIFY_PIN`, le libellé *bankOk* désigne le comportement signifiant le succès d'authentification de la banque.

Propriété 1. Pendant la phase d'utilisation, il n'est pas possible pour la banque de s'authentifier.

between $\texttt{STORE\_DATA}_{ok}$ **and** `(mode=invalid)` **absence** $\texttt{VERIFY\_PIN}_{bankOk}$

Propriété 2. Après personnalisation, la commande `INIT_TRANSACTION` doit toujours être immédiatement suivie de la commande `COMMIT_TRANSACTION`

after `(mode=use)` $\texttt{COMMIT\_TRANSACTION}_{ok}$ **response** $\texttt{INIT\_TRANSACTION}_{ok}$

Propriété 3. Pendant la phase de personnalisation, on ne peut pas réaliser de transaction.

before $\texttt{STORE\_DATA}_{ok}$ **absence** $\texttt{INIT\_TRANSACTION}_{ok}$

Propriété 4. Lorsque la carte est bloquée, la commande `PIN_CHANGE_UNBLOCK` nécessite pour réussir une authentification réussie de la banque :

between `(mode=invalid)` **and** `(mode=dead)`
$\texttt{VERIFY\_PIN}_{bankOk}$ **precedes** $\texttt{PIN\_CHANGE\_UNBLOCK}_{ok}$

Nous présentons ci-après la contribution principale de cet article, à savoir la production des scénarios de tests décrits en partie 2 à partir des propriétés décrites dans cette dernière partie.

4. Stratégies de génération de scénarios

La conception des scénarios de tests se déroule en deux temps. D'abord, l'appel, en fonction du contenu de l'expression, à un ensemble de primitives, qui permettent, dans un second temps, de produire les scénarios eux-mêmes. Les primitives servent à faire le lien entre les patrons de propriété considérés et les scénarios. Chaque primitive est en charge de produire une partie de scénario. L'idée est d'enchaîner les primitives de manière à obtenir des scénarios complets. Ainsi, pour chaque patron, on définit un *besoin de test* qui représente une manière d'exercer la propriété considérée. Ce besoin de test va se traduire par une succession d'appel de primitives, nommée stratégie, et visant à produire un scénario, exerçant précisément la propriété considérée et répondant au besoin de test choisi.

Notre contribution est ainsi une architecture extensible permettant de générer des tests à partir de propriétés, par l'intermédiaire de stratégies qui visent à produire des scénarios. Cette architecture est extensible au sens où de nouvelles primitives, ainsi que de nouvelles stratégies, peuvent être ajoutées aux existantes.

Nous présentons d'abord quelques-unes des primitives utilisées, puis nous verrons leur emploi à travers des stratégies basées sur des patrons de propriété.

4.1. *Définition des primitives de génération de scénarios*

Nous décrivons ici 4 primitives utilisées par la suite pour produire des scénarios de tests. Nous considérons dans celles-ci un paramètre n fourni par l'utilisateur qui permet de fixer la longueur maximale des séquences de tests. Il est à noter que ce paramètre a une influence directe sur le temps de calcul des cas de tests. En effet, plus la longueur des scénarios est importante, plus il y a de combinaisons à prendre en compte.

Dans les descriptions qui suivent, lo représente une liste d'opérations du modèle $(o_1, \ldots, o_i)$, et lp représente une liste de prédicats $(p_1, \ldots, p_j)$. Pour chaque primitive, nous décrivons informellement son intention, puis nous donnons son schéma de tests associé, suivant la syntaxe présentée en partie 2. Par ailleurs, nous supposons que les comportements des opérations se décomposent en 2 catégories : les comportements libellés "prog", qui permettent au système de progresser (en faisant évoluer les valeurs des variables d'état), par opposition aux comportements qualifiés de "non-prog" qui ne modifient aucune valeur du système. Cette distinction est réalisée automatiquement par simple analyse statique de l'expression des comportements considérés, et est utilisée pour réduire l'explosion combinatoire durant l'énumération ultérieure des chemins.

Primitive 1. **CoverStop(**lo, lp**)** Cette primitive permet de récupérer les plus longues séquences finissant par un état particulier, en couvrant les différentes manières d'atteindre cet état. Elle réalise une couverture des comportements de toutes les opérations en profondeur. Ce calcul s'arrête lorsque la profondeur maximale a été atteinte. Elle conserve toutes les séquences qui terminent par un comportement des opérations de lo satisfaisant les prédicats de lp. On trouve ainsi plusieurs variantes en fonction de la vacuité des listes lo et lp.

$$
\begin{aligned}
[\![CoverStop(lo, \emptyset)]\!] &= [\$OP/w\{prog\}]^{0..n-2}.[\$OP]?.[o_1|\ldots|o_i] \\
[\![CoverStop(\emptyset, lp)]\!] &= [\$OP/w\{prog\}]^{0..n-1}.[\$OP] \rightsquigarrow p_1 \wedge \ldots \wedge p_j \\
[\![CoverStop(lo, lp)]\!] &= [\$OP/w\{prog\}]^{0..n-2}.[\$OP]?.[o_1|\ldots|o_i] \rightsquigarrow p_1 \wedge \ldots \wedge p_j
\end{aligned}
$$

Primitive 2. **CoverExact(**lo, lp**)** Cette primitive est une variante de la précédente. Si elle réalise également une couverture des comportements en profondeur, elle ne permet pas aux opérations intermédiaires (resp. aux états intermédiaires) d'appartenir aux opérations de lo (resp. de satisfaire les prédicats de lp).

$$
\begin{aligned}
[\![CoverExact(lo, \emptyset)]\!] &= [\$OP/w\{prog\}\backslash lo]^{0..n-2}.[\$OP\backslash lo]?.[o_1|\ldots|o_i] \\
[\![CoverExact(\emptyset, lp)]\!] &= ([\$OP/w\{prog\}] \rightsquigarrow (\neg p_1 \vee \ldots \vee \neg p_j))^{0..n-1}.[\$OP] \rightsquigarrow (p_1 \wedge \ldots \wedge p_j) \\
[\![CoverExact(lo, lp)]\!] &= ([\$OP/w\{prog\}\backslash lo] \rightsquigarrow (\neg p_1 \vee \ldots \vee \neg p_j))^{0..n-2} \ . \\
&\quad [\$OP\backslash lo] \rightsquigarrow (\neg p_1 \vee \ldots \vee \neg p_j) \ . \ [o_1|\ldots|o_i] \rightsquigarrow (p_1 \wedge \ldots \wedge p_j)
\end{aligned}
$$

Primitive 3. **CoverExcept(**lo, lp**)** Cette primitive est une variante de la précédente, dans laquelle, on cherche à couvrir tous les comportements sauf ceux de la liste d'opérations lo (resp. atteindre tous les états, sauf ceux de la liste de prédicats lp).

$$\begin{aligned}
[\![CoverExcept(lo, \emptyset)]\!] &= [\$OP/w\{prog\}\backslash lo]^{0..n-1}.[\$OP\backslash lo] \\
[\![CoverExcept(\emptyset, lp)]\!] &= ([\$OP/w\{prog\}] \rightsquigarrow (\neg p_1 \vee \ldots \vee \neg p_j))^{0..n-1}.[\$OP] \rightsquigarrow \neg p_1 \vee \ldots \vee \neg p_j \\
[\![CoverExcept(lo, lp)]\!] &= ([\$OP/w\{prog\}\backslash lo] \rightsquigarrow (\neg p_1 \vee \ldots \vee \neg p_j))^{0..n-1} \ . \\
&\quad [\$OP\backslash lo] \rightsquigarrow (\neg p_1 \vee \ldots \vee \neg p_j)
\end{aligned}$$

Primitive 4. **Activate(**lo**)** Cette primitive permet l'activation de tous les comportements de l'opération.

$$[\![Activate(lo)]\!] = [o_1]|\ldots|[o_i]$$

Voyons à présent l'emploi de ces primitives à partir des patrons considérés.

4.2. *Des propriétés aux scénarios, en passant par les primitives*

Les propriétés que nous étudions se limitent aux propriétés de sûreté, les propriétés de vivacités étant liées à l'atteignabilité (indécidable) des événements. Nous proposons, pour chaque patron de propriété, des tests issus de l'interprétation "littérale" de la propriété, pouvant ainsi servir de cas nominal ("ce qu'on veut voir satisfait"). Nous proposons également des cas dérivés permettant de tester des variantes de cette propriété, visant à exercer des cas implicites de robustesse ("ce qui ne doit pas arriver"). Les stratégies produites dépendent du type d'élément mis en jeu (événement ou prédicat d'état).

A chacun des scénarios ainsi produits est associé un *besoin de test*, informel, qui permet de savoir à quelle intention le scénario répond. En combinant cette description informelle avec l'instanciation faite du patron considéré, il devient possible d'associer à chaque ensemble de tests issu d'une propriété, l'intention qui est à l'origine de la création de ces tests, sous la forme d'une phrase en langage naturel.

Par souci de concision, et pour éviter l'effet "catalogue", nous ne donnons ici que les combinaisons dont sont issues les propriétés présentées en partie 3.2 sur l'étude de cas. Néanmoins, le lecteur intéressé pourra trouver une plus grande panoplie de combinaisons dans [ROY 08]. Notons que toutes les combinaisons possibles n'ont pas été traitées, mais peuvent l'être à condition de définir les stratégies adéquates. Néanmoins les primitives présentées ici permettent de couvrir un grand nombre de combinaisons portée/schéma de propriété.

4.2.1. ***between*** A ***and*** B ***absence*** C *ou "Entre* A *et* B*,* C *ne peut pas arriver"*

Cette combinaison permet d'exprimer le fait que C ne peut arriver dans un intervalle débutant par A et finissant par B. On produira pour cette propriété des cas de tests nominaux (Besoin de test 1.1) dans lequels on atteint l'événement B, après avoir atteint l'événement A et sans chercher à passer par C[1]

$$CoverStop(A); CoverExcept(C, B); CoverExact(B)$$

1. Dans les stratégies, la notation $primitive(X)$ est un raccourci se lisant $primitive(X, \emptyset)$ si X est un événement, ou $primitive(\emptyset, X)$ si X est un prédicat.

Sur la propriété 1, **between** $\texttt{STORE\_DATA}_{ok}$ **and** `mode=invalid` **absence** $\texttt{VERIFY\_PIN}_{bankOk}$ nous obtenons ainsi :

$$CoverStop([\texttt{STORE\_DATA}_{ok}], \emptyset); CoverExcept([\texttt{VERIFY\_PIN}_{bankOk}], \texttt{mode=invalid}]); \\ CoverExact(\emptyset, [\texttt{mode=invalid}])$$

Néanmoins cette propriété suggère plus une notion de robustesse, que nous pouvons exercer en testant une séquence dans laquelle on se préoccupe de construire des tests mettant en jeu l'intervalle A–B avec en plus des tentatives d'activation systématiques de C au sein de cet intervalle (Besoin de test 1.2). Ce qui donne, directement sur l'exemple :

$$CoverStop([\texttt{STORE\_DATA}_{ok}], \emptyset); CoverExcept([\texttt{VERIFY\_PIN}_{bankOk}], [\texttt{mode=invalid}]); \\ Activate([\texttt{VERIFY\_PIN}]); CoverStop(\emptyset, [\texttt{mode=invalid}])$$

Ici, nous produirons des tests qui contiennent des appels à VERIFY_PIN qui, lorsqu'ils seront réalisés, tenteront d'authentifier la banque. En toute rigueur, si le modèle est correct, ces appels devront retourner un code d'erreur. Si l'implantation contient une erreur, elle pourra par exemple autoriser cet appel, ce qui se traduira par un code de retour différent.

4.2.2. ***after*** A B ***response*** C *ou "Après A, B est la réponse à C"*

Cette combinaison permet de décrire des enchaînements insécables d'opérations et/ou d'états. On produira ici des cas de test nominaux (Besoin de test 2.1), visant à enchaîner les opérations considérées dans le contexte souhaité. Sur la propriété 2, **after** `(mode=use)` $\texttt{COMMIT\_TRANSACTION}_{ok}$ **response** $\texttt{INIT\_TRANSACTION}_{ok}$, nous obtenons ainsi le cas nominal donné par l'enchaînement de stratégies suivantes, directement appliqué à l'exemple :

$$CoverStop(\emptyset, [\texttt{mode=use}]); CoverExact([\texttt{INIT\_TRANSACTION}_{ok}], \emptyset); \\ Activate(\texttt{COMMIT\_TRANSACTION}_{ok})$$

Néanmoins, il est également intéressant de rompre les enchaînements en intercalant des appels à des opérations visant à rompre l'enchaînement supposé des deux événements B et C (Besoin de test 2.2). Sur l'exemple, ceci est donné par l'appel aux stratégies suivantes :

$$CoverStop(\emptyset, [\texttt{mode=use}]); CoverExact([\texttt{INIT\_TRANSACTION}_{ok}], \emptyset); \\ CoverExcept([\texttt{COMMIT\_TRANSACTION}_{ok}], \emptyset); Activate(\texttt{COMMIT\_TRANSACTION})$$

4.2.3. ***before*** A ***absence*** B

Cette combinaison permet de décrire la non-survenue d'un événement B entre l'état initial du système et l'arrivée d'un événement A. Sur la propriété 3, **before** $\texttt{STORE\_DATA}_{ok}$ **absence** $\texttt{INIT\_TRANSACTION}_{ok}$, on teste d'abord un cas nominal (Besoin de test 3.1), dans lequel on cherche à activer `STORE_DATA` sans passer par un appel à `INIT_TRANSACTION`.

$$CoverExcept([\texttt{STORE\_DATA}_{ok}, \texttt{INIT\_TRANSACTION}_{ok}], \emptyset); Activate(\texttt{STORE\_DATA}_{ok})$$

Nous considérons aussi un scénario visant à révéler un cas d'erreur potentiel, dans lequel on tente de satisfaire un appel à `INIT_TRANSACTION` avant d'atteindre `STORE_DATA` (Besoin de test 3.2) :

$$CoverExcept([\texttt{STORE\_DATA}_{ok}, \texttt{INIT\_TRANSACTION}_{ok}], \emptyset); \\ Activate(\texttt{INIT\_TRANSACTION}); CoverExact(\texttt{STORE\_DATA}_{ok})$$

4.2.4. ***between*** A ***and*** B, C ***precedes*** D

Cette combinaison exprime la précédence de l'événement C par rapport à D dans l'intervalle A et B. En prenant le cas nominal, nous cherchons à atteindre A, puis C, D et enfin B (Besoin de test 4.1). Sur la propriété 4, **between** `mode=invalid` **and** `mode=dead` $\texttt{VERIFY\_PIN}_{bankOk}$ **precedes** $\texttt{PIN\_CHANGE\_UNBLOCK}_{ok}$, nous obtenons le scénario suivant :

$$CoverExact(\emptyset, [\texttt{mode=invalid}]); CoverStop(\texttt{VERIFY\_PIN}_{bankOk}, \emptyset);$$
$$CoverStop(\texttt{PIN\_CHANGE\_UNBLOCK}, \emptyset); CoverStop(\emptyset, [\texttt{mode=dead}])$$

Nous considérons en plus, un cas d'erreur potentiel, lorsque dans l'intervalle A–B on tente de satisfaire un appel à D sans être passé par C (Besoin de test 4.2).

$$CoverExact(\emptyset, [\texttt{mode=invalid}]); CoverStop(\texttt{PIN\_CHANGE\_UNBLOCK}, \emptyset);$$
$$CoverStop(\emptyset, [\texttt{mode=dead}])$$

Pour les besoins d'expérimentations, un prototype a ainsi été développé prenant en compte les possibilités d'extensions par de nouvelles primitives et/ou de nouvelles stratégies.

En ajoutant la traçabilité des tests, qui permet de savoir pour un ensemble de tests quelle était la propriété d'origine, nous sommes ainsi en mesure de produire un rapport de génération de tests qui permet à l'ingénieur validation de connaître et de comprendre quels tests couvrent quelle propriété, comment et pourquoi. Ces éléments sont très importants d'un point de vue industriel, car ils permettent d'augmenter la confiance que l'ingénieur aura dans la suite de tests qui aura été produite.

5. Expérimentation

Nous terminons sur quelques métriques liées à l'étude de cas considérée. Nous avons réalisé un modèle B de l'application Demoney, ainsi qu'une implantation qui nous sert de système sous test pour exécuter les tests que nous produisons. L'expérimentation s'est déroulée de la manière suivante. D'abord nous avons généré les tests à partir de l'ensemble de propriétés données en partie 3.2.2. Les expressions régulières obtenues ont ensuite été dépliées et instanciées par le mécanisme d'animation symbolique. Les tests ont ensuite été exécutés sur une implantation Java de l'application. Dans l'objectif d'évaluer la pertinence des tests produits, nous avons réalisé une analyse mutationnelle. Nous avons ainsi créé des mutants[2] dans lesquels les erreurs sont guidées par les propriétés que nous souhaitons exercer. Par exemple, nous pouvons créer un mutant qui autorise la rupture de l'enchaînement des opérations INIT_TRANSACTION et COMMIT_TRANSACTION.

Pour l'expérience, nous nous sommes principalement intéressés à produire, manuellement et pour chaque propriété, des mutants dont le comportement est différent de celui du programme d'origine. Nous sommes ainsi sûrs que les erreurs ainsi introduites peuvent être détectées par les tests adéquats. En guise de comparaison, nous avons également tenté de détecter ces mêmes mutants avec des batteries de tests produits par l'outil LEIRIOS Test Generator [JAF 07], commercialisé par la société Smartesting[3]. A titre indicatif, il s'agit de deux campagnes de tests ; l'une pour la personnalisation, et l'autre pour la phase d'utilisation. La première comporte 44 tests, d'une longueur moyenne de 1.9 opérations, générés en moins de 3 minutes. La seconde présente 59 tests, d'une longueur moyenne de 7.6 opérations et générée en moins de 5 minutes.

2. Un mutant est une variante d'un programme supposé correct dans lequel une erreur a été introduite.
3. anciennement LEIRIOS Technologies – `http://www.smartesting.com`

Prop.	Besoin de test	# tests	Temps	Long. moy.	Mut.	Total	LTG
1	Cas nominal (1.1)	96	109 s	9.4	6/6	6/6	6/6
	Appels à C dans A–B (1.2)	302	179 s	11.3	6/6		
2	Cas nominal (2.1)	61	29 s	9.6	2/7	7/7	2/7
	Rupture C-B (2.2)	1527	308 s	12	7/7		
3	Cas nominal (3.1)	576	313s	5.9	3/7	7/7	3/6
	Appel à B avant A (3.2)	996	1673 s	7.7	6/7		
4	Cas nominal (4.1)	48	300 s	19.8	5/6	6/6	5/6
	Appel de D sans C (4.2)	172	730 s	13.9	3/6		

Tableau 1. *Résultats de l'expérience pour les 4 propriétés considérées*

Les résultats des expériences sont donnés par le tableau 1. Celui-ci sert principalement à mesurer l'efficacité des stratégies liées aux propriétés considérées. La colonne *Prop.* donne le numéro de la propriété considérée dans la liste donnée en partie 3.2.2. La colonne *Besoin de test* décrit le besoin de test qui a motivé le scénario. Les colonnes suivantes donnent ensuite le nombre de tests (colonne *# tests*) produits à partir du scénario, le temps de calcul[4] total (colonne *temps*) comprenant le dépliage du scénario et l'animation des séquences sur l'animateur de BZ-Testing-Tools, et la longueur moyenne des tests (colonne *long. moy.*). La colonne *mut.* donne le score de détection de mutants associés à la propriété, tandis que la colonne *total* donne le score de l'ensemble des tests liés à la propriétés sur les mutants. La dernière colonne donne à titre indicatif le score réalisé par les tests produits par LTG.

Nous constatons que la génération des tests est réalisée dans des temps de calculs raisonnables, malgré l'explosion combinatoire inhérente au dépliage des expressions régulières décrivant les scénarios. Les tests produits sont efficaces vis-à-vis de diverses erreurs potentielles, puisqu'ils permettent de les détecter, alors qu'une approche comme celle de LTG n'y parvient pas forcément. Ce résultat est à nuancer, car l'outil LTG étant conçu pour assurer une couverture des comportements des opérations "en isolation", il n'est donc pas surprenant qu'une approche dédiée au test de propriétés soit plus efficace. Bien évidemment, il ne faut pas voir l'approche proposée ici comme un mécanisme de test à part entière, mais plutôt comme un complément pour le test automatique, ciblant des propriétés et des besoins de tests bien précis. Une amélioration néanmoins concerne la traçabilité des tests car il est possible, par notre approche de savoir précisément quelle propriété a été testée, et à quel besoin répondaient les scénarios qui ont été engendrés.

6. Autres travaux

La notion de propriété est souvent employée dans le contexte de la génération de tests. Les approches proposées par [HON 02, AMM 01, TAN 04] prennent en compte des formules LTL subissant des mutations pour ensuite utiliser un model-checker dans le but de produire des traces menant à un contre-exemple de cette propriété, comme présenté dans [GAR 99]. D'autres travaux plus récents [FRA 08] définissent la notion de "property-relevant test cases" (cas de tests pertinents pour une propriété) introduisant un nouveau critère de couverture qui peuvent être utilisés pour déterminer des cas de tests positifs et négatifs. Cette volonté de générer des tests qui illustrent, mais aussi exercent, la propriété est commune à notre approche. Néanmoins nous nous différencions au niveau de la mise en pratique. Nous n'employons pas un model checker mais des techniques symboliques à contraintes qui permettent de parcourir des espaces d'états potentiellement plus grands.

4. Temps mesuré sur un Intel Core 2 Duo, 1.8 GHz, 1 Go RAM

Un travail similaire à celui présenté dans cet article a été proposé dans [MAS 07], dans lequel les auteurs considèrent une propriété nominale exprimée sous la forme d'une expression régulière qui est ensuite mutée. L'approche présentée dans ce papier est relativement similaire dans son essence. Néanmoins l'expressivité du langage d'expressions régulières reste relativement faible, comparé aux patrons de propriétés de Dwyer. Par contre, certaines des mutations réalisées peuvent être réemployées pour ajouter de nouveaux besoins de tests, et par conséquent de nouvelles stratégies, aux scénarios que nous avons proposé.

Différents outils sont basés sur des propriétés pour générer des cas de tests. Citons l'outil j-Post [FAL 08] qui utilise une propriété exprimée dans une logique de traces pour générer des cas de tests à partir de systèmes de transitions à entrées/sorties. Similairement, STG [CLA 01] s'appuie également sur le formalisme des ioSTS pour décrire le système considéré et un objectif de test associé qui sera utilisé pour produire les tests. Une différence majeure avec notre approche est le fait que STG demande à l'utilisateur de formaliser sa propriété sous la forme d'un ioSTS, alors que notre approche, comme j-Post, permet l'expression de la propriété à un plus haut niveau.

7. Conclusion et Perspectives

Nous avons présenté dans cet article une technique originale de génération de tests à partir de patrons de propriété génériques. Ces propriétés sont utilisées pour produire des scénarios de tests, décrits par des expressions régulières sur les opérations du système, laissant les paramètres abstraits. Ces derniers sont instanciés suite au dépliage de l'expression et son évaluation par un moteur d'animation symbolique. Nous avons présenté une expérimentation sur une étude de cas réaliste, et nous avons évalué le pouvoir de détection des tests générés qui s'est révélé améliorer les simples tests fonctionnels générés automatiquement sur des critères de couverture structurelle des opérations du modèle. Par ailleurs, l'utilisation de propriétés permet de faire un lien direct de traçabilité entre les tests et les exigences qu'ils permettent de couvrir.

L'une des premières perspectives de recherche est la réduction de l'explosion combinatoire. En effet, même si notre approche évite l'énumération des paramètres, les combinaisons d'enchaînements de comportements sont potentiellement denses. L'expérimentation menée sur une étude de cas représentative montre qu'un grand nombre de cas de tests est produit par l'application de notre démarche. Par ailleurs, nous avons constaté durant l'expérimentation que les mutants étaient généralement tués par un sous-ensemble assez important de la suite de tests, laissant ainsi penser à la présence de classes d'équivalence permettant de regrouper les tests produits. Une possibilité pour cela serait de jouer plus finement sur le pilotage des scénarios qui permettrait de réduire considérablement le nombre de tests produits. Une autre option potentielle serait de trouver des critères permettant de réduire les suites de tests obtenues après le dépliage des scénarios.

La question du choix des valeurs pour l'instanciation des paramètres des opération pourrait également être abordée. A l'heure actuelle, les stratégies ne couvrent que des valeurs dites "intelligentes" (une valeur pour un atome ou les valeurs extremum pour les valeurs numériques). Une extension visant à introduire une sélection aléatoire des données de tests est à envisager pour éviter la trop grande proximité de valeurs de variables potentiellement interdépendantes.

Pour finir, nous prévoyons une diffusion de cette technologie dans le cadre d'un plug-in pour la plateforme RODIN dédiée à la modélisation en B événementiel[5].

Remerciements. L'auteur souhaite remercier Sébastien Roy pour sa contribution au travail présenté dans cet article, ainsi que les relecteurs de l'Atelier AFADL pour leurs remarques.

5. http://www.event-B.org

8. Bibliographie

[ABR 96] ABRIAL J., *The B-Book*, Cambridge University Press, 1996.

[AMM 01] AMMAN P., DING W., XU D., « Using a Model Checker to Test Safety Properties », *ICECCS'01, 7-th Int. Conf. on Engineering of Complex Computer Systems*, Washington, DC, USA, 2001, IEEE Computer Society.

[BEI 95] BEIZER B., *Black-Box Testing : Techniques for Functional Testing of Software and Systems*, John Wiley & Sons, New York, USA, 1995.

[BOU 04] BOUQUET F., LEGEARD B., PEUREUX F., « CLPS-B : A Constraint Solver to Animate a B Specification », *International Journal on Software Tools for Technology Transfer, STTT*, vol. 6, n° 2, 2004, p. 143–157, Springer.

[BOU 06] BOUQUET F., DADEAU F., GROSLAMBERT J., JULLIAND J., « Safety Property Driven Test Generation from JML Specifications », *FATES/RV'06, 1st Int. WS on Formal Approaches to Testing and Runtime Verification*, vol. 4262 de *LNCS*, Seattle, WA, USA, août 2006, Springer, p. 225–239.

[CAM 05] CAMPBELL C., GRIESKAMP W., NACHMANSON L., SCHULTE W., TILLMANN N., VEANES M., « Testing Concurrent Object-Oriented Systems with Spec Explorer. », FITZGERALD J., HAYES I. J., TARLECKI A., Eds., *International Symposium of Formal Methods Europe (FM'05)*, vol. 3582 de *Lecture Notes in Computer Science*, Newcastle, UK, July 18-22 2005, Springer, p. 542-547.

[CLA 01] CLARKE D., JÉRON T., RUSU V., ZINOVIEVA E., « STG : a tool for generating symbolic test programs and oracles from operational specifications », *ESEC/FSE-9 : Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, New York, NY, USA, 2001, ACM, p. 301–302.

[DAD 08] DADEAU F., DE KERMADEC A., TISSOT R., « Combining Scenario- and Model-Based Testing to ensure POSIX Compliance », *ABZ'2008, International Conference on ASM, B and Z*, vol. 5238 de *LNCS*, London, United Kingdom, septembre 2008, Springer, p. 153–166.

[DWY 99] DWYER M. B., AVRUNIN G. S., CORBETT J. C., « Patterns in property specifications for finite-state verification », *ICSE '99 : Proceedings of the 21st international conference on Software engineering*, Los Alamitos, CA, USA, 1999, IEEE Computer Society Press, p. 411–420.

[FAL 08] FALCONE Y., MOUNIER L., FERNANDEZ J.-C., RICHIER J.-L., « j-POST : a Java Toolchain for Property-Oriented Software Testing », *MBT'08, Int. Workshop on Model-Based Testing*, Budapest, Hungary, 2008.

[FRA 08] FRASER G., WOTAWA F., « Using Model-Checkers to Generate and Analyze Property Relevant Test-Cases », *Software Quality Journal*, vol. 16, 2008, p. 161-183.

[GAR 99] GARGANTINI A., HEITMEYER C., « Using Model Checking to Generate Tests from Requirements Specifications », *Procs of the Joint 7th Eur. Software Engineering Conference and 7th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, 1999.

[HON 02] HONG H., LEE I., SOKOLSKY O., URAL H., « A Temporal Logic Based Theory of Test Coverage and Generation », *TACAS'02, 8th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, vol. 2280 de *LNCS*, London, UK, 2002, Springer-Verlag, p. 327–341.

[JAF 07] JAFFUEL E., LEGEARD B., « LEIRIOS Test Generator : Automated Test Generation from B Models », *B'2007, the 7th Int. B Conference*, vol. 4355 de *LNCS*, Besançon, France, janvier 2007, Springer, p. 277–281.

[JUL 08] JULLIAND J., MASSON P.-A., TISSOT R., « Generating Security Tests in Addition to Functional Tests », *AST'08, 3rd Int. workshop on Automation of Software Test*, Leipzig, Germany, mai 2008, ACM Press, p. 41–44.

[LED 04] LEDRU Y., DU BOUSQUET L., MAURY O., BONTRON P., « Filtering TOBIAS Combinatorial Test Suites », WERMELINGER M., MARGARIA T., Eds., *Fundamental Approaches to Software Engineering, 7th Int. Conf., FASE 2004*, vol. 2984 de *LNCS*, Barcelona, Spain, 2004, Springer, p. 281-294.

[MAR 01] MARLET R., METAYER D. L., « Security properties and Java Card specificities to be studied in the SecSafe project », 2001.

[MAS 07] MASSON P.-A., JULLIAND J., PLESSIS J.-C., JAFFUEL E., DEBOIS G., « Automatic Generation of Model Based Tests for a Class of Security Properties », *A-MOST'07, 3rd int. Workshop on Advances in Model Based Testing*, London, UK, juillet 2007, ACM Press, p. 12–22.

[ROY 08] ROY S., « Conception de Stratégies pour Tester les Propriétés de Sûreté », Mémoire de Master 2 Recherche, Université de Franche-Comté, LIFC, 2008.

[TAN 04] TAN L., SOKOLSKY O., LEE I., « Specification-based Testing with Linear Temporal Logic », *IRI'2004, IEEE Int. Conf. on Information Reuse and Integration*, novembre 2004, p. 413–498.

[TRE 02] TRENTELMAN K., HUISMAN M., « Extending JML Specifications with Temporal Logic », *AMAST '02 : Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*, London, UK, 2002, Springer-Verlag, p. 334–348.