Formula Negator, Outil de négation de formule.

Aymerick Savary[1,2], Mathieu Lassale[1,2], Jean-Louis Lanet[1] et Marc Frappier[2]

[1] Université de Limoges
[2] Université de Sherbrooke

Résumé. Cet article présente un outil de génération de tests de vulnérabilité, appelé VTG, fondé sur la mutation de modèles abstraits. Les modèles sont exprimés en Event-B. Les tests de vulnérabilité sont obtenus par mutation de modèles Event-B en niant des gardes des événements et des axiomes des contextes. L'API TOM est utilisée pour effectuer la ré-écriture des formules de logique. Le vérifieur de modèles ProB est utilisé pour générer les tests par animation des modèles Event-B.

1 Introduction

La génération de tests de vulnérabilité [8] est une méthode permettant de mettre en évidence des failles de sécurité potentielles. Elle est notamment avantageuse pour des applications pour lesquelles nous ne possédons pas le code source (boite noire). Cette méthode repose sur la génération de test à base de mutants [5] et le test à base de modèle [10]. Le VTG [1] (*Vulnerability Tests Generator*) est une implémentation de cette méthodologie. La première version de l'outil a permis de tester l'approche sur différents cas d'études tels que le BCV (*i.e. Byte Code Verifier*) Java Card [4] et le protocole EMV [7] (*Europay Mastercard Visa*). Cette première étude a montré la validité de l'approche pour une sous-partie du BCV et du protocole EMV. La nouvelle version [9] de la méthode a permis d'étendre ses capacités en décomposant le processus de génération de tests de vulnérabilité en deux sous-processus, (*c.f.* Fig. 1).

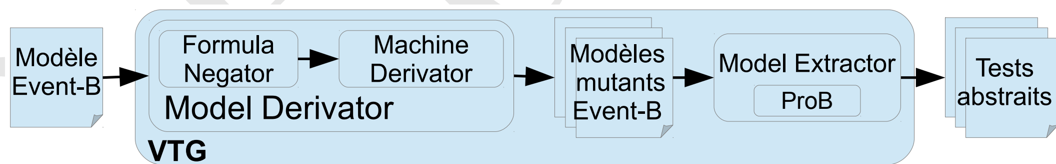

Fig. 1. Décomposition du VTG en sous-processus

Model Derivator permet de muter un modèle initial en un ensemble de modèles mutants. Ces derniers sont ensuite envoyés au processus *Model Extractor* qui extrait des tests abstraits des modèles mutants. Le processus *Model Derivator* est décomposé en deux sous-processus, *Formula Negator* qui permet de muter des formules et *Machine Derivator* qui combine ces mutations avec des machines. Ce dernier processus peut être remplacé par *Context Derivator* pour dériver des contextes. Dans cet article, nous nous attarderons, dans la section 3, sur le détail du fonctionnement du processus *Formula Negator* qui constitue la partie la plus intéressante de cette contribution.

2 Choix d'implémentation

Dans nos recherches, nous utilisons le langage de modélisation Event-B. Un modèle Event-B est une combinaison de contextes et de machines. Les contextes représentent la partie statique en définissant les constantes et les axiomes. Les machines représentent la partie dynamique en définissant les variables, les invariants et les événements. Les événements sont composés d'une garde et d'une action (similaire à des pré/post-conditions). Les axiomes et les gardes correspondent aux contraintes que l'on souhaite muter. Cette mutation repose sur des règles de négation.

Pour implémenter cette négation, nous avons utilisé l'API de Rodin [2] et l'API TOM [3]. L'API de Rodin permet de manipuler des modèles Event-B et celle de TOM permet de parcourir un arbre de données orienté objet et de le modifier selon des règles de réécritures. Ces dernières correspondent à nos règles de négation. Rodin est un programme à code ouvert et son analyseur de formules repose sur un programme TOM. Pour l'analyseur de *Formula Negator*, nous avons donc réutilisé l'analyseur fourni avec Rodin. La négation est ensuite effectuée à l'aide de réécritures TOM. *Formula Negator* est donc décomposé en trois parties: l'extraction des formules, l'analyseur et le réécriveur.

Pour l'extraction de tests, nous avons utilisé le vérifieur de modèle ProB [6]. *Model Extractor* se base pour le moment sur l'interface en ligne de commande de ProB. Cependant, nous envisageons d'utiliser l'API de ProB. Cela nous permettait de guider plus précisément la recherche de solutions et de sélectionner plus précisément les tests que nous souhaitons extraire.

3 *Formula Negator*

L'algorithme 1 représente la première étape qui consiste à analyser les différents fichiers composant un modèle Rodin. Pour simplifier la description de l'outil, nous considérons qu'un modèle correspond à l'ensemble des fichiers contenus dans un projet Rodin. *Formula Negator* prend en entrée un projet Rodin, extrait les axiomes pour chaque contexte et les gardes de chaque événement de chaque machine.

Algorithm 1 formulaExtractor

```
1:  procedure formulaExtractor(rodinProject)
2:      for all context in rodinProject do
3:          for all axiom in context do
4:              analyseAndRewrite(axiom)
5:          end for
6:      end for
7:      for all machine in rodinProject do
8:          for all event in machine do
9:              for all guard in event do
10:                 analyseAndRewrite(guard)
11:             end for
12:         end for
13:     end for
14: end procedure
```

Ces formules sont ensuite transmises à un analyseur qui, pour chaque formule, analyse son arbre syntaxique et détermine la négation à appliquer. L'analyse de l'arbre est seulement effectuée pour une profondeur de deux (noeud courant et ses fils, mais pas

DRAFT

ses sous-fils). Si la négation est récursive, l'algorithme est répété sur les fils devant être récursivement niés. Cette analyse s'arrête lorsque aucune règle de négation ne peut être appliquée. À la fin de l'exécution on obtient la liste des négations applicables à la formule passée en entrée.

Prenons par exemple la formule suivante : $x < 24 \vee x = 43 - 1$. Dans la suite, nous utiliserons les symboles et les règles de négation suivantes :

- $i_1, i_2 \in \mathbb{N}$
- p_1, p_2 des prédicats

1. $neg(p_1 \vee p_2) \rightsquigarrow neg(p_1) \wedge neg(p_2)$
2. $neg(i_1 < i_2) \rightsquigarrow \{\ i_1 > i_2, i_1 = i_2\}$
3. $neg(i_1 = i_2) \rightsquigarrow \{i_1 < i_2,\ i_1 > i_2\}$

L'analyse de la formule est assimilée à la première règle de négation. Cette règle propage ensuite la négation aux fils du "$\vee$". Le premier fils, $x < 24$, concorde avec la seconde règle de négation. La formule est donc réécrite en $x > 24$ et $x = 24$. La sous-formule $x = 43 - 1$, est réécrite $x < 43 - 1$ et $x > 43 - 1$. On obtient alors les réécritures suivantes par combinaison des différentes négations :

1. $x > 24 \wedge x < 43 - 1$
2. $x > 24 \wedge x > 43 - 1$
3. $x = 24 \wedge x < 43 - 1$
4. $x = 24 \wedge x > 43 - 1$

4 Fonctionnement de TOM

Les programmes TOM peuvent être compilés pour générer du code; plusieurs langages sont supportés (ex: C, C++, Java). Nous utilisons Java. Deux types de fichiers TOM sont nécessaires afin de générer un programme d'analyse d'arbres syntaxiques Event-B. Le premier fichier contient la définition des catégories syntaxiques de la grammaire de Event-B; il est tiré de l'API Rodin; un exemple est donné à la fig. 2. La première ligne décrit le symbole "=", appelé `Equal` et de type `Predicate`; ce symbole a deux fils de type `Expression`. Les lignes suivantes donnent des règles de vérification de type utilisées par Rodin et qui ne sont pas utilisées pour notre réécriture.

Le second fichier décrit les réécritures à effectuer en utilisant un appariement de formes (*pattern matching*). L'implémentation de la règle de réécriture 3 pour l'égalité est illustrée à la Fig. 3. L'opérateur `%match` (ligne 1) dénote l'appariement de forme; il est similaire à un `switch`/`case` en Java. Chaque forme correspond à un `case`. La forme est donnée à la ligne 3, et le code à exécuter pour générer une nouvelle formule est donné aux lignes 4 à 11.

Une règle récursive comme la règle 1 sur la disjonction est implémentée par un appel récursif sur chaque fils afin de générer les réécritures des fils et de les combiner pour donner les réécritures de la disjonction.

5 *Model Extractor*

Un test est constitué de quatre parties: préambule, corps, identification et postambule. Le corps correspond à la faute que nous voulons faire afin de mettre en évidence une défaillance du système sous test. L'identification est la partie du test qui permet de

DRAFT

```
1 %op Predicate Equal (left: Expression, right: Expression) {
2     is_fsym(t) { t != null && t.getTag() == Formula.EQUAL }
3     get_slot(left,t) { ((RelationalPredicate) t). getLeft() }
4     get_slot(right,t) { ((RelationalPredicate) t).getRight() }
5 }
```

Fig. 2. Grammaire Rodin définie en TOM

```
1  %match(formula) {
2      ...
3      Equal(left,right)->
4      {
5      Formula<?> f1,f2;
6      f1=ff.makeRelationalPredicate(Formula.GT,`left,`right,null);
7      f2=ff.makeRelationalPredicate(Formula.LT,`left,`right,null);
8      Set<Formula> s = new Set<>();
9      s.add(f1);
10     s.add(f2);
11     return  s;
12     }
13     ...
14 }
```

Fig. 3. Analyseur du *Formula Negator*

mettre en évidence la défaillance. Le préambule permet de conduire le système dans un état où le corps peut être exécuté. Le postambule permet de ramener le système dans un état valide après l'exécution d'une faute.

Dans le cas d'une mutation de machine, le corps d'un test correspond à un événement muté. Cet événement muté a été généré par *Machine Derivator* par mutation de sa garde. Pour trouver un préambule, nous utilisons un vérifieur de modèles en spécifiant qu'il est impossible d'exécuter cet événement. Tous les contre-exemples correspondent à des préambules. Le postambule est ensuite recherché de la même façon. Pour trouver un test, nous spécifions simplement qu'il est impossible d'exécuter l'événement sous test et d'arriver dans un état terminal.

Dans le cas d'une mutation de contexte, un test est uniquement représenté par un corps, qui correspond à un ensemble de valeurs. Toute instanciation de ce contexte constitue un test.

6 Conclusion

Nos premiers travaux sur la génération de tests de vulnérabilité avaient permis de mettre en évidence des vulnérabilités dans des cartes à puce. Ces études de faisabilité, bien qu'appliquées à de tout petits exemples, étaient encourageantes. Cependant, le VTG avait été construit en mode prototypage en essayant de prendre en compte de fréquentes modifications de la partie théorique. Cela rendait difficile son utilisation pour des modèles de taille plus importante. La méthode étant désormais stable, nous développons actuellement la nouvelle version du VTG. *Formula Negator*, que nous avons présenté dans cet article et qui correspond au premier maillon de la chaine permettant de générer ces tests de vulnérabilité.

DRAFT

Nous avons vu que l'utilisation de TOM nous a permis de facilement implémenter notre méthode de génération de tests de vulnérabilité. Cependant, l'utilisateur désirant ajouter de nouvelles règles ou utiliser son propre jeux de règles doit actuellement modifier notre programme TOM. Il serait intéressant de proposer une notation plus abstraite ayant une syntaxe proche de nos règles de négation. Une couche additionnelle au logiciel *Formula Negator* prendrait en entrée un fichier contenant les règles de négation exprimées avec cette notation et génèrerait les programmes de réécriture TOM correspondant.

References

1. A. Savary, J.-L Lanet, M. Frappier, T. Razafindralambo, J.D.: VTG - Vulnerability Test Generator, a Plug-in for Rodin. Workshop Deploy 2012 (2012)
2. Rodin website: `http://wiki.event-b.org/index.php/Main_Page`
3. TOM website: `http://tom.loria.fr/wiki/index.php5/Main_Page`
4. Hamadouche, S., Lanet, J.L., Mezghiche, M.: Méthode d'Analyse de Vulnérabilité Appliquée à un Composant de Sécurité d'une Carte à Puce. Rencontres sur la Recherche en Informatique (R2I)
5. Jia, Y., Harman, M.: An Analysis and Survey of the Development of Mutation Testing. IEEE Transactions on Software Engineering 37(5), 649–678 (Sep 2011)
6. Leuschel, M., Butler, M.: ProB: A model checker for B. FME 2003: Formal Methods pp. 855–874 (2003)
7. Ouerdi, N., Azizi, M., Ziane, M.H., Azizi, A., Savary, A.: Security Vulnerabilities Tests Generation from SysML and Event-B Models for EMV Cards. International Journal of Security and Its Applications 8(1), 373–388 (2013)
8. Savary, A., Frappier, M., Lanet, J.L.: Automatic Generation of Vulnerability Tests for the Java Card Byte Code Verifier. In: 2011 Conference on Network and Information Systems Security. pp. 1–7. IEEE (2011)
9. Savary, A., Frappier, M., Lanet, J.: Detecting Vulnerabilities in Java-Card Bytecode Verifiers Using Model-Based Testing. Integrated Formal Methods (2013)
10. Utting, M., Legeard, B.: Practical model-based testing: a tools approach. Morgan Kaufmann Publishers (2010)

DRAFT