Université de Montréal

# Vérification des patrons temporels d'utilisation d'API sans exécution du code: une approche et un outil

par

**Erick F. Raelijohn**

Département de mathématiques et de statistique

Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de

Maître ès sciences (M.Sc.)

en Informatique

août 2020

© Erick F. Raelijohn, 2020

# Université de Montréal

Faculté des études supérieures et postdoctorales

Ce mémoire intitulé

**Vérification des patrons temporels d'utilisation d'API sans exécution du code: une approche et un outil**

présenté par

**Erick F. Raelijohn**

a été évalué par un jury composé des personnes suivantes :

*Eugène Syriani*
(président-rapporteur)

*Michalis Famelis*
(directeur de recherche)

*Houari Sahraoui*
(codirecteur)

*Liam Paull*
(membre du jury)

# Sommaire


La réutilisation est une pratique courante lors du développement de logiciel. Bien souvent, cette réutilisation se fait à travers l'utilisation des librairies. Cette dernière met ses fonctionnalités à disposition des développeurs en utilisant les Interfaces de Programmation d'Application (API). En théorie, les développeurs qui utilisent les API n'ont pas forcément besoin de se préoccuper de comment les éléments internes de cette API fonctionnent. En effet, les API mettent leurs fonctionnalités à disposition des développeurs sans forcément dévoiler ce qui se passe à l'interne. Cependant, pour utiliser correctement une API il est nécessaire de respecter des contraintes d'utilisation qui sont à la fois implicites et explicites ainsi que des modèles d'utilisation.

L'usage des librairies et des API est très commun dans le domaine du développement de logiciel. Cela permet aux développeurs d'utiliser les fonctionnalités proposées par l'API et ainsi de se concentrer directement sur la tâche qu'ils doivent effectuer. Toutefois, apprendre et se familiariser avec les contraintes d'usage des API sont des tâches ardues et exigent un effort cognitif considérable de la part du développeur. Les chercheurs ont tenté de corriger ce problème en étudiant les modèles d'utilisation et en analysant les traces d'utilisation de code client pour s'assurer de leurs conformités. Néanmoins, les analyses dynamiques ne sont pas possibles pendant les phases précoces de développement du logiciel, car cela requiert une implémentation minimum et l'exécution du code.

Nous proposons l'outil Temporal Usage PAttern Checker (Tupac). Une approche basée sur l'analyse statique interprocédural pour vérifier la conformité du code client aux modèles d'utilisation pendant la phase de développement. Tupac peut être déployé dans un environnement de développement (IDE) et ainsi fournir des informations relatives à l'utilisation des API plus tôt pendant la phase de développement du logiciel.

Nous avons évalué notre approche sur quatre projets Java avec quatre API. Les résultats ont démontré que TUPAC a une bonne précision et un taux de rappel intéressant. De plus, nous avons pu conclure qu'en moyenne cela prend une demi-seconde pour vérifier la conformité d'un patron pour un projet tout entier. Cela démontre que TUPAC peut être déployé dans un rythme de codage régulier.

**Mots clés: Patrons d'utilisation d'API, analyse statique, vérification de modèles**

# Summary

In modern software development, reuse takes the form of using libraries that expose their functionality via Application Programming Interfaces (APIs). In theory, APIs allow developers to write client code that reuses library code without needing to know its internals. In practice, correctly using APIs requires respecting explicit and implicit constraints and usage patterns. This allows developers to use functionality proposed by API so that they can focus directly on the task they want to achieve. APIs require a significant effort from the developer to learn various usage constraint. Ignoring such patterns could lead to errors and design flaws. These often cannot be detected prior to integration and system testing. Researchers have attempted to solve this problem by extracting API usage patterns and analyzing client code traces for conformance. However, dynamic analysis is still impossible to perform early without a minimum of integration and execution.

We propose the Temporal Usage PAttern Checker (TUPAC) for API, an interprocedural static analysis approach that can verify that client code conforms to temporal API usage patterns as it is being developed. TUPAC can be deployed inside an Integrated Development Environment (IDE), thus providing developers with feedback about API usage much earlier in the development process.

We evaluated the effectiveness of our approach on four projects with four different APIs. Our evaluation shows that TUPAC has good precision and interesting recall. Crucially, we also show that it takes, on average, half a second to check an entire project for conformance to a pattern, meaning that it can realistically be deployed in the regular coding rhythm.

**Keywords: API usage patterns, static analysis, model checking.**

# Table des matières

**Sommaire** .... 3

**Summary** .... 5

**Liste des tableaux** .... 9

**Liste des figures** .... 10

**Liste des sigles et des abréviations** .... 11

**Remerciements** .... 12

**Chapitre 1. INTRODUCTION** .... 13

1.1. Contexte .... 13

1.2. Problématique et solution proposée .... 13

1.3. Contribution .... 15

1.4. Organisation du mémoire .... 16

**Chapitre 2. CONTEXTE ET ÉTAT DE L'ART** .... 17

2.1. Définition d'une API .... 17

2.2. Assistance disponible pour les développeurs .... 18

2.3. Patron d'utilisation d'API .... 18

2.4. Extraction des patrons d'utilisation .... 19

2.4.1. Recherche sur les patrons d'utilisation non ordonnés .... 19

2.4.2. Recherche sur les patrons d'utilisation ordonnés .... 20

2.4.3. Patrons d'utilisation pour assister les développeurs ........ 20
Amélioration de la documentation ........ 20
Rédaction du code ........ 21
Phase de compilation et de test ........ 21

**Chapitre 3. EXEMPLE ILLUSTRATIF ........ 23**

3.1. Mauvaise utilisation ........ 23

3.2. Réparation possible du mauvais usage ........ 24

3.3. Utilisation des patrons ........ 24

**Chapitre 4. FORMULATION DE L'APPROCHE ........ 29**

4.1. Définition ........ 29
4.1.1. Graphe de flot de contrôle (CFG) ........ 29
4.1.2. Graphe d'appel (CG) ........ 30
4.1.3. Graphe profond (DG) ........ 31

4.2. Détails de l'approche ........ 33
4.2.1. Génération du Graphe profond ........ 34
4.2.2. Transformation du Graphe profond en modèle SMV ........ 36
4.2.3. Vérification du modèle ........ 38

**Chapitre 5. IMPLÉMENTATION ........ 41**

5.1. Analyse du code Java ........ 42
Génération des modèles CFG, CG et DG ........ 43

5.2. Vérificateur de modèle ........ 45

5.3. Interface utilisateur ........ 46

**Chapitre 6. VALIDATION ........ 50**

6.1. Cadre expérimental ........ 50

6.2. Résultats ........ 53
6.2.1. ***RQ1:*** Est-ce que notre approche est capable de détecter les violations de patrons d'utilisation à partir d'une analyse statique? ........ 53
6.2.2. **RQ2:** Est-ce qu'une analyse de type interprocédurale offre une meilleure performance comparée à une simple analyse intraprocédurale? ........ 54
6.2.3. **RQ3:** *TUPAC* est-il assez rapide pour fournir des résultats utilisables au développeur sans être une entrave à sa productivité? ........ 55
6.3. Menaces à la validité ........ 57
6.4. Limitations et travaux futurs ........ 58
**Chapitre 7. CONCLUSION** ........ 59
**Références bibliographiques** ........ 61
**Annexe A. Modèle SMV** ........ 64

# Liste des tableaux

4.1 Comment les concepts du Graphe profond sont représentés en syntaxe SMV . 37

4.2 trace d'exécution possible. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

6.1 Projets et API utilisés pour la validation . . . . . . . . . . . . . . . . . . . . . . . 52

6.2 Nombre de patrons traités . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

6.3 Taille des patrons test. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

6.4 Résultats de l'analyse manuelle (en bleu) et Tupac (en vert). . . . . . . . . . . 53

6.5 Résultat après mutation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

6.6 Moyenne par patron . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57

6.7 Moyenne par projet en ms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57

# Liste des figures


1.1 Vue globale de notre approche. . . . . 16

3.1 Graphe de flot de contrôle (CFG) de la méthode `close()`. . . . . 27

4.1 CG de la méthode `safeClose()` dans Listing 3.2. . . . . 33

4.2 CFG des méthodes dans Listing 3.2. . . . . 33

4.3 Graphes de la méthode `manager()` dans Listing 4.1. . . . . 39

4.4 DG de la méthode `safeClose()` dans Listing 3.2. . . . . 40

5.1 Architecture globale de notre approche. . . . . 41

5.2 Les IJavaELement dans Eclipse. . . . . 42

5.3 Processus de vérification de modèle. . . . . 45

5.4 Menu affiché vue d'Eclipse . . . . . 47

5.5 Panneau de configuration de Tupac. . . . . 48

5.6 Panneau des résultats. . . . . 48

5.7 Panneau de visualisation de contrexemple. . . . . 49

6.1 Inter VS intraprocédural résultat positif . . . . . 55

6.2 Inter VS intraprocédural résultat négatif. . . . . 56

# Liste des sigles et des abréviations

---

API : (*Application Programming Interface*) Interface de Programmation d'Application

CFG : (*Control Flow Graph*) Graphe de flot de contrôle

CG : (*Call Graph*) Graphe d'appel

DG : (*Deep Graph*) Graphe Profond

FNR : (*False Negative Rate*) Taux de faux négatif

ICFG: (*Incomplete Control Flow Graph*) Graphe de Flot de Contrôle Incomplet

IDE : (*Integrated Development Environment*) Environnement de Développement Intégré

OBDD: (*Ordered Binary Decision Diagram*) Diagramme de Décision Binaire Ordonné

TNR : (*True Negative Rate*) Taux de Vrai Positif

# Remerciements

Je tiens tout d'abord à exprimer ma gratitude au tout puissant pour la réalisation de ce mémoire.

Je voudrais remercier mes Directeurs de recherche Michalis Famelis et Houari Sahraoui sans quoi ce mémoire ne serait pas abouti. Merci d'avoir été d'un soutien que ce soit moral ou financière, des conseils et lumières tout au long de mon séjour au laboratoire de Génie Logiciel de l'Université de Montréal.

Je tiens à remercier les membres de ce laboratoire pour leur gentillesse abondante et de m'avoir offert un lieu de travail paisible et de leur bienveillance, en particulier Vasco Soussa pour ses conseils enrichissants.

Enfin et surtout, ma famille et mes amis de leur soutien quotidien et de leurs encouragements, sans quoi tout ça n'aurait pas eu lieu.

# Chapitre 1

## INTRODUCTION

### 1.1. Contexte

La réutilisation est un concept fondamental en génie logiciel. De manière générale, cela prend la forme d'utilisation de bibliothèques de logiciels et d'infrastructure logicielle via une Interface de Programmation d'Application ("Application Programming Interface": API). Cette dernière expose des points de fonctionnalité, permettant au code client d'utiliser la librairie sans avoir besoin de connaitre ses internes.

Malgré ses avantages, l'utilisation d'une API n'est pas triviale. Un usage typique des librairies se limite rarement à l'utilisation d'une seule méthode de cette API. Supposons qu'il existe une API permettant d'écrire dans une ressource. Pour utiliser cette API, le développeur aurait besoin d'utiliser `open()` pour ouvrir la ressource, `write()` pour écrire, `flush()` pour nettoyer et finalement `close()` pour fermer la ressource. Utiliser ce genre de combinaison n'est pas forcément simple et exige des efforts cognitifs de la part du développeur. De plus, une méthode peut également être surchargé, c'est-à-dire définie pour plusieurs types de paramètres, ce qui complique davantage son utilisation.

### 1.2. Problématique et solution proposée

Apprendre à utiliser une API nécessite un investissement considérable en temps et en efforts [**17**]. En plus de la documentation, les développeurs s'appuient également sur des exemples et du soutien de la communauté en ligne. Cela est dû au fait qu'une API n'est pas qu'un amas de méthodes. C'est un outil complexe qui a été créé pour être utilisé selon des scénarios d'utilisation spécifiques.

Certes, ces scénarios d'utilisation peuvent être documentés à différents degrés de qualité, mais cela dépend de l'attention du créateur de l'API en question. Pour complémenter la documentation, plusieurs chercheurs se sont concentrés sur l'extraction de tels scénarios à partir des codes sources ouverts sous forme de *Patron d'utilisation d'API* .

En bref, les patrons d'utilisation peuvent être vus comme des groupes de méthodes qui sont souvent utilisés conjointement dans un code client [**31**]. Lorsque ces groupes incluent des propriétés temporelles tel que l'ordre d'appel, ils sont appelés des patrons temporels [**23**]

Lorsque des patrons d'utilisation d'API sont disponibles, qu'ils soient documentés manuellement ou automatiquement découverts, la question est de savoir comment tirer parti de ces modèles d'utilisation pour alléger la charge des développeurs, en vérifiant la conformité de leur code ou en les guidant lors de l'utilisation de l'API.

En pratique, les développeurs qui cherchent à alléger leur charge cognitive s'appuient généralement sur l'assistance des Environnements de développement intégrés ("Integrated Development Environment": IDE). Cette assistance peut être de diverses formes, notamment d'aide contextuelle qui affiche une partie de la documentation, de complétion de code pour augmenter la productivité ou d'un vérificateur de syntaxe en temps réel. Toutes ces fonctionnalités sont fournies localement au niveau même de l'instruction ou de la méthode. Par conséquent, ces fonctionnalités ne peuvent gérer des propriétés temporelles comme celles présentes dans les patrons d'usage. Ces derniers sont généralement utilisés en analysant des traces d'exécution obtenues par analyse dynamique.

Dans ce mémoire, nous voulons tirer parti de ces patrons d'utilisation pour augmenter la productivité des développeurs en les aidant à mieux utiliser les API, indépendamment de la qualité de la documentation disponible ou du support de la communauté en ligne. Pour ainsi réduire l'écart entre la recherche sur les patrons temporels et leur utilisation dans la pratique en vue de contribuer à la capacité des IDE à réellement alléger la charge cognitive des développeurs.

Pour ce faire, nous introduisons Temporal Usage PAttern Checker (Tupac), une approche qui permet aux développeurs de tirer parti des connaissances sur les patrons d'utilisation des API directement à l'intérieur de l'IDE, et cela pendant la phase de développement. À partir d'un ensemble de patrons d'utilisation, Tupac effectue une analyse statique du code client

tel qui est écrit et utilise un vérificateur de modèle pour générer des commentaires sur l'utilisation de l'API au développeur. TUPAC a été implémenté en tant que plugin Eclipse [1].

Nous avons effectué une évaluation empirique de l'exactitude et de la scalabilité de TUPAC. Nous avons constaté que, malgré le fait que nous n'utilisions pas d'analyse dynamique, TUPAC est tout aussi capable de détecter les violations d'utilisation avec en moyenne une précision de 0.87 et un rappel de 0.65. En termes de performance, nous avons observé que TUPAC prend en moyenne une demi-seconde pour vérifier un patron pour un projet entier.

## 1.3. Contribution

Une vue globale de notre approche est représentée par la Figure 1.1 où les principales étapes sont mises en évidence. Pour fournir un résultat, notre approche a besoin d'un code source ainsi que d'une série de patrons d'utilisation d'API écrite en format LTL comme entrée. Un modèle de représentation du code sera généré sous forme de GRAPHE PROFOND, qui sera ensuite traduit en un modèle SMV. La conformité de ce dernier au patron d'utilisation sera vérifié en utilisant un vérificateur de modèle NuXmv [**5**]. Les résultats seront par la suite formatés et affichés à l'utilisateur.

Les principales contributions de cette thèse sont :

(1) Une technique qui utilise l'analyse statique interprocédurale pour capturer l'exécution possible d'un code client qui utilise une API. L'analyse interprocédurale se base sur un GRAPHE PROFOND qui est généré à partir de la fusion du graphe d'appel et du graphe de flot de contrôle du code client ;

(2) Une approche qui effectue une vérification des modèles des patrons d'utilisation temporels sur un automate fini en effectuant une traduction du GRAPHE PROFOND du code client en un modèle SMV pouvant être utilisé par un vérificateur de modèle ;

(3) Un plugin Eclipse, qui consiste à aider les développeurs à exploiter les patrons d'utilisation temporels pendant l'écriture du code. Ce plugin est l'implémentation de notre approche et qui formate les résultats pour qu'ils soient directement intégrés dans l'interface d'Eclipse.

[1] `https://bitbucket.org/ErickFifa/artefacts/`

## 1.4. Organisation du mémoire

La suite de ce mémoire est organisée comme suit: le Chapitre 2 aborde les différents travaux existants en relation avec les patrons d'utilisation temporels. Ensuite, le Chapitre 3 présente un exemple illustratif. Le Chapitre 4 qui est le corps de ce mémoire contient les définitions formelles ainsi que les détails de l'approche. Le Chapitre 5 fournit des détails sur l'implémentation suivie par le Chapitre 6 qui discute la validation avec les détails concernant les méthodes utilisées et les résultats. Enfin, nous concluons dans le Chapitre 7.

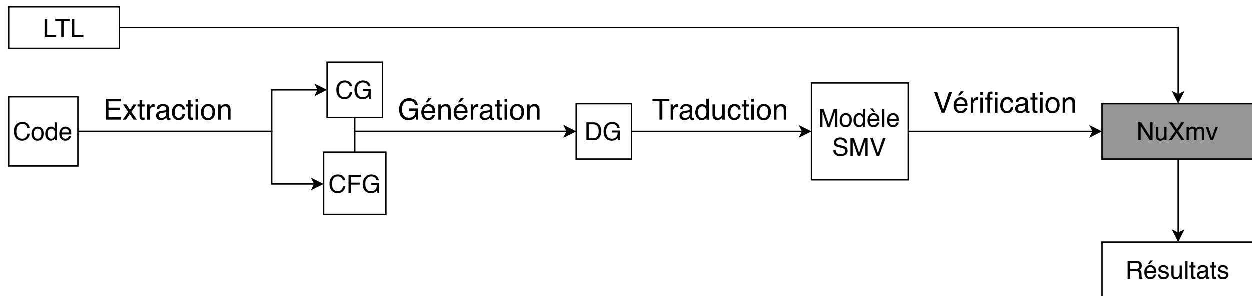



**Fig. 1.1.** Vue globale de notre approche.

# Chapitre 2

---

## CONTEXTE ET ÉTAT DE L'ART

### 2.1. Définition d'une API

En quelques mots, une API est une interface de programmation. Le mot-clé est “interface" car il démontre l'objectif d'une API qui est de permettre et de faciliter l'interaction entre l'Application et le programme qui veut l'utiliser.

Les API n'offrent pas toutes les mêmes fonctionnalités. De ce fait, nous pouvons les classifier en trois groupes:

(1) L'API pour les systèmes d'exploitation , celle qui sert d'interface entre le programme et le matériel. Elle permet de connaitre par exemple les interactions de l'utilisateur avec la machine (Déplacement de la souris, pression sur une touche, etc.);

(2) L'API pour langage de programmation, qui est le type qui propose diverses fonctionnalités au développeur. Elle peut prendre diverses formes. Pour n'en citer que quelques-unes, il existe des API de structure de données, des API de connexion, des API de cryptage, etc.

(3) L'API Web, le plus populaire, qui permet d'exploiter les services et données du site web. Par exemple, API Graph[1] et GitHub REST API v3[2] sont proposées par Github et Facebook pour permettre d'exploiter les données présentes sur leur site .

D'une certaine manière, nous pouvons dire qu'une API sert à élever le niveau d'abstraction pour que le développeur n'ait pas besoin de réinventer la roue et par la même occasion augmenter leur productivité.

---

[1] `https://developers.facebook.com/docs/graph-api/`

[2] `https://developer.github.com/v3/`

## 2.2. Assistance disponible pour les développeurs

Fournir des informations utiles aux développeurs est une chose à laquelle les chercheurs se sont consacrés de diverses manières. Les travaux de Rastkar*et al.* [**20**] et Murphy dans [**13**] soulignent le fait que fournir des connaissances fiables aux développeurs en fonction de leurs besoins est la façon dont des outils comme l'IDE pourraient mieux aider les développeurs dans leur tâche quotidienne.

De même, un travail de Robillard *et al.* [**21**] se concentrent sur la génération de connaissances contextuelles à la demande du développeur. Kersten *et al.* [**10**] filtrent les informations fournies par l'IDE pour éviter un afflux d'informations non essentielles vers le développeur qui affectera sa productivité.

Sur la même base, mais avec une assistance plus centrée sur l'API, Xu *et al.* [**29**] présentent MULAPI, un outil qui recommande des API liées aux fonctionnalités à partir de documents de requête de fonctionnalité.

Il existe d'autres approches qui ne consistent pas à donner des recommandations, mais à évaluer l'existence d'une utilisation non recommandée des API. Wen *et al.* [**28**] proposent MutAPI, un outil permettant de détecter les schémas de mauvaise utilisation des API. Utilisant l'analyse des mutations, Amann *et al.* [**2**] introduisent MuDETECT un détecteur d'utilisation hors du commun d'API qui utilise des graphes comme nouvelle représentation des utilisations d'API.

## 2.3. Patron d'utilisation d'API

Les développeurs utilisent fréquemment des séquences d'instructions stylisées pour utiliser les API. Nous appelons ces séquences stylisées *patron d'utilisation d'API*. Les patrons d'utilisation peuvent être très utiles pour développer des systèmes utilisant des API. D'une part, les patrons peuvent être utilisés pour recommander des séquences d'appels d'API afin d'aider les développeurs à réaliser une tâche particulière. L'apprentissage de l'utilisation correcte des API est ainsi facilité d'autant plus que nous savons que la découverte de sous-ensembles d'API pertinents pour une tâche donnée est difficile [**22**].

D'autre part, si nous supposons que les patrons représentent une utilisation correcte de l'API, nous pouvons nous en servir pour trouver des utilisations hors du commun de l'API dans un code client. Cela peut aider pendant les phases de développement telles que

l'assurance qualité (par exemple, avec les révisions de code) et la maintenance du système (par exemple, pendant le débogage).

Prenons par exemple l'utilisation de l'API `java#util#Iterator` : quel que soit le scénario d'application particulier, nous utilisons généralement `hasNext()` d'abord pour vérifier s'il y a un élément à l'itération, puis déplacer le pointeur de l'itérateur en utilisant `next()`. Il peut exister de multiples variations de ce patron, que nous appelons *alternatives*. Dans l'exemple précédent, nous avons différentes alternatives selon le type de boucle (`while`/ `for`) ou l'ordre d'itération (par exemple, en utilisant `hasPrevious()` et `previous()`).

En pratique, trouver des patrons d'utilisation nécessite beaucoup d'efforts de la part des développeurs. En effet, la documentation de l'API est généralement basée sur les unités. Dans la documentation de `java#util#Iterator`[3] le mode d'utilisation décrit précédemment n'est expliqué ni dans la documentation de l'interface Iterator ni dans la documentation de `hasNext()` ou `next()`. Par conséquent, les développeurs doivent s'appuyer sur des exemples d'utilisation fournis par la communauté. Et si l'API de l'Iterator Java est largement connue et comprise, ce n'est probablement pas le cas pour toutes les autres API.

## 2.4. Extraction des patrons d'utilisation

La découverte et l'extraction des patrons d'utilisation des API sont des tâches non triviales et qui exigent beaucoup d'efforts de la part des développeurs. Ainsi, plusieurs chercheurs ont essayé différentes techniques pour récupérer les modèles d'utilisation des API.

### 2.4.1. Recherche sur les patrons d'utilisation non ordonnés

Zhong *et al.* [**31**] ont présenté l'outil MAPO, un "framework" pour l'extraction des patrons d'utilisation de l'API à partir des dépôts open source. Nam *et al.* [**14**] ont présenté un algorithme d'extraction de patrons d'utilisation nommée MARBLE, qui se focalise en particulier sur les patrons de type "Boilerplate". C'est une section de code répétitif non ou très peu changé qui revient fréquemment. MARBLE a pour but de faciliter la détection de certains problèmes reliés à l'utilisation de l'API. FOCUS quant à lui est un système de

[3]`docs.oracle.com/javase/8/docs/api/java/util/Iterator.html`

recommandation proposé par Nguyen *et al.* [**17**] qui permet non seulement d'extraire des patrons d'utilisation d'API, mais aussi de proposer des suggestions au développeur concernant les appels de fonction et les modèles d'utilisation appropriés.

Un travail similaire est effectué par Nguyen *et al.* [**16**] qui est un outil de complétion de code sensible au contexte basé sur un modèle de patron d'utilisation fréquente de l'API. Saied *et al.* [**23**] propose d'identifier automatiquement les collections de bibliothèques qui sont couramment utilisées ensemble par le développeur afin de récupérer les modèles de co-utilisation des bibliothèques.

### 2.4.2. Recherche sur les patrons d'utilisation ordonnés

Une autre approche possible pour l'extraction des patrons d'utilisation est l'usage des patrons temporels logiques notamment, sous forme de Logique Temporelle Linéaire (LTL) [**9**]. Cette dernière est composée de variables propositionnelles, de connecteurs logiques booléens et de variables temporelles spécifiques à la logique temporelle modale, à savoir les opérateurs Suivant (X), Finalement (F), Globalement (G) et jusqu'à (U). Une variable propositionnelle dans un patron représente un appel à une méthode.

Yang *et al.* [**30**] utilisent un simple gabarit de propriétés *"$event_0$ toujours suivi de $event_1$"* basé sur le Modèle de réponse ("Response template") de Dwyer *et al.* [**7**] . Lo *et al.* [**12**] ont proposé une approche multiévènement qui utilise le modèle *"à chaque fois qu'une série d'évènements se produit, éventuellement, d'autres séries d'évènements se produiront ".* Saied *et al.* [**24**] nous avons proposé une approche non basée sur des modèles pour récupérer des patrons d'utilisation complexes et non triviaux en utilisant la programmation génétique.

### 2.4.3. Patrons d'utilisation pour assister les développeurs

Notre vision pour aider les développeurs à utiliser au mieux l'API en tirant parti de l'utilisation des patrons temporels [**19**] est de proposer des recommandations au cours des différentes phases de développement:

*Amélioration de la documentation*

La rédaction et la compréhension d'une formule LTL sont un défi, car elles nécessitent un haut niveau d'expertise en matière de logique formelle. Il est donc utile de traduire les

modèles de LTL dans un langage naturel plus facile à comprendre pour les humains. Les approches existantes prennent généralement une approche basée sur l'utilisation d'ensembles de modèles prédéterminés [**7, 15**]. Cependant, comme les patrons d'utilisation des API peuvent également être exploités automatiquement, nous aurons besoin de techniques indépendantes des gabarits. Une approche consiste à utiliser des techniques d'apprentissage automatique et de traduction basés sur des exemples [**27**]. Un autre défi consiste à intégrer les descriptions de modèles générées à la documentation existante de l'API.

*Rédaction du code*

Dans cette phase, nous voulons fournir un retour d'information aux développeurs en recommandant l'appel de méthode API et en générant également un avertissement à la volée en cas de violation potentielle d'un patron. Le processus diffère des phases de test et de compilation car cette fois, les développeurs sont encore en train d'écrire le code, ce qui signifie que nous devrons calculer les chemins d'exécution sur un CFG incomplet ("Incomplete Control Flow Graph":ICFG).

*Phase de compilation et de test*

Les développeurs devraient pouvoir exploiter les patrons pendant la phase de compilation et de test, complétant ainsi les messages de compilation et les suites de tests existantes par des connaissances supplémentaires sur l'utilisation des API. Cela leur permettrait de trouver des problèmes qui seraient autrement difficiles à détecter sans une analyse statique ciblée. Notre approche consiste à les aider à déterminer la méthode ou au moins la partie du code sur laquelle ils doivent concentrer leurs efforts pour utiliser correctement une API donnée.

Dans ce mémoire, nous allons nous concentrer sur cette phase où nous pourrions mettre en évidence une violation potentielle d'un patron qui constitue une mauvaise utilisation potentielle de toutes les méthodes API concernées.

Bien qu'un grand nombre d'études soient consacrées à la récupération des patrons d'utilisation des API. Les travaux concernant leur application dans la pratique sont en revanche peu nombreux, encore moins comme une implémentation dans un IDE tel qu'un plugin Eclipse. Le travail se rapprochant le plus à TUPAC est Grapacc [**16**], un plugin qui analyse statiquement le code client pour extraire son "groum" [**18**], une représentation graphique du code. Une recommandation est ensuite proposée au développeur en trouvant des similitudes

entre le groum généré et les groums disponibles dans la base de données de patrons d'utilisation. Toutefois, Grapacc ne fournit aucunne information quant à l'utilisation d'un API et se focalise surtout sur la recommendation de bouts de code.

Notre approche Tupac fournit des informations aux développeurs en vérifiant la conformité d'un patron d'utilisation pour les informer sur une éventuelle utilisation erronée des API ou dans le cas opposé, d'une bonne application du patron. Cela se fait en récupérant des informations du code client structurées en tant que graphe. Ces informations sont récupérées en effectuant une analyse statique du code client [**6**]. Tupac considère les patrons d'utilisation d'API sous forme de formule Logique Temporelle Linéaire (LTL), Un exemple de patron d'utilisation écrit en LTL est le patron $\phi_1$, décrit dans le Chapitre 3.

On vérifie la conformité du code écrit avec les patrons d'utilisation en faisant appel à un vérificateur de modèle NuXmv [**5**]. Cela peut se faire en traduisant le graphe représentant le code entant que langage d'entrée du vérificateur de modèle. Dans notre cas, en un Vérificateur de Modèle Symbolique (SMV). En résumé: NuXmv prend comme entrée un modèle SMV ainsi que des spécifications LTL, et retourne un résultat: positif si les spécifications sont respectées par le modèle ou dans le cas échéant, négatif avec un contrexemple.

# Chapitre 3

## EXEMPLE ILLUSTRATIF

Dans ce chapitre, pour mieux motiver et illustrer l'approche que nous proposons, nous considérons le scénario d'un développeur nommé Dev qui travaille sur une application mobile nécessitant un stockage persistant. Dev travaille sur la classe `DataManager` qui utilise l'API SQLite d'Android pour gérer la connexion entre la base de données et l'application. Il est important de souligner que Dev n'a pas d'expériences dans l'usage de cette API.

### 3.1. Mauvaise utilisation

Nous montrons un extrait de la classe `DataManager` de Dev dans Listing 3.1. La classe `DataManager` hérite de la classe API `SQLiteClosable`. Nous pouvons voir que dans l'extrait du code dans Listing 3.1, plus précisément à la ligne 5, Dev déclare une méthode `close()`. Cette dernière fait appel à la méthode `onAllReferenceReleasedFromContainer()` à la ligne 11. La documentation de l'API SQLite[1] décrit que la méthode `onAllReferenceReleasedFromContainer()` est utilisée pour libérer toutes les références de l'objet et peut éventuellement fermer la base de données si la dernière connexion est supprimée.

On peut observer que `DataManager` utilise `SQLiteCompiledSql` (ligne 3) c'est-à-dire des instructions SQL compilées . Cependant, comme décrit précédemment, la ligne 11 pourrait fermer la base de données. Si Dev était plus expérimenté avec l'API, il saurait qu'une erreur est générée si la base de données est fermée avant que toutes les déclarations SQL compilées ne soient finalisées. Comme Dev n'a pas d'expérience, il est peu probable qu'il écrive un test qui permettrait de découvrir le problème.

[1] `https://developer.android.com/reference/android/database/sqlite/SQLiteClosable`

**Listing 3.1.** Version originale de la méthode `close()`.

```
1  public class DataManager extends SQLiteClosable {
2      ...
3      Map<String, SQLiteCompiledSql> CQueries = Maps.newHashMap();
4      ...
5      public void close() {
6          Iterator<Entry<SQLiteClosable, Object>> iter = mPrograms.entrySet().iterator();
7          while (iter.hasNext()) {
8              Map.Entry<SQLiteClosable, Object> entry = iter.next();
9              SQLiteClosable program = entry.getKey();
10             if (program != null)
11                 program.onAllReferencesReleasedFromContainer();
12 }}}
```

## 3.2. Réparation possible du mauvais usage

Un développeur plus expérimenté pourrait désaffecter toutes les instructions SQL compilées avant de fermer la connexion à la base de données. Par expérience, il saurait que pour éviter l'erreur, il est nécessaire d'appeler la méthode `releaseSqlStatement()` avant que la dernière référence à `SQLiteClosable` soit enlevée.

Ce patron est respecté dans la version corrigée de la méthode `close()`, appelée `safeClose()`, présentée dans Listing 3.2. Dans cette version, avant d'invoquer la méthode `onAllReferenceReleasedFromContainer()`, nous faisons appel à la méthode `safeRelease()` dans la ligne 5, qui s'assure de libérer en toute sécurité les instructions SQL compilées (lignes 14-17).

## 3.3. Utilisation des patrons

La réparation que nous avons proposée dans la section précédente est un exemple d'utilisation qui pourrait être considéré comme une bonne pratique pour l'utilisation d'API. Il existe plusieurs techniques [**8, 24, 26**] qui permettent d'extraire automatiquement de tels patrons d'utilisation, de règles ou directives à partir de codes disponibles publiquement.

Pour notre exemple, la règle relative à la désaffectation des requêtes SQL compilées peut être représentée par la formule logique temporelle linéaire (LTL) suivante : $\phi_1$ :

$$\phi_1 = \ G(\neg rs) \rightarrow ((o \vee c \vee rr)\ U\ rs)$$

où les variables $rs$, $o$, $c$, et $rr$ représentent les méthodes suivantes:

**rs:** `SQLiteCompiledSql#releaseSqlStatement()`

**o:** `SQLiteClosable#onAllReferencesReleasedFromContainer()`

**c:** `SQLiteClosable#close()`

**rr:** `SQLiteClosable#onAllReferencesReleased()`

Ce patron suit le gabarit de propriété de l'universalité de Dwyer *et al.* [**7**] et peut être interprété comme suit : $rs$ doit être invoqué au moins une fois avant que $o$ ou $c$ ou $rr$ soient invoqués à leur tour. Les opérateurs $G$ (Globalement) et $U$ (Jusqu'à) sont des opérateurs LTL.

Notre objectif est de tirer parti de patron tel que $\phi_1$ pour aider Dev à pallier son inexpérience avec l'API SQLite. Cela pourrait se faire de différentes manières, par exemple en l'aidant à rédiger de meilleurs cas de test, par une analyse dynamique, ou en recommandant la partie de la documentation appropriée ou les extraits de code qui sont pertinents. Ici, nous nous concentrons sur l'amélioration immédiate de l'expérience de codage de Dev. Nous nous posons notamment les questions suivantes : pouvons-nous lui fournir des informations pertinentes en temps utile à propos de la conformité de son code par rapport à $\phi_1$ et tout cela directement dans son IDE ? De plus, pouvons-nous générer ce genre d'information même lorsque le code n'est pas complet ou ne peut être entièrement exécuté ?

Notre idée de base est d'effectuer une analyse statique du code client écrit par Dev. Le Graphe de flot de contrôle (CFG) de la méthode `Datamanager.close()`, présenté dans Figure 3.1, est un bon point de départ si nous pouvons utiliser un vérificateur de modèle standard tel que NuSMV [**4**] pour trouver un chemin d'exécution qui représente une susceptible violation d'un patron d'utilisation. Le fait de trouver un tel chemin indique qu'il y a un risque qu'une fois que le code est exécuté, le patron soit violé. Pour en revenir à notre exemple, nous pouvons facilement observer que le patron n'est pas respecté puisqu'il n'y a pas de chemin contenant $rs$, c'est-à-dire que tous les chemins d'exécution possibles sur le CFG représentent une violation du patron $\phi_1$.

Compte tenu des résultats fournis par le vérificateur de modèles, nous pourrions aider Dev de différentes manières. Nous pouvons l'informer que son code viole un patron d'utilisation de l'API, et lui montrer graphiquement le chemin d'exécution responsable trouvé par le vérificateur de modèle. Cela permettrait de le guider pour trouver le problème dans son code et par la même occasion le corriger. Cette visualisation graphique du parcours peut également aider à détecter tout comportement inattendu et montrer ainsi à Dev comment il devrait étendre la couverture de ses cas de tests. Par conséquent, les résultats de l'analyse statique pourraient contribuer à améliorer la mise en place de l'analyse dynamique. En bref, si nous pouvions donner à Dev ces informations directement dans l'IDE, cela pourrait l'inciter à déboguer davantage et à corriger son implémentation. Pour que cela fonctionne, nous devons nous assurer que l'analyse statique ne prend pas trop de temps, au risque de perturber le rythme de codage de Dev et ainsi affecter sa productivité.

Mais qu'en est-il de `safeClose()`, présenté dans Listing 3.2? Son CFG ne contient pas l'appel à `releaseSQLStatement` car il se trouve dans la méthode de l'aide `safeRelease()`. En d'autres termes, nous avons besoin d'un système plus sophistiqué, soit une technique d'analyse statique interprocédurale. Dans le chapitre suivant, nous décrivons Tupac, une technique qui supporte à la fois l'analyse statique intraprocédural et interprocédural des codes clients pour vérifier la conformité des patrons temporels d'utilisation des API, en respectant les contraintes de fonctionnement qui visent à ne pas perturber le rythme régulier du codage des développeurs dans leur IDE.

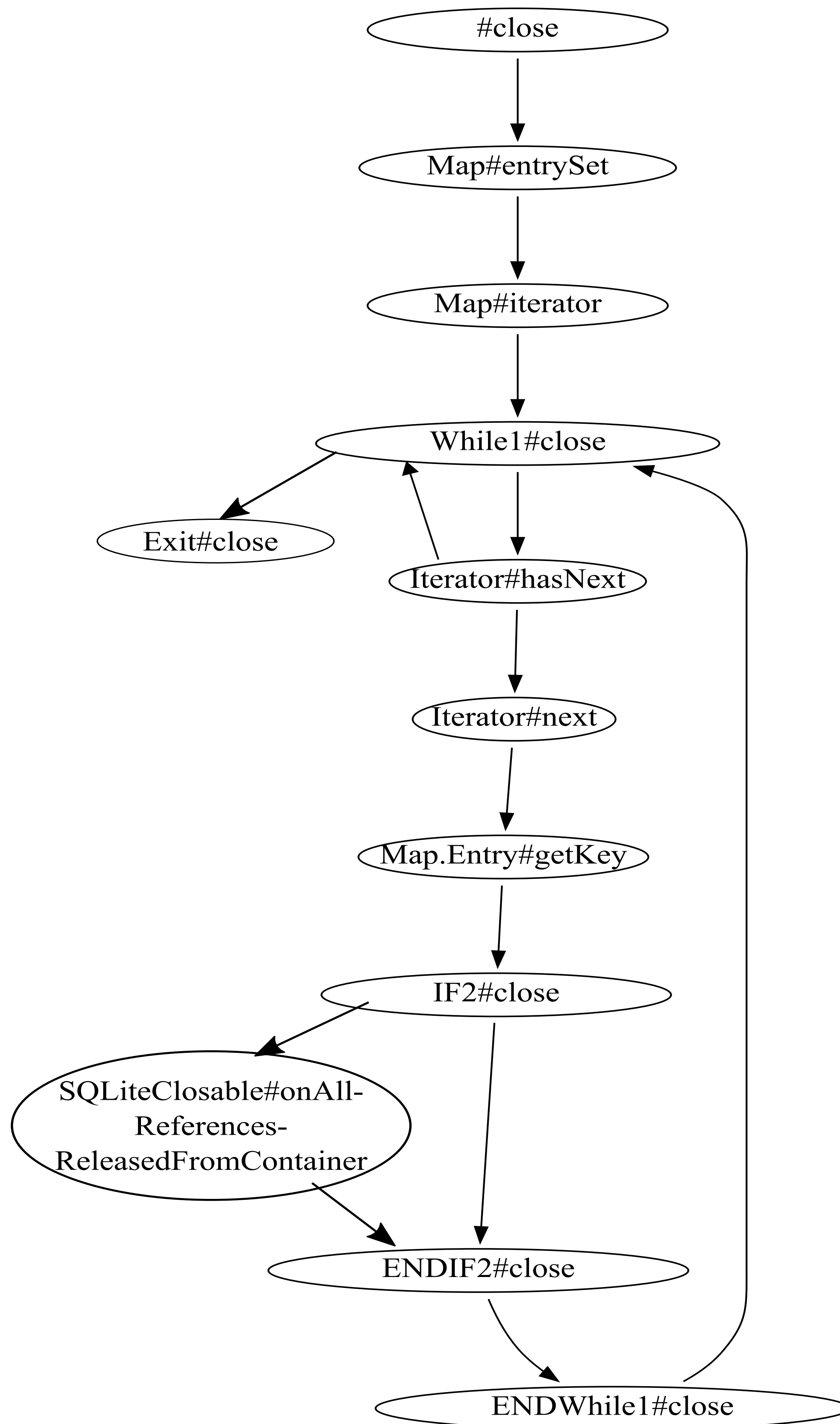



**Fig. 3.1.** Graphe de flot de contrôle (CFG) de la méthode `close()`.

**Listing 3.2.** Version correcte de la méthode `close()`.

```
1  public class DataManager extends SQLiteClosable {
2      ...
3      public void safeClose() {
4          Iterator<Entry<SQLiteClosable, Object>> iter = mPrograms.entrySet().iterator();
5          this.safeRelease();
6              while (iter.hasNext()) {
7                  Map.Entry<SQLiteClosable, Object> entry = iter.next();
8                  SQLiteClosable program = entry.getKey();
9                  if (program != null)
10                     program.onAllReferencesReleasedFromContainer();
11         }
12             onAllReferencesReleased();
13     }}
14     public void safeRelease(){
15         for (SQLiteCompiledSql compiledSql : this.CQueries.values())
16             compiledSql.releaseSqlStatement();
17         CQueries.clear();
18 }}
```

# Chapitre 4

## FORMULATION DE L'APPROCHE

Dans ce chapitre, nous décrivons les abstractions de code utilisées par TUPAC pour représenter des programmes. Nous supposons un paradigme orienté objet, selon lequel un logiciel est constitué d'un ensemble de classes, chacune d'entre elles contenant un certain nombre de méthodes, qui chacune d'entre elles à leur tour contiennent un ensemble ordonné d'instructions.

Nous désignons l'ensemble de toutes les méthodes d'un système par $\mathcal{M}$, et l'ensemble contenant l'union de toutes les instructions de toutes les méthodes par $\mathcal{I}$. Par souci de simplicité, nous supposons ici que les instructions enchainées, telle que l'instruction `mPrograms.entrySet().iterator()` dans la ligne 5 du Listing 3.1 sont déchainées (Comme par exemple:`x=mPrograms.entrySet() ; i=x.iterator()`)

## 4.1. Définition

### 4.1.1. Graphe de flot de contrôle (CFG)

Une abstraction que TUPAC utilise est le Graphe de flot de contrôle (CFG), un modèle de programmes informatiques qui permet de visualiser les traces d'exécution possibles [**1**]. Dans ce mémoire, nous utilisons la définition suivante:

**Définition 4.1.1** (Graphe de flot de contrôle (CFG))**.** *Le CFG de la méthode $k$ est l'ensemble:*

$$\langle M, M_0, M_f, F, n_{CFG} \rangle$$

*$M$ est un ensemble de noeuds représentant les instructions de $k$ sans arc entrant sauf pour l'entrée et sans arc sortant sauf pour la sortie; $M_0$ est le noeud initial tel que $M_0 \in M$*

*représentant le point d'entrée de k; $M_f$ est l'ensemble des noeuds de sortie tel que $M_f \in M$ représentant le point de sortie de k; $F$ est un ensemble d'arcs orientés tel que $F \subseteq M \times M$ représentant le flot de contrôle dans k; $n_{CFG}$ est une fonction de dénomination $n_{CFG} : M \rightarrow \mathcal{I}$ qui étiquète chaque noeud avec l'instruction qu'il représente*

Les déclarations conditionnelles, les boucles et les déclarations d'essai (try-catch) sont représentées par des noeuds qui ont plus d'un arc de contrôle sortant. Ils sont appelés *noeuds prédicat*. Pour simplifier la représentation, nous supposons que chaque noeud prédicat est associé à un pseudonoeud "END" qui lui est correspondant (par exemple, un `if` est associé à un `endif`, et ainsi de suite).

À titre d'exemple, le CFG de la méthode `DataManager.close()` de Listing 3.1 est présenté dans la Figure 3.1, où les noeuds sont étiquetés par les instructions correspondantes dans Listing 3.1. L'ensemble $M$ de ce CFG est l'ensemble de tous les noeuds ; $M_0$ est le noeud Map#entrySet ; $M_f$ est l'ensemble qui contient le noeud while1#close ; et l'ensemble $E$ est l'ensemble de tous les arcs.

### 4.1.2. Graphe d'appel (CG)

Chaque chemin sur ce CFG représente un possible chemin d'exécution de la méthode. Le CFG permet certaines formes d'analyse intraprocédurale (c'est-à-dire au sein d'une même méthode) mais ne fournit aucune information interprocédurale (c'est-à-dire entre les méthodes). Pour représenter les appels interprocéduraux, nous utilisons un type particulier de Graphe de flot de contrôle, le Graphe d'appel (CG), défini comme suit :

**Définition 4.1.2** ( Graphe d'appel (CG)). *Le CG d'un système est un graphe dirigé défini comme un tuple*

$$\langle V, V_e, E, n_{CG} \rangle$$

*$V$ est un ensemble de noeuds représentant les méthodes des classes du système ; $V_e$ est un noeud dans $V$ représentant le point d'entrée d'exécution du système ; $E$ est l'ensemble des arcs $\langle V_i, V_j \rangle$ représentant les relations appelant appelé entre les méthodes représentées par $V$ ; et $n_{CG}$ est une fonction de dénomination $n_{CG} : V \rightarrow \mathcal{M}$ qui étiquète chaque noeud avec la méthode qu'il représente.*

Nous partons du principe que $E$ ne contient que des relations appelant appelé qui respectent des règles de visibilité pertinentes. Nous limittons notre approche à l'analyse statique, par conséquent nous ne prenons pas en considération les appels polymorphes, l'injection de code, etc. Étant donné un sous-ensemble particulier de méthodes du système et une profondeur d'appel donnée, nous pouvons toujours construire une tranche finie. Dans ce qui suit, lorsque nous parlons de CG, nous faisons toujours référence à ces tranches finies pertinentes.

Par exemple, le CG de l'application de Dev est constitué des méthodes de ses classes, ainsi que des méthodes de l'API qu'elle utilise. En supposant que Dev utilise la version corrigée de `close()` de Listing 3.2, nous montrons une tranche de ce CG qui est centrée sur la méthode `safeClose()` dans la Figure 4.1. Dans cette tranche, la méthode `safeClose()` est le point d'entrée $V_e$, et l'ensemble $V$ ne contient que les méthodes de la classe `DataManager` et les appels à l'API SQLite d'Android. Dans la figure, nous indiquons les relations appelant-appel sous forme de flèches. Par exemple, puisque la méthode `safeClose()` appelle la méthode de l'API `SQLiteCloseable.onAllReferencesReleasedFromContainer()` les deux noeuds correspondants sont reliés par un arc.

### 4.1.3. Graphe profond (DG)

Bien que le CG permet de visualiser les dépendances interprocédurales entre les méthodes, il ne permet pas de déterminer le fonctionnement interne de chaque méthode appelée. Afin de pouvoir analyser à la fois les dépendances interprocédurales et le fonctionnement interne de chaque méthode, nous utilisons un hybride du CFG et du CG, que nous appelons Graphe profond (DG). Le DG combine les informations pertinentes du CFG et du CG pour permettre une analyse interprocédurale de l'utilisation des API.

**Définition 4.1.3** (Graphe profond (DG))**.** *Le DG d'un système est un graphe dirigé défini comme un tuple :*

$$\langle N, N_0, \Delta, n_{DG} \rangle$$

*$N$ est un ensemble de noeuds représentant des instructions dans les méthodes du système ; $N_0$ est un noeud tel que $N_0 \in N$ représentant le point d'entrée du système ; $\Delta$ est un ensemble d'arcs tels que $\Delta \subseteq N \times \mathcal{M} \times N$ qui étiquète chaque flot de contrôle avec la méthode dans*

*$\mathcal{M}$ qui l'a déclenchée; et $n_{DG}$ est une fonction de dénomination $n_{DG} : N \to \mathcal{I}$ qui étiquète chaque noeud avec la méthode qu'il représente. Un arc $\langle s,l,t \rangle$ dans $\Delta$ est représenté par $s \xrightarrow{l} t$*

Nous détaillons la construction du DG dans la Section 4.2. Intuitivement, la DG utilise le CG d'un système pour relier entre eux les CFG des méthodes. Le DG de la méthode `safeClose()` du Listing 3.2 est présenté dans la figure 4.4. Cette dernière combine le CFG des deux méthodes `safeClose()` et `safeRelease()` dans une représentation unique, en utilisant les relations appelant appelé décrit dans la tranche du CG centrée sur `safeClose()`.

Un chemin sur un CFG, un CG, ou un DG, représente un flot de contrôles possibles dans le système. Nous adoptons une perspective d'analyse statique et notre objectif est de raisonner sur les systèmes sans les exécuter. Ainsi, nous appelons ces chemins *traces* et même si ces chemins ne correspondent pas exactement aux traces réelles d'exécution du système ils représentent des exécutions potentielles du système et ont le potentiel de donner des indications sur l'utilisation correcte des API.

**Définition 4.1.4** (Trace)**.** *Un chemin d'exécution ou trace, est un ensemble d'exécutions possibles d'un programme $P$ selon un point d'entrée donné. Il est défini comme suit :*

$$T =< t_0,t_1,...,t_n >$$

*où $t_i$ est un ensemble de noeuds, de sorte qu'une seule trace commence à un point d'entrée $V_0$ et se termine à un noeud qui n'a pas d'arc sortant $V_e$.*

Le défi est maintenant de trouver ces traces d'exécution et de vérifier si chacune de ces traces respecte une spécification LTL donnée. C'est principalement la tâche des vérificateurs de modèle: vérifier si un modèle satisfait une propriété. Pour ce faire, nous utilisons un vérificateur de modèle symbolique (SMV).

**Définition 4.1.5** (Modèle SMV)**.** *Comme décrit dans* [**4**]*, un modèle SMV est un modèle de transition d'état ou structure de Kripke* [**11**]*. Elle est définie comme suit :*

$$M =< S,I,R,L >$$

*où $S$ est un ensemble fini d'états; $I$ un ensemble d'états initiaux tels que, $I \subseteq S$; $R$ la relation de transition telle que, $R \subseteq S \times S$; $L$ une fonction d'étiquetage de $S$ tel que, $L : S \to V$*

Avec un modèle SMV $M$ et une formule LTL $\phi$, le vérificateur de modèle trouve l'ensemble de tous les états qui satisferait $\phi$, à savoir, $s \in S|M,s \vdash \phi$. Les modèles SMV sont représentés en utilisant le langage de NuSMV[**5**]. Un programme satisfait $\phi$ si tous les $s$ satisfont $\phi$. Dans

le cas où $s_n \not\models \phi$, $s_n$ est considéré comme un contrexemple. C'est-à-dire une trace d'exécution du programme qui viole $\phi$.

Le modèle SMV de la méthode `safeClose()` du Listing 3.2 est présenté dans Listing A.1.

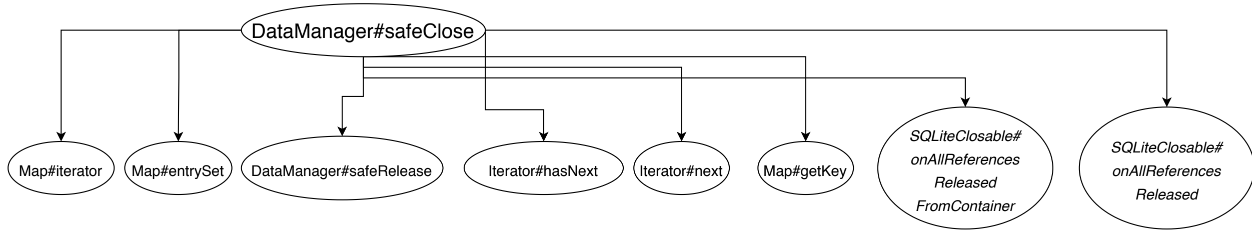



**Fig. 4.1.** CG de la méthode `safeClose()` dans Listing 3.2

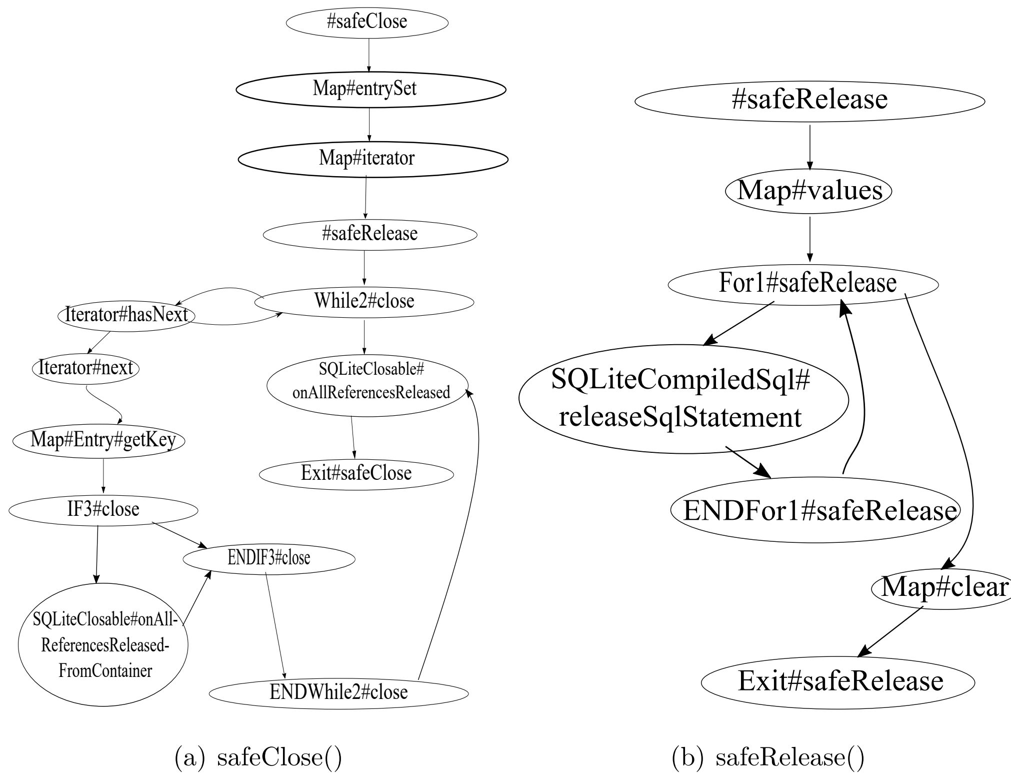



**Fig. 4.2.** CFG des méthodes dans Listing 3.2.

## 4.2. Détails de l'approche

Dans cette section, nous décrivons plus en détail notre algorithme pour la génération du Graphe profond ainsi que sa conversion en un Automate fini pour permettre la validation de la spécification.

### 4.2.1. Génération du Graphe profond

Pour construire le Graphe profond nous commençons par initialiser $N_0$ par le CFG du point d'entrée $V_e$ du CG. Chaque noeud du CFG sélectionné est ajouté au DG. Ensuite, le CFG est traversé de $N_0$ en suivant $F$. Chaque arc du DG est étiqueté en utilisant le nom de la classe et de la méthode faisant l'appel. Cette procédure est répétée pour chaque méthode de la classe du système, c'est-à-dire pour chaque élément de $V$. Cette procédure est suffisante pour une vérification intraprocédurale. Cependant, une autre étape est nécessaire pour pouvoir effectuer une représentation interprocédurale.

À ce stade , notre DG contient plusieurs amas de CFG avec des arcs étiquetés, mais il ne contient pas encore de liaison (arc) entre ces différents CFG. Pour ce faire, après que chaque CFG a été rajouté au DG et que les arcs sont nommés tels que décrit précédemment, nous sélectionnons le point d'entrée du système et nous suivons $F$. Chaque méthode $k$ est vérifiée si c'est une méthode du code client. Si c'est le cas, nous rajouterons un arc de $k$ vers son CFG et un autre arc est rajouté pour lier son point de sortie $M_f$ vers la prochaine méthode $next(k)$. À titre d'exemple, même si le CG (Figure 4.1) montre également les relations appelant appelé entre `SafeClose()` et `Iterator.hasNext()`, nous n'avons pas développé son CFG dans le DG puisque ce dernier n'est pas une méthode du code client. Au lieu de cela, il apparait comme un noeud régulier dans le DG. Cependant, nous avons développé le CFG de la méthode `safeRelease()` puisqu'il s'agit d'une méthode du code client et qu'il pourrait donc contenir des appels API. Les CFG de ces deux méthodes sont visibles dans la Figure 4.2.

Nous détaillons l'algorithme pour générer un Graphe profond dans Algorithme 1. Dans ce dernier, nous utilisons une procédure `prune(CFG)` qui consiste à enlever les noeuds non nécessaires. Cette procédure élague les instructions telles que les déclarations et nous permet de garder uniquement les noeuds qui contiennent des appels de méthode. De plus, pour les noeuds prédicats, nous rajoutons un ID pour bien les distinguer entre eux surtout lors des imbrications.

**Algorithm 1** Création du Graphe profond

```
1: procedure ConstructDG(S_CFG : {cfg : ⟨M, M_0, M_f, F, n_CFG⟩}, cg:⟨V, V_e, E, n_CG⟩ )
   Output: dg : ⟨N, N_0, Δ, n_DG⟩
2:     A) Initialisation
3:     g = getCFG(V_e).M_0
4:     prune(g)
5:     N_0 = g
6:     N = N_0
7:     Δ = ∅
8:     n_dg = ∅
9:     B) Construire une forêt de CFG annotés
10:    for each v ∈ V − N_0 do
11:        c = getCFG(v)
12:        prune(c)
13:        N := N ∪ c.M
14:        n_dg := n_dg ∪ c.n_cfg
15:        for each edge s → t ∈ c.F do
16:            Δ := Δ ∪ s →^v t
17:        end for
18:    end for
19:    C) Relier la forêt des CFG
20:    for each y ∈ S_CFG do
21:        for each s → t ∈ y.F do
22:            m_s = y.n_CFG(s)
23:            m_c = getCFG(m_s)
24:            if m_c = ∅ then continue        ▷ c.-à-d s est un prédicat ou un appel API
25:            end if
26:            if m_c = y then continue        ▷ nous ignorons les récursions
27:            end if
```

28: $l = y.n_{CFG}(y.M_0)$
29: $\Delta := \Delta \cup s \xrightarrow{l} m_c.M_0$
30: $\Delta := \Delta \cup m_c.M_f \xrightarrow{l} t$
31: $\Delta := \Delta - (s \xrightarrow{l} t)$
32: **end for**
33: **end for**
34: **return** dg:$\langle N, N_0, \Delta, n_{dg} \rangle$
35: **end procedure**
36: **procedure** GETCFG($v \in V$)
37: **for each** $y \in S_{CFG}$ **do**
38: **if** $n_{CG}(v) = y.n_{CFG}(y.M_0)$ **then**
39: **return** $y$
40: **end if**
41: **end for**
42: **return** $\emptyset$
43: **end procedure**

#### 4.2.2. Transformation du Graphe profond en modèle SMV

La transformation du GRAPHE PROFOND en un modèle SMV ou en tout autre *Automate fini* revient à traduire chaque élément du DG en différent élément du modèle (Tableau 4.1). Les règles de cette traduction sont comme suit:

- Le Graphe devient un module;
- Les éléments de $N$ deviennent les éléments d'une variable *State* de type énuméré;
- Chaque étiquette dans $\Delta$ devient une variable booléenne;
- $\Delta$ devient des règles de transition pour aller d'un "State" a un autre;
- Chaque élément de $\Delta$ : $(s \xrightarrow{l} t)$ devient une ligne dans les règles de transition suivant le gabarit suivant:

$$State = (s) \wedge l \wedge rules : \{t\}$$

- Pour chaque noeud avec plusieurs arcs de sortie, nous appliquons des contraintes de déterminisme: *rules*.

**Tableau 4.1.** Comment les concepts du Graphe profond sont représentés en syntaxe SMV

| DG | SMV |
|---|---|
| $N$ | Variable *State* de type énuméré |
| $N_0$ | *init(State)* |
| $\Delta$ | *Next(State)* |
| $n_{DG}$ | Chaque étiquette devient une variable booléenne utilisée dans *Next(State)* |

Ces contraintes de déterminisme servent à reproduire le flot de contrôle. Pour le contrôler, et éviter que des traces non souhaitées ne se produisent, des contraintes sont ajoutées.

Considérons un programme très basique présenté dans Listing 4.1. En observant le code, nous pouvons intuitivement déduire que l'ordre d'exécution de la méthode dans `Stack#manager` est : `push(),pop(),push()` et finalement `peek()`. Maintenant, si nous regardons son DG dans la Figure 4.3. Nous pouvons remarquer qu'à partir du noeud de départ `Stack#manager`, il y a plusieurs moyens possibles d'atteindre le noeud `Exit#manager`. Nous présentons ces traces dans le Tableau 4.2. Il est nécessaire d'ajouter des règles lors de la traversée du DG pour éviter que des traces non présentes dans le code ne se produisent. Ainsi, afin d'éviter que $t_1$ ne se produise, la transition dans l'automate doit contenir la règle suivante: la transition de `push` à `peek` est restreinte. Elle n'est autorisée que si la transition entre `push` et `pop` a été effectuée (c'est-à-dire vraie). Pour éviter de boucler entre `push` et `pop` (trace $t_2$), une restriction est ajoutée à la transition `push` $\rightarrow$ `pop` de sorte que, une fois qu'elle est vraie, elle devient et reste fausse. Comme le DG est une représentation graphique que le développeur peut voir, ces règles prennent surtout leur importance lorsqu'elles sont appliquées à l'automate. Le modèle SMV de la méthode `manager()` de Listing 4.1 est disponible dans l'Annexe A dans Listing A.2.

Une autre considération est nécessaire concernant la condition des prédicats. Les prédicats permettent au moins deux chemins d'exécution, l'un où la condition est vraie et l'autre lorsqu'elle est fausse. Supposons que dans une méthode du programme $p$, il y ait deux prédicats, $IF_1$ et $IF_2$, où nous annotons les conditions comme étant $c_1$ et $c_2$ pour $IF_1$ et $IF_2$ respectivement. Si $c_1$ et $c_2$ sont disjoint exclusivement ($c_1 \oplus c_2$), ce qui veut dire qu'ils ne peuvent pas être vrai en même temps, alors l'instruction dans $IF_1$ et $IF_2$ ne peut pas

être exécutée dans les deux cas, même si dans le Graphe profond cela reste un chemin d'exécution possible. Étant donné que nous n'examinons pas ces conditions, nous n'avons aucun moyen de déterminer si elles sont mutuellement exclusives.

**Listing 4.1.** méthode manager() de la classe Stack

```
1  public class Stack {
2    public void manager() {
3      push(data);
4      pop();
5      push(data);
6      peek();
7    }}
```

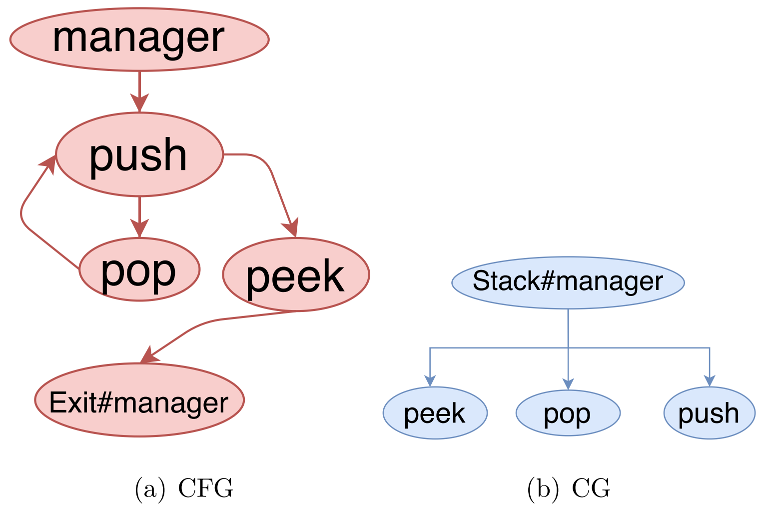
**Tableau 4.2.** trace d'exécution possible.

| Trace | Ordre d'appel |
|---|---|
| $t_0$ | push,pop,push,peek |
| $t_1$ | push,peek |
| $t_2$ | push,pop,push,pop,... |

### 4.2.3. Vérification du modèle

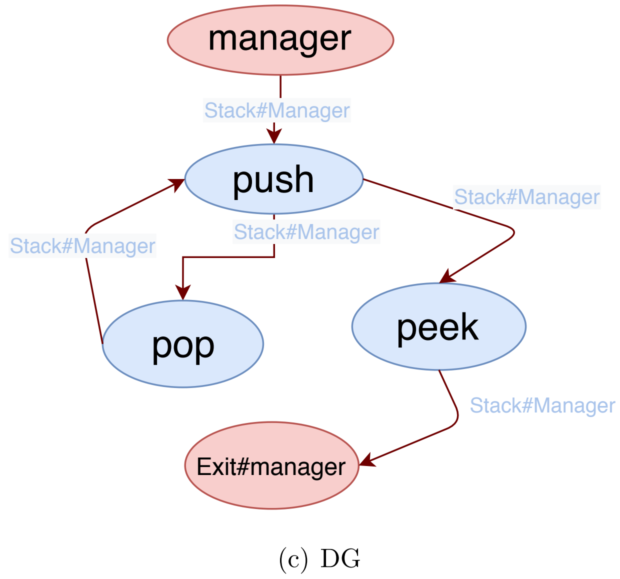
La vérification de modèle est une sorte de test qui vise à trouver rapidement une violation des propriétés. Ici, les propriétés représentent une bonne utilisation des API. Le modèle à vérifier est le modèle SMV qui a été créé à partir du DG. Par conséquent, il contient toutes traces d'exécution possible du programme. Le vérificateur de modèle que nous utilisons pour cette approche est un vérificateur de modèle symbolique NuXmv [**4**].

Nous pouvons nous attendre à deux types de retours de NuXmv. (1) Le patron a été vérifié et aucune trace ne viole le patron, nous pouvons alors considérer que le programme ne possède aucune mauvaise utilisation. Dans ce cas, NuXmv retourne la réponse "TRUE". (2) Le patron est violé, ce qui veut dire qu'une ou plusieurs traces d'exécution représentent une mauvaise utilisation de l'API. Si c'est le cas, NuXmv retourne un contrexemple qui équivaut à la trace d'exécution du programme qui viole le patron.




(a) CFG

(b) CG




(c) DG

**Fig. 4.3.** Graphes de la méthode `manager()` dans Listing 4.1.

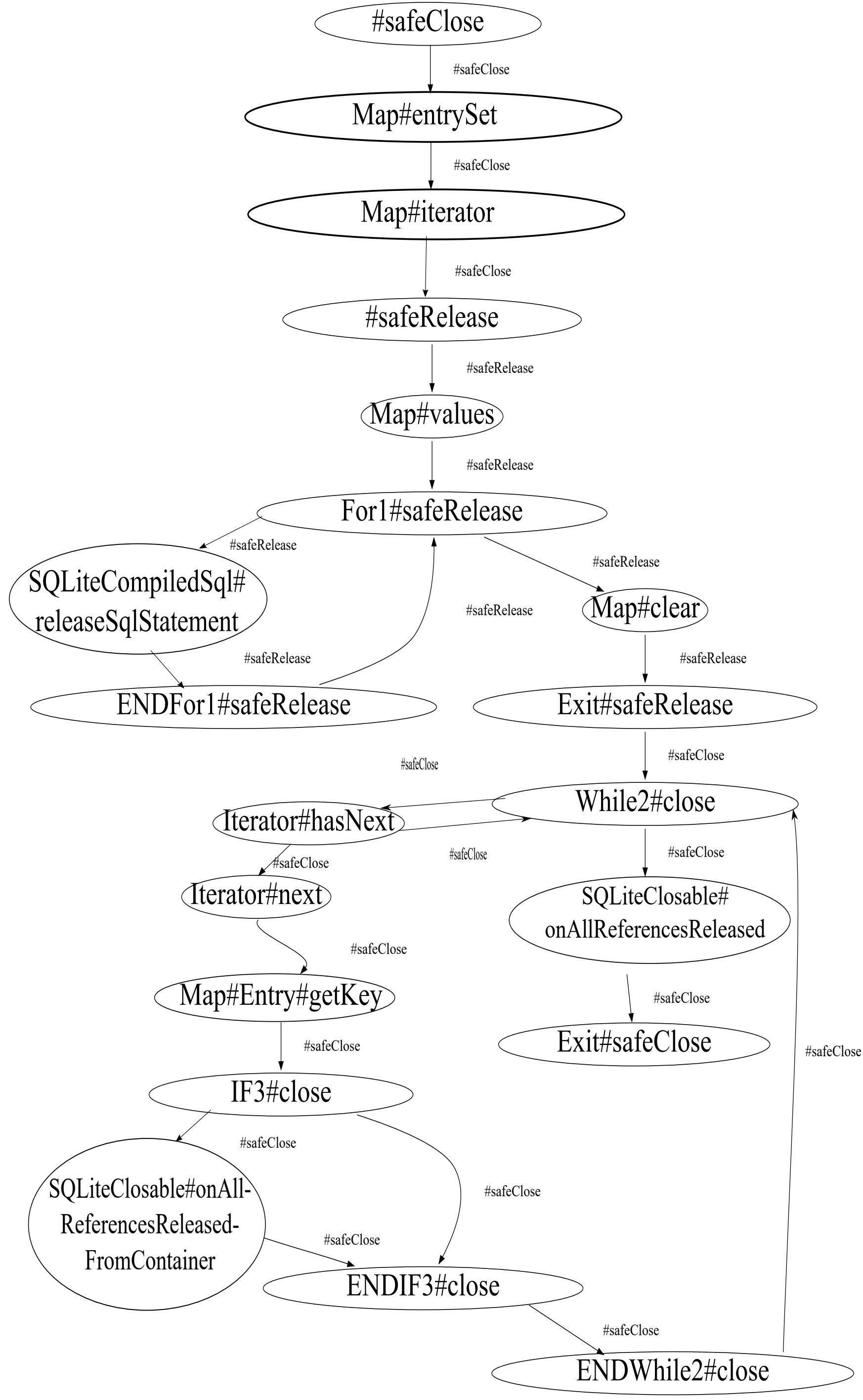



**Fig. 4.4.** DG de la méthode `safeClose()` dans Listing 3.2.

# Chapitre 5

---

# IMPLÉMENTATION

Dans ce chapitre, nous discuterons brièvement l'implémentation de cette approche. Figure 5.1 montre l'architecture globale de notre approche avec trois modules principaux: (1)IDE, (2)TUPAC, (3)vérificateur de modèle.

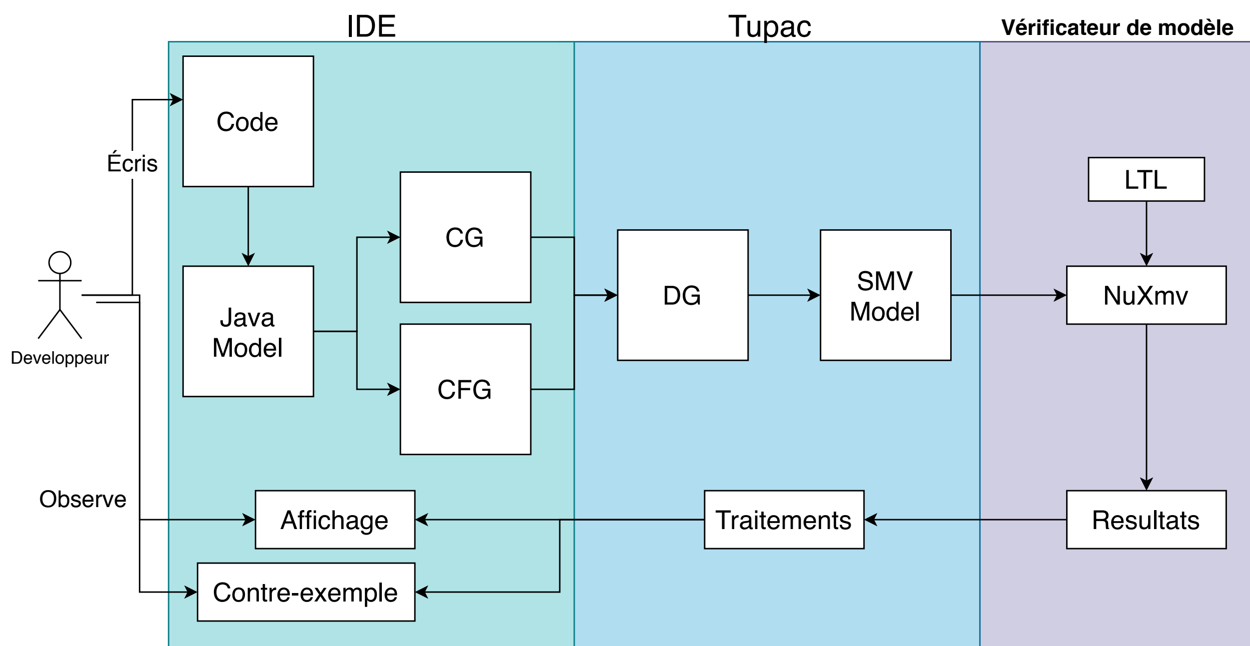



**Fig. 5.1.** Architecture globale de notre approche.

## 5.1. Analyse du code Java

Eclipse offre une API qui se trouve dans le paquet `org.eclipse.jdt.core`[1] Bien que cette API permet la création, l'édition et même la construction d'un programme Java, nous nous en servirons uniquement pour faire l'analyse du code client.

La première étape de l'implémentation est de pouvoir analyser le code client. Pour ce faire nous utilisons une représentation appelée "Java Model" d'un projet Java. En effet, chaque projet Java est représenté en interne par Eclipse en utilisant ce modèle. Ce modèle est une représentation légère donc rapide à générer et est aussi tolérant aux fautes ce qui est nécessaire pour traiter les codes encore incomplets. Ce modèle est utilisé par exemple pour générer la vue "Breadcrumb" dans Eclipse. L'API *eclipse.jdt* considère chaque composant du projet Java en un `élément`. Chaque élément correspond à une interface *IJavaElement* Figure 5.2 montre une description des IJavaElement dans l'IDE.

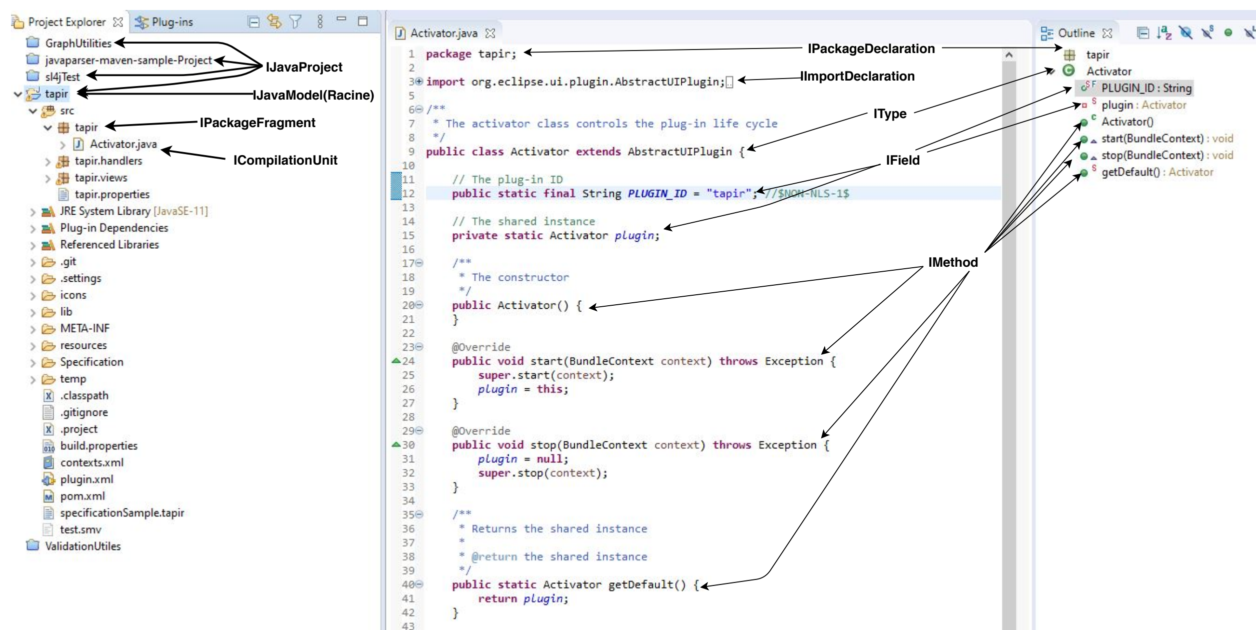



**Fig. 5.2.** Les IJavaELement dans Eclipse.

Les éléments qui nous intéressent le plus sont: invocation et déclaration de méthode, prédicats et constructeur de classe. Les instanciations sont également utilisées pour résoudre le typage . Ces éléments ont des attributs différents selon leur type. Par exemple, les éléments qui sont des prédicats(if,while) ont un attribut condition.

[1]`https://www.eclipse.org/jdt/core/index.php`

**Listing 5.1.** Méthode writeAll()

```
1   public class Helper {
2     // ...
3    public void writeAll(List aList) {
4      Iterator<Object> it = aList.iterator();
5      while(it.hasNext()){
6          String result = it.next().toString();
7          Helper.writeInFile(result);
8      }}}
```

Supposons par exemple une méthode que nous montrons dans Listing 5.1. Afin de construire le GRAPHE PROFOND de la méthode `Helper#writeAll` nous traitons le modèle Java pour obtenir le CG et le CFG. Nous itérons à travers chacun de ces éléments. En partant de la ligne 4 de Listing 5.1, cet élément est une déclaration variable. Il faut d'abord résoudre son type, dans notre cas, la variable `it` est un type `Iterator`. Maintenant, chaque fois que `it` appelle une méthode, nous allons l'attribuer à son type, c'est-à-dire `Iterator`. La même procédure est suivie pour chaque argument de la méthode `Classe#writeAll`. Ensuite, nous voulons analyser le côté droit de la déclaration en tant qu'élément. Comme nous pouvons le constater, il s'agit de `aList#iterator()` et c'est un élément de type invocation de méthode, la procédure est la suivante :

- Vérifier si l'appelant n'est pas une autre invocation de méthode. Dans ce cas, l'appelant est `aList` qui n'est pas une méthode. Si c'était le cas, alors nous le traiterons comme un élément;
- Convertir l'appelant en son type. Dans notre cas, le type `alist` est donné dans la déclaration `writeAll`, c'est un type `List`;
- Ajouter un noeud sous la forme de : type d'appelant + nom de la méthode. Pour cet exemple, il s'agit de `List#Iterator`;
- Ajouter un arc au noeud récemment ajouté et au noeud précédent;
- Annoter l'arc précédemment ajouté en utilisant le nom de la classe#Méthode où l'appel a lieu

À ce stade, nous en avons fini avec la ligne 4 et nous passons à la suivante. La ligne suivante est une boucle *While*. Les attributs de la déclaration While sont: (1) la condition de l'élément et (2) le corps de l'élément, qui est le contenu de la boucle. La procédure est la suivante :

- Ajoutez un noeud au graphique avec le nom du prédicat : while avec un ID;
- Ajouter un arc au noeud récemment ajouté et au noeud précédent;
- Traiter la condition de la boucle entant qu'élément. Dans notre cas, `it#hasNext()`, En suivant la même procédure pour traiter les invocations de méthodes, nous obtenons un noeud `Iterator#hasNext()`. De plus, un arc supplémentaire est ajouté du noeud de l'élément conditionnel au noeud while;
- Analyser le corps de l'élément (le contenu du while) comme un ensemble d'éléments;
- Pour terminer la boucle `While`, nous ajoutons un noeud "End" pour indiquer la fin de la boucle;
- Enfin, nous marquons la fin de la méthode en ajoutant un noeud "Exit" au graphique.

Selon le type de boucle, des étapes supplémentaires sont nécessaires pour encapsuler complètement l'exécution conditionnelle d'un programme.

- **Élément de type *While* et *For*:** ajouter un arc entre le noeud final et le noeud prédicat. Ensuite, ajoutez un arc entre le noeud de prédicat et l'élément suivant.
- **Élément de type *if*:** ajouter un bord entre le noeud if et son noeud final correspondant.
- **Élément *Try-catch*:** pour chaque noeud créé à l'intérieur du "Try", ajouter un bord au début du noeud de "Catch"

Les étapes décrites précédemment ne sont que des aperçus très au niveau des différentes phases nécessaires à la création d'un DG. Le code source de TUPAC est disponible pour avoir plus de détails.

## 5.2. Vérificateur de modèle

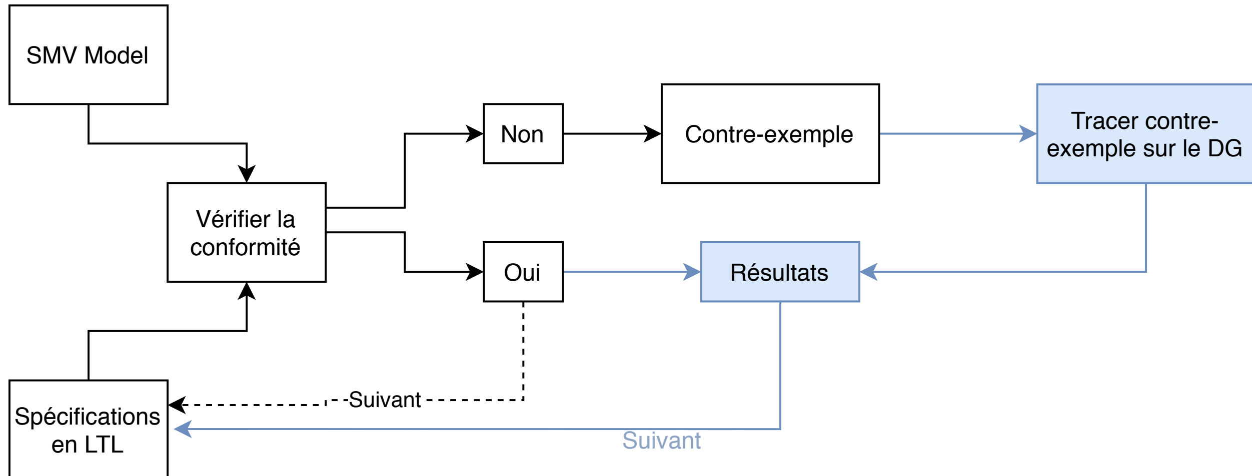



**Fig. 5.3.** Processus de vérification de modèle.

La vérification de modèle est une méthode formelle qui utilise les mathématiques et la logique pour décrire et surtout vérifier des systèmes. Un vérificateur de modèle utilise la recherche d'espace d'état pour vérifier si un système à états finis satisfait des spécifications données. C'est un processus de vérification entièrement automatique. Toutefois l'explosion de l'espace d'état est un des problèmes majeurs de la vérification de modèle. Bryant *et al.* [**3**] suggère que l'utilisation des diagrammes de décision binaires ordonnés ("Ordered Binary Decision Diagram":OBDD) peut résoudre efficacement ce problème.

Le vérificateur de modèle NuXmv [**4**] utilisé par Tupac est un vérificateur de modèle symbolique basé sur les OBDD. Figure 5.3 montre une vue globale du fonctionnement de NuXmv en noir, ainsi que les processus supplémentaires nécessaires pour Tupac différencié en bleu.

Effectivement, le vérificateur de modèle NuXmv nécessite quelques modifications pour pouvoir être utilisé par Tupac. L'un des problèmes majeurs est son exécution. Il a été créé pour vérifier un modèle sans interruption en un seul processus, ce qui veut dire que la moindre malformation d'une spécification peut faire arrêter le processus et par conséquent arrêter la vérification.

Ce genre de malformation peut arriver dans le cas où l'une des méthodes dans la spécification n'est pas présente dans le modèle. Considérons par exemple la méthode `writeAll()`

dans Listing 5.1 et un patron $p$ tel que Iterator#previous ne soit jamais appelée aprés Iterator#hasNext ce qui donne en LTL (en se basant sur le gabarit absence de Dwyer[**7**]):

$$G(state = Iterator\#hasNext \rightarrow G(!state = Iterator\#previous))$$

Or, comme nous pouvons le constater dans Listing 5.1, la méthode `Iterator#previous` n'est jamais appelée.

Si nous vérifions la conformité de la méthode `writeAll()` à la spécification $p$, NuXmv retournerai `"undefined identifier 'Iterator#previous'"` ce qui est normal puisque cette méthode n'est jamais utilisée. Malgré tout, cela est considéré comme une erreur et arrête le processus. Pour contourner ce problème, il nous est nécessaire de vérifier chaque spécification de manière séparée.

Cela engendre plusieurs conséquences néfastes. Pour commencer, du point de vue de la performance, il est important de savoir qu'avant chaque vérification, le modèle doit être initialisé. Cette tâche n'est normalement faite qu'une fois avant de vérifier le set de spécification. Toutefois, comme il faut vérifier chaque spécification séparément, le modèle doit être initialisé autant de fois que le nombre de spécifications à vérifier. De plus, cela introduit une certaine ambigüité dans la réponse fournie. Nous avons décidé de considérer les patrons qui contiennent des méthodes non définies dans le modèle comme *ne s'applique pas*. En d'autres termes, $p$ ne s'applique pas à la méthode `writeAll()` dans Listing 5.1, ce qui n'est pas tout à fait exact puisqu'aucune trace d'exécution de cette méthode ne représente une violation de $p$.

## 5.3. Interface utilisateur

Tupac a été créé dans l'optique de ne pas perturber la productivité des développeurs. De ce fait, nous avons implémenté notre approche en tant que plug-in. Ainsi, toutes les configurations nécessaires ainsi que les résultats sont faits à l'intérieur même de l'IDE pour que le développeur n'ait pas besoin de changer de contexte. Toutes les interfaces que Tupac propose sont accessibles dans le menu vu d' Eclipse (Figure5.4)

Comme nous l'avons mentionné dans la Section 4.1, le développeur doit spécifier diverses informations à Tupac comme la classe et la méthode qui sera le point d'entrée, la liste des spécifications, le dossier contenant l'installation de NuXmv ainsi que le type de vérification

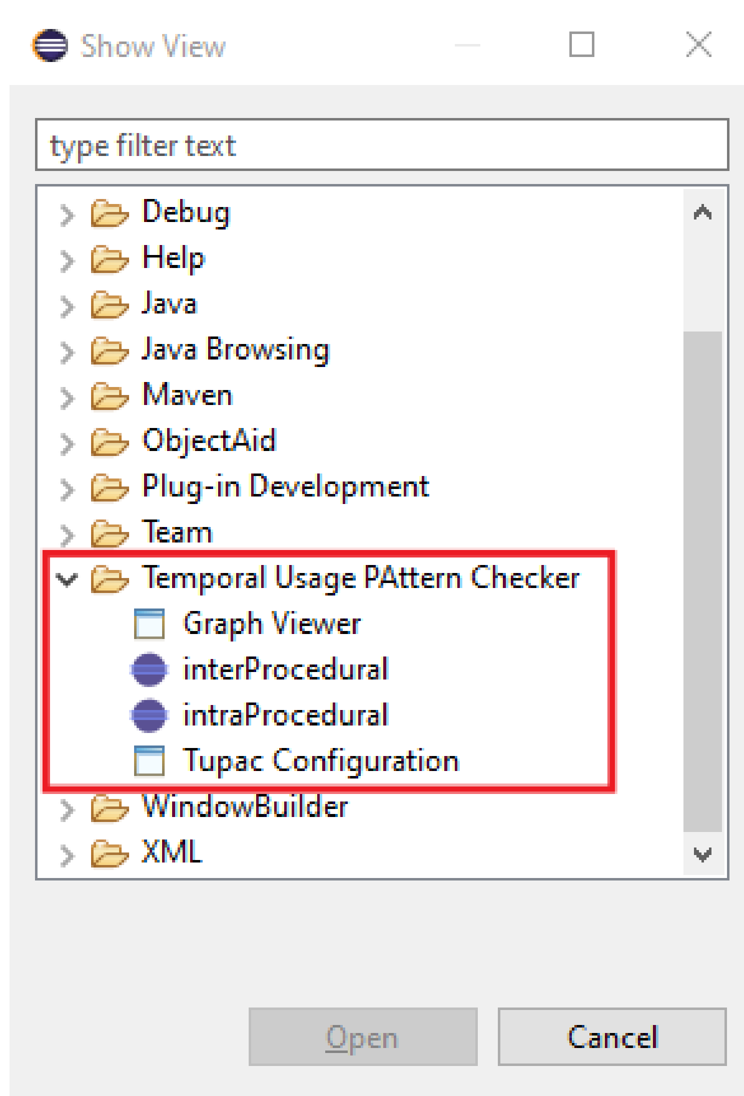



**Fig. 5.4.** Menu affiché vue d'Eclipse

(interprocédural ou intraprocédural). Figure 5.5 offre un aperçu du panneau de configuration de Tupac.

Après avoir fait les configurations au préalable, le développeur peut lancer la vérification à chaque fois que l'éditeur est sauvegardé. La motivation derrière cela est que nous considérons que lorsqu'un développeur sauvegarde son code, il est arrivé à un certain point où il a accompli une tâche ou au moins une partie. Le fait est qu'un développeur est moins susceptible de sauvegarder au milieu de l'écriture d'une instruction.

Comme le démontre Figure 5.6, les résultats sont affichés dans un panneau sous forme de tableau où chaque patron est numéroté et est suivi d'une brève description si disponible.

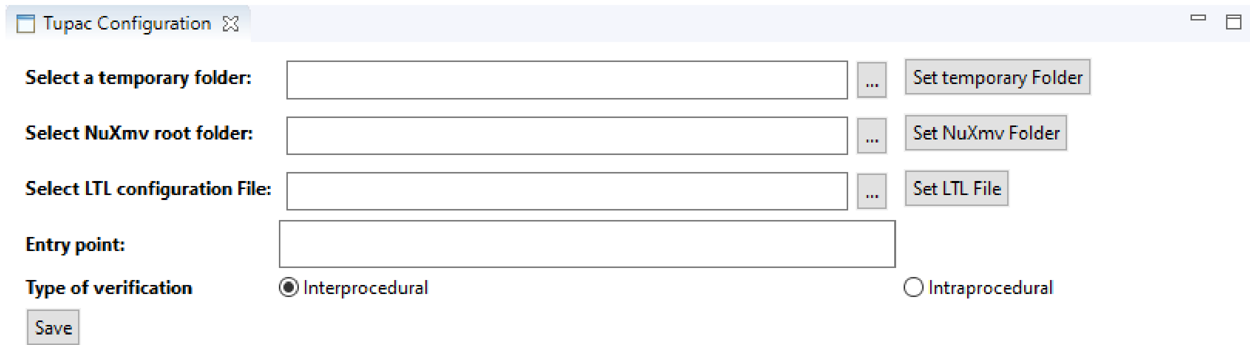



**Fig. 5.5.** Panneau de configuration de Tupac.

interProcedural

| ID | Pattern | Result | Description |
|---|---|---|---|
| 0 | LTLSPEC F(State = File#listFiles) -> ((!(State = File#isDirectory) & !(State = File#l... | True | |
| 1 | LTLSPEC F(State = File#isDirectory) -> (!(State = File#listFiles) U (State = File#is... | True | |
| 2 | LTLSPEC G(((State = List#get) & !(State = List#size) & F(State = List#size)) -> ((S... | True | |
| 3 | LTLSPEC F(State = List#get) -> (!(State = List#get) U ((State = List#add) & !(Stat... | True | |
| 4 | LTLSPEC G((!(State = List#add) & (State = List#get)) -> (!(State = List#size) W (S... | True | |
| 5 | LTLSPEC G((!(State = List#add) & (State = List#get) & F(State = List#add)) -> (!(... | True | |
| 6 | LTLSPEC G(((State = List#get) & !(State = List#size) & F(State = List#size)) -> (!(... | True | |
| 7 | LTLSPEC G(((State = List#size) & XF(State = List#add)) -> XF((State = List#add) ... | True | |
| 8 | LTLSPEC G(((State = List#size) & XF(State = List#get)) -> XF((State = List#get) ... | True | |
| 9 | LTLSPEC G((State = List#get) -> G!(State = List#size)) | True | |
| 10 | LTLSPEC G!(State = List#add) \| F((State = List#add) & (!(State = List#size) W (Sta... | True | |

**Fig. 5.6.** Panneau des résultats.

Pour visualiser un contrexemple, les résultats de NuXmv sont mis en évidence sur le DG pour que le développeur puisse prendre connaissance du chemin d'exécution qui viole le patron et ainsi faire les changements nécessaires. Comme démontré dans Figure 5.7, le chemin d'exécution qui viole un patron donné est représenté par les arcs en rouge.

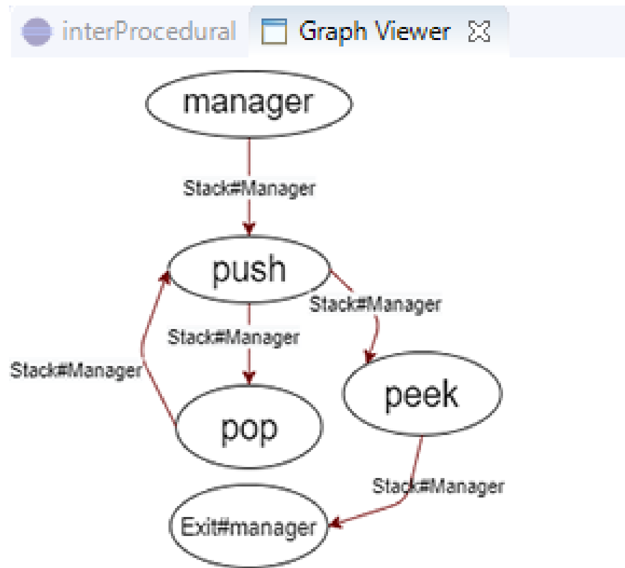



**Fig. 5.7.** Panneau de visualisation de contrexemple.

# Chapitre 6

## VALIDATION

Afin de pouvoir évaluer notre approche et ainsi déterminer sa précision aussi bien que son utilité, nous avons défini les questions de recherche suivantes :

- **RQ1:** Est-ce que notre approche est capable de détecter les violations de patrons d'utilisation à partir d'une analyse statique ?
- **RQ2:** Est-ce qu'une analyse de type interprocédurale offre une meilleure performance comparée à une simple analyse intraprocédurale?
- **RQ3:** TUPAC est-il assez rapide pour fournir des résultats utilisables au développeur sans être une entrave à sa productivité?

### 6.1. Cadre expérimental

Nous avons utilisé quatre programmes Java open source, provenant d'un site communautaire Programcreek [1] pour extraire des patrons d'utilisation à partir de quatre API largement utilisées indiquées dans le Tableau 6.1.

Pour extraire les patrons d'utilisation à partir de ces projets, nous avons commencé par isoler les appels de méthode d'API à partir des traces d'exécution. Ces dernières sont recueillies à partir de scénarios d'utilisation typiques. Nous avons par la suite extrait séparément les patrons par API pour chaque projet. Ensuite, nous avons sélectionné tous les patrons qui se chevauchent, c'est-à-dire les patrons qui sont présents dans chaque projet. Nous avons considéré ces patrons comme étant de "bons" modèles d'utilisation de ces API.

[1] `programcreek.com`

Désormais pour éviter toute confusion, nous allons désigner ces "bons" patrons d'utilisation par le terme $p_{API}$. Le Tableau 6.2 offre un aperçu des patrons que nous avons traités pour chaque projet et le nombre de $p_{API}$ que nous avons déduit de ces traces d'exécution.

Nous avons fait une analyse manuelle de chaque projet avec $p_{API}$ pour pouvoir faire une comparaison avec les résultats de notre approche. Il y a deux possibilités de résultat lors de la vérification d'un patron $p$. Elles sont définies comme suit :

- **Positif:** Le patron est respecté;
- **Negatif:** Le patron n'est pas respecté. Cela peut arriver selon deux cas de figure: (1) Au moins une trace d'exécution du programme enfreint $p$ ou (2) $p$ ne s'applique ou n'est pas pertinent pour le programme

Pour compléter notre évaluation, nous avons aussi simulé un usage typique des développeurs. Pour ce faire, nous avons effectué une mutation de chaque projet en utilisant les mutations suivantes:

- **Permutation**: les méthodes d'API ainsi que les méthodes qui contiennent des méthodes d'API sont permutées de place. Cela a été fait pour simuler une mauvaise sélection de méthode de l'API à utiliser fait par le développeur;
- **Suppression**: les méthodes d'API sont supprimées pour simuler un oubli ou une omission;
- **Duplication**: les méthodes contenant un appel de méthode d'API sont dupliquées au niveau interprocédural pour reproduire un comportement inattendu pendant l'exécution.

Nous avons aussi évalué l'utilisabilité du plugin TUPAC en termes de temps pour générer le GRAPHE PROFOND et fournir le résultat aux développeurs. Nous avons observé le temps de calcul requis par TUPAC pour fournir des résultats. Tout d'abord, nous avons mesuré le temps nécessaire à la génération du GRAPHE PROFOND. Une fois que ce dernier est généré, nous avons enregistré le temps d'exécution du vérificateur de modèle pour produire le résultat.

Il est important de noter que nous avons séparé l'observation du temps pour générer le GRAPHE PROFOND et le processus de vérification du modèle. La motivation derrière cela est issue d'une expérience pilote précédente où nous avons observé que NuXmv prend 92

pour cent du temps pour calculer le résultat parce qu'il nécessite l'utilisation d'IO pour fonctionner.

Nous avons conçu Tupac de manière à ce qu'il puisse fonctionner avec différents contrôleurs de modèle. Par conséquent, nous en avons déduit que la séparation des résultats offrira peut-être un aperçu plus intéressant.

Nous avons procédé comme suit, à partir du patron d'API que nous avons extrait, nous avons sélectionné au hasard divers patrons. La sélection a été faite en vue d'avoir une variété de patrons aux profondeurs variables. La profondeur du patron fait référence au nombre de propositions qu'il renferme. Plus il y a de propositions, plus un modèle est complexe et inévitablement prend plus de temps à vérifier. Un aperçu du jeu de données est montré dans le Tableau 6.3

| Projets | Variable | util#Map | util#List | util#Set | io#File |
|---|---|---|---|---|---|
| Html compressor | p1 | x | x | - | x |
| doc-to-pdf-converter | p2 | - | - | - | x |
| jar2Java | p3 | x | - | x | x |
| JTar | p4 | x | - | x | x |

**Tableau 6.1.** Projets et API utilisés pour la validation

| API | util#Set | util#Map | util#List | io#File | Chevauchement |
|---|---|---|---|---|---|
| nb. Patron | 12 | 1376 | 76 | 480072 | 26 |

**Tableau 6.2.** Nombre de patrons traités

| API | Nb. Patron | Profondeur des patrons |
|---|---|---|
| **io#File** | 72 | 5 à 11 |
| **util#List** | 14 | 2 à 7 |
| **util#Set** | 12 | 2 à 14 |
| **util#Map** | 26 | 2 à 8 |

**Tableau 6.3.** Taille des patrons test.

## 6.2. Résultats

### 6.2.1. *RQ1:* Est-ce que notre approche est capable de détecter les violations de patrons d'utilisation à partir d'une analyse statique?

Nous voulons évaluer l'exactitude des résultats fournis par TUPAC, pour voir s'ils sont utiles, c'est-à-dire s'ils fournissent réellement un aperçu pertinent plutôt que de semer la confusion chez le développeur en introduisant des informations erronées.

Pour ce faire, nous avons comparé les résultats de l'analyse manuelle à ceux de TUPAC. Les résultats présentés dans le Tableau 6.4 révèlent plusieurs détails intéressants.

| | **p1** | **p2** | **p3** | **p4** |
|---|---|---|---|---|
| **Positif** | 17 | 1 | 15 | 16 |
| **Négatif** | 9 | 25 | 11 | 10 |
| **Positif trouvé** | 11 | 2 | 8 | 7 |
| **Négatif confirmé** | 15 | 24 | 18 | 19 |
| **Précision** | 1 | 0,5 | 1 | 1 |
| **Rappel** | 0,65 | 1 | 0,53 | 0,44 |
| **f-score** | 0,79 | 0,67 | 0,70 | 0,61 |

**Tableau 6.4.** Résultats de l'analyse manuelle (en bleu) et TUPAC (en vert).

Premièrement, les résultats globaux sont cohérents, à l'exception de $p2$ où la précision est inférieure à 1 avec un rappel maximal. Une explication possible est que seule une API sur quatre dans $p_{API}$ est utilisée dans $p2$. Ainsi, avec un petit nombre de patrons positifs, il est plus facile d'avoir un bon rappel. Deuxièmement, les patrons faux positifs sont quasi inexistants(0,03%) et nous dénotons un très bon taux de vrai négatif (TNR) :0.99 ce qui signifie que les modèles qui sont faux sont plus susceptibles d'être marqués comme faux. Cependant, il est important de noter que les résultats ne sont pas parfaits puisque 21% des patrons qui sont considérés comme négatif par TUPAC, sont des faux négatifs. En effet, le taux de faux négatif (FNR) est égal à 0,35.

Nous avons simulé une mauvaise utilisation des API en générant une mutation de chaque projet. Les résultats obtenus sont présentés dans le Tableau 6.5. Nous pouvons constater une légère diminution du nombre des patrons positifs dans l'ensemble. Après une analyse

plus approfondie des projets mutants, nous avons remarqué que les projets qui ont plus d'opérations de suppression sont plus susceptibles de rompre un patron positif. C'est particulièrement le cas pour le projet $p2$ qui conduit à la suppression complète des patrons positifs. Nous supposons que la suppression des appels API simule le cas où le développeur a omis d'appeler une méthode particulière.

Il est important de souligner que sur ce résultat les patrons considérés comme $VPD$ et N/A sont comptés comme négatif. Or, certains des patrons que nous avons ne couvrent pas le cas où, si une méthode du modèle est manquante cela est considéré comme non applicable plutôt que violé. Dwyer *et al* [**7**] a classé ce type de patron comme **"Order type"**. Ce sont des patrons qui encapsulent des informations sur l'ordre relatif dans lequel la méthode se produisent, par opposition à **"Occurence type"** qui parlent de l'occurrence d'une méthode donnée .

| | **p1** | **p2** | **p3** | **p4** |
|---|---|---|---|---|
| **Positif** | 9 | 0 | 8 | 5 |
| **Négatif** | 17 | 26 | 18 | 21 |

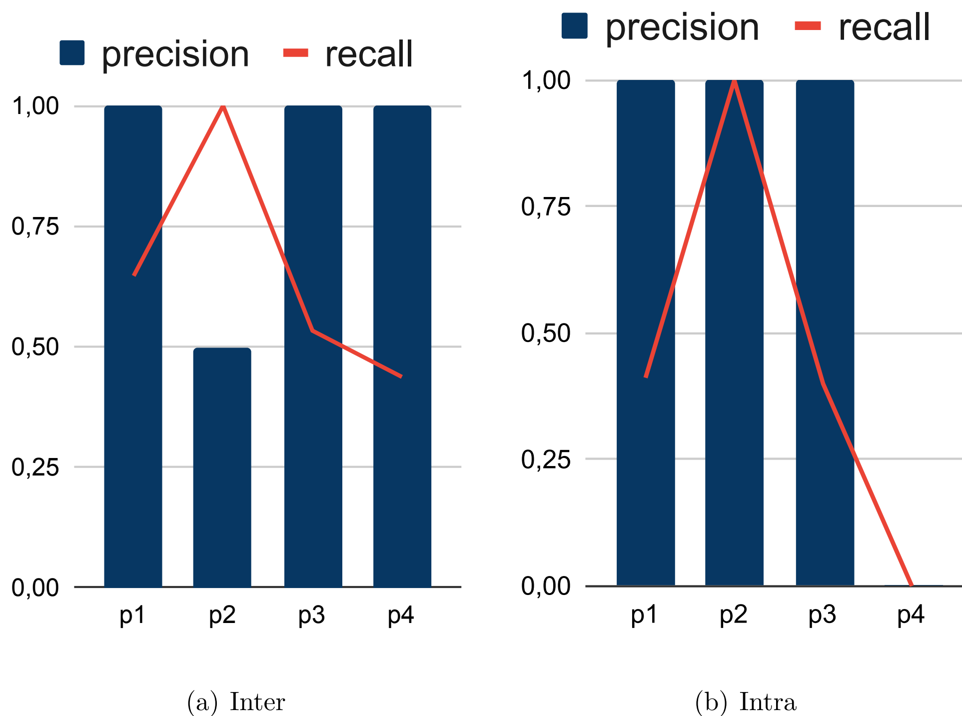
**Tableau 6.5.** Résultat après mutation

Bien que les résultats ne sont pas parfaits avec une précision moyenne de 0,87 et un rappel de 0,65, nous pouvons affirmer que notre approche est en effet capable de détecter les violations de patrons d'utilisation en utilisant l'analyse statique.

### 6.2.2. RQ2: Est-ce qu'une analyse de type interprocédurale offre une meilleure performance comparée à une simple analyse intraprocédurale?

Afin d'effectuer une analyse intraprocédurale, nous avons vérifié toutes les méthodes qui contiennent au moins un appel de méthode API. Nous présentons les résultats comparatifs de la détection des patrons positifs dans Figure 6.1 et des patrons négatifs dans Figure 6.2. Nous pouvons voir qu'en général, l'analyse interprocédurale offre une meilleurs précision et rappel sur les quatre projets. Même si l'analyse intraprocédurale semble offrir une bonne précision, nous ne pouvons pas en dire autant concernant le rappel. Effectivement, on observe une réduction drastique du rappel, surtout pour $p4$. Cela pourrait s'expliquer par le fait que ce projet en particulier compte 39 méthodes au total, ce qui est le plus élevé des quatre projets.

Intuitivement, nous pouvons supposer qu'il existe une corrélation entre l'interaction entre les procédures et le nombre de méthodes. Ces résultats semblent donner du poids à l'idée que la plupart du temps, les méthodes d'API ne sont pas concentrées dans une seule méthode du code client mais sont plutôt dispersées dans le projet. En comparant les résultats concernant la détection des patrons négatifs, nous pouvons voir que l'analyse intraprocédurale offre une performance légèrement inférieure à l'interprocédurale. Cela nous amène à penser que la violation des patrons peut se produire à un niveau interprocédural, ce qui est cohérent puisque les patrons ont été extraits de la trace d'exécution, c'est-à-dire dans une perspective interprocédurale, indépendamment de la méthode ou de la classe qui a fait l'appel.



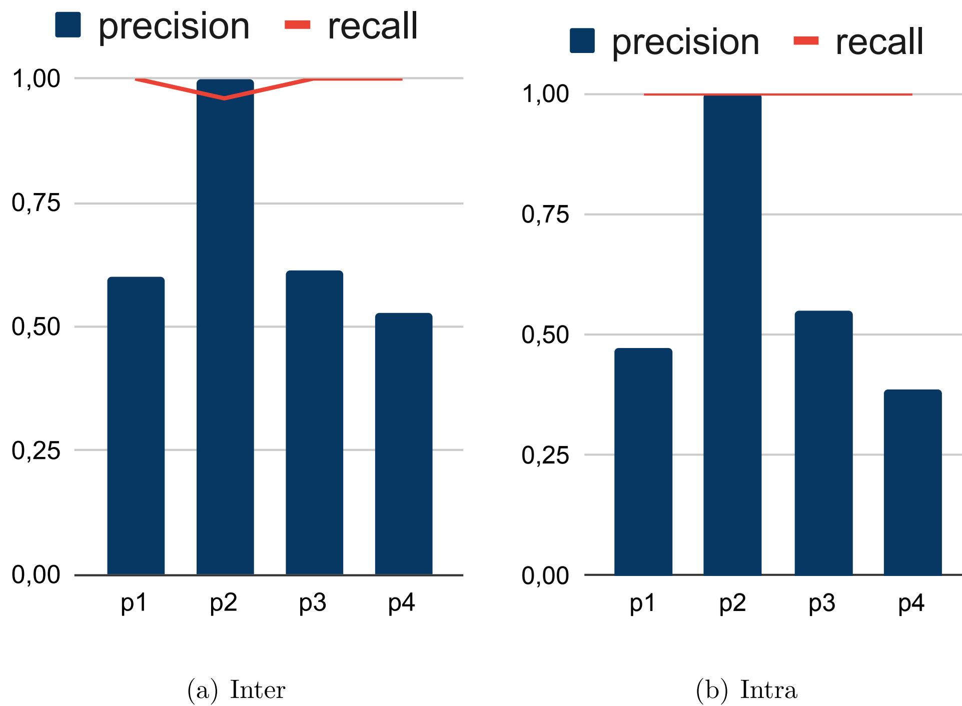

(a) Inter

(b) Intra

**Fig. 6.1.** Inter VS intraprocédural résultat positif

### 6.2.3. RQ3: *TUPAC* est-il assez rapide pour fournir des résultats utilisables au développeur sans être une entrave à sa productivité?

Les résultats affichés dans le Tableau 6.6 montrent quelques détails intéressants. Premièrement, les résultats sont tous cohérents, ils ont tout en moyenne. 695,84ms. Nous pouvons également remarquer que l'utilitaire API Map a été le plus rapide de tous, car il est le moins




(a) Inter

(b) Intra

**Fig. 6.2.** Inter VS intraprocédural résultat négatif.

complexe avec une profondeur de deux à huit. Deuxièmement, nous remarquons que même si les utilitaires API Set et File ont relativement la même complexité, il a fallu 5 000ms de plus à l'API File pour produire un résultat. Cela signifie que le nombre de patrons à vérifier a plus d'impact que la complexité.

Nous présentons un résultat plus détaillé par projet et par API dans le Tableau 6.7. Un détail notable est la cohérence du temps de génération du Graphe profond (colonne 3 du Tableau 6.7) avec la taille du projet.

Nous avons également remarqué que le projet Jar2Java prend environ 65 000ms de plus que le compresseur Html, même s'il est plus petit. Après une enquête plus approfondie, nous avons constaté que Jar2Java contient plus de classes utilisant io#File que le compresseur Html, qui sont respectivement de six et de trois. Comme il y a plus de patrons à vérifier pour l'API io#File, cela confirme notre déclaration précédente, à savoir que le nombre de modèles à vérifier a plus d'impact en termes de temps que sa complexité.

En dernier lieu, mais non des moindres, le temps nécessaire pour calculer et générer le Graphe profond est d'environ en moyenne deux pour cent du temps total. Cela signifie

que TUPAC est très dépendant et ne peut être aussi rapide que le vérificateur de modèle utilisé.

Ces résultats nous permettent de conclure que TUPAC est assez rapide pour fournir des retours au développeur sans altérer sa productivité.

| API | Moyenne(ms) | Moyenne par patron(ms) |
|---|---|---|
| **io#File** | 58679,50 | 814,99 |
| **util#List** | 10783,00 | 770,21 |
| **util#Set** | 8541,50 | 711,79 |
| **util#Map** | 12645,00 | 486,35 |

**Tableau 6.6.** Moyenne par patron

| **Projet** | Taille(LOC) | DG(ms) | io#File | util#List | util#Set | util#Map | **Moy. par projet (ms)** |
|---|---|---|---|---|---|---|---|
| **p1** | 5309 | 592 | 6101 | 2214 | - | 2623 | **3646,00** |
| **p2** | 592 | 183 | 5140 | - | - | - | **5140,00** |
| **p3** | 2879 | 758 | 217611 | 19352 | 15758 | 32950 | **71417,75** |
| **p4** | 1314 | 270 | 5866 | - | 1325 | 2362 | **3184,33** |

**Tableau 6.7.** Moyenne par projet en ms

## 6.3. Menaces à la validité

Notre évaluation est confrontée à diverses menaces qui pèsent sur la validité et qui sont brièvement évoquées ci-dessous.

En premier lieu, le choix des projets lors de cette étude a été fait en se basant sur un site communautaire Programcreek. Ce site met à la disposition des développeurs des extraits de code contenant des méthodes d'API. Ces extraits de code se trouvent dans des programmes disponibles sur des dépôts publics. Le site permet aux développeurs de voter positivement pour les extraits de code qui les ont aidés à comprendre l'utilisation d'un API donné, ou de voter négativement dans le cas contraire. Pour sélectionner les API, nous avons pris les quatre premières librairies Java dans leur classement "Top Java Classes". Ensuite, nous

avons sélectionné les projets qui ont le plus de votes positifs tout en priorisant les projets qui utilisent les API simultanément.

Toujours en ce qui concerne les projets utilisés, il est probable que nos résultats seront plus solides si nous élargissons notre jeu de données. Toutefois, ceci n'a pas été possible, car faire une vérification manuelle de chaque patron pour chaque API pour chaque projet est une tâche qui nécessite énormément de ressources. En effet, afin de pouvoir juger si les résultats rendus par TUPAC sont corrects ou non, il nous faut des données de référence. Vu qu'aucun autre outil ne peut être utilisé pour avoir ces données de références, l'alternative a été de faire une vérification manuelle. Certes, cela introduit une sérieuse menace à la validité, car les résultats ont été générés par les sujets qui ont mené l'expérience, toutefois, nous considérons ces résultats comme étant des résultats exploratoires afin d'avoir une vision du potentiel de notre approche.

Une autre menace à la validité des résultats obtenus concerne le type d'utilisation dont nous nous sommes servies pour générer les traces d'exécution. Afin d'atténuer cette menace, nous nous sommes basés sur les exemples fournis dans la documentation lorsque disponibles, ou dans le cas contraire, nous avons créé une exécution minimum fonctionnelle du projet afin de réduire autant que possible le risque d'utilisation non envisagée par le développeur du projet.

## 6.4. Limitations et travaux futurs

Bien que les résultats exploratoires soient encourageants, des améliorations peuvent être apportées. Pour avoir de meilleurs résultats, nous pourrions augmenter le nombre de projets et d'API. Toutefois pour obtenir des conclusions rigoureuses il faudrait avoir une donnée de référence sans passer par une analyse manuelle. Enfin, comme énoncé dans les menaces à la validité, les mutations que nous avons définies pour simuler une mauvaise utilisation des API peuvent être améliorées. Nous pourrons ajouter un autre type de mutation "Déplacement". Cela permettrait de déplacer les appels de méthodes API et les méthodes contenant des appels API vers un autre endroit du code. Cela a intuitivement plus de sens, car les résultats de la comparaison entre l'analyse interprocédurale et intraprocédurale démontrent que les méthodes d'API sont dispersées dans le projet.

# Chapitre 7

## CONCLUSION

À travers ce mémoire, nous avons proposé une méthode pour exploiter les patrons d'utilisation écrits en LTL afin d'aider les développeurs dans l'utilisation des API. Notre méthode se base sur l'analyse statique pour vérifier la conformité du code client aux patrons et ainsi fournir des retours aux développeurs concernant l'utilisation des API au moment même où ils écrivent le code sans attendre une analyse dynamique, cette dernière étant couteuse en ressource et requérant que le code soit exécutable.

Plusieurs étapes sont nécessaires pour pouvoir vérifier la conformité d'un code client. Nous commençons par extraire les modèles représentatifs du code à partir du graphe d'appel et du graphe de flot de contrôle. Ces deux graphes vont nous permettre de générer le Graphe profond qui sera par la suite converti en automate. Ce dernier est ensuite vérifié en utilisant un vérificateur de modèle. Le but de se modèle est de déterminer toutes les traces d'exécution possibles du code client et par conséquent détecter les possibles violations du patron d'utilisation. Ces possibles violations sont rapportées à l'intérieur même de l'IDE pour que le développeur n'ait pas besoin de changer de contexte et ainsi ne pas affecter sa productivité.

Notre approche a été évaluée sur quatre projets Java avec quatre API. Cette évaluation a été faite en vue de répondre aux questions de recherche concernant la capacité de Tupac a détecter les violations de patron tant bien au niveau interprocédural qu'au niveau intraprocédurale, ainsi que le temps nécessaire pour traiter un projet.

Nous avons aussi pu avoir une idée quant aux capacités de Tupac à détecter les violations de patrons en comparant les résultats retournés à une analyse manuelle. Ces résultats démontrent que notre approche a en moyenne 0.87 de précision avec un taux de 0.65 de

rappel. De plus, pour simuler d'éventuelles mauvaises utilisations des API, nous avons créé des mutants de chaque projet. Nous avons pu constater que les projets qui ont plus d'opérations de suppression, qui a pour but de simuler l'omission d'un développeur, sont plus susceptibles de rompre un patron positif. En ce qui concerne le type d'analyse, nous avons pu déterminer qu'une analyse interprocédurale a généré de meilleurs résultats comparés à une analyse intraprocédurale. Cela suggère que la plupart du temps, les méthodes d'API ne sont pas concentrées dans une seule méthode du code client, mais dispersées à l'intérieur du projet. Enfin, les résultats du test de scalabilité nous ont permis de déterminer que TUPAC prend en moyenne 695,84ms pour vérifier la conformité d'un patron pour un projet. Dans ce temps total, TUPAC prend en moyenne deux pour cent pour générer le GRAPHE PROFOND.

Malgré des résultats très encourageants, plusieurs améliorations peuvent-être proposée. Un algorithme plus sophistiqué pour générer le CFG et le CG serait un début. Cela améliorera grandement la performance de TUPAC à vérifier les patrons d'utilisation puisque le modèle sera plus sain et complet. De plus, il sera nécessaire de faire la détection de prédicats qui sont mutuellement exclusifs. Cela pourrait se faire en utilisant les mêmes stratégies de détection utilisées par Saied *et al.* [**25**]. Cela permettra d'augmenter la précision du GRAPHE PROFOND et possiblement réduire le temps nécessaire à la vérification du modèle.

Utiliser les patrons d'utilisation pour aider le développeur, surtout, sans exécution du code n'est pas une tâche triviale. Toutefois, ces résultats, bien qu'exploratoires, suggèrent que c'est un domaine qui pourrait être grandement exploité, non seulement pour aider les développeurs, mais aussi en vue de donner encore plus de valeur à la contribution des nombreux chercheurs travaillant sur la détection et l'extraction de patrons d'utilisation d'API.

# Références bibliographiques


[1] Frances E. Allen : Control flow analysis. *In Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. Association for Computing Machinery.

[2] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen et Mira Mezini : Investigating next steps in static API-Misuse detection. *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 265–275, 2019.

[3] Randal E. Bryant : Symbolic boolean manipulation with ordered binary-decision diagrams. 1992.

[4] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani et Armando Tacchella : NuSMV 2: An OpenSource tool for symbolic model checking. *In* Ed Brinksma et Kim Guldstrand Larsen, éditeurs : *Computer Aided Verification*, 2002.

[5] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia et Marco Roveri : NuSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[6] Barthélémy Dagenais et Laurie Hendren : Enabling static analysis for partial java programs. *In Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, OOPSLA 08, page 313328, New York, NY, USA, 2008. Association for Computing Machinery.

[7] Matthew B. Dwyer, George S. Avrunin et James C. Corbett : Patterns in property specifications for finite-state verification. *In Proceedings of the 21st International Conference on Software Engineering*, ICSE 99, page 411420, New York, NY, USA, 1999. Association for Computing Machinery.

[8] Heather J. Goldsby et Betty H. C. Cheng : Automatically discovering properties that specify the latent behavior of UML models. *In* Dorina C. Petriu, Nicolas Rouquette et Øystein Haugen, éditeurs : *Model Driven Engineering Languages and Systems*, pages 316–330, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[9] Michael Huth et Mark Ryan : *Logic in Computer Science: Modelling and reasoning about systems.* Cambridge university press, 2004.

[10] Mik Kersten et Gail C. Murphy : Using task context to improve programmer productivity. *In Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT 06/FSE-14, page 111, New York, NY, USA, 2006. Association for Computing Machinery.

[11] Saul A. Kripke : Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16(1963):83–94, 1963.

[12] David Lo, Siau-Cheng Khoo et Chao Liu : Mining temporal rules for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(4):227–247.

[13] G. C. Murphy : Beyond integrated development environments: Adding context to software development. *In 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 73–76, 2019.

[14] D. Nam, A. Horvath, A. Macvean, B. Myers et B. Vasilescu : MARBLE: Mining for boilerplate code to identify api usability problems. *In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[15] R. Nelken et Nissim Francez : Bilattices and the semantics of natural language questions. *Linguistics and Philosophy*, 25, 10 1999.

[16] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen et T. N. Nguyen : GraPacc: A graph-based pattern-oriented, context-sensitive code completion tool. *In 2012 34th International Conference on Software Engineering (ICSE)*, pages 1407–1410, 2012.

[17] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, L. Ochoa, T. Degueule et M. Di Penta : FOCUS: A recommender system for mining api function calls and usage patterns. *In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1050–1060, 2019.

[18] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi et Tien N. Nguyen : Graph-based mining of multiple object usage patterns. *In Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE 09, page 383392, New York, NY, USA, 2009. Association for Computing Machinery.

[19] E. Raelijohn, M. Famelis et H. Sahraoui : A vision for helping developers use APIs by leveraging temporal patterns. *In 2019 IEEE/ACM 7th International Conference on Formal Methods in Software Engineering (FormaliSE)*, pages 95–98, 2019.

[20] S. Rastkar, G. C. Murphy et A. W. J. Bradley : Generating natural language summaries for crosscutting source code concerns. *In 2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 103–112, 2011.

[21] M. P. Robillard, A. Marcus, C. Treude, G. Bavota, O. Chaparro, N. Ernst, M. A. Gerosa, M. Godfrey, M. Lanza, M. Linares-Vásquez, G. C. Murphy, L. Moreno, D. Shepherd et E. Wong : On-demand developer documentation. *In 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 479–483, 2017.

[22] Martin Robillard et Robert Deline : A field study of API learning obstacles. *Empirical Software Engineering*, 16:703–732, 12 2011.

[23] Mohamed Aymen Saied, Ali Ouni, Houari Sahraoui, Raula Gaikovina Kula, Katsuro Inoue et David Lo : Improving reusability of software libraries through usage pattern mining. *Journal of Systems and Software*, 145:164 – 179, 2018.

[24] Mohamed Aymen Saied, Erick Raelijohn, Edouard Batot, Michalis Famelis et Houari Sahraoui : Towards assisting developers in API usage by automated recovery of complex temporal patterns. *Information and Software Technology*, 119, 2020.

[25] Mohamed Aymen Saied, Houari A. Sahraoui et Bruno Dufour : An observational study on API usage constraints and their documentation. *In 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, pages 33–42, 2015.

[26] Anas Shatnawi, Hudhaifa Shatnawi, Mohamed Aymen Saied, Zakarea Al Shara, Houari Sahraoui et Abdelhak Seriai : Identifying Software Components from Object-Oriented APIs Based on Dynamic Analysis. *In Proceedings of the 26th Conference on Program Comprehension*, 2018.

[27] Harold Somers : Review article: Example-based machine translation. *Machine Translation*, 14(2):113–157, 1999.

[28] M. Wen, Y. Liu, R. Wu, X. Xie, S. Cheung et Z. Su : Exposing library API misuses via mutation analysis. *In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 866–877, 2019.

[29] Congying Xu, Xiaobing Sun, Bin Li, Xintong Lu et Hongjing Guo : MULAPI: Improving API method recommendation with API usage location. *J. Syst. Softw.*, 142:195–205, 2018.

[30] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat et Manuvir Das : Perracotta: Mining temporal API rules from imperfect traces. *In Proceedings of the 28th International Conference on Software Engineering*, New York, NY, USA, 2006. Association for Computing Machinery.

[31] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei et Hong Mei : MAPO: Mining and recommending API usage patterns. *In Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, page 318343, Berlin, Heidelberg, 2009. Springer-Verlag.

# Annexe A

## Modèle SMV

**Listing A.1.** Modèle SMV de la méthode safeClose()

```
1
2  MODULE main
3  VAR
4      eDataManager#safeClose : boolean;
5      eDataManager#safeRelease : boolean;
6      r0 : boolean;
7      r1 : boolean;
8      r01 : boolean;
9      r02 : boolean;
10     r03 : boolean;
11     r04 : boolean;
12     r001 : boolean;
13     r002 : boolean;
14     state : {DataManager#safeClose,Map#entrySet, Map#iterator, DataManager#safeRelease,
           Map#values,
15     For1#safeRelease, SQLiteCompiledSql#releaseSqlStatement,ENDFor1#safeRelease,Map#clear
           ,Exit#DataManager#safeRelease,
16     While2#safeClose,SQLiteClosable#onallReferencesReleased, Exit#DataManager#safeClose,
           Iterator#hasNext,
17     Iterator#next, Map#Entry#getKey, IF3#safeClose, SQLiteClosable#
           onAllReferenceReleasedFromContainer,ENDIF3#safeClose,ENDWhile2#safeClose, ERROR};
18
```

```
19 ASSIGN
20     init(eDataManager#safeClose) := TRUE;
21     init(eDataManager#safeRelease) :=FALSE;
22     init(r0) := FALSE;
23     init(r1) := FALSE;
24     init(r01) := FALSE;
25     init(r02) := FALSE;
26     init(r03) := FALSE;
27     init(r04) := FALSE;
28     init(r001) := FALSE;
29     init(r002) := FALSE;
30     init(state) := DataManager#safeClose;
31
32 next(state) := case
33     state = DataManager#safeClose & eDataManager#safeClose: {Map#entrySet};
34     state = Map#entrySet & eDataManager#safeClose: {Map#iterator};
35     state = Map#iterator & eDataManager#safeClose: {DataManager#safeRelease};
36     state = DataManager#safeRelease & eDataManager#safeRelease: {Map#values};
37     state = Map#values & eDataManager#safeRelease: {For1#safeRelease};
38     state = For1#safeRelease & eDataManager#safeRelease &r0 : {SQLiteCompiledSql#
           releaseSqlStatement};
39     state = SQLiteCompiledSql#releaseSqlStatement & eDataManager#safeRelease: {ENDFor1#
           safeRelease};
40     state = ENDFor1#safeRelease & eDataManager#safeRelease: {For1#safeRelease};
41     state = For1#safeRelease & eDataManager#safeRelease & r1: {Map#clear};
42     state = Map#clear & eDataManager#safeRelease: {Exit#DataManager#safeRelease};
43     state = Exit#DataManager#safeRelease & eDataManager#safeClose: {While2#safeClose};
44     state = While2#safeClose & eDataManager#safeClose & r02: {SQLiteClosable#
           onallReferencesReleased};
45     state = SQLiteClosable#onallReferencesReleased & eDataManager#safeClose: {Exit#
           DataManager#safeClose};
46     state = While2#safeClose & eDataManager#safeClose & r01: {Iterator#hasNext};
47     state = Iterator#hasNext & eDataManager#safeClose & r03: {Iterator#next};
48     state = Iterator#hasNext & eDataManager#safeClose & r04 : {While2#safeClose};
```

```
49     state = Iterator#next & eDataManager#safeClose: {Map#Entry#getKey};
50     state = Map#Entry#getKey & eDataManager#safeClose: { IF3#safeClose};
51     state = IF3#safeClose & eDataManager#safeClose & r001: {SQLiteClosable#
           onAllReferenceReleasedFromContainer};
52     state = SQLiteClosable#onAllReferenceReleasedFromContainer & eDataManager#safeClose:
           {ENDIF3#safeClose};
53     state = IF3#safeClose & eDataManager#safeClose & r002: {ENDIF3#safeClose};
54     state = ENDIF3#safeClose & eDataManager#safeClose: {ENDWhile2#safeClose};
55     state = ENDWhile2#safeClose & eDataManager#safeClose: {While2#safeClose};
56     state = Exit#DataManager#safeClose & eDataManager#safeClose: {Exit#DataManager#
           safeClose};
57     TRUE  : {ERROR};
58 esac;
59
60 next(eDataManager#safeClose) := case
61     eDataManager#safeClose = TRUE : TRUE;
62     state = Map#clear & eDataManager#safeRelease : TRUE;
63     TRUE : FALSE;
64 esac;
65
66 next(eDataManager#safeRelease) := case
67     state = Map#iterator & eDataManager#safeClose = TRUE : TRUE;
68     eDataManager#safeRelease = TRUE : TRUE;
69     TRUE : FALSE;
70 esac;
71
72 next(r1) :=case
73     state = ENDFor1#safeRelease & eDataManager#safeRelease = TRUE : TRUE;
74     TRUE : FALSE;
75 esac;
76
77 next(r0) :=case
78     state = Map#values & eDataManager#safeRelease = TRUE : TRUE;
79     TRUE : FALSE;
```

```
80  esac;
81
82  next(r01) :=case
83       state = Exit#DataManager#safeRelease & eDataManager#safeRelease = TRUE : TRUE;
84       state = ENDWhile2#safeClose & eDataManager#safeRelease = TRUE : TRUE;
85      TRUE : FALSE;
86  esac;
87
88  next(r02) :=case
89      state = Iterator#hasNext & eDataManager#safeRelease = TRUE : TRUE;
90      TRUE : FALSE;
91  esac;
92
93  next(r04) :=case
94      r01 = TRUE : TRUE;
95      TRUE : FALSE;
96  esac;
97
98  next(r03) :=case
99      r01 = TRUE : TRUE;
100     TRUE : FALSE;
101 esac;
102
103 next(r001) :=case
104 state = Map#Entry#getKey & eDataManager#safeClose = TRUE : TRUE;
105 TRUE : FALSE;
106 esac;
107
108 next(r002) :=case
109 state = Map#Entry#getKey & eDataManager#safeClose = TRUE : TRUE;
110 TRUE : FALSE;
111 esac;
```

**Listing A.2.** Modèle SMV de la méthode manager() de la classe Stack

```
1  MODULE main
2  VAR
3      eStack#manager : boolean;
4      r1 : boolean;
5      r2 : boolean;
6      state : {manager , ExitStack#manager , push , pop , peek, ERROR };
7
8  ASSIGN
9      init(r1) := FALSE;
10     init(r2) := FALSE;
11     init(eStack#manager) := TRUE;
12     init(state) :=manager;
13
14 next(state) := case
15     state = (manager) & eStack#manager : {push};
16     state = (push) & eStack#manager & r1: {pop};
17     state = (pop) & eStack#manager : {push};
18     state = (push) & eStack#manager & r2: {peek};
19     state = (peek) & eStack#manager : {ExitStack#manager};
20     state = (ExitStack#manager) & eStack#manager : {ExitStack#manager};
21     TRUE : {ERROR};
22 esac;
23
24 next(eStack#manager) := case
25     eStack#manager = TRUE : TRUE;
26     TRUE : FALSE;
27 esac;
28
29 next(r1) :=case
30     state =manager & r1 = FALSE : TRUE;
31     state = pop & r1 = TRUE : FALSE;
32     TRUE : FALSE;
33 esac;
```

```
34
35 next(r2) :=case
36     state = push & r1 = TRUE : FALSE;
37     TRUE : TRUE;
38 esac;
```