Inférence de modèles dirigée par la logique métier

William Durand, Sébastien Salva

Inférence de modèles dirigée par la logique métier

William Durand, Sébastien Salva
LIMOS - UMR CNRS 6158,
PRES Clermont-Ferrand University, FRANCE,
william.durand@isima.fr, sebastien.salva@udamail.fr

Abstract

De nombreux travaux utilisent des modèles formels pour effectuer de la ==vérification de propriétés ou de la génération de tests. Cependant, produire ces modèles reste une tâche complexe et fastidieuse. L'inférence de modèles est un domaine de recherche récent qui répond partiellement à cette problématique. Cette technique consiste à générer des modèles à partir de tests automatiques ou d'informations sur l'application. Cet article propose une nouvelle approche de génération de modèles à partir de traces d'exécution (séquences d'actions) extraites depuis une application. Intuitivement, un expert humain est capable de reconnaître des comportements fonctionnels parmi ces traces, en appliquant des règles de déduction. Nous proposons une plateforme capable de reproduire ce principe en utilisant un système expert basé sur des règles d'inférence. Ces règles sont organisées en couches et permettent de construire des modèles IOSTS partiels (Input Output Symbolic Transition System), qui deviennent de plus en plus abstraits au fur et à mesure que l'on s'élève dans la pile de règles. Comme cette solution se base sur des traces issues d'une application en cours d'exécution, cet ensemble de traces peut être potentiellement trop réduit. Pour augmenter cet ensemble automatiquement, notre solution fournit également un Robot explorateur guidé par des stratégies de couverture, permettant de découvrir de nouveaux états de l'application, et ainsi de produire de nouvelles traces.

Mots clés : IOSTS, inférence de modèles, test automatique.

1 Introduction

Le cycle de vie d'une application n'est souvent accompagné que de peu de documentation et ceci tend à plusieurs problématiques. Premièrement, la phase de test, qui s'appuie généralement sur cette documentation, donne une couverture de test très incomplète. Par la suite, la maintenance devient ardue car sa compréhension et sa modification nécessite de se plonger dans le code source, dans la mesure où il est disponible. Une première solution, connue dans le monde de la recherche depuis plusieurs décennies, consiste à définir des modèles formels exprimant les comportements fonctionnels d'une application. Un tel modèle représente de la documentation, et peut également permettre la génération automatique de suites de tests grâce à des techniques de test basées modèle. Cependant, la production de modèles formels s'avère être une tâche complexe et lourde. Aujourd'hui, seules des spécifications partielles sont proposées dans la plupart des cas. A nouveau, nous retrouvons les problématiques énoncées précédemment, à savoir une génération de test partielle et le manque de documentation.

Dans cet article, nous nous intéressons à l'inférence de modèles, un domaine de recherche récent qui tend à aider à l'obtention de modèles. Cette approche peut s'appliquer pendant la phase de conception d'un logiciel, mais elle s'applique particulièrement bien lorsque l'application existe et fonctionne déjà. Elle cible en priorité les phases de maintenance et d'évolution. L'inférence de modèles permet d'étudier et de comprendre le fonctionnement d'une application en générant une spécification. Grâce à du test automatique ou à l'étude de traces d'exécution, des méthodes et outils sont ainsi capables de générer des modèles partiels [MBN03, ANHY12, MvDL12, YPX13], qui peuvent être employés pour générer automatiquement des cas de test de non-regression [AFT+12].

Mais ils pourraient aussi être utilisés comme base pour retrouver et écrire une spécification complète. Cependant, les modèles produits offrent peu de sémantique et restent souvent proches des traces d'exécution. De plus, ces méthodes d'inférence de modèles ne prennent généralement en compte que des applications événementielles, c'est-à-dire des applications qui offrent une interface graphique permettant aux utilisateurs d'interagir, tout en répondant à une séquence donnée par l'utilisateur. En effet, elles peuvent être explorées, par test automatique, en remplissant les interfaces par des données de test et en déclenchant des événements (clic, etc.).

Nous proposons ici une nouvelle solution pour générer des modèles à partir de traces d'exécution en supposant que les applications ne sont pas obligatoirement événementielles. Intuitivement, nous sommes partis du postulat suivant : un expert humain, qui est capable d'écrire des spécifications, est généralement capable de lire des traces d'exécution et de reconnaître des comportements fonctionnels, en s'appuyant sur sa connaissance de l'application. Nous avons choisi d'injecter cette notion d'expertise et de connaissance dans une méthode d'inférence de modèles pour produire non pas un modèle mais plusieurs, offrant ainsi différents niveaux d'abstraction et exprimant une sémantique de plus en plus riche. Cette notion de connaissance est reproduite par un système expert qui comporte des règles formalisées par une logique des prédicats du premier ordre. En appliquant ces règles sur des traces d'une application, puis de modèles en modèles, les déductions de l'expert humain sont simulées pour inférer de nouveaux modèles qui gagnent en abstraction. Les modèles obtenus dans ce travail sont des IOSTSs (Input Output Symbolic Transition Systems [FTW05]). Comme notre solution repose sur les traces issues d'une application en cours d'exécution, les modèles produits sont intimement liés à la richesse de cet ensemble de traces. Un ensemble trop pauvre en termes d'information mènera à des modèles très partiels. Pour pallier cela, notre approche est également composée d'un Robot explorateur qui va augmenter cet ensemble de traces via du test automatique. A la différence des méthodes existantes, ce robot peut produire de nouvelles traces et découvrir de nouveaux états d'une application en suivant des stratégies d'exploration définies par des règles d'inférence. Ces stratégies peuvent ainsi être modifiées comme désiré suivant le type d'application.

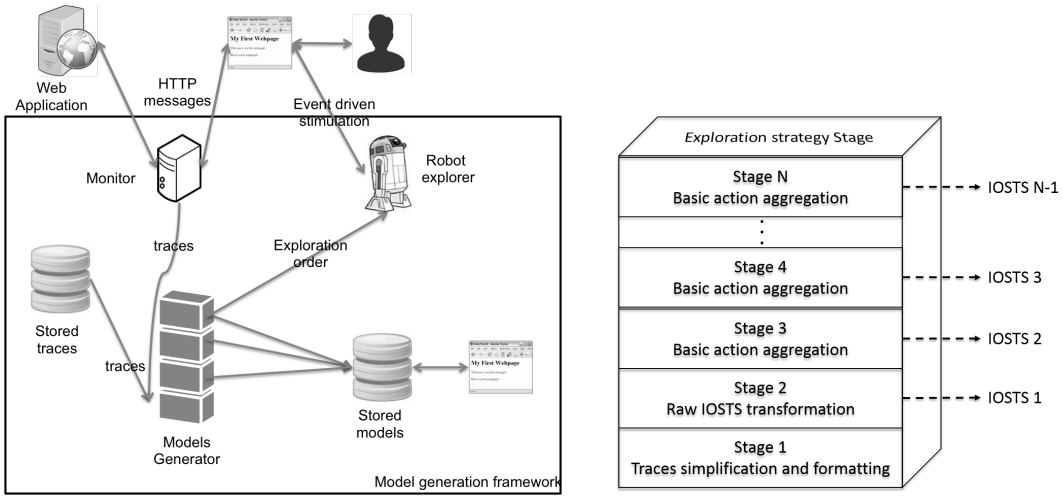
Dans la section suivante, nous décrivons de façon générale le fonctionnement de notre plateforme rassemblant le Robot générateur de traces et le Générateur de modèles. Puis, par manque de place, nous détaillons uniquement ce dernier mais de façon concrète, en ciblant le contexte des applications Web. Nous rappelons quelques définitions et notations sur les IOSTSs en Section 3. Par la suite, nous décrivons notre Générateur de modèles architecturé en couches en Section 4. Nous comparons notre méthode à quelques travaux et concluons en Section 5.

2 Présentation générale de l'approche

Notre approche a pour but de générer des modèles formels qui expriment des comportements fonctionnels d'une application à partir de ses traces d'exécution. L'une des originalités fortes de cette approche réside dans la génération incrémentale de plusieurs modèles qui capturent le comportement de l'application à différents niveaux d'abstraction. De façon générale, ces modèles sont partiels et leur expressivité dépend de la richesse des traces, exprimée en quantité d'actions. Le nombre de modèles n'est pas strictement limité, bien qu'il doive être fini.

Intuitivement, notre méthode de génération de modèles provient de l'idée suivante : un expert métier, capable de concevoir des modèles, peut également diagnostiquer le fonctionnement d'une application en lisant ses traces grâce à ses connaissances et à un raisonnement logique. Ces connaissances peuvent être formalisées sous forme de règles suivant une logique des prédicats du premier ordre et être exploitées pour construire automatiquement des modèles. Nous avons choisi de décomposer cette expertise et connaissance en différents modules comme le montre la Figure 1(a).

Le *Générateur de modèles* est la pièce maîtresse de notre solution. Il reçoit des traces en entrée, qui peuvent être envoyées par un *Moniteur*, qui a pour objectif de collecter des traces à la volée. Le Générateur de modèles repose sur un système expert, autrement dit un moteur d'intelligence artificielle, permettant de simuler le raisonnement d'un expert en utilisant des règles d'inférence qui



(a) Plateforme de génération de modèles

(b) Générateur de modèles

Figure 1

expriment la connaissance de cet expert. Dans notre cas, cette connaissance métier est organisée sous forme d'une architecture hiérarchisée en couches. Chacune rassemble un ensemble de règles qui permettent de créer deux IOSTSs (excepté pour la première couche). Plus la couche est élevée et plus le modèle généré est abstrait. Ces modèles sont successivement stockés et peuvent être analysés par la suite par des experts ou par des outils de vérification.

L'ensemble des traces peut ne pas être suffisant pour générer des modèles pertinents et/ou suffisamment complets. Il est alors possible de collecter plus de traces par test automatique dans le cas où les applications sont événementielles. Dans notre approche, le *Robot explorateur* est chargé de cette exploration automatique. Mais, à la différence de la plupart des techniques de test automatique [MBN03, ANHY12, MvDL12, AFT+12, YPX13], notre robot ne progresse pas à l'aveugle, ne se base pas sur du test aléatoire et n'utilise pas une stratégie d'exploration fixe. Notre robot est guidé de façon intelligente par le Générateur de modèles qui applique une stratégie d'exploration décrite par des règles d'inférence. Ces règles interprètent les modèles en cours de génération à la volée, et renvoient une liste d'états symboliques à explorer par le robot. Le Moniteur ou le Robot récupèrent les traces produites par le test automatique et les fournissent au Générateur de modèles et ainsi de suite.

Cette approche, telle qu'elle est conçue, offre ainsi de nombreux avantages :

- il est possible de l'appliquer sur tout type d'application ou système à condition qu'il produise des traces. Ces dernières peuvent avoir été stockées au préalable et/ou peuvent être produites par du test automatique si l'application analysée est événementielle,

- l'exploration de l'application est guidée par une stratégie qui peut être modifiée en fonction des besoins liés à l'application analysée. Cette stratégie offre l'avantage de pouvoir cibler certains états de l'application quand le nombre d'états est trop grand pour être complètement visité en un temps raisonnable,

- la connaissance encapsulée par le système expert peut être utilisée pour couvrir des ensembles de traces provenant de plusieurs applications de même type (Web, etc.),

- mais les règles peuvent aussi être spécialisées et ajustées pour une application spécifique dans le but de générer des IOSTSs plus précis. Cela devient particulièrement intéressant pour comprendre une application et inférer différents niveaux d'abstraction,

- notre méthode est à la fois flexible et évolutive. Elle ne produit non pas un mais plusieurs IOSTSs selon le nombre de couches, qui n'est pas limité et peut évolué en fonction du type de l'application. Chaque modèle exprime des comportements de l'application à un niveau d'abstraction donné. Il peut être utilisé pour faciliter la génération d'un modèle final, pour appliquer des techniques de vérification (vérifier la satisfiabilité de certaines propriétés) ou encore pour automatiquement générer des suites de tests fonctionnels.

Par manque de place, nous ne présentons dans ce papier que la partie traitant de l'inférence de modèles, mais auparavant, nous donnons quelques définitions.

3 Définition du modèle IOSTS et notations

Un IOSTS est un modèle de type automate étendu composé de deux ensembles de variables, un ensemble de variables internes permettant de stocker des informations et un ensemble de paramètres enrichissant ses actions. Les transitions portent les actions, des gardes et des assignations sur des variables internes et des paramètres. L'ensemble des actions est séparé par des actions entrantes, commençant par ? et des actions sortantes, commençant par !. Les premières expriment les actions attendues par le système, tandis que les secondes expriment des actions produites par le système. Un IOSTS possède des états symboliques (locations).

Definition 1 (Input/Output Symbolic Transition System (IOSTS)) *Un IOSTS $\mathcal{S}$ est un tuple $< L, l0, V, V0, I, \Lambda, \rightarrow >$, tel que :*

- *L est l'ensemble dénombrable d'états symboliques, $l0$ est l'état symbolique initial,*
- *V est l'ensemble de variables internes, I est l'ensemble de paramètres. Nous notons D_v le domaine dans lequel une variable v prend des valeurs. L'attribution de valeurs à un ensemble de variables $Y \subseteq V \cup I$ est défini par des valuations de telle sorte qu'une valuation est une fonction $v : Y \rightarrow D$. $v_\emptyset$ représente la valuation nulle. D_Y décrit l'ensemble des valuations pour l'ensemble Y de variables. Les variables internes sont initialisées par la valuation $V0$ sur V, qui est supposée unique,*
- *Λ est l'ensemble des actions symboliques $a(p)$, avec $p = (p_1, ..., p_k)$ un ensemble fini de paramètres dans $I^k (k \in \mathbb{N})$. p est également supposé unique. $\Lambda = \Lambda^I \cup \Lambda^O \cup \{!\delta\}$: Λ^I correspond à l'ensemble des action d'entrées, Λ^O est l'ensemble des actions de sorties, δ est la quiescence,*
- *$\rightarrow$ est l'ensemble des transitions. Une transition $(l_i, l_j, a(p), G, A)$, partant de l'état symbolique $l_i \in L$ et arrivant à $l_j \in L$, notée $l_i \xrightarrow{a(p),G,A} l_j$ est étiquetée par :*
  - *une action $a(p) \in \Lambda$,*
  - *une garde G sur $(p \cup V \cup T(p \cup V))$ qui doit être satisfaite pour tirer la transition. $T(p \cup V)$ est un ensemble de fonctions qui retournent des booléens uniquement (c'est-à-dire des prédicats) sur $p \cup V$,*
  - *une fonction d'assignement A qui met à jour des variables internes. A est de la forme $(x := A_x)_{x \in V}$, tel que A_x est une expression sur $V \cup p \cup T(p \cup V)$.*

Un IOSTS est associé à un IOLTS (Input/Output Labelled Transition System) pour formuler sa sémantique. De façon intuitive, un IOLTS sémantique correspond à un automate valué ne contenant pas de variables et qui est souvent infini : les états d'IOLTS sont étiquetés par des valuations sur les variables internes et les transitions transportent des actions et des valuations sur l'ensemble des paramètres. La sémantique d'un IOSTS $\mathcal{S} =< L, l_0, V, V_0, I, \Lambda, \rightarrow >$ est l' IOLTS $[\![\mathcal{S}]\!] =< Q, q_0, \Sigma, \rightarrow >$ composé d'états valués dans $Q = L \times D_V$, $q_0 = (l_0, V_0)$ est l'état initial, Σ l'ensemble des actions valuées et $\rightarrow$ est la relation de transition. La définition de l' IOLTS sémantique peut être trouvée dans [FTW05].

Definition 2 (Séquences d'exécution et traces) *Pour un IOSTS* $\mathcal{S} = < L, l0, V, V0, I, \Lambda, \rightarrow >$, *interprété par son IOLTS sémantique* $[\![\mathcal{S}]\!] =< Q, q_0, \Sigma, \rightarrow >$, *une séquence d'exécution de* $\mathcal{S}$, $q_0\alpha_0 q_1 ... q_{n-1}\alpha_{n-1}q_n$ *est une séquence de termes* $q_i\alpha_i q_{i+1}$, *avec* $\alpha_i \in \Sigma$ *une action valuée et* q_i, q_{i+1} *deux états de* Q.

$Run(\mathcal{S}) = Run([\![\mathcal{S}]\!])$ *est l'ensemble des séquences d'exécution de* $[\![\mathcal{S}]\!]$. *Il s'en suit qu'une trace d'une séquence d'exécution* r *est définie par la projection* $proj_\Sigma(r)$ *sur les actions.* $Traces_F(\mathcal{S}) = Traces_F([\![\mathcal{S}]\!])$ *est l'ensemble des traces des séquences d'exécution terminées par des états de* $F \times D_V$.

4 Inférence de modèles partiels

Comme décrit dans la Section 2, le Générateur de modèles est principalement composé d'un système expert, qui infère des IOSTSs à partir d'un ensemble de traces provenant d'une application ou d'un système. Un système expert est fondé sur un moteur de règles qui utilise ces dernières pour faire des déductions ou des choix. Dans un tel système, la base de connaissances est séparée du raisonnement : la connaissance est exprimée par une base de faits qui est analysée par des règles qui adoptent souvent un chaînage avant. Ces règles produisent de nouveaux faits et ainsi de suite. Le moteur s'arrête lorsque plus aucune règle ne peut être activée. Notre Générateur de modèles prend en entrée des traces qui correspondent à la base de faits initiale. Les règles d'inférence sont ici organisées en couches, pour tenter de correspondre avec le comportement d'un expert humain. Ces couches sont présentées en Figure 1(b). Chacune est composée de règles qui s'appliquent sur la base de faits courante, via le moteur d'inférence qui déduit de nouveaux faits. Lorsque aucune règle d'une couche ne peut plus être instanciée, la nouvelle base de faits est enregistrée et sera utilisée par la couche suivante.

La première couche commence par formater puis élaguer l'ensemble de traces donné en entrée. Habituellement, quand un expert humain doit lire des traces d'une application, il commence par les filtrer pour ne conserver que celles qui ont du sens vis-à-vis de l'application. Cette couche va effectuer ces opérations automatiquement. Pour cela, il faut demander à l'expert comment il décide d'ignorer certaines traces et sur quels informations il se base. Ce sont ces choix que nous formalisons sous forme de règles dans cette première couche.

Nous appelons traces structurées, les traces conservées et sur lesquelles des informations ont été identifiées, par opposition aux traces brutes qui proviennent du Moniteur. L'ensemble de ces traces structurées, noté ST, est ensuite transmis à la couche supérieure. Ce procédé est effectué de manière incrémentale. Chaque fois que de nouvelles traces sont données au Générateur de modèles, elles sont formatées et filtrées avant d'être envoyées à la couche 2 sous forme de traces structurées.

Les couches restantes produisent deux IOSTSs chacune : le premier IOSTS $\mathcal{S}_i$ possède une structure en arbre dérivée des traces. Le second, noté $App(\mathcal{S}_i)$ est une approximation de l'IOSTS précédent qui capture potentiellement plus de comportements mais qui peuvent s'avérer incorrects. Ces deux IOSTSs sont minimisés grâce à une technique de minimisation par bisimulation [Fer89].

Le rôle de la seconde couche consiste à effectuer une première transformation des traces structurées en IOSTS. Intuitivement, les actions valuées d'une trace sont successivement traduites en transitions IOSTS en respectant la définition de l'IOSTS sémantique. Dans cette couche, les IOSTSs ne sont pas regénérés à chaque fois que de nouvelles traces sont reçues. Ils sont complétés à la volée.

Les couches 3 à N (N étant un entier fini) sont composées de règles qui simulent la capacité d'un expert humain à analyser des transitions dans le but de déduire la sémantique de l'application. Ces analyses et déductions ne sont souvent pas réalisées d'une seule traite. C'est pourquoi le Générateur de modèles est architecturé par un nombre non-préalablement défini mais fini de couches. Chacune d'entre elles prend un IOSTS S_{i-1} en entrée, qui est le modèle résultant de la couche précédente. Cet IOSTS, qui représente la base de faits au travers de ses transitions, est analysé par des règles pour inférer un nouvel IOSTS qui, au mieux, est plus riche sémantiquement que le précédent. Les couches les plus basses (niveau 3 au moins) sont composées de règles génériques qui peuvent être appliquées sur plusieurs applications de même type. Par exemple, nous rassemblerons les règles permettant d'identifier de manière fiable un protocole réseau au sein d'une même couche. Certaines

de ces règles permettent d'enrichir des transitions qui vont s'avérer pertinentes pour les couches supérieures. D'autres règles peuvent effectuer des agrégations simples de transitions successives en une seule, composée d'une action plus abstraite. Les couches les plus hautes possèdent des règles plus précises qui peuvent être dédiées à une application spécifique. Ces règles peuvent être employées pour effectuer des agrégations de transitions ou pour enrichir le sens d'une action, toujours en étant fortement liées au métier de l'application. Si un expert du domaine pouvait écrire la plupart des règles des couches précédentes, les couches les plus hautes nécessitent une expertise sur l'application cible.

Afin que la génération de modèles puisse se faire de façon déterministe et finie, les règles de ces différentes couches doivent respecter les hypothèses suivantes :

1. (complexité finie) : une règle ne peut s'appliquer qu'un nombre fini de fois sur une même base de faits,
2. (justesse) : les règles d'inférence sont Modus Ponens,
3. (pas d'élimination de connaissance implicite) : après l'application d'une règle r exprimée par la relation $r : T_i \rightarrow T_{i+1} (i \geq 2)$, avec T_i une base de faits comportant des Transitions, pour toute transition $t = (l_n, l_m, a(p), G, A)$ extraite de T_{i+1}, l_n est accessible depuis l_0.

Dans la suite, nous détaillons ces différentes couches en prenant l'exemple du contexte des applications Web et nous donnons des exemples de règles. Nous avons choisi le moteur d'inférence Drools [1], qui accepte des règles de la forme *When condition sur les faits Then actions sur les faits.* Ainsi, les règles présentées seront de ce format. Drools est un outil flexible, écrit en Java, qui emploie des bases de faits implantés par des objets. Pour correspondre à la définition d'un IOSTS, nous avons des faits de type *Location* et *Transition.* Cette dernière classe est composée de deux états symboliques *Linit*, *Lfinal*, ainsi que de deux collections de données *Guard* et *Assign* décrivant les gardes et assignations liées à une transition d'un IOSTS comme défini en section 3.

4.1 Couche 1 : représentation des traces et filtrage

Les traces d'une application Web sont basées sur les messages HTTP (requêtes et réponses). Le protocole HTTP est conçu de manière à ce que chaque requête HTTP soit obligatoirement suivie d'une seule réponse HTTP. De ce fait, les traces données à la couche 1 sont des séquences de couples (requête HTTP, réponse HTTP). Cette couche commence par formater ces couples afin d'obtenir un format de traces strict et plus simple à analyser.

Une requête HTTP est un message textuel contenant un verbe HTTP (*GET*, *POST*, etc), suivi d'un identifiant uniforme de ressource (URI). Elle peut également contenir des en-têtes tels *Host*, *Connection* ou *Accept.* La réponse HTTP correspondante est également textuelle et contient au moins un code de retour. Cette réponse peut aussi contenir des en-têtes (par exemple *Content-Type*, *Content-Length*) et un corps de réponse. Ce sont ces informations que nous identifions dans nos traces structurées.

La proposition ci-après permet transformer ces données textuelles en actions valuées structurées.

Definition 3 (Traces Structurées) *Soit $t = req_1, resp_1, ..., req_n, resp_n$ une trace HTTP brute, composée d'une séquence alternée de requêtes HTTP req_i et de réponses HTTP $resp_i$. La trace structurée σ de t est la séquence $(a_1(p), \theta_1)...(a_n(p), \theta_n)$ telle que :*

- *a_i est le verbe HTTP utilisé pour effectuer la requête req_i,*
- *p est l'ensemble des paramètres $\{URI, request, response\}$,*
- *θ_i est la valuation $p \rightarrow D_p$ qui assigne une valeur à chaque variable p. θ est obtenu des valeurs extraites dans req_i et $resp_i$.*

[1] http://www.jboss.org/drools/

L'ensemble des traces structurées est noté ST.

Habituellement, un utilisateur exécute, via un navigateur Web, une requête qui représente la requête principale. Pour autant, le navigateur va généralement déclencher d'autres sous-requêtes qui permettent de rapatrier, par exemple, des images ou des fichiers CSS et JavaScript. De manière générale, ces requêtes n'ont aucune valeur ajoutée en matière de sémantique pour une application. C'est pourquoi nous proposons un ensemble de règles dans la couche 1 qui permettent de reconnaître ces sous-requêtes et de les éliminer.

De telles sous-requêtes peuvent être identifiées de plusieurs manières. Par exemple, si une image est récupérée, l'URI de cette requête se termine souvent par une extension de fichier de type image. De plus, lorsque l'en-tête $Content - Type$ d'une réponse est fourni, il peut également être analysé pour reconnaître toute sous-requête non pertinente. En nous basant sur ces informations, nous avons créé des ensembles de règles dont le format est le suivant :

```
rule "Filter"
when
    $t: HttpVerb(condition on the content)
then
    retract($t);
end
```

Ces règles prennent une condition sur le contenu des requêtes et des réponses, et suppriment toute action valuée non désirée. En guise d'exemple concret, la Figure 2 permet de supprimer les actions qui permettent de récupérer des images de type PNG.

```
rule "Filter PNG images"
when
    \$va: Get(request.mime_type = 'png' or
    request.file_extension = 'png')
then
    retract(\$va);
end
```

Figure 2: Exemple de règle de filtrage

Après le déclenchement de ces règles de niveau 1, nous obtenons un ensemble de traces formatées *ST* composées d'actions valuées, à partir desquelles la couche suivante peut extraire les premiers IOSTSs.

4.2 Couche 2 : transformation des traces en IOSTSs

Intuitivement, cette transformation est fondée sur la Définition 2 et la transformation en IOLTS sémantique. En fait, deux IOSTSs sont construits : le premier structuré, sous forme d'arbre, représente les traces. Le second est une sur-approximation du premier IOSTS. Ceux-ci sont construits en appliquant les étapes suivantes :

1. des séquences d'exécution sont calculées à partir des traces structurées en injectant des états entre les actions valuées,
2. le premier IOSTS nommé $\mathcal{S}$ est dérivé de ces séquences d'exécution et, est ensuite minimisé par bisimulation [Fer89],
3. le second IOSTS, noté $App(\mathcal{S})$ est obtenu à partir de $\mathcal{S}$, en fusionnant certains états symboliques, puis en appliquant également une technique de minimisation par bisimulation.

La base de faits résultante est composée d'objets *Transition(Action,Guard,Assign,Linit,Lfinal)*, composés respectivement d'une action, d'une garde, d'une assignation de variables internes, d'un état symbolique de départ et d'un état symbolique d'arrivée. Le second IOSTS est donné avec des objets de type AppTransition.

Traces vers Séquences d'exécution

Etant donné une trace σ, une séquence d'exécution r est construite en injectant des états sur les côtés droit et gauche de chaque action valuée de σ. En gardant à l'esprit la définition d'un IOLTS sémantique, un état est un tuple de la forme $((URI, k), v_\emptyset)$ avec $v_\emptyset$ la valuation nulle et (URI, k) un tuple composé d'une URI et d'un entier ($k \geq 0$). (URI, k) sera par la suite un état symbolique de l'IOSTS généré. Comme nous souhaitons préserver l'ordre séquentiel des traces, quand une URI déjà rencontrée est une nouvelle fois détectée, l'état résultant est composé de l'URI couplée à un entier k qui est incrémenté pour créer un nouvel état unique.

La traduction des traces structurées ST en un ensemble de séquences d'exécution SR est réalisée par l'Algorithme 1. Il gère un ensemble $States$ qui stocke les états construits. Toutes les séquences d'exécution r de $Runs$ commencent par le même état $(l0, v_\emptyset)$. L'Algorithme 1 couvre ensuite chaque action (a_i, θ_i) de r pour construire le prochain état s. Il extrait la valuation $URI = val$ de θ_i, qui donne la valeur de l'URI de la prochaine ressource atteinte après l'action a_i. L'état $s = ((val, k+1), v_\emptyset)$ est construit avec k tel qu'il existe $((URI, k), v_\emptyset) \in States$ composé du plus grand entier $k \geq 0$. La séquence d'exécution r est complété avec l'action valuée (a_i, θ_i) suivie de l'état s. Enfin, SR rassemble toutes les séquences d'exécution construites.

Algorithm 1: Traduction de traces en séquences d'exécution

```
   input  : Ensemble de Traces ST
   output: Ensemble de séquences d'exécution SR
1  States := ∅;
2  foreach trace σ = (a_0, θ_0)...(a_n, θ_n) ∈ ST do
3      r:= null;
4      for 0 ≤ i ≤ n do
5          if i==0 then
6              r := r.(s0, v_∅)
7          extraire la valuation URI = val de θ_i;
8          if ((val, 0), v_∅) ∉ States then
9              s := ((val, 0), v_∅);
10         else
11             s := ((val, k+1), v_∅) avec k ≥ 0 le plus grand entier tel que ((val, k), v_∅) ∈ States;
12         States := States ∪ {s};
13         r := r.(a_i, θ_i).s
14     SR := SR ∪ {r}
```

4.2.1 Génération d'IOSTSs

Le premier IOSTS $\mathcal{S}$ est directement dérivé de l'ensemble SR. Il correspond à un arbre composé de chemins, chacun exprimant une trace commençant par le même état initial. Cet IOSTS est ensuite minimisé.

Definition 4 *Etant donné un ensemble de séquences d'exécution SR, l'IOSTS $\mathcal{S}$ est appelé l'IOSTS arbre de SR et correspond à $< L_\mathcal{S}, l0_\mathcal{S}, V_\mathcal{S}, V0_\mathcal{S}, I_\mathcal{S}, \Lambda_\mathcal{S}, \rightarrow_\mathcal{S} >$ tel que :*
$L_\mathcal{S} = \{l_i \mid \exists r \in SR, (li, v_\emptyset)$ est un état de $r\}$, $l0_\mathcal{S}$ est l'état symbolique initial tel que $\forall s \in SR$, s commence par $(l0_\mathcal{S}, v_\emptyset)$, $V_\mathcal{S} = \emptyset$, $V0_\mathcal{S} = v_\emptyset$, $\Lambda_\mathcal{S} = \{a_i(p) \mid \exists s \in SR, (a_i(p), \theta_i)$ est une action valuée de $s\}$, $\rightarrow_\mathcal{S}$ et I_s sont définis par la règle d'inférence suivante appliquée à tout élément $s \in SR$:

$$\frac{s_i(a_i(p),\theta_i)s_{i+1} \text{ est un terme de } s, s_i=(l_i,v_\emptyset), s_{i+1}=(l_{i+1},v_\emptyset), G_i= \bigwedge_{(x_i=vi\in\theta_i)} x_i == vi}{l_i \xrightarrow{a_i(p),G_i,(x:=x)_{x\in V}}_\mathcal{S} l_{i+1}}$$

Ici, aucune variable n'a été modifiée et c'est pour cela que la fonction identité est appliquée.

En partant de l'IOSTS arbre $\mathcal{S}$, une sur-approximation de $\mathcal{S}$ peut maintenant être logiquement déduite en fusionnant ensemble tous les états symboliques de la forme $(URI, k)_{k>0}$ en un seul. Ce choix est arbitraire et dépend du type d'applications analysées. Cet IOSTS, qui peut potentiellement être cyclique, exprime généralement plus de comportements et devrait être fortement réduit en comptant moins d'états. Mais c'est également une approximation qui peut révéler de nouvelles séquences d'actions qui n'existent pas dans les traces de départ. Ce modèle peut être particulièrement intéressant pour aider à la création d'un modèle complet ou à améliorer la couverture de méthodes de test spécifiques, comme le test de sécurité, grâce aux nouveaux comportements représentés. Cependant, il est clair qu'une méthode de test de conformité ne doit pas prendre ce modèle en tant que référence pour générer des cas de test.

Definition 5 *Soit $\mathcal{S}$ l'IOSTS arbre de SR. L'approximation de $\mathcal{S}$, noté App($\mathcal{S}$), est l'IOSTS* $< L_{App}, l0_{App}, V_{App}, V0_{App}, I_{App}, \Lambda_{App}, \rightarrow_{App} >$ *tel que :*
$L_{App} = \{(URI) \mid (URI, k) \in L_{\mathcal{S}}, k \geq 0\}$, $l0_{App} = l0_{\mathcal{S}}$, $V_{App} = V_{\mathcal{S}}$, $V0_{App} = V0_{\mathcal{S}}$, $\Lambda_{App} = \Lambda_{\mathcal{S}}$, $\rightarrow_{App} = \{(URI_m) \xrightarrow{a(p),G,A} (URI_n) \mid (URI_m, k) \xrightarrow{a(p),G,A} (URI_n, l) \in \rightarrow_{\mathcal{S}} (k \geq 0, l \geq 0)\}$.

Un IOSTS $\mathcal{S}$ et son approximation sont composés de séquences de transitions dérivées des traces structurées ST. Ainsi définis, les comportements de ST et $\mathcal{S}$ sont équivalents puisque les séquences d'exécution (ordonnées) sont transformées en chemins de $\mathcal{S}$. L'approximation de $\mathcal{S}$ partage des comportements de $\mathcal{S}$ et ST mais peut également contenir de nouveaux comportements. Ceci est défini par la Proposition suivante.

Proposition 6 *Soit ST un ensemble de traces structurées et SR l'ensemble de séquences d'exécution. Si $\mathcal{S}$ est l'IOSTS arbre de SR, nous avons $Traces(\mathcal{S}) = ST$ et $Traces(App(\mathcal{S})) \supseteq ST$.*

Par soucis de lisibilité, nous ne présentons pas ici les règles de la couche 2 qui correspondent aux définitions et algorithmes décrits ci-avant.

Exemple 4.1 Pour illustrer cette couche, nous prenons en exemple des traces obtenues depuis l'application Web GitHub [2] après avoir appliqué les actions suivantes : connexion à un compte existant, sélection d'un projet existant, puis déconnexion. Ces quelques actions produisent déjà de nombreuses requêtes et réponses HTTP. En effet, un navigateur envoie trente requêtes HTTP en moyenne pour afficher une seule page de l'application Web GitHub. En filtrant les traces de notre exemple, nous récupérons l'ensemble des traces structurées suivant où les requêtes et réponses HTTP ont été omises, là encore par soucis de lisibilité :

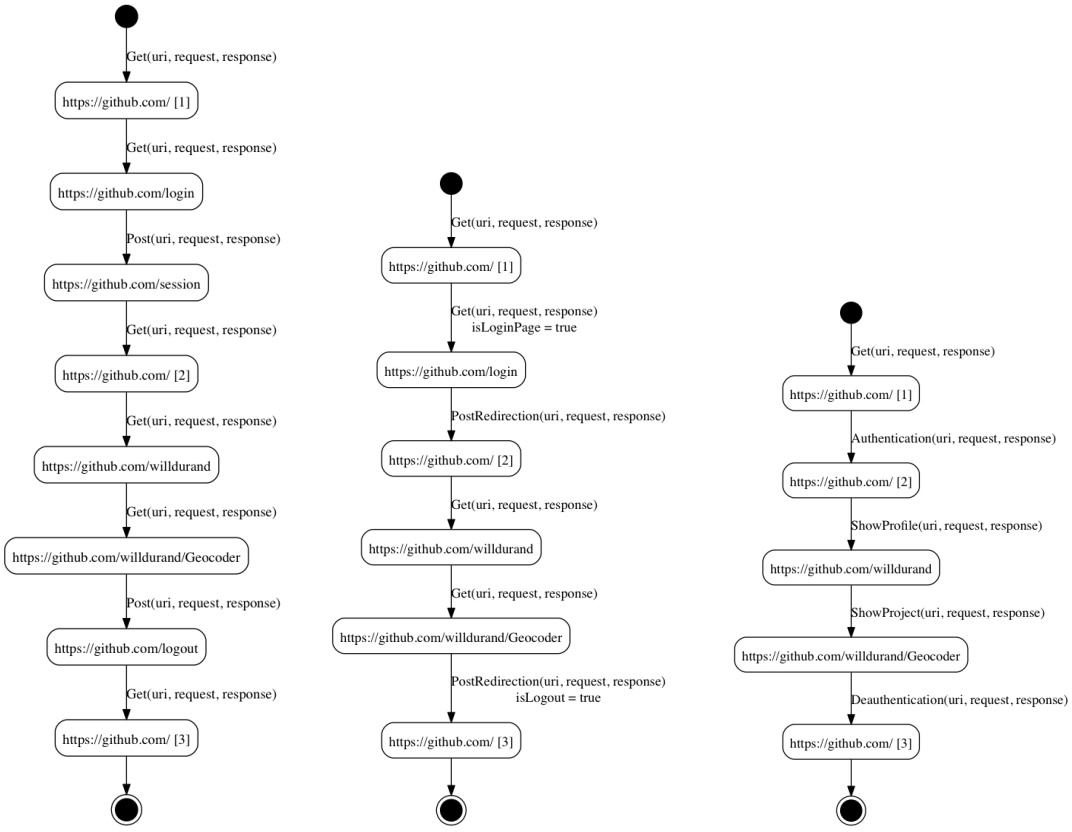
```
1 GET(https://github.com/)
  GET(https://github.com/login)
3 POST(https://github.com/session)
  GET(https://github.com/)
5 GET(https://github.com/willdurand)
  GET(https://github.com/willdurand/Geocoder)
7 POST(https://github.com/logout)
  GET(https://github.com/)
```

Après application des règles de niveau 2, nous obtenons un IOSTS présenté en Figure 3(a). Les états symboliques sont étiquetés avec l'URI trouvée dans la requête, accompagnée d'un entier pour conserver la structure arborescente de la trace de départ. Les actions sont composées du verbe HTTP et des variables URI, request, et response. Cet IOSTS reflète précisément le comportement de la trace mais reste toujours difficile à lire. Des actions offrant un plus haut niveau d'abstraction seront déduites dans les couches supérieures.

4.3 Couches 3-N : montée en abstraction des IOSTSs

Les transitions de l'IOSTS généré à l'étape précédente comportent des données extraites des requêtes et réponses HTTP. Comme indiqué précédemment, les règles des couches plus hautes

[2] https://www.github.com/



(a) IOSTS $\mathcal{S}$ (b) IOSTS $\mathcal{S}_2$ (c) IOSTS $\mathcal{S}_3$

Figure 3: IOSTSs générés avec notre exemple

analysent ces données pour tenter d'apporter plus de sens au modèle et, souvent, de réduire la taille de l'IOSTS initial.
A partir d'un IOSTS $\mathcal{S}$ représenté avec des objets *Transition* et *Location* donnés par la couche inférieure, chaque couche exécute ensuite la séquence d'étapes suivante :

1. exécution des règles d'inférence et déduction de nouveaux faits (IOSTS $\mathcal{S}_i$, $i > 1$),
2. construction de l'IOSTS $App(\mathcal{S}_i)$ et minimisation des deux IOSTSs,
3. stockage des 2 IOSTSs.

Sans perte de généralité, nous limitons la structure des règles afin de garder un lien entre les IOSTSs générés. Ainsi, chaque règle de la couche i ($i \geq 3$) va soit enrichir le sens des actions d'une transition, soit combiner des séquences de transitions en une seule pour rendre plus abstrait le nouvel IOSTS. Il en résulte qu'un IOSTS $\mathcal{S}_i$ est composé exclusivement d'états symboliques provenant du premier IOSTS $\mathcal{S}$. Par conséquent, pour une transition ou un chemin de $\mathcal{S}_i$, il est possible de retrouver le chemin complet associé dans $\mathcal{S}$. Ceci est capturé dans la Proposition suivante :

Proposition 7 *Soit $\mathcal{S}$ le premier IOSTS généré à partir de l'ensemble de traces structurées ST. L'IOSTS $\mathcal{S}_i(i > 1)$ produit par la couche i possède un ensemble d'états symboliques $L_{\mathcal{S}_i}$ tel que $L_{\mathcal{S}_i} \subseteq \mathcal{S}$.*

Dans la suite, nous détaillons deux couches spécialisées pour des applications Web.

4.3.1 Couche 3

Comme indiqué en Section 2, la couche 3 comporte un ensemble des règles génériques qui peuvent être appliquées sur un large ensemble d'applications appartenant à la même catégorie.
Son rôle est de :

- déduire une signification pour certaines transitions et de les enrichir. Dans cette étape, nous avons choisi d'ajouter des assignations de variables internes aux transitions. Celles-ci auront pour but d'aider à la déduction d'actions plus abstraites dans les couches supérieures,
- créer des agrégations génériques de transitions. Lorsque les contenus de quelques transitions successives respectent une condition donnée, alors celles-ci sont remplacées par une seule transition transportant une nouvelle action.

Par exemple, la règle présentée en Figure 4 permet de reconnaître la navigation vers une page de connexion. Si le contenu de la réponse de toute requête envoyée avec le verbe HTTP *GET* contient un formulaire de connexion, alors cette transition est marquée comme étant une page de connexion. La Figure 5 présente la règle qui marque les transitions qui sont utilisées pour se déconnecter.

La règle de la Figure 6 est un exemple d'agrégation simple de transitions. Son but est d'inférer que, lorsqu'une requête envoyée avec la méthode *POST* possède une réponse identifiée comme étant une redirection (en se basant sur le statut HTTP 301 ou 302) et qu'elle est suivie d'une requête *GET*, alors ces deux transitions sont réduites en une seule action *PostRedirection*.

```
rule "Identify Login Page"
when
    $t: Transition(Action == GET, Guard.
      response.content contains('login-form'))
then
    modify ($t) { Assign.add("isLoginPage:=true") }
end
```

Figure 4: Règle de reconnaissance d'une page de connexion

```
rule "Identify Logout Request"
when
    $t: Transition(Action == GET, Guard.
      request.uri matches("/logout"))
then
    modify ($t1) { Assign.add("isLogout:=true") }
end
```

Figure 5: Règle de reconnaissance d'une requête de déconnexion

Exemple 4.2 Si nous appliquons ces règles sur l'IOSTS présenté en Figure 3(a), nous obtenons le nouvel IOSTS de la Figure 3(b). Sa taille est réduite puisqu'il possède désormais 6 transitions contre 9 auparavant. Cependant, ce nouvel IOSTS ne reflète pas encore très bien le scénario de connexion. Des règles permettant la déduction d'actions plus abstraites sont nécessaires, et c'est au niveau suivant que l'on va les retrouver.

4.3.2 Couche 4

Cette couche a pour but d'inférer un IOSTS de plus haut niveau d'abstraction, qui devrait avoir une dimension plus réduite mais également être composé d'actions ayant une sémantique plus forte. Ces règles peuvent avoir différentes formes :

```
rule "Identify Redirection after a Post"
when
    $t1: Transition(Action == POST and
       (Guard.response.status == 301 or Guard.response.
       status == 302) and $l1final := Lfinal)
    $t2: Transition(Action == GET, linit == $l1final,
       $l2linit:=Linit)
    not (Transition (Linit == $l2linit))
then
    insert(new Transition("PostRedirection", Guard(
       $t1.Guard, $t2.Guard), Assign($t1.Assign,
       $t2.Assign), $t1.Linit, $t2.Lfinal );
    retract($t1);
    retract($t2);
end
```

Figure 6: Aggregation simple

- elles peuvent être appliquées sur une transition uniquement. Dans ce cas, les règles modifient l'action de la transition pour donner plus de sens,
- elles peuvent aussi agréger plusieurs transitions successivement pour obtenir une transition étiquetée avec une action plus abstraite.

Nous présentons trois exemples de règles ci-dessous. La première illustrée en Figure 7 a pour but de reconnaître l'authentification d'un utilisateur. Cette règle se base sur la variable interne *isLoginPage* ajoutée au niveau 3. Cette règle se lit comme suit : si une page de "login" est présentée à l'utilisateur, puis qu'une redirection est effectuée après avoir déclenchée une requête $POST$, alors cet enchaînement décrit une phase d'authentification, et ces deux transitions peuvent être réduites en une seule, marquée avec l'action "Authentication". De la même manière, la seconde règle en Figure 8 reconnaît une phase de déconnexion et construit une transition marquée avec l'action "Deauthentication". Cette règle indique que lorsqu'une action *PostRedirection* est observée alors l'action résultante est une "Deauthentication".

```
rule "Identify Authentication"
when
    $t1: Transition(Action == GET,
       Assign contains "isLoginPage:= true",
    $t1final:=Lfinal)
    $t2: Transition(Action == PostRedirection,
       linit == $t1lfinal, $t2linit:=Linit)
    not (Transition (linit == $t2linit))
then
    insert(new Transition("Authentication",
       Guard($t1.Guard,$t2.Guard), Assign($t1.Assign,
       $t2.Assign), $t1.Linit, $t2.Lfinal );
    retract($t1);
    retract($t2);
end
```

Figure 7: Reconnaissance de l'action Authentication

```
rule "Identify Deauthentication"
when
    $t: Transition(action == PostRedirection,
       Assign contains "isLogout:=true")
then
    modify ($t) (setAction "Deauthentication"));
end
```

Figure 8: Reconnaissance de l'action Deauthentication

Il est également possible de proposer des règles d'inférences spécifiques à une application cible, de manière à enrichir le modèle en matière de connaissance. Si l'on reprend l'exemple de l'application Web GitHub, celle-ci possède une grammaire d'URL qui lui est propre (via un système de routing). Les utilisateurs de GitHub possèdent une page de profil disponible à l'adresse suivante : *https://github.com/username* où *username* est un nom d'utilisateur. Cependant, il existe des mot-clés réservées par GitHub, comme *edu* et *explorer*. La règle décrite en Figure 9 se base sur cette notion pour produire une nouvelle action nommée "ShowProfile", afin d'ajouter plus de sémantique au modèle. Nous avons suivi la même logique pour créer une nouvelle action nommée "ShowProject" puisque tout projet hébergé sur la plateforme GitHub possède une URL qui lui est propre, à savoir : *https://github.com/username/project_name*. La règle correspondante est décrite en Figure 10.

```
rule "GitHub profile pages"
when
    $t: Transition(action == GET, (
       Guard.request.uri matches "/[a-zA-Z0-9]+$",
       Guard.request.uri not in [ "/edu", "/explorer" ]
        ))
then
    modify ($t) (setAction("ShowProfile"));
end
```

Figure 9: Reconnaissance de choix de profil utilisateur

```
rule "GitHub project pages"
when
    $t: Transition(action == GET,
       Guard.request.uri matches "/[a-zA-Z0-9]+/.+$")
then
    modify ($t) (setAction("Showprofile"));
end
```

Figure 10: Reconnaissance de choix de projet

Exemple 4.3 L'application des ces quatre règles permet de créer un IOSTS final présenté en Figure 3(c). Cet IOSTS peut maintenant être utilisé pour mieux cerner les fonctionnalités de l'application. En effet, les actions ont plus de sens qu'initialement et décrivent clairement le fonctionnement de l'application. Des outils de vérification ou de test peuvent aussi être employés sachant que nous avons gardé un lien entre les IOSTSs $_i$ et celui d'origine S composé de toutes les actions.

5 Travaux relatifs et conclusion

L'inférence de modèles est un domaine de recherche récent qui se penche sur la génération de modèles partiels décrivant des comportements fonctionnels d'applications. Plusieurs travaux de natures différentes ont été proposés sur ce domaine.

Par exemple, dans [ZZXM11], les auteurs infèrent des spécifications depuis des documentations d'API écrites en langage naturel. De telles spécifications permettent de détecter des déviances entre les implémentations et leurs documentations. Mais encore faut-il avoir ce type de documents. Observer une application en cours d'exécution semble être une meilleure alternative, et c'est ce que décrivent Pradel et al. dans [PG09] : des spécifications sont construites à partir d'une application en cours d'exécution, en extrayant les séquences d'appels de méthode depuis de grandes quantités de traces. Les modèles produits sont très détaillés, mais ne reflètent pas les fonctionnalités de l'application. La plupart des autres travaux produisent des modèles

fonctionnels d'applications événementielles au moyen de tests automatiques. Certains travaux [MBN03, ANHY12, MvDL12, AFT+12, YPX13] proposent de retrouver des modèles d'applications événementielles (application de bureau, Mobiles ou Web) en effectuant du test automatique. Certains de ces travaux expérimentent l'application en boite blanche via du test concolique pour explorer et retrouver des chemins d'exécution [ANHY12]. D'autres solutions [YPX13] recouvrent une spécification en analysant le code de l'application pour trouver les événements associés aux interfaces et en effectuant une exploration avec une stratégie de parcours en profondeur. Mais une majorité des méthodes est orientée test en boite noire. Certains outils comme GUITAR [MBN03] produisent des graphes de flots d'événements et des arbres illustrants des séquences d'actions. Toutes ces méthodes proposent soit des modèles très vastes montrant toutes les intéractions possibles, soit des modèles très simples uniquement composés des événements. Seuls les travaux [MvDL12, YPX13] proposent de réduire le nombre d'états des modèles générés en rassemblant les GUIs qui ont des éléments non éditables similaires.

Notre proposition prend une autre direction en inférant plusieurs modèles exprimant différents niveaux d'abstraction au moyen de règles d'inférence qui capturent la connaissance d'un expert. La première contribution de cette approche réside dans la flexibilité et l'extensibilité amenée par l'utilisation des règles d'inférence. Les mêmes règles peuvent en effet être appliquées sur plusieurs applications de même type ou sur une seule application avec des règles spécifiques. Dans ce dernier cas, seules quelques règles doivent être proposées. Notre approche peut être appliquée à des applications événementielles ou non. Mais pour ce type d'application, nous proposons un module d'exploration automatique guidé par des stratégies qui peuvent être revues suivant l'application analysée. De plus, les méthodes précédentes emploient principalement soit un parcours en profondeur, soit un parcours en largeur pour explorer l'application. Nous proposons une couche qui permet d'implanter tout type de stratégie via des règles d'inférence.

La plateforme présentée en Section 2 est en cours d'implantation dans un outil appelé *Autofunk* (Automatic Functional model inference). Pour l'instant, nous l'avons appliqué sur l'application Web GitHub, sur un enregistrement composé de 840 requêtes HTTP, avec un Générateur de modèles incluant 5 couches rassemblant 18 règles. Parmi celles-ci, 3 sont dédiées à l'application Github. Après le filtrage de traces (Couche 1), nous avons obtenu un premier IOSTS composé de 28 transitions. Les 4 couches suivantes ont construit, en quelques secondes, un dernier IOSTS $\mathcal{S}_4$ composé de 13 transitions. L'IOSTS approximation $App(\mathcal{S}_4)$ est illustré en Figure 11. La plupart des actions ont une signification précise reflétant les actions faites par l'utilisateur pendant l'enregistrement des traces. Ainsi, il est facile de déduire que l'utilisateur a crée, choisi, effacé et lu des publications de projets.

Par la suite, nous avons l'intention de considérer d'autres applications, et notamment les systèmes industriels de notre partenaire Michelin. Cependant, avec ce type de système, d'autres problématiques sont soulevées. Par exemple, ces systèmes industriels comportent souvent des actions asynchrones. Nos règles d'inférence ne prennent pour l'instant pas en compte ce type d'actions. De plus, l'écriture de règles peut devenir aussi difficile que l'écriture d'un modèle. C'est pourquoi nous sommes en train de travailler sur une interface qui aidera à la conception de règles.

References

[AFT+12] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 258–261, New York, NY, USA, 2012. ACM.

[ANHY12] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 59:1–59:11, New York, NY, USA, 2012. ACM.

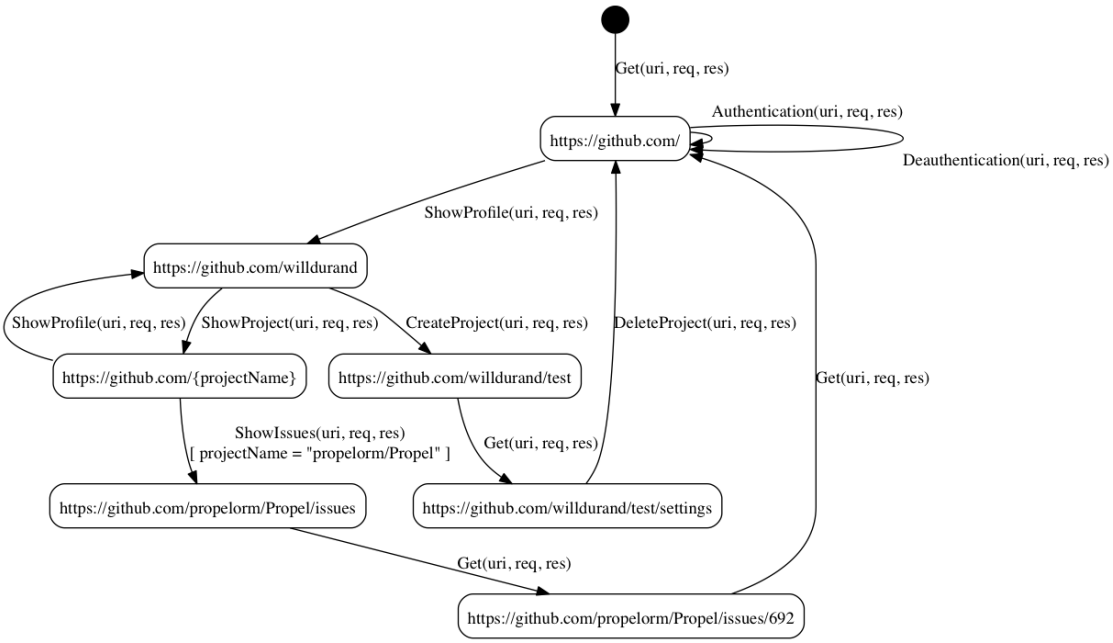


Figure 11: IOSTS $App(\mathcal{S}_4)$ obtenu depuis l'application Web Github

[Fer89] Jean-Claude Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13:13–219, 1989.

[FTW05] L. Frantzen, J. Tretmans, and T.A.C. Willemse. Test Generation Based on Symbolic Specifications. In J. Grabowski and B. Nielsen, editors, *FATES 2004*, number 3395 in Lecture Notes in Computer Science, pages 1–15. Springer, 2005.

[MBN03] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of the 10th Working Conference on Reverse Engineering*, WCRE '03, pages 260–, Washington, DC, USA, 2003. IEEE Computer Society.

[MvDL12] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3:1–3:30, 2012.

[PG09] Michael Pradel and Thomas R. Gross. Automatic generation of object usage specifications from large method traces. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 371–382, Washington, DC, USA, 2009. IEEE Computer Society.

[YPX13] Wei Yang, Mukul R. Prasad, and Tao Xie. A grey-box approach for automated gui-model generation of mobile applications. In *Proceedings of the 16th international conference on Fundamental Approaches to Software Engineering*, FASE'13, pages 250–265, Berlin, Heidelberg, 2013. Springer-Verlag.

[ZZXM11] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring specifications for resources from natural language api documentation. *Autom. Softw. Eng.*, 18(3-4):227–261, 2011.