AFADL 2014

Actes des 13èmes journées sur les

Approches Formelles dans l'Assistance au Développement de Logiciels

Edités par Catherine Dubois et Régine Laleau

11 et 12 juin 2014
Conservatoire National des Arts et Métiers (CNAM), Paris, France

Préface

La 13ème édition d'AFADL, atelier francophone sur les Approches Formelles dans l'Assistance au Développement de Logiciels, se tiendra les 11 et 12 juin 2014 au Conservatoire National des Arts et Métiers à Paris. Elle est organisée conjointement avec 3 autres manifestations : CAL 2014, Conférence sur les Architectures Logicielles, CIEL 2014, Conférence en IngénieriE du Logiciel et les journées nationales du GDR Génie de la Programmation et du Logiciel (GPL). Cet événement permettra ainsi à toute la communauté francophone des chercheurs en génie logiciel de se retrouver et échanger.

L'atelier AFADL rassemble de nombreux acteurs académiques et industriels intéressés par la mise en œuvre de techniques formelles aux divers stades du développement des logiciels et/ou des systèmes. Il a pour objectif de mettre en valeur les travaux récents effectués autour de thèmes comme :

- les techniques et outils formels contribuant à assurer un bon niveau de confiance dans la construction de logiciels et de systèmes,
- les méthodes et processus permettant d'exploiter efficacement les techniques et outils formels disponibles ou proposés,
- les méthodes et processus permettant l'utilisation de techniques formelles différentes et hétérogènes dans un même développement,
- les leçons tirées de la mise en œuvre de ces outils ou principes sur des études de cas ou des applications industrielles.

Nous aurons l'honneur d'accueillir, en association avec les journées nationales du GDR GPL, les conférences CAL et CIEL, les orateurs invités suivants :

- Roland Ducournau (LIRMM, U. Montpellier) : *Les talons d'Achille de la programmation par objets*,
- Christine Paulin (Université Paris XI, LRI) : *Preuves formelles d'algorithmes probabilistes*
- Gérard Morin (Esterel Technologies) : *SCADE Model-Based Requirements Engineering*

Les actes d'AFADL 2014 comprennent 6 articles longs, 6 articles courts présentant tous des résultats nouveaux. Un des articles présente un projet ANR. Enfin 7 présentations concernent des résultats dèjà présentés dans des conférences internationales. Cette dernière catégorie, nouvelle pour AFADL, offre la possibilité de présenter à la communauté française des résultats publiés récemment. Comme les années précédentes, les contributions couvrent un large éventail de techniques, méthodes et applications. Enfin, une session posters et démonstration d'outils a été organisée conjointement avec les conférences et journées co-localisées.

Nous remercions les membres du comité de programme pour leur travail qui a contribué à produire un programme de qualité, ainsi que tous les auteurs qui ont soumis un article et sans qui il n'y aurait plus d'atelier AFADL.

Nous remercions les membres du comité d'organisation des journées AFADL-CAL-CIEL-GPL 2014 qui ont pris en charge tous les aspects logistiques.

Le 26 mai 2014

Catherine Dubois
Régine Laleau
Présidentes du comité de programme AFADL 2014

Comité de programme

Présidentes

Catherine Dubois, CEDRIC - ENSIIE - Evry
Régine Laleau, LACL - Université Paris-Est, Créteil

Membres

Yamine Ait Ameur, IRIT - INPT-ENSEEIHT, Toulouse
Béatrice Bérard, LIP6 - Université Pierre et Marie Curie, Paris
Sandrine Blazy, IRISA - Université Rennes 1, Rennes
Frédéric Boniol, ONERA, Toulouse
Jean-Michel Bruel, IRIT - Université de Toulouse, Toulouse
Pierre Casteran, LABRI - Universite Bordeaux 1, Bordeaux
Sylvain Conchon, LRI - Université Paris-Sud, Orsay
Christèle Faure, SafeRiver, Paris
Akram Idani, LIG - Université Joseph Fourier, Grenoble
Jacques Julliand, FEMTO-ST - Université de Franche-Comté, Besançon
Florent Kirchner, CEA LIST, Saclay
Arnaud Lanoix, LINA - Université de Nantes, Nantes
Yves Ledru, LIG - Université Joseph Fourier, Grenoble
Pascale Le Gall, MAS - École Centrale Paris
Yves Le Traon, Université de Luxembourg
Nicole Levy, CEDRIC - CNAM, Paris
Dominique Méry, LORIA - Université de Lorraine, Nancy
Jean-Marc Mota, Thalès Research & Technology, Palaiseau
Ioannis Parissis, LCIS - ESISAR, Valence
François Pessaux, U2IS - ENSTA, Paris
Pascal Poizat, LIP6 - Université Paris Ouest, Nanterre
Marie-Laure Potet, VERIMAG - ENSIMAG, Grenoble
Marc Pouzet, LIENS - ENS, Paris
Vlad Rusu, INRIA, Lille
Sylvie Vignes, LCTI - Télécom Paris-Tech, Paris
Laurent Voisin, Systerel, Aix en provence
Helene Waeselynck, LAAS-CNRS, Toulouse
Virginie Wiels, ONERA / DTIM, Toulouse
Nicky Williams, CEA LIST, Saclay

Table des matières

Articles longs

Articles courts

Présentation de projet

Présentations de recherches publiées récemment

Modélisation et validation formelle des règles d'exploitation ferroviaires

Rahma Ben Ayed[1], Simon Collart-Dutilleul[1], Philippe Bon[1],
Yves Ledru[2] et Akram Idani[2]

[1] Univ. Nord de France, IFSTTAR/COSYS-ESTAS, 20 rue Elisée Reclus, F-59650, Villeneuve d'Ascq, France
{rahma.ben-ayed,simon.collart-dutilleul,philippe.bon}@ifsttar.fr
http://www.ifsttar.fr
[2] UJF-Grenoble 1/Grenoble-INP/UPMF-Grenoble 2/CNRS, LIG UMR 5217, F-38041, Grenoble, France
{yves.ledru,akram.idani}@imag.fr
http://www.liglab.fr

Résumé Le système européen de surveillance du trafic ferroviaire (en anglais, European Rail Traffic Management System, ERTMS) est un système complexe de contrôle/commande et de signalisation ferroviaire mettant en œuvre des règles européennes d'exploitation ferroviaires. Cet article propose une étude de cas basée sur deux scénarios extraits de ces règles, un scénario nominal d'autorisation de mouvement et un scénario exceptionnel de franchissement d'un arrêt. En effet, on trouve dans ces scénarios des aspects fonctionnels et de sécurité. Ces aspects nécessitent, d'une part, une modélisation fonctionnelle enrichie par des modèles décrivant la politique de sécurité et les autorisations données aux agents agissant sur le système, et d'autre part, une validation formelle. Pour ce faire, nous avons utilisé la plate-forme B4MSecure, fondée sur l'approche IDM (Ingénierie Dirigée par les Modèles), produisant à partir des modèles UML des spécifications formelles B. L'objectif de ces spécifications résultantes est de valider ces scénarios à l'aide d'outils d'animation et de preuve de spécifications B afin de garantir une analyse rigoureuse de la fonctionnalité et de la politique de sécurité.

1 Introduction

La sécurité des systèmes critiques ferroviaires est un enjeu majeur des systèmes d'aujourd'hui du fait de leur complexité et des conséquences graves pouvant découler d'erreur de conception. C'est pourquoi, leur validation et leur vérification constituent des tâches d'envergure ayant une place prépondérante dans leur cycle de développement. Ce faisant, un éventail de méthodes formelles existe dans l'optique de mener rigoureusement ces activités. Fondées sur des bases mathématiques, ces méthodes peuvent pallier la complexité et l'ambiguïté des spécificités des systèmes critiques, dès lors qu'elles permettent la spécification et le développement de systèmes, ainsi que la validation et la vérification automatique des propriétés de sécurité ferroviaire.

Dans le cadre du projet ANR « Vers la formalisation des exigences ferroviaires et leur traçabilité » (Performing Enhanced Rail Formal Engineering Constraints Traceability, PERFECT), notre travail se focalise sur les systèmes ERTMS et s'oriente vers la modélisation et la validation de leurs aspects fonctionnels et de sécurité par la méthode formelle B. Plusieurs travaux de recherche ont été menés dans le cadre de la validation et de la vérification des spécifications ERTMS par des méthodes et des techniques formelles. Nous pouvons citer le projet *OpenETCS* mené par Systerel [1] grâce à son expertise dans la maîtrise des systèmes complexes et en particulier des méthodes formelles, telles que la méthode B. Il consiste à consolider l'ensemble des spécifications ERTMS avec des méthodologies formelles et des techniques de preuve.

Dans cet article, nous présentons la modélisation de deux scénarios d'autorisation de mouvement des trains et de franchissement d'un arrêt dans le système ERTMS. Cette modélisation comprend un modèle fonctionnel, qui décrit les principaux concepts de ces mouvements de trains, et un modèle qui précise les responsabilités de chaque intervenant (conducteur de train, agent de circulation, ordinateurs embarqué et au sol). Ce deuxième modèle, dit de sécurité, est exprimé comme un modèle de contrôle d'accès, en utilisant l'outil B4MSecure.

Dans la section 2, nous décrivons notre étude de cas comprenant deux scénarios de règles d'exploitation ferroviaires du système ERTMS/ETCS. Ensuite, nous présentons dans la section 3 le modèle fonctionnel et les modèles de sécurité, ainsi que leur transformation en spécifications B à l'aide de la plate-forme B4MSecure dans la section 4. La section 5 montre la validation formelle des spécifications résultantes en utilisant l'animateur ProB et le prouveur Atelier B. Finalement, la section 6 conclut et présente les améliorations que l'on pourrait apporter à notre travail.

2 Étude de cas

ERTMS [2] est un projet industriel majeur implémenté par huit membres d'UNIFE [3] en Europe. Ce projet vise à harmoniser la signalisation ferroviaire en Europe tout en garantissant la sécurité des circulations. En effet, chaque pays possède son propre système de signalisation ferroviaire implémenté et géré par des entreprises ferroviaires nationales. Chaque système est alors considéré comme indépendant et non-interopérable avec les autres systèmes, ce qui provoque un surcoût financier très important dédié au passage de frontières imposant par exemple le changement de locomotive et/ou du système de signalisation embarqué.

Le système ERTMS est composé du système européen de contrôle des trains (en anglais, European Train Control System, ETCS) qui est le système de contrôle/commande, ainsi que le système de communication GSM-R (en anglais, Global System for Mobile communications - Railways) pour la transmission de

1. SYSTEREL : http ://www.systerel.fr/
2. European Rail Traffic Management System : http ://www.ertms.net
3. European Rail Industry : http ://www.unife.org

données entre l'ETCS embarqué (le système à bord du train) et l'ETCS sol (le système au sol).

Afin de respecter des impératifs de sécurité et d'assurer la bonne gestion des circulations, des règles d'exploitation ferroviaires sont définies régissant la sécurité ferroviaire. Ces règles définissent l'ensemble des interactions entre les systèmes «temps réels» embarqués et les opérateurs tels que le conducteur et l'agent de circulation, notamment dans les modes dégradés.

Notre étude de cas est extraite des principes et des règles d'exploitation du système ERTMS/ETCS niveau 2 appliqués à la ligne de grande vitesse LGV-Est Européenne [2] et des spécifications décrites dans [1], disponibles sur le site d'ERA [4]. Nous avons choisi deux scénarios pour notre étude, un scénario nominal d'autorisation de mouvement (en anglais Movement Authority, MA) et un scénario exceptionnel de franchissement d'un arrêt ETCS (en anglais Override EOA). Le document de référence [2] est un document industriel qui n'est pas encore public et n'est pas dans sa version définitive. Seule l'utilisation du système ETCS niveau 2 est concernée par la présente étude.

2.1 Scénario nominal d'autorisation de mouvement (MA)

En ETCS niveau 2, le train reçoit une MA en « mode » nominal. Celle-ci est une autorisation donnée au train de circuler sur une distance donnée en tant que mouvement supervisé. La MA est mise à jour au fur et à mesure que le train avance.

Une MA est la traduction ETCS d'un itinéraire tracé sur l'infrastructure dont tout ou partie est affecté au train. Cette règle de signalisation se base notamment sur la position du train, sur l'occupation de l'infrastructure par d'autres trains, sur des règles d'exploitation de sécurité et sur les tables horaires de chacun des trains (par exemple l'heure d'arrivée en gare), elles-mêmes dépendantes de règles d'exploitation propres à chaque ligne.

La MA est caractérisée par (cf. paquet 15 de [1]) :

La section représente une distance par rapport au repère géographique du train. Elle est composée éventuellement de sous-sections. Une MA peut être appliquée sur une ou plusieurs sections. La dernière section est appelée la section de fin.

La fin d'autorisation de mouvement (End Of Authority, EOA) est le « lieu » jusqu'au quel le train est autorisé à se mouvoir, où la vitesse but indiquée sur l'interface conducteur-machine (Driver Machine Interface, DMI) est égale à zéro. Elle peut correspondre à un repère d'arrêt ETCS.

La vitesse cible à l'EOA est la vitesse autorisée à l'EOA. Lorsque la vitesse cible n'est pas nulle, l'EOA est appelé la limite de l'autorisation de mouvement (Limit Of Authority, LOA). Cette vitesse cible peut être limitée dans le temps.

4. European Railway Agency : http ://www.era.europa.eu

Le point de danger est un point au-delà de l'EOA qui peut être atteint par l'extrémité avant du train sans risque d'une situation dangereuse.

Le délai d'attente est un délai qui peut être attaché à chaque section. Il est utilisé pour la révocation de l'itinéraire associé lorsque le train ne l'a pas encore emprunté. Il peut être aussi attaché à la section de fin de la MA. Dans ce cas, il est utilisé pour la révocation de la dernière section quand elle est occupée par le train.

Ce scénario, décrit en détail dans [11], se déroule en interaction entre la machine responsable de la gestion de sécurité à bord du train appelée *OnboardSafetyManagement* qui demande une MA, la machine responsable de la gestion de sécurité au sol appelée *TracksideSafetyManagement* qui reçoit la demande et propose une MA après avoir effectué des vérifications et le conducteur qui peut lire la MA proposée via le DMI après sa validation par l'OnboardSafetyManagement. Le conducteur doit respecter les consignes à travers le DMI.

2.2 Scénario exceptionnel (Override EOA)

Override EOA est un scénario déclenché par le conducteur dans des situations dégradées spécifiques en absence de MA. Lorsqu'il est activé par le conducteur, ce scénario permet à un train de franchir un repère d'arrêt ETCS ou un EOA après avoir reçu de l'agent de circulation une autorisation Override EOA et de désactiver certaines protections. Cette autorisation est donnée sous forme d'un ordre écrit. Ce dernier est un message de sécurité délivré par l'agent de circulation au conducteur dans le but de fournir des instructions. Il peut être délivré physiquement ou bien faire l'objet d'une transmission verbale par téléphone ou par radio sol train selon les modalités d'application de la réglementation technique de sécurité relative à la communication. Il existe plusieurs types d'ordres écrits d'ETCS01 à ETCS07. Nous pouvons citer par exemple l'ordre écrit ETCS01 d'autorisation de franchir un EOA traité dans notre étude de cas, l'ordre écrit ETCS03 d'obligation de rester à l'arrêt, l'ordre écrit ETCS04 d'annulation de l'ordre écrit ETCS03, etc. Un ordre écrit contient au minimum le type de l'autorisation, le numéro de l'autorisation, l'heure et la date de sa délivrance, le poste qui le délivre, à quel train il s'adresse, à quel endroit il s'applique et une indication claire, précise et sans ambiguïté des actions à effectuer.

Le scénario d'Override EOA est effectué comme suit :

OverrideEOA.1 Le conducteur (Driver) demande un Override EOA à travers le DMI. Il peut aussi faire avancer, freiner, arrêter le train à travers le DMI.

OverrideEOA.2 La machine responsable de la gestion de sécurité à bord du train (OnboardSafetyManagement) traite la demande d'Override EOA et la transmet au système au sol (TracksideSystem).

OverrideEOA.3 L'agent de circulation (TrafficAgent) reçoit la demande du système au sol (TracksideSystem), crée un ordre écrit et l'autorise. Il peut aussi le modifier et/ou le supprimer.

OverrideEOA.4 L'agent de circulation (TrafficAgent) transmet l'ordre écrit autorisé au système à bord (OnboardSystem).

OverrideEOA.5 La machine responsable de la gestion de sécurité à bord du train (OnboardSafetyManagement) traite cette autorisation afin qu'elle soit affichée sur le DMI.

OverrideEOA.6 Le conducteur (Driver) suit les indications et les consignes affichées sur le DMI suite à cette autorisation.

Les ordres et les instructions, entre autres les instructions de MA et d'Override EOA, sont affichés sur le DMI sous forme de messages textuels ou de symboles. Le DMI permet alors la communication entre le système à bord du train et le conducteur. Ce dernier doit respecter les indications et acquitter les consignes sur le DMI. Il peut aussi informer le système en entrant des informations.

Notre étude de cas basée sur ces deux scénarios révèle l'existence d'interactions entre le système et les agents agissant sur le système. Pour bien modéliser ces interactions, une séparation de la fonctionnalité du système et de la politique de sécurité est nécessaire. Cette politique de contrôle d'accès permettra de modéliser qui est responsable de quelle action. Dans ce qui suit, nous décrivons la modélisation fonctionnelle du système renforcée par la politique de sécurité.

3 Modélisation

Le couplage des méthodes formelles, en l'occurrence la méthode B, et semi-formelles telles qu'UML est un sujet étudié depuis des années grâce à leurs complémentarités [7]. D'une part, UML possède un aspect structurel, intuitif et synthétique grâce à l'utilisation de ses diagrammes. D'autre part, la méthode B permet de construire des spécifications précises et détaillées et d'identifier les ambiguïtés mieux qu'en UML. Les diagrammes UML utilisés sont les diagrammes de classes permettant de décrire le système et ses fonctionnalités. L'aspect dynamique permettant la description du comportement du système n'est pas encore abordé dans nos travaux de recherche.

Les deux scénarios de la section précédente ne se réalisent qu'après le déroulement d'une séquence d'actions par des agents qui possèdent des rôles bien déterminés. Chaque action est autorisée pour un agent et ne peut être exécutée que par ce dernier. Ainsi, chaque agent n'a l'autorisation d'exécuter que des actions qui lui sont permises. De ce fait, des permissions associées aux actions et des rôles associés aux agents peuvent être mis en œuvre dans la modélisation de ces scénarios. Ces concepts peuvent découler des principes de contrôle d'accès à base de rôles (Role Based Acces Control, RBAC) [10] appliqués principalement dans les systèmes d'information. Pour cette raison, nous avons eu recours à l'emploi des principes de RBAC dans notre modélisation, et ce, au travers de l'outil B4MSecure.

B4MSecure [5] [9] est une plate-forme Eclipse fondée sur l'approche IDM. Cette plate-forme permet de modéliser graphiquement en UML un diagramme de classes fonctionnel et des diagrammes de classes pour la politique de contrôle d'accès en utilisant le profil RBAC. L'outil transforme automatiquement tous

5. B4MSecure : http ://b4msecure.forge.imag.fr/

ces modèles en des spécifications B dans le but de les valider formellement. Les travaux de recherche menés dans le cadre du projet Selkis [5], [6] et [7] montrent l'utilité et l'efficacité de cette plate-forme pour la validation formelle des scénarios dans le domaine d'un SI médical tout en détectant les séquences d'opérations pernicieuses.

3.1 Modèle fonctionnel

Le modèle fonctionnel permet de décrire les entités du système impliquées dans la réalisation des scénarios, les caractéristiques de chaque entité, les opérations effectuées sur chaque entité et les relations entre les entités. Les principes de modélisation des diagrammes de classes UML : classes, attributs de classes, méthodes de classes, les relations (agrégation, composition, généralisation/spécialisation), ainsi que les associations et les classes associatives permettent la modélisation fonctionnelle désirée. En effet, les classes sont utilisées pour la modélisation des entités. Les attributs de classes et les méthodes de classes sont utilisés respectivement pour la modélisation des caractéristiques des entités et les opérations effectuées sur les entités. Les relations (agrégation, composition, etc.), les associations et les classes associatives sont utilisées pour la modélisation des relations entre les entités.

La Fig. 1 représente le système ETCS (la classe ETCSSystem) contenant le sous-système à bord (la classe OnboardSystem) qui fait partie du train ETCS (la classe TrainETCS) et le sous-système au sol (la classe TracksideSystem). L'autorisation de mouvement et l'ordre écrit sont modélisés par deux classes (les classes MA et ETCSOrder) contenant les attributs qui les caractérisent. Chacun est associé à un train ETCS et est affiché sous forme de consignes sur le DMI, qui fait partie du système à bord, après son traitement et sa validation. Comme décrit dans la section 2.1, une MA est transmise par le systèmes au sol et elle est composée d'une ou plusieurs sections. Sur une section, au maximum un train peut circuler.

3.2 Politique de sécurité

Des permissions sont accordées aux usagers du système selon les rôles qui leur sont attribués. Les concepts de permission et de rôle sont modélisés conformément au modèle RBAC, et ce, au travers d'une extension d'UML inspirée du profil SecureUML [8]. En effet, une action n'est opérationnelle pour un rôle que si ce rôle a la permission de l'exécuter. Selon le profil RBAC, un rôle est représenté par une classe stéréotypée « Role » et une permission est représentée par une classe associative stéréotypée « Permission » entre le rôle et l'entité du système sur laquelle s'applique cette permission.

Les modèles de sécurité de notre étude de cas enrichissent le modèle fonctionnel par la modélisation de quatre rôles, à savoir le conducteur (Driver), l'agent de circulation (TrafficAgent), la machine de gestion de sécurité à bord (OnboardSafetyManagement) et la machine de gestion de sécurité au sol (TracksideSafetyManagement). La Fig. 2 est un modèle de sécurité décrivant les permissions

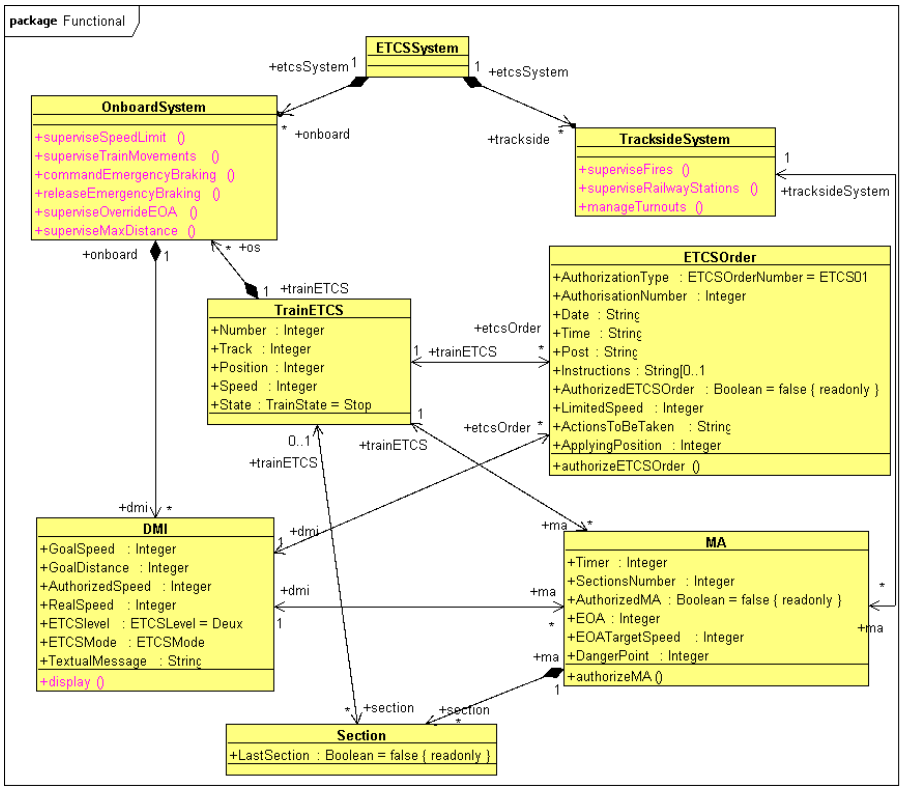


Figure 1. Le modèle fonctionnel

accordées aux rôles sur un ordre écrit. Le conducteur du train ne peut que lire un ordre écrit et suivre les indications appropriées sur le DMI alors que l'agent de circulation peut le créer, le modifier, l'autoriser et/ou le supprimer. De la même manière, nous décrivons pour chaque entité du modèle fonctionnel un modèle de sécurité contenant les permissions accordées aux rôles pouvant agir sur cette dernière.

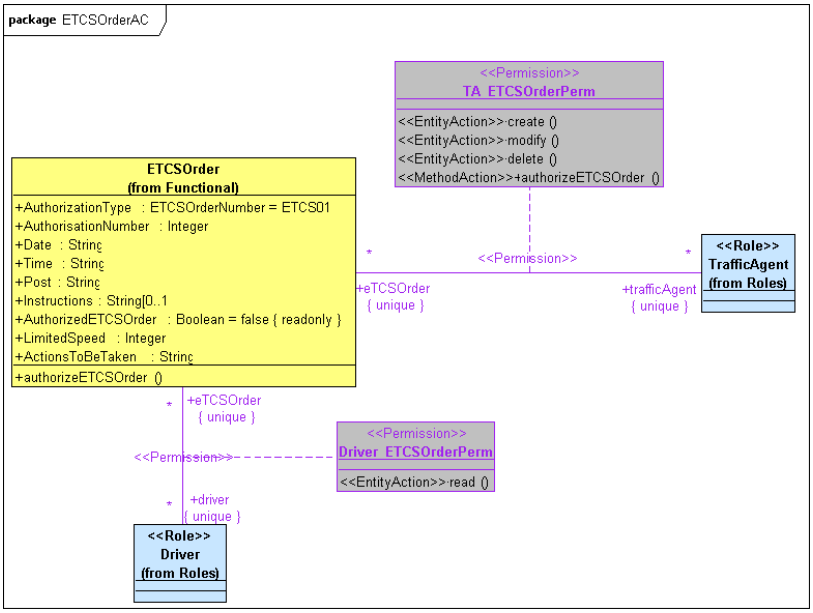


Figure 2. Permissions associées à la classe ETCSOrder

4 Transformation en spécifications B

La méthode B est une méthode de spécification formelle et d'analyse rigoureuse de la fonctionnalité et du comportement d'un système. Elle couvre toutes les phases du cycle de vie de développement logiciel allant de la spécification vers l'implémentation. Elle est basée sur des notations mathématiques fondées sur la logique du premier ordre et la théorie des ensembles où l'état du système est modélisé par des types abstraits de données prédéfinies. Elle permet la modélisation des aspects statiques et dynamiques du logiciel structurés dans des machines abstraites. L'aspect statique est caractérisé par les ensembles, les constantes, les propriétés, les variables et les invariants, tandis que l'aspect dynamique est décrit par l'initialisation et les opérations.

Dans le cadre du couplage des notations UML et B, la plate-forme dispose d'une base de règles de transformations de UML vers B. Dans cette section, nous illustrons la transformation du modèle fonctionnel ainsi que les modèles de sécurité en spécifications B.

La transformation automatique des modèles avec la plate-forme B4MSecure permet d'obtenir une unique machine B abstraite pour le modèle fonctionnel appelée « Functional » et une unique machine B pour tous les modèles de sécurité appelée « RBAC_Model ». La traduction du modèle fonctionnel est inspirée des approches de transformation existantes de UML vers B [7].

La spécification B décrit les ensembles (*SETS*), les variables abstraites (*ABSTRACT_VARIABLES*), les invariants (*INVARIANT*) et les opérations (*OPERATIONS*). La Fig. 3 donne quelques extraits du modèle fonctionnel. Les classes *MA* et *ETCSOrder* sont transformées respectivement en un ensemble *MA_AS* et un ensemble *ETCSORDER* (déclarés dans la clause SETS) qui représentent les instances possibles de ces classes et les variables abstraites *MA* et *ETCSOrder* (déclarées dans la clause *ABSTRACT_VARIABLES*) qui représentent les instances existantes de ces classes. Les invariants des lignes 15 et 16 montrent l'inclusion de *MA* dans *MA_AS* et de *ETCSOrder* dans *ETCSORDER*. Les attributs des classes (*MA__AuthorizedMA* et *ETCSOrder__AuthorizedETCSOrder*) et les associations (*MAOfTrainETCS* entre la classe *MA* et *TrainETCS* et *ETCSOrderOfTrainETCS* entre la classe *ETCSOrder* et *TrainETCS*) sont représentés sous forme de variables (lignes 9, 10, 11 et 12). Les invariants des lignes 17 et 18 sont relatifs aux attributs dans le but de spécifier leurs types (le type booléen « *BOOL* »). Les invariants des lignes 19 et 20 sont relatifs aux associations pour spécifier leurs multiplicités. Les méthodes des classes, ainsi que les constructeurs, les destructeurs, les getters et les setters sont transformés en opérations dans la partie dynamique du modèle B. Ces opérations respectent les invariants de types.

La spécification des opérations peut être enrichie par des pré-conditions et des substitutions, ainsi que la machine par des invariants. Cette tâche peut être préalablement effectuée dans le modèle UML afin de générer automatiquement des opérations bien spécifiées et des invariants de la machine autres que les invariants de types. En effet, la plate-forme B4MSecure permet d'ajouter des annotations en B sur le modèle UML pour spécifier des invariants et sur les méthodes des classes pour spécifier des pré-conditions et des substitutions d'opération. Les pré-conditions des lignes 31 et 36 et les substitutions d'opération des lignes 32 et 37 peuvent être ajoutées dans le modèle UML sous forme d'annotations respectivement sur les méthodes *authorizeMA* et *authorizeETCSOrder* des classes *MA* et *ETCSOrder*.

La machine associée au modèle de sécurité inclut via un opérateur *INCLUDES* la machine associée au modèle fonctionnel pour vérifier les propriétés de sécurité. Cette machine appelée « RBAC_Model » ajoute des variables relatives aux permissions. Par exemple, *PermissionAssignement* est une fonction totale de l'ensemble *PERMISSIONS* vers le produit cartésien *(ROLES * ENTITIES)*, *isPermitted* est une relation entre les deux ensembles *ROLES* et *Operations*. *PERMISSIONS*, *ENTITIES* et *Operations* sont des ensembles définis dans *RBAC_Model*,

```
1  Machine
2   Functional
3  SETS
4   MA_AS;
5   ETCSORDER; ...
6  ABSTRACT_VARIABLES
7   MA,
8   ETCSOrder,
9   MA__AuthorizedMA,
10  ETCSOrder__AuthorizedETCSOrder,
11  MAOfTrainETCS,
12  ETCSOrderOfTrainETCS,
13  ...
14 INVARIANT
15  MA <: MA_AS
16  & ETCSOrder <: ETCSORDER
17  & MA__AuthorizedMA : MA --> BOOL
18  & ETCSOrder__AuthorizedETCSOrder : ETCSOrder --> BOOL
19  & MAOfTrainETCS : MA --> TrainETCS
20  & ETCSOrderOfTrainETCS : ETCSOrder --> TrainETCS
21  & ...
22 INITIALISATION
23  MA := {}
24  || ETCSOrder := {}
25  ||  MA__AuthorizedMA := {}
26  ||  ETCSOrder__AuthorizedETCSOrder := {}
27  ||...
28 OPERATIONS
29  MA__authorizeMA(Instance)=
30   PRE Instance : MA
31     & MA__AuthorizedMA(Instance) = FALSE
32   THEN MA__AuthorizedMA(Instance) := TRUE
33  END;
34  ETCSOrder__authorizeETCSOrder(Instance)=
35   PRE Instance : ETCSOrder
36     & ETCSOrder__AuthorizedETCSOrder(Instance) = FALSE
37   THEN ETCSOrder__AuthorizedETCSOrder(Instance) := TRUE
38  END; ...
39 END
```

Figure 3. Machine « *Functional* »

alors que *ROLES* est un ensemble défini dans la machine incluse *UserAssignments*. Elle ajoute aussi des contraintes d'autorisation d'accès aux opérations du modèle fonctionnel. Elle associe une opération sécurisée correspondante à une opération fonctionnelle pour vérifier si l'utilisateur courant (le rôle) possède une permission d'effectuer cette opération. Afin de limiter l'accès à une opération fonctionnelle, un prédicat est ajouté dans l'opération sécurisée du modèle de sécurité qui vérifie si cette opération figure parmi les opérations permises par l'utilisateur courant *currentRole*. Ces prédicats apparaissent dans les lignes 23 et 29 de la Fig. 4 pour les opérations d'autorisation de *MA* et d'*ETCSOrder*.

```
1  Machine
2   RBAC_Model
3  INCLUDES
4   Functional, UserAssignments
5  SEES
6   ContextMachine
7  SETS
8   ENTITIES = {MA_Label, ETCSOrder_Label, ...};
9   Attributes = {MA_AuthorizedMA_Label,ETCSOrder_AuthorizedETCSOrder_Label...};
10  Operations ={MA_authorizeMA_Label, ETCSOrder_authorizeETCSOrder_Label...}...
11 VARIABLES
12  PermissionAssignement, isPermitted, ...
13 INVARIANT
14  PermissionAssignement: PERMISSIONS --> (ROLES * ENTITIES)
15  & isPermitted: ROLES <-> Operations ...
16 INITIAISATION
17  PermissionAssignement :=
18  {(OSM_MAPerm|->(OnboardSafetyManagement|->MA_Label)),
19   (TA_ETCSOrderPerm|->(TrafficAgent|->ETCSOrder_Label))...}
20 OPERATIONS
21  secure_MA__authorizeMA(Instance)=
22   PRE Instance: MA & MA__AuthorizedMA(Instance) = FALSE
23   THEN SELECT MA__authorizeMA_Label : isPermitted[currentRole]
24        THEN MA__authorizeMA(Instance)
25        END
26  END;
27  secure_ETCSOrder__authorizeETCSOrder(Instance)=
28   PRE Instance: ETCSOrder & ETCSOrder__AuthorizedETCSOrder(Instance) = FALSE
29   THEN SELECT ETCSOrder__authorizeETCSOrder_Label : isPermitted[currentRole]
30        THEN ETCSOrder__authorizeETCSOrder(Instance)
31        END
32  END; ...
33 END
```

Figure 4. Machine « *RBAC_ Model* »

5 Validation des modèles formels

La vérification et la validation de modèles basées sur la méthode B ont pour objectif de garantir que les modèles respectent bien les exigences de départ. Nous nous focalisons sur la validation formelle des spécifications B générées automatiquement par la plate-forme B4MSecure au moyen des outils ProB et Atelier B.

ProB supporte plusieurs techniques de validation telles que l'animation et le model checking. Nous l'avons utilisé en tant qu'animateur pour tester les deux scénarios de la section 2. Une telle animation simule l'évolution de l'état du système. Le premier état est calculé à partir de l'initialisation. Les états suivants dépendent des opérations dont la pré-condition est vérifiée.

La transformation des modèles UML en spécifications B génère des opérations B qui respectent les invariants de types (les types des attributs de classes, la multiplicité des associations entre les classes, etc.) dans le modèle fonctionnel et la politique de sécurité (les rôles, les permissions, etc.) dans le modèle de sécurité. L'animation avec ProB permet de valider, suivant les droits de chaque rôle, certaines opérations du modèle fonctionnel et de vérifier la séquence d'opérations qui construit les deux scénarios. Pour la validation de nos exigences de sécurité ferroviaire, il est judicieux d'ajouter des contraintes tant pour le modèle fonctionnel que pour le modèle de sécurité afin d'assurer la conformité des modèles B enrichis à ces exigences. L'animation de la séquence d'opérations avec ProB révèle le besoin d'ajouter des contraintes de sécurité ferroviaire à certaines opérations sous forme de pré-conditions ou bien à la spécification en sa globalité sous forme d'invariants. Ces contraintes peuvent être exprimées sous forme d'annotations en B dans les modèles UML afin qu'elles soient transformées automatiquement par la plate-forme B4MSecure dans les spécifications générées en B.

Comme mentionné dans la section précédente, le modèle de sécurité inclut le modèle fonctionnel et chaque opération sécurisée du modèle de sécurité est associée à une opération du modèle fonctionnel sur laquelle s'ajoutent des contraintes d'autorisation d'accès. De ce fait, l'opération du modèle de sécurité appelante doit contenir les mêmes pré-conditions de l'opération du modèle fonctionnel appelée.

La pré-condition ci-dessous a été ajoutée à l'opération « $DMI\_\_advance$ » du modèle fonctionnel (Fig. 5) dans lequel les classes et leurs attributs sont spécifiés. De même, elle a été ajoutée à l'opération sécurisée correspondante « $secure\_DMI\_\_advance$ » du modèle de sécurité (Fig. 6). Cette opération sécurisée doit être exécutée par le conducteur du train à travers le DMI. Selon les règles ferroviaires décrites par notre étude de cas, le conducteur ne fait avancer le train qu'après avoir reçu l'autorisation de franchir un EOA ou bien une MA. Cette pré-condition est exprimée comme suit : Le conducteur ne peut avancer le train à travers le DMI que s'il existe un ETCSOrder (une instance déjà créée de ETCSOrder) qui a été autorisé (son attribut booléen $AuthorizedETCSOrder$ est à $TRUE$) ou une MA (une instance déjà créée de MA) qui a été autorisée (son attribut booléen $AuthorizedMA$ est à $TRUE$).

$$\exists\, \text{etcsorder}.(\text{etcsorder} \in \text{ETCSOrder} \wedge \text{ETCSOrder\_\_AuthorizedETCSOrder}(\text{etcsorder}) = \text{TRUE}) \\ \vee\, \exists\, \text{ma}.(\text{ma} \in \text{MA} \wedge \text{MA\_\_AuthorizedMA}(\text{ma}) = \text{TRUE})$$

Une autre contrainte a été ajoutée à la spécification B générée du modèle fonctionnel sous forme d'invariant qui exprime la propriété de sécurité ferroviaire suivante : sur une section de voie, au maximum un train peut circuler. Alors,

```
DMI__advance(Instance)=
   PRE
      Instance : DMI &
      #etcsorder.(etcsorder:ETCSOrder & ETCSOrder__AuthorizedETCSOrder(etcsorder)=TRUE)
or #ma.(ma:MA & MA__AuthorizedMA(ma) = TRUE)
   THEN
      TrainETCS__State(OnboardSystem_in_TrainETCS(DMI_in_OnboardSystem(Instance))) := Advance
   END
;
```

Figure 5. L'opération « *DMI_ _ advance* » du modèle fonctionnel

```
secure_DMI__advance(Instance)=
   PRE
      Instance : DMI &
      #etcsorder.(etcsorder:ETCSOrder & ETCSOrder__AuthorizedETCSOrder(etcsorder)=TRUE)
or #ma.(ma:MA & MA__AuthorizedMA(ma) = TRUE)
   THEN  SELECT
       DMI__advance_Label : isPermitted[currentRole]
       THEN
          DMI__advance(Instance)
       END
   END
;
```

Figure 6. L'opération « *secure_ DMI_ _ advance* » du modèle de sécurité

quelque soient les trains t1 et t2 appartenant aux instances existantes de la classe *TrainETCS* et au codomaine de la fonction *TrainETCSSection* et tel que le train t1 est différent du train t2 implique les images de t1 et t2 par la fonction inverse de *TrainETCSSection* (correspondantes aux sections occupées par ces deux trains) sont différentes. *TrainETCSSection* est une fonction partielle de l'ensemble *Section* vers l'ensemble *TrainETCS* qui correspond à l'association entre la classe *Section* et la classe *TrainETCS*.

$$\forall\ (t1,t2).(t1 \in \text{TrainETCS} \wedge t2 \in \text{TrainETCS} \wedge t1 \in \text{ran}(\text{TrainETCSSection}) \wedge t2 \in \text{ran}(\text{TrainETCSSection}) \wedge t1 \neq t2 \Rightarrow \text{TrainETCSSection}^{-1}(t1) \neq \text{TrainETCSSection}^{-1}(t2))$$

Cette contrainte est exprimée sous forme d'invariant qui doit être respecté par les opérations de la machine fonctionnelle, ainsi que la machine de sécurité qui l'inclut.

Nous avons utilisé l'Atelier B [6] pour prouver les spécifications B générées automatiquement et puis ces spécifications après l'ajout des contraintes. Atelier B est un outil industriel développé par la société ClearSy qui permet une utilisation opérationnelle de la méthode formelle B pour des développements de logiciels prouvés sans défaut. Il dispose d'un prouveur automatique pour la démonstration de la plupart des obligations de preuves justes et d'un prouveur interactif pour la détection des erreurs et la finalisation de la preuve. Nous avons alors utilisé le prouveur automatique et un ensemble de commandes du prouveur interactif pour prouver les obligations de preuves restantes dues aux contraintes ajoutées. Les obligations de preuves générées du modèle fonctionnel sont toutes prouvées. Par contre, des obligations de preuves du modèle de sécurité ne sont

6. Atelier B : http ://www.atelierb.eu/

pas encore prouvées à cause d'une limitation du prouveur qui ne parvient pas à raisonner sur toutes les définitions décrites dans le modèle de sécurité. Le modèle de sécurité est considéré comme un filtre qui ne propose pas de nouvelles opérations et n'ajoute pas un comportement supplémentaire. Il permet donc de limiter l'accès à certaines opérations en fonction du rôle. Les principaux invariants à respecter sont ceux qui sont liés aux concepts de RBAC, ce qui est déjà réalisé par la transformation automatique à l'aide de l'outil B4MSecure.

6 Conclusion

Notre étude de cas est basée sur deux scénarios extraits des règles d'exploitation ferroviaires ERTMS/ETCS appliquées sur la ligne à grande vitesse LGV Est-Européenne. Ces scénarios contiennent des aspects fonctionnels liés à la fonctionnalité du système (modèle fonctionnel) et des aspects de sécurité liés au contrôle d'accès (modèles de sécurité). Notre objectif est de pouvoir les modéliser et les valider formellement. Basée sur l'approche IDM, la plate-forme B4MSecure nous a permis, d'une part, de modéliser ces règles en diagrammes de classe UML renforcés par un profil d'extension UML pour la politique de contrôle d'accès, et d'autre part, de les transformer en spécifications B afin de les valider formellement au moyen de l'animateur ProB et du prouveur Atelier B. Une amélioration sur l'architecture des spécifications générées en B pourrait être intégrée à la plate-forme B4MSecure. Elle s'agit de remplacer l'inclusion du modèle fonctionnel par le modèle de sécurité par un raffinement du modèle fonctionnel. Le modèle de sécurité réduit l'espace d'état du modèle fonctionnel grâce à des règles de contrôle d'accès. De ce fait, considérer le modèle de sécurité comme un raffinement du modèle fonctionnel permet de respecter les invariants du modèle fonctionnel et de préserver les propriétés sans augmenter la taille des invariants à prouver ainsi que celle des obligations de preuves.

UML permet la modélisation avec des différentes vues graphiques des aspects différents du système grâce à l'utilisation des profils. Dans notre étude de cas, les scénarios décrits mettent en œuvre des actions et des interactions entre les rôles et les entités du système. Nous visons dans des travaux futurs à explorer les diagrammes UML dynamiques tels que les diagrammes de séquence de telle sorte que nous pourrions valider formellement la séquence des opérations après sa transformation en spécifications B. [3] propose une transformation des diagrammes UML de séquence en des spécifications formelles CSP afin de valider les exigences décrites par ces diagrammes. Les travaux réalisés dans [4] proposent aussi une approche pour valider des spécifications UML avec des diagrammes de classe et des diagrammes de séquence en les transformant en spécifications B définissant une nouvelle machine B de simulation.

L'exploitation de l'approche utilisant B4Msecure pour une nouvelle application s'intéressant à l'analyse de la sécurité ferroviaire pour une portion de règles d'exploitation est plutôt concluante. En ce qui concerne les aspects dynamiques du modèle, l'approche existante avoue quelques limites et des extensions sont à envisager.

Remerciements Ces travaux de recherche ont été soutenus par le projet Perfect (ANR-12-VPTT-0010) et en partie par le projet Selkis (ANR-08-SEGI-018) et l'ARC6 de la Région Rhône-Alpes.

Références

1. ALCATEL, ALSTOM, ANSALDO SIGNAL, BOMBARDIER, INVENSYS RAIL, SIEMENS : ETCS-Baseline 2, System Requirements Specification - Subset026. 7 chapters (2006)
2. RFF : Principes et règles d'exploitation du système ETCS - Particularités en cas de superposition à un autre système de signalisation. PROJET 0T Révision (2012) (to be published)
3. Rasch, H., Wehrheim, H. : Checking the Validity of Scenarios in UML Models. Proceedings of the 7th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems, pp. 67-82. Springer, Heidelberg (2005)
4. Truong, N.T. and Souquières, J. : Test of object-based specifications using B notations. Hal-INRIA, hal-00015031. LORIA (2005)
5. Ledru, Y., Idani, A., Milhau, J., Qamar, N., Laleau, R., Richier, J.L., Labiadh, M.A. : Taking into Account Functional Models in the Validation of IS Security Policies. Advanced Information Systems Engineering Workshops. LNCS, vol. 83, pp. 592-606. Springer, Heidelberg (2011)
6. Milhau, J., Idani, A., Laleau, R., Labiadh, M.A., Ledru, Y., Frappier, M. : Combining UML, ASTD and B for the formal specification of an access control filter. Innovations in Systems and Software Engineering, vol. 7, pp. 303-313. Springer (2011)
7. Idani, A., Labiadh, M.A., Ledru, Y. : Infrastructure dirigée par les modèles pour une intégration adaptable et évolutive de UML et B. Ingénierie des Systèmes d'Information Journal, vol. 15, pp. 87-112 (2010)
8. Lodderstedt, T., Basin, D.A., Doser, J. : SecureUML : A UML-Based Modeling Language for Model-Driven Security. Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 426-441. Springer, Heidelberg (2002)
9. Idani, A., Ledru, Y., Labiadh, M.A. : B4MSecure : une plateforme IDM pour la modélisation et la validation de politiques de sécurité en Systèmes d'Information. Journées Francophones sur les Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL), pp. 85-89 (2013)
10. Sandhu, R., Ferraiolo, D., Kuhn, R. : The NIST Model for Role-based Access Control : Towards a Unified Standard. Proceedings of the Fifth ACM Workshop on Role-based Access Control, pp. 47-63 (2000)
11. Ben Ayed, R., Collart-Dutilleul, S., Bon, P., Idani, A., Ledru, Y. : B Formal Validation of ERTMS/ETCS Railway Operating Rules. Y. Ait Ameur and K.-D. Schewe (Eds.) : ABZ 2014. LNCS, vol. 8477, pp. 124-129. Springer, Heidelberg (2014)

Une proposition pour l'ajout de dimensions dans la programmation de logiciels embarqués

Frédéric Boniol

ONERA-Toulouse, France
frederic.boniol@onera.fr

Abstract. Le but de cet article est d'étudier l'enrichissement du langage de programmation Lustre [HCRP91] par un système de calcul de dimension. Par dimension, on entend des notions comme les *mètres*, les *degrés*, les *secondes* ou des dimensions composées comme les *mètres par seconde*. On montre que cet enrichissement est assez minimal, et repose sur un système d'inférence également très simple[1].

1 Introduction

La production des logiciels embarqués semble aujourd'hui bien maîtrisée. Ces logiciels surveillent et commandent des dispositifs physiques par le biais de capteurs et d'actionneurs. Un bon exemple est le système de commande de vol d'un avion. L'avion, les capteurs (d'altitude, de vitesse, etc.), le logiciel de commande, et les actionneurs (les gouvernes de vol) forment une boucle fermée fonctionnant en permanence. La maîtrise des technologies logicielles a permis la réalisation de tels systèmes dans les avions commerciaux civils depuis 30 ans, et depuis plus longtemps encore pour les avions militaires. Ces logiciels sont développés aujourd'hui à l'aide de modèles et outils considérés comme formels SCADE [Dor08] reposant lui-même sur le langage flot de donnée synchrone Lustre [HCRP91].

Si ces technologies sont considérées comme des succès, elles souffrent cependant d'une lacune étonnante. Bien que dédiées à la programmation de systèmes cyber-physiques, elles ne permettent pas la spécification et la vérification des dimensions physiques des données manipulées, telles que le *mètre* pour des longueurs, le *mètre par seconde* pour des vitesses, ou encore le *radian par seconde carré* pour des accélérations angulaires. Si on comprend que ces dimensions physiques ne présentent aucun intérêt pour la génération du code, elles sont néanmoins importantes pour comprendre ce que calcule le logiciel et pour en vérifier la cohérence. Par exemple, même si le calcul d'une altitude en *mètres* est correct, il sera considéré comme faux s'il est utilisé par un composant qui l'attend en *pieds*. De même, bien qu'elles soient toutes deux de type réel, il n'est pas correct d'additionner une distance et une vitesse. De telles confusions d'unités dans les programmes ont parfois eu des conséquences graves comme la perte de la sonde "Mars Climate Orbiter"[2].

[1] Ce travail a été supporté par le projet P financé par le programme FUI 2011.

[2] voir http://fr.wikipedia.org/wiki/Mars_Climate_Orbiter

Plusieurs travaux ont étudié la notion de dimension dans des langages de programmation tels que Ada [Geh85] ou C [JS06,GM05]. [Nov95] a montré que les dimensions reposent sur une structure algébrique donnant aux opérations sur les unités des propriétés d'associativité et de commutativité. A partir de cette formalisation, il propose une classification des dimensions en 8 catégories (*length, time*, etc.), chaque catégorie étant raffinée en unités (*meter, foot*, etc. pour *length*) convertibles les unes dans les autres via une procédure de conversion. En parallèle, [Gou94] a proposé une extension du système de type de Standard ML permettant un typage des quantités numériques par l'incorporation des dimensions physiques. [Ken96] a proposé une formalisation mathématique de l'algèbre des dimensions utilisée par [Gou94] et en a étudié les aspects théoriques. C'est sur ces travaux que nous nous appuyons en les simplifiant et les restreignant au langage Lustre. Tous ces travaux présupposent soit un corpus de dimensions standard, soit un corpus de dimensions défini par le programmeur au moyen de primitives spécifiques. Plus récemment [GM05] a étudié le dimensionnement d'un programme C standard sans corpus de dimensions. L'idée consiste à introduire des variables de dimension et, en exploitant les instructions du programme, inférer son schéma dimensionnel par un ensemble de contraintes entre les dimensions d'entrée et les dimensions de sortie.

Malgré ces avancées les langages utilisés industriellement aujourd'hui n'implantent toujours pas cette notion de dimension. Une limite possible des approches précédentes, pointée par [BM08], réside peut-être dans leur trop grande généralité. Les langages de programmation visés sont souvent généralistes et de bas niveau tels que C. Partant de ce constant, [BM08] propose d'étudier un langage spécifique pour la programmation de robot supportant l'analyse de dimension. La portée de cette analyse est plus large que dans les travaux précédents au sens où les dimension sont utilisées à l'exécution pour vérifier la compatibilité des unités lorsque le robot (mobile) se connecte à de nouveaux capteurs. La nature dynamique du programme et du système est ici centrale.

La programmation des systèmes de contrôle commande repose également sur des langages de haut niveau et spécifiques tels que Lustre. Nous suivons donc dans cet article une approche similaire à [BM08] en proposant d'intégrer les dimensions dans Lustre et non pas dans C. En revanche, les systèmes visés étant statiques (pas de mobilité et de changement de connexion en cours d'exécution), nous explorons une approche de dimensionnement par typage, similaire à celle suivie dans utilisée par [Gou94], mais à la compilation uniquement et non pas à l'exécution.

2 Un exemple introductif

2.1 Première version

Pour illustrer le problème, considérons le système de vol figure 1. Ce système est composé de trois sous-systèmes : (1) ADIRS (Air Data and Inertial Reference System) qui à partir de capteurs situés dans l'avion produit un ensemble de mesures représentant l'état de l'avion (altitude, vitesse, etc.) ; (2) FMS (Flight

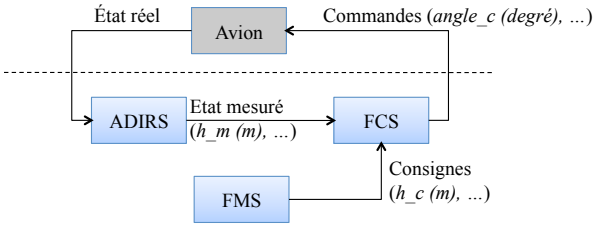


Fig. 1. Exemple de système cyber-physique

Management System) qui gère le plan de vol et qui envoie des consignes de vol au (3) FCS (Flight Control System) qui à partir de ces consignes et de l'état mesuré de l'avion produit des commandes pour les gouvernes de vol. Ces trois systèmes forment une boucle fermée avec l'avion. A ce niveau de description, les données produites et consommées sont décrites dans des documents de conception. Elles sont caractérisées par une dimension, le *mètre* par exemple pour l'altitude h_m mesurée par l'ADIRS et pour la consigne d'altitude h_c produite par le FMS, ou le *degré* pour la consigne de braquage de gouverne $angle\_c$ calculée par le FCS.

Au niveau suivant, chaque sous-système est raffiné, puis programmé dans un formalisme adéquat. A titre d'exemple, le système FCS (simplifié) est raffiné en un programme Lustre figure 2. Après la déclaration d'un ensemble de constantes (lignes 1 à 7), ce programme prend en entrée l'altitude mesurée (`h_m`) et l'altitude de consigne (`h_c`), toutes deux de type `real`, et produit en sortie un angle de gouverne commandé (`angle_c`) également de type `real` (ligne 9). Le corps du programme est décrit par trois équations encadrées par les mots clefs `let` et `tel` (ligne 14 à 23). La première équation (ligne 15) calcule l'erreur d'altitude `erreur_h` comme la différence entre la consigne et la mesure. La seconde équation (lignes 16-18) définit `angle_c`. Si `erreur_h` est supérieure à la constante `erreur_h_switch`, c'est-à-dire à 50 *mètres*, la consigne de braquage de gouverne `angle_c` est égale à `angle_max`, c'est-à-dire 5,2 *degrés* ; si `erreur_h` est inférieure à -50 *mètres*, `angle_c` est positionnée à -5,2 *degrés*; et enfin lorsque `erreur_h` est dans l'intervalle $[-50m; 50m]$, le braquage de la gouverne est calculé via une loi de type PID dont le coefficient proportionnel est `kp`, le coefficient intégral est `ki` et le coefficient dérivé est nul (équation de `pid` ligne 19). Le terme proportionnel du PID est `erreur_h * kp`, tandis que le terme intégral est `ki * integral` où `integral` est défini par l'équation lignes 20-22. Le pas d'intégration est `delta_t` = 0,005 *seconde* ; l'intégration est réalisée sur les `duree_integr` = 10000 derniers pas, soit 50 *secondes*, et est initialisée à `init_integr` = 0. Ce calcul intégral utilise l'opérateur `fby`(x, n, i) qui retourne i lors des n premiers cycles puis lors des cycles suivants la valeur que x avait n cycles avant.

Le programme figure 2 est suffisant pour générer un code embarqué. Notons cependant que la description que nous venons d'en faire est plus riche que le programme lui-même. Ce dernier ne contient aucune information de dimen-

```
1  const erreur_h_switch = 50.0;
2  const angle_max = 5.2;
3  const kp = 0.1014048;
4  const ki = 0.0048288;
5  const delta_t = 0.005;
6  const duree_integr = 10000;
7  const init_integr = 0.;
8
9  node FCS (h_m : real; h_c : real) returns (angle_c : real)
10 var
11         erreur_h : real;
12         pid : real;
13         integral : real;
14 let
15         erreur_h = h_c - h_m;
16         angle_c = if (erreur_h > erreur_h_switch) then angle_max
17                 else if (erreur_h < -erreur_h_switch) then -angle_max
18                 else pid ;
19         pid = (erreur_h * kp) + (ki * integral) ;
20         integral = init_integr -> (pre(integral)
21                                    + (erreur_h * delta_t)
22                                    - fby(erreur_h, durre_integr, 0.)*delta_t);
23 tel
```

Fig. 2. Programme LUSTRE du FCS

sion (les *mètres*, *degrés*, *secondes*). Certes, ces informations sont inutiles pour la génération de code. En revanche, elles sont utiles pour vérifier la correction du programme. Reprenons ce dernier. Si à partir des documents de conception du niveau supérieur, on peut inférer que `h_m` et `h_c` sont en *mètres*, et donc inférer que `erreur_h` est également en *mètres*, il n'est pas possible de calculer l'unité de `angle_c`, et donc vérifier que le programme FCS produit une sortie conforme à la spécification de niveau supérieur. L'inférence de la dimension de `angle_c` nécessite la dimension des constantes du programme, dimensions qui ne sont pas spécifiées. De même l'unité de temps n'est pas spécifiée. Or les valeurs des coefficients `kp` et `ki` du PID dépendent des choix d'unités. Ces valeurs seraient très différentes si les unités étaient *pied*, *radian* et *milliseconde*. En l'absence d'information d'unité, il est donc impossible de vérifier que la sortie est bien en *degré*. Il serait également impossible de détecter des incohérences telles que l'utilisation d'une valeur de `kp` en *radian par mètres* avec une valeur de `angle_max` en *degré*. Ce genre d'erreur est généralement détecté en simulation, mais il pourrait l'être en amont par une analyse statique du programme. Une telle analyse, appelée *calcul de dimensions*, suppose l'introduction de cette notion dans le programme. Notons pour achever de convaincre le lecteur que l'analyse dimensionnelle fait partie de la boite à outils de base de l'ingénieur. Toute personne qui manipule des équations y a naturellement recours. L'introduction de cette analyse dans la vérification des programmes parait donc naturelle.

2.2 Proposition informelle d'une extension dimensionnée

Reprenons le FCS et imaginons en une version *dimensionnée*. Cette version est donnée en figure 3. Les ajouts par rapport à la version non dimensionnée sont sur-lignés en rouge. La première idée consiste à ajouter en tête du programme

```
1  dim = m, deg, s;
2  const erreur_h_switch = 50.0 dim(m);
3  const angle_max = 5.2 dim(deg);
4  const kp = 0.1014048 dim(deg*m-1);
5  const ki = 0.0048288 dim(deg*m-1*s-1);
6  const delta_t = 0.005 dim(s);
7  const duree_integr = 10000;
8  const init_integr = 0. dim(m*s);
9
10 node FCS (h_m : real dim(m); h_c : real dim(m)) returns (angle_c : real)
11 var
12         erreur_h : real;
13         pid : real;
14         integral : real;
15 let
16         erreur_h = h_c - h_m;
17         angle_c = if (erreur_h > erreur_h_switch) then angle_max
18                 else if (erreur_h < -erreur_h_switch) then -angle_max
19                 else pid ;
20         pid = (erreur_h * kp) + (ki * integral) ;
21         integral = init_integr -> (pre(integral)
22                                    + (erreur_h * delta_t)
23                                    - fby(erreur_h, durre_integr, 0. dim(m))*delta_t);
24 tel
```

Fig. 3. Programme LUSTRE du FCS étendu avec des unités

la déclaration des unités de base utilisées (ligne 1). Ces unités sont `m` pour *mètre*, `deg` pour *degré*, et `s` pour *seconde*. Le choix est de laisser le programmeur définir ses propres unités plutôt que de lui imposer un corpus standard. Chaque constante est ensuite déclarée avec son unité (lignes 2 à 6 et 8). Par exemple, `erreur_h_switch` est de dimension `m`, et vaut donc 50 `m` (ligne 2). Les constantes du PID sont définies complètement avec leur dimension (lignes 4 et 5). De même chaque valeur littérale est annotée avec sa dimension (ligne 23), cette dimension pouvant être une dimension physique comme c'est le cas ligne 23 où la dimension spécifique "sans dimension" comme nous le verrons plus loin. Enfin, les entrées du programme sont également déclarées avec leur dimension. `h_m` et `h_c` sont reçues en `m` (ligne 10).

Cette extension syntaxique amène plusieurs remarques.

1. Si seules trois unités de base ont été introduites en ligne 1, des dimensions composées ont été construites et utilisées à partir de ces unités. Pour ce faire, on a introduit deux opérations sur les dimensions : la multiplication ($*$) et l'inversion (-1). La déclaration de dimension ligne 5 par exemple spécifie que `ki` est de dimension `deg*m-1*s-1` à savoir *degré* par *mètre* et par *seconde*. On s'attend alors à ce que l'expression `erreur_h * kp` ligne 20 soit de dimension `m*(deg*m-1)`, c'est-à-dire `deg`. Cette première remarque amène à penser que la notion de dimension repose sur une structure algébrique sous-jacente dans laquelle, par associativité et commutativité de l'opération $*$ puis par simplification entre `m` et `m-1`, on puisse dériver que les dimensions `m*(deg*m-1)` et `deg` sont équivalentes.

2. En appliquant le raisonnement précédent, on semble pouvoir inférer les dimensions de l'ensemble des expressions du programme. Par exemple, l'équation ligne 16 dit que `erreur_h` est de dimension `m`. De même, par une inférence un peu plus longue, on peut obtenir que `pid` est de dimension `deg`, et au final que `angle_c` est également en `deg`. Si l'inférence arrive à donner une dimension à tous les flots du programme, alors le programme est bien dimensionné. Dans le cas contraire, Le programme est mal dimensionné. De ce point de vue, le calcul de dimension est similaire à un calcul de type ou au calcul d'horloge des langages synchrones.
3. Notons que pour inférer la dimension de `erreur_h` et de `pid`, on a utilisé le polymorphisme du point de vue des dimensions des opérations arithmétiques. Les équations ligne 16, lignes 20, et lignes 21-23 font toutes intervenir une opération arithmétique (soustraction ou addition). Or elles portent sur des dimensions différentes. Elles doivent donc être polymorphes du point de vue des dimensions. La seule contrainte imposée par l'addition et la soustraction est de porter sur des flots de même dimension, quelle que soit celle-ci. A l'inverse, la multiplication et la division n'impliquent aucune contrainte. On peut par exemple diviser une température par des radians. En revanche, on ne peut pas additionner ces deux grandeurs. Un peu plus formellement, cela pourrait être exprimé par le typage (sur les dimensions) :

$$\forall d_1, d_2 : \mathit{dimensions} \begin{cases} + : (d_1, d_1) \to d_1 \\ - : (d_1, d_1) \to d_1 \\ * : (d_1, d_2) \to d_1 * d_2 \\ / : (d_1, d_2) \to d_1 * d_2^{-1} \end{cases}$$

4. Remarquons que la constante `duree_integr` ligne 7 n'est pas dimensionnée. Implicitement, elle est sans dimension. Cela amène à penser que l'ensemble des unités de base du programme, déclaré en ligne 1, est implicitement augmenté de l'unité "*sans dimension*". Les constantes et les flots d'entrée non dimensionnés par le programmeur sont implicitement *sans dimension*.
5. Enfin, notons que seules les contantes et les flots d'entrée du programme ont été dimensionnés explicitement. Le flot de sortie `angle_c` (ligne 10) et les trois flots internes `erreur_h`, `pid` et `integral` (lignes 12-14) ne sont pas dimensionnés. Pour autant, ils ne sont pas *sans dimension*. Avant le calcul de dimension, leur dimension est simplement inconnue. Comme dans un système d'inférence de types, il sera nécessaire d'introduire pour ces quatre flots des variables de dimension, qui seront ensuite calculées en suivant un principe similaire au principe d'unification de l'inférence de types.

3 Formalisation

La formalisation recherchée doit permettre de :

– définir la nature mathématique de l'ensemble des dimensions,

– à partir de cette définition, trouver un moyen opératoire, si possible simple, pour tester l'équivalence de deux expressions de dimensions (par exemple `m*deg*m-1` et `deg`), puis une méthode pour réduire une expression de dimension à sa forme la plus simple (par exemple `m*deg*m-1` réduite à `deg`);
– et enfin, proposer un ensemble de règles d'inférence de dimensions applicables à un programme Lustre dimensionné.

Pour un programme donné P, on notera dans la suite $U_P = \{u_1, \ldots, u_N\}$ l'ensemble des N unités de base déclarées en tête du programme. Chaque u_i est une chaîne de caractères, telle que `deg`. Nous supposerons que U_P est ordonné par l'ordre lexicographique. Par exemple, dans le cas du programme figure 3, $U_{FCS} = \{deg, m, s\}$.

On notera également $(pr_i)_{i\in\mathbb{N}^*}$ la suite des nombres premiers ($pr_1 = 2$,..., $pr_i = i^{ième}$ nombre premier).

Enfin on notera $'\alpha_1$, $'\alpha_2$... des variables de dimension (par similarité aux variables de type $'a$ en OCaml), et α_1, α_2... (sans "quote") des variables à valeurs dans les rationnels $\mathbb{Q}$.

3.1 Groupe des dimensions

L'ensemble des dimensions possibles engendré par U_P, noté $(\mathbb{D}_{U_P}, *)$ forme un groupe infini défini par la grammaire suivante :

$$\begin{array}{rll} d ::= & \mathbb{1} & \\ | & u^n & u \in U_P, n \in \mathbb{Z} \\ | & ('\alpha)^n & '\alpha \text{ variable de dimension}, n \in \mathbb{Z} \\ | & d * d & \end{array}$$

où $\mathbb{1}$ est l'unité "*sans dimension*".

Comme l'a montré [Ken96], ce groupe possède de bonnes propriétés mathématiques définies par les axiomes ci-dessous et qui engendrent la relation d'équivalence "$\equiv$" entre dimensions :

– Commutativité et associativité de * :

$$\forall d_1, d_2 \in \mathbb{D}_{U_P}, \qquad d_1 * d_2 \equiv d_2 * d_1$$

$$\forall d_1, d_2, d_3 \in \mathbb{D}_{U_P}, \qquad d_1 * (d_2 * d_3) \equiv (d_1 * d_2) * d_3$$

– $\mathbb{1}$ est l'élément neutre de * :

$$\forall d \in \mathbb{D}_{U_P}, \qquad d * \mathbb{1} \equiv \mathbb{1} * d \equiv d$$

– additivité des exposants :

$$\forall u \in U_P, \forall '\alpha, \forall a, b \in \mathbb{Z}, \qquad u^a * u^b \equiv u^{a+b} \text{ et } ('\alpha)^a * ('\alpha)^b \equiv ('\alpha)^{a+b}$$

– nullité de l'exposant :

$$\forall u \in U_P, \forall '\alpha, \qquad u^0 \equiv ('\alpha)^0 \equiv \mathbb{1}$$

Considérons par exemple la dimension $d = m * (deg * m^{-1})$. En appliquant successivement les axiomes ci-dessus, on montre que

$$\begin{aligned} d &= m * (deg * m^{-1}) \\ &\equiv m * (m^{-1} * deg) && \text{(commutativité)} \\ &\equiv (m * m^{-1}) * deg && \text{(associativité)} \\ &\equiv (m^0) * deg && \text{(additivité de l'exposant)} \\ &\equiv \mathbb{1} * deg && \text{(nullité de l'exposant)} \\ &\equiv deg && \text{(élément neutre)} \end{aligned}$$

Formellement, $(\mathbb{D}_{U_P}, *)$ forme un groupe abélien libre de type fini dont le rang est le cardinal de U_P.

3.2 Test d'équivalence

La question est alors de décider, au moyen d'une méthode calculatoire, si deux dimensions sont équivalentes au sens de la relation $\equiv$ définie ci-dessus. Une façon de décider de l'équivalence de deux expressions est d'identifier une transformation qui projette toutes expressions équivalentes en une forme unique. Le groupe des dimensions possédant une opération de multiplication et une opération inverse, il est naturel de le projeter dans l'espace des rationnels $\mathbb{Q}$ en suivant un encodage de type "encodage de Gödel"[3]. Intuitivement, l'idée consiste à

- projeter chaque unité de base de U_P vers un nombre premier (un nombre unique pour chaque unité),
- projeter chaque variable de dimension vers une variable dans $\mathbb{Q}$,
- associer l'opération exposant sur les unités ou les variables de dimension, à l'opération exposant dans $\mathbb{Q}$,
- associer l'opération de multiplication $d_1 * d_2$ sur les dimensions à l'opération de multiplication $d_1 \cdot d_2$ sur $\mathbb{Q}$,
- et projeter l'unité *sans dimension* sur la valeur 1.

Ce faisant, on transforme une expression de dimension en une expression sur $\mathbb{Q}$.

Definition 1. *Soit la fonction RF (pour Rational Form) définie par induction sur* $(\mathbb{D}_{U_P}, *)$ *:*

$$\begin{aligned} RF(\mathbb{1}) &\stackrel{\text{def}}{=} 1 \\ RF(u_i^n) &\stackrel{\text{def}}{=} pr_i^n \quad (i^{ième} \text{ nombre premier exposant } n) \\ RF(('\alpha)^n) &\stackrel{\text{def}}{=} \alpha^n \\ RF(d_1 * d_2) &\stackrel{\text{def}}{=} RF(d_1) \cdot RF(d_2) \end{aligned}$$

RF associe à toute dimension d une expression rationnelle unique de la forme

$$RF(d) = \frac{p}{q} \Pi_i \alpha_i^{n_i}$$

où $p, q \in \mathbb{N}^*$*, et où chaque* α_i *est une variable dans* $\mathbb{Q}$ *correspondant à une variable de dimension* $'\alpha_i$ *apparaissant dans d avec l'exposant* n_i*.*

[3] `http://en.wikipedia.org/wiki/Godel_numbering`

A titre d'exemple, en prenant $U = \{deg, m, s\}$, alors $RF(deg) = 2$, $RF(m) = 3$, $RF(s) = 5$, et par exemple

$$RF(deg * {}'\alpha * m^{-1} * s^{-1}) = \frac{2}{15}\alpha$$

Proposition 1. *RF préserve l'équivalence de dimension :*

$$\forall d_1, d_2 \in (\mathbb{D}_{U_P}, *), d_1 \equiv d_2 \Rightarrow RF(d_1) = RF(d_2)$$

Proposition 2. *Inversement, deux expressions de dimension ayant même forme rationnelle sont équivalentes :*

$$\forall d_1, d_2 \in (\mathbb{D}_{U_P}, *), RF(d_1) = RF(d_2) \Rightarrow d_1 \equiv d_2$$

C'est cette deuxième proposition qui nous donne un test facile pour décider de l'équivalence de deux expressions de dimension.

3.3 Forme normale

Pour toute dimension d de $(\mathbb{D}_{U_P}, *)$, $RF(d)$ peut être décomposée en facteurs premiers sous la forme

$$RF(d) = pr_1^{a_1} \cdot \ldots \cdot pr_N^{a_N} \cdot \Pi_i \alpha_i^{b_i}$$

où les $a_i, b_i \in \mathbb{Z}$ et où chaque α_i est une variable dans $\mathbb{Q}$ correspondant à une variable de dimension ${}'\alpha_i$ apparaissant dans d. Chaque a_i représente l'exposant de u_i dans d (rappelons que N est le nombre d'unités de base du programme P), et chaque b_i représente l'exposant de la variable de dimension ${}'\alpha_i$ dans d.

A titre d'exemple, en prenant $U = \{deg, m, s\}$, $RF(m * {}'\alpha^2 * s^{-1} * s^{-1}) = pr_1^0 \cdot pr_2^1 \cdot pr_3^{-2} \cdot \alpha^2$.

Definition 2. *Soit $d \in (\mathbb{D}_{U_P}, *)$, notons $pr_1^{a_1} \cdot \ldots \cdot pr_N^{a_N} \cdot \Pi_i \alpha_i^{b_i} = RF(d)$, l'ensemble des variables α peut être éventuellement vide. Soit $i_1 \ldots i_l$ les indices compris entre 1 et N tel que les a_{i_j} sont non nuls et tels que les autres a_k $k \neq i_j$ $\forall j$ sont nuls. On définit la forme normale de d par*

$$NF(d) \stackrel{\text{def}}{=} \begin{cases} \mathbb{1} & \text{si tous les } a_i \text{ sont nuls et si } RF(d) \text{ ne contient aucune variable} \\ \Pi_i {}'\alpha_i^{b_i} & \text{si tous les } a_i \text{ sont nuls et si } RF(d) \text{ contient des variables} \\ u_{i_1}^{a_{i_1}} * \ldots * u_{i_l}^{a_{i_l}} & \text{si au moins un } a_i \text{ est non nul et si } RF(d) \text{ ne contient aucune variable} \\ u_{i_1}^{a_{i_1}} * \ldots * u_{i_l}^{a_{i_l}} * \Pi_i {}'\alpha_i^{b_i} & \text{si au moins un } a_i \text{ est non nul et si } RF(d) \text{ contient des variables} \end{cases}$$

Par exemple $NF(s * m * {}'\alpha_1 * s^{-1} * s^{-1} * {}'\alpha_1^{-1}) = m * s^{-1}$.

Les unités et les variables apparaissent de façon unique dans $NF(d)$ et sont rangées par ordre lexicographique pour les unités puis par ordre des indices pour les variables. Il s'ensuit que :

Proposition 3. $\forall d \in (\mathbb{D}_{U_P}, *)$, $NF(d)$ *est unique.*

$NF(d)$ est la représentation équivalente la plus courte de d.

Proposition 4. $\forall d_1, d_2 \in (\mathbb{D}_{U_P}, *), d_1 \equiv d_2 \Leftrightarrow NF(d_1) = NF(d_2)$.

3.4 Un calcul de dimensions

Reste alors à expliciter les règles du calcul de dimensions. Nous suivons une formalisation similaire à un système d'inférence de type à la ML. Nous notons "$\Gamma, C \vdash dim(exp) = d$" pour exprimer que dans l'environnement de dimensionnement Γ, si les contraintes dans C sont satisfaites, alors l'expression exp est de dimension d.

Pour simplifier la présentation, nous supposerons que le calcul de dimension intervient après une première passe syntaxique qui

1. vérifie que les unités de base utilisées par le programmeur ont été déclarées,
2. remplace toutes valeurs littérales (par exemple 0. ligne 23) par une constante de même dimension déclarée en tête du programme,
3. ajoute la dimension $\mathbb{1}$ aux constantes et flots d'entrée non dimensionnés par le programmeur,
4. range les unités de base par ordre lexicographique,
5. ajoute à chaque flot interne ou flot de sortie `flow_name` une variable de dimension `'flow_name`, le but du système de type étant d'inférer automatiquement la valeur de ces variables,
6. réécrit toutes les dimensions en leur forme normale,
7. et réécrit les opérations $\texttt{fby}(x, n, i)$ en l'expression équivalente

$$\underbrace{i \texttt{ -> pre}(i \texttt{ -> pre}(\ldots \texttt{ -> pre}}_{n \text{ fois}}(x))\ldots))$$

qui pendant les n premiers cycles retourne la valeur i puis à chaque cycle t retourne la valeur de x à $t - n$.

Le système d'inférence de dimension est composé de 12 règles :

– **Dimensionnement des constantes et des flots.** L'environnement de dimensionnement associe à chaque constante et chaque flot la dimension qui lui est syntaxiquement associée dans le programme :

$$\frac{\texttt{const } cst = lit \texttt{ dim}(d);}{\Gamma, C \vdash dim(cst) = d} \text{ [Const]}$$

$$\frac{\begin{array}{c}\texttt{node}(x_1 : t_1 \texttt{ dim}(d_1); \ldots; x_n : t_n \texttt{ dim}(d_n)) \\ \texttt{returns}(x'_1 : t'_1 \texttt{ dim}(d'_1); \ldots; x'_m : t'_m \texttt{ dim}(d'_m))\end{array}}{\begin{array}{c}\Gamma, C \vdash dim(x_1) = d_1, \ldots, \Gamma, C \vdash dim(x_n) = d_n \\ \Gamma, C \vdash dim(x'_1) = d'_1, \ldots, \Gamma, C \vdash dim(x'_m) = d'_m\end{array}} \text{ [Node]}$$

$$\frac{\texttt{var } x_1 : t_1 \texttt{ dim}(d_1); \ldots; x_n : t_n \texttt{ dim}(d_n);}{\Gamma, C \vdash dim(x_1) = d_1, \ldots, \Gamma, C \vdash dim(x_n) = d_n} \text{ [Var]}$$

– **Dimensionnement des opérations arithmétiques addition et soustraction.** Une addition ou une soustraction sont dimensionnées si les deux opérandes ont la même dimension. Cette nouvelle contrainte est ajoutée dans l'ensemble des contraintes de l'environnement de dimensionnement. Le résultat hérite alors de la dimension des opérandes.

$$\frac{\Gamma, C \vdash dim(exp_1) = d_1 \quad \Gamma, C \vdash dim(exp_2) = d_2}{\Gamma, C \cup \{RF(d_1) = RF(d_2)\} \vdash dim(exp_1 + exp_2) = d_1} \text{[Add]}$$

$$\frac{\Gamma, C \vdash dim(exp_1) = d_1 \quad \Gamma, C \vdash dim(exp_2) = d_2}{\Gamma, C \cup \{RF(d_1) = RF(d_2)\} \vdash dim(exp_1 - exp_2) = d_1} \text{[Soust]}$$

– **Dimensionnement des opérations arithmétiques multiplication et division.** A l'inverse de l'addition et de la soustraction, la multiplication et la division n'imposent aucune contrainte.

$$\frac{\Gamma, C \vdash dim(exp_1) = d_1 \quad \Gamma, C \vdash dim(exp_2) = d_2}{\Gamma, C \vdash dim(exp_1 * exp_2) = NF(d_1 * d_2)} \text{[Multi]}$$

$$\frac{\Gamma, C \vdash dim(exp_1) = d_1 \quad \Gamma, C \vdash dim(exp_2) = d_2}{\Gamma, C \vdash dim(exp_1 / exp_2) = NF(d_1 * d_2^{-1})} \text{[Div]}$$

Notons que pour simplifier les expressions de dimension stockées dans l'environnement Γ, la dimension des résultats de ces opérations est normalisée.

– **Dimensionnement des opérations de comparaison.** Deux flots ne peuvent être comparés que si ils ont même dimension. Dans ce cas, l'expression booléenne exprimant cette comparaison est sans dimension.

$$\frac{\Gamma, C \vdash dim(exp_1) = d_1 \quad \Gamma, C \vdash dim(exp_2) = d_2}{\Gamma, C \cup \{RF(d_1) = RF(d_2)\} \vdash dim(exp_1 \ comp \ exp_2) = \mathbb{1}} \text{[Comp]}$$

avec $comp \in \{<, >, =, \leq, \geq\}$.

– **Dimensionnement des expressions `if-then-else`.** Si la condition est sans dimension, et si les deux branches ont la même dimension, alors l'expression `if-then-else` hérite de cette dimension.

$$\frac{\Gamma, C \vdash dim(exp_1) = d_1 \quad \Gamma, C \vdash dim(exp_2) = d_2}{\Gamma, C \cup \{RF(d_1) = RF(d_2), RF(b) = 1\} \vdash dim(\texttt{ite}(b, exp_1, exp_2)) = d_1} \text{[ITE]}$$

(où `ite` est l'abréviation de `if-then-else`).

– **Dimensionnement des expressions temporelles.** L'opération d'initialisation (`->`) impose que ses deux opérandes (la valeur initiale et l'expression pour les instants suivants) aient même dimension.

$$\frac{\Gamma, C \vdash dim(exp_1) = d_1 \quad \Gamma, C \vdash dim(exp_2) = d_2}{\Gamma, C \cup \{RF(d_1) = RF(d_2)\} \vdash dim(exp_1 \ \texttt{->} \ exp_2) = d_1} \text{[Init]}$$

L'opérateur `pre` ne change pas la dimension du flot.

$$\frac{\Gamma, C \vdash dim(exp_1) = d_1}{\Gamma, C \vdash dim(pre(exp_1)) = d_1} \text{ [Pre]}$$

– **Dimensionnement des équations.** Enfin, une équation propage la dimension de sa partie droite vers sa partie gauche.

$$\frac{x \texttt{ = } exp \in Eqs \quad \Gamma, C \vdash dim(exp) = d}{\Gamma, C \cup \{'x = RF(d)\} \vdash dim(x) = d} \text{ [Equation]}$$

Notons qu'aucune règle ne contient de prémisse négative. Si deux règles sont applicables à un instant donné de l'analyse, alors l'application de l'une n'empêchera pas l'application de l'autre. Le système de règle est donc confluent. Le résultat de l'inférence ne dépend pas de l'ordre d'application des règles.

Si à l'issue de l'application des règles sur un programme donné, les contraintes de C admettent une solution unique dans l'environnement Γ, alors le programme est bien dimensionné, et la dimension de l'ensemble des flots peut être calculée. Dans le cas inverse, le programme est mal dimensionné.

Notons enfin deux points intéressants:

– les contraintes générées par les règles d'inférences ne sont que des contraintes d'égalité;
– ensuite, un programme Lustre sémantiquement correct étant déterministe, si les dimensions de toutes les entrées et de toutes les constantes sont données, alors sit le système de contraintes généré par les règles d'inférence admet une unique solution, soit il n'admet aucune solution; dans ce cas le programme bien que sémantiquement correct est mal dimensionné.

Ces deux points laissent à penser que l'inférence des dimensions d'un programme Lustre est une opération plus simple que l'inférence de type à la ML, et même que l'inférence de dimension d'un programme impératif à la Java ou C. Cette question, à l'état de conjecture, ne sera pas approfondie dans cet article.

3.5 Application à l'exemple

Reprenons l'exemple du FCS dans lequel les 6 étapes préliminaires décrites au début de la section précédente ont été réalisées (à l'exception du remplacement de l'occurrence de `fby` pour des raisons de lisibilité). Le nouveau programme est décrit figure 4. Quatre variables de dimensions ont été introduites (lignes 11, 13, 14 et 15). On procède alors de façon itérative :

– Soient l'environnement vide Γ_0 (qui ne contient aucun dimensionnement) et l'ensemble de contraintes vide $C_0 = \emptyset$. A partir de cet environnement, seules les règles Const, Node et Var sont applicables, et donnent $C_1 = C_0$ et Γ_1 l'environnement qui associe à chaque constante et chaque flot sa dimension telle que explicitée dans le programme.

```
1  dim = deg, m, s;
2  const erreur_h_switch = 50.0 dim(m);
3  const angle_max = 5.2 dim(deg);
4  const kp = 0.1014048 dim(deg*m-1);
5  const ki = 0.0048288 dim(deg*m-1*s-1);
6  const delta_t = 0.005 dim(s);
7  const duree_integr = 10000 dim(I);
8  const init_integr = 0. dim(m*s);
9  const L0 = 0. dim(m);
10
11 node FCS (h_m : real dim(m); h_c : real dim(m)) returns (angle_c : real dim('angle_c))
12 var
13         erreur_h : real dim('erreur_h);
14         pid : real dim('pid);
15         integral : real dim('integral);
16 let
17         erreur_h = h_c - h_m;
18         angle_c = if (erreur_h > erreur_h_switch) then angle_max
19                 else if (erreur_h < -erreur_h_switch) then -angle_max
20                 else pid ;
21         pid = (erreur_h * kp) + (ki * integral) ;
22         integral = init_integr -> (pre(integral)
23                                    + (erreur_h * delta_t)
24                                    - fby(erreur_h, durre_integr, L0)*delta_t);
25 tel
```

Fig. 4. Programme LUSTRE du FCS étendu avec des unités et normalisé

- Plusieurs règles peuvent alors être appliquées. Citons à titre d'exemple la règle Soust qui déduit de la ligne 17 que $dim(\texttt{h\_c - h\_m}) = \texttt{m}$ et qui produit la contrainte triviale $\{3 = 3\}$ (rappelons que $RF(deg) = 2$, $RF(m) = 3$, et $RF(s) = 5$). Plus intéressant, la règle Comp appliquée aux expressions booléenne lignes 18 et 19 déduit que ces deux expressions sont de dimension $\mathbb{1}$ avec la contrainte $erreur\_h = 3$ (signifiant que `erreur_h` doit être en `m`). Citons enfin la règle Equation appliquée ligne 17 qui génère également la contrainte $erreur\_h = 3$.
- Au final, l'ensemble de contraintes produit est

$$C_{final} = \{\ \tfrac{2}{3}\,erreur\_h = \tfrac{2}{15}\,integral;\ erreur\_h = 3;\ integral = 15;\\ pid = \tfrac{2}{3}\,erreur\_h;\ integral = 5\,erreur\_h;\ pid = 2;\ angle\_c = 2;\ \}$$

  Cet ensemble de contrainte admet une unique solution, d'où on déduit :

$$\texttt{'erreur\_h} = \texttt{m}, \texttt{'integral} = \texttt{m*s}, \texttt{'pid} = \texttt{deg}, \texttt{'angle\_c} = \texttt{deg}$$

  et donc que la sortie du programme est bien en *degrés*.

4 Discussion

L'objectif de cet article était de proposer l'introduction de dimensions dans des langages de programmation de systèmes embarqués tels que LUSTRE. Ces langages, de part leur nature déclarative et flots de données, se prêtent naturellement à un typage statique de flots par des expressions de dimension. Les informations de dimension sont des annotations formelles utilisées par un analyseur

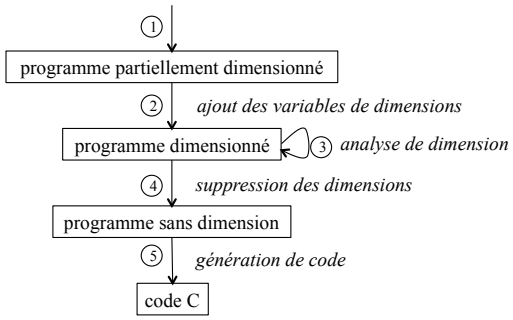


Fig. 5. Processus de conception intégrant l'étape d'analyse de dimension

de dimension, similaire à un calcul de type ou un calcul d'horloge, qui permet de décider si un programme est bien dimensionné. Un processus de développement intégrant l'analyse de dimension est donné figure 5 :

1. Le processus commence par l'écriture par le programmeur d'un programme partiellement dimensionné (par exemple le programme figure 3), c'est-à-dire explicitant les unités utilisées et les dimensions des constantes et des flots d'entrée (étape 1).
2. Ce programme est complété par l'ajout des dimensions implicites : dimension $\mathbb{1}$ ou variables de dimension (par exemple le programme figure 4) (étape 2).
3. L'analyse de dimension est ensuite effectuée (étape 3).
4. Puis, les annotations de dimensions sont retirées pour obtenir une version standard (sans dimension, par exemple le programme figure 2) du programme (étape 4).
5. Version qui est ensuite compilée et transformée en code C (étape 5).

Ce processus amène deux remarques. Notons d'une part que l'analyse de dimension est effectuée à la compilation, avant la génération de code, et n'impacte pas celle-ci. L'intérêt est de ne pas nécessiter de modification du générateur de code. En revanche, cela interdit l'exploitation des informations de dimension à l'exécution, par exemple pour tester les dimensions des entrées fournies au programme lors d'une exécution en mode interactif. Notons ensuite que l'extension proposée du langage Lustre a la bonne propriété d'être conservatrice, au sens où si aucune dimension n'est renseignée lors de l'étape 1, l'analyseur de dimension inférera que tous les flots du programme sont sans dimension. En ce sens, il est encore possible au programmeur de ne renseigner aucune dimension et continuer à "faire comme avant".

Plusieurs questions restent ouvertes et feront l'objet des travaux suivants :

– *La complexité de l'inférence de dimension.* Comme dit rapidement et sans justification à la fin de la section 3.4, la nature flot de données à assignation unique et la propriété de déterminisme du langage Lustre devrait simplifier

en pratique, voire réduire la complexité théorique, du calcul de dimension, par rapport aux approches plus générales proposées par [Ken96] ou plus récemment par [KL13]. Cette affirmation devra être démontrée par une étude ultérieure sur la complexité théorique du calcul et sa mise en pratique.
- *Les expressions non dimensionnables.* Comme le remarquait [Ken96], certaines expressions ne peuvent pas être dimensionnées par l'approche proposée. C'est le cas de $\sqrt{x}$. Cela nécessiterait de traiter des exposants rationnels (moyennant l'abandon du plongement des dimensions dans $\mathbb{Q}$ et son remplacement par un calcul formel sur les unités).
- *Le cas des nœuds importés ou génériques.* Imaginons un système FCS commandant plusieurs gouvernes via plusieurs lois de type PID. Il serait naturel de coder une seule loi PID générique et de l'instancier autant de fois que nécessaire. Mais chaque instance contrôlant un paramètre différent de l'avion (altitude, vitesse, angle, etc.), la loi générique ne peut intégrer des dimensions connues à l'avance. A l'image de l'addition, elle doit pouvoir être instanciée sur des dimensions différentes, et donc dimensionnée de façon polymorphe.
- *Conversion de dimension.* Comme l'a proposé [Nov95], une troisième perspective serait la prise en compte des multiples ou diviseurs d'une dimension, par exemple m et cm, ou de conversion entre unités, par exemple m et $foot$.

Remerciements. L'auteur tient à remercier vivement Rémi Delmas pour les discussions fructueuses qui ont initié ce travail, et les relecteurs pour leurs remarques très judicieuses.

References

[BM08] G. Biggs and B. MacDonald. A pragmatic approach to dimensional analysis for mobile robotic programming. *Autonomous Robots*, 25(4):405–419, 2008.

[Dor08] F.X. Dormoy. SCADE 6 a model based solution for safety critical software development. In *Embedded Real-Time Systems Conference*, 2008.

[Geh85] N. H. Gehani. Ada's derived types and units of measure. *Software: Practice and Experience*, 15(6):555–569, 1985.

[GM05] Philip Guo and Stephen Mccamant. Annotation-less unit type inference for c. In *Final Project, 6.883: Program Analysis, CSAIL, MIT*, 2005.

[Gou94] J. Goubault. Inférence d'unités physiques en ML. In *JFLA'94*, 1994.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[JS06] L. Jiang and Z. Su. Osprey: a practical type system for validating dimensional unit correctness of c programs. In *Proceedings of the 28th international conference on Software engineering*, pages 262–271. ACM, 2006.

[Ken96] A.J. Kennedy. *Programming Languages and Dimensions.* PhD thesis, University of Cambridge, April 1996.

[KL13] Sebastian Krings and Michael Leuschel. Inferring physical units in B models. In *Proceedings of SEFM'2013*, LNCS 8137. Springer, 2013.

[Nov95] G.S Novak. Conversion of units of measurement. *Software Engineering, IEEE Transactions on*, 21(8):651–661, 1995.

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.

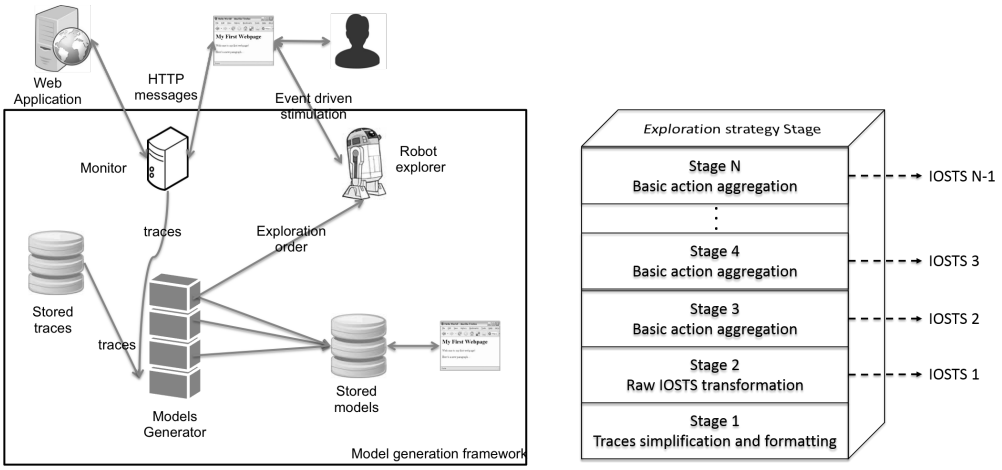
Dans la section suivante, nous décrivons de façon générale le fonctionnement de notre plateforme rassemblant le Robot générateur de traces et le Générateur de modèles. Puis, par manque de place, nous détaillons uniquement ce dernier mais de façon concrète, en ciblant le contexte des applications Web. 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

$States := \emptyset$;
foreach *trace* $\sigma = (a_0, \theta_0)...(a_n, \theta_n) \in ST$ **do**
  r:= null;
  for $0 \leq i \leq n$ **do**
    if $i==0$ **then**
      $r := r.(s0, v_\emptyset)$
    extraire la valuation $URI = val$ de θ_i;
    if $((val, 0), v_\emptyset) \notin States$ **then**
      $s := ((val, 0), v_\emptyset)$;
    else
      $s := ((val, k+1), v_\emptyset)$ avec $k \geq 0$ le plus grand entier tel que $((val, k), v_\emptyset) \in States$;
    $States := States \cup \{s\}$;
    $r := r.(a_i, \theta_i).s$
  $SR := SR \cup \{r\}$

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 :

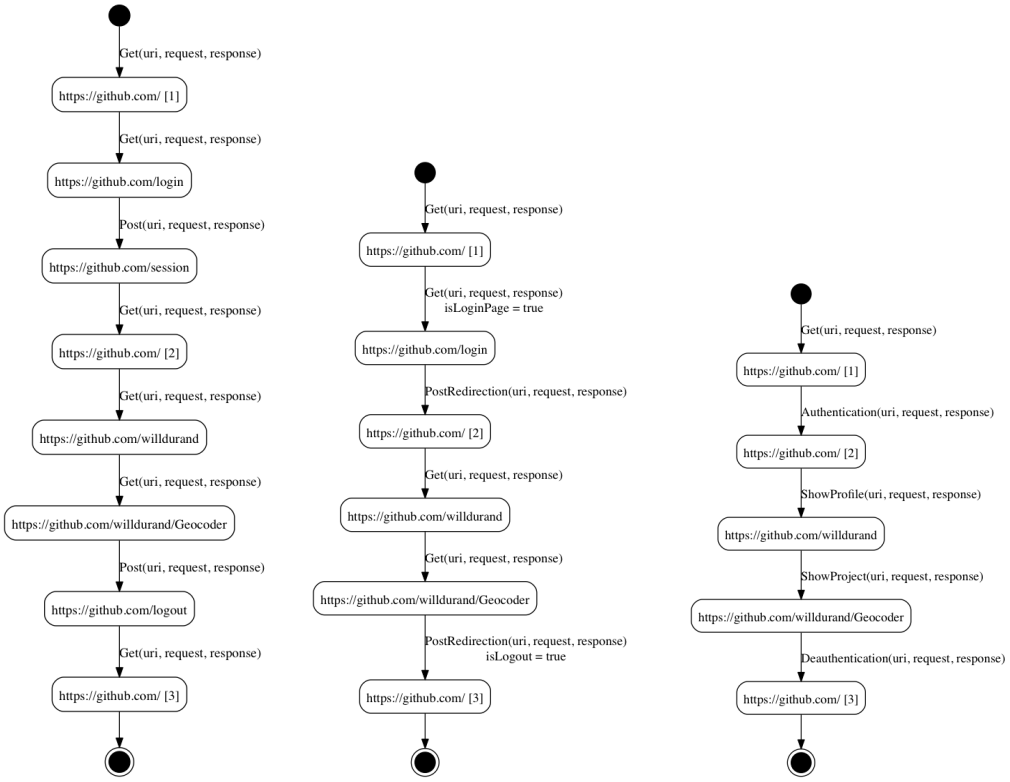
```
1 GET(https://github.com/)
  GET(https://github.com/login)
3 POST(https://github.com/session)
  GET(https://github.com/)
5 GET(https://github.com/willdurand)
  GET(https://github.com/willdurand/Geocoder)
7 POST(https://github.com/logout)
  GET(https://github.com/)
```

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.

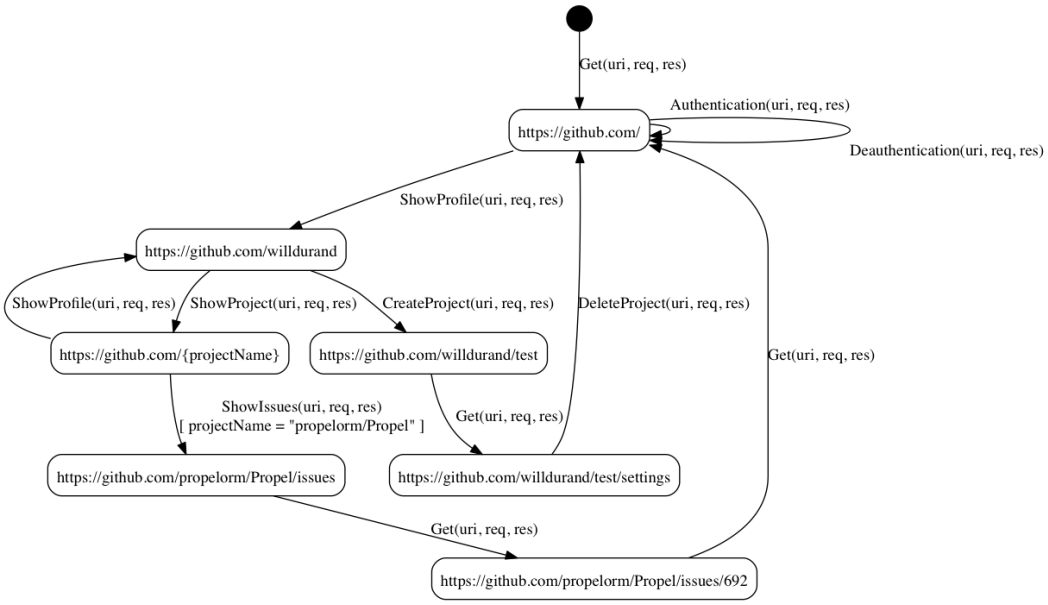


Figure 11: IOSTS $App(\mathcal{S}_4)$ obtenu depuis l'application Web Github

[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.

Adapting LTL model checking for inferring biological parameters

Emmanuelle Gallet[1], Matthieu Manceny[2],
Pascale Le Gall[1], and Paolo Ballarini[1]

[1] Laboratoire MAS, Ecole Centrale Paris, 92195 Châtenay-Malabry, France
email: {emmanuelle.gallet, pascale.legall, paolo.ballarini}@ecp.fr
[2] Laboratoire LISITE, ISEP, 28 Rue Notre-Dame-des-Champs 75006 Paris, France
email: matthieu.manceny@isep.fr

Abstract. The identification of parameters of Genetic Regulatory Networks (GRN) poses a problem of combinatorial explosion, even for networks of small size. In this paper, we propose a computer-aided methodology for reverse engineering of parameters of discrete models of GRN. Provided that the biological knowledge on gene expression levels can be expressed as LTL formulas, we adapt classical LTL model checking algorithms for dealing with parameters by using symbolic execution and constraint solving techniques to search for accepting cycles. As a result, we obtain constraints that parameters have to verify to make their associated models consistent with biological knowledge.

Keywords: LTL Model Checking, Symbolic Execution, Constraint Solving Techniques, Parameter Identification, Genetic Regulatory Network, Thomas Discrete Modelling.

1 Introduction

Formal modelling of genetic regulatory networks. We are interested in the modelling of *Genetic Regulatory Networks* (GRN) that coordinate the biological interactions at genetic level. Our goal is to understand how gene expression can be regulated through the presence or absence of regulatory proteins, taking into account the interdependence nature of regulations. To understand the time evolutions of gene expression of a GRN, also known as the *dynamics*, various continuous and discrete modelling approaches have been advocated for supporting analysis techniques. However, most of them suffer from the need of determining biological parameters which play a major role for describing the possible dynamics and are difficult to estimate. Indeed, not all of the dynamics included within a GRN are consistent with biological knowledge or observations. This knowledge can be used to determine the value of some parameters or can be translated in the form of constraints that parameters should comply with. By abstracting continuous dynamics using a discrete-step asynchronous dynamics, the R. Thomas discrete modelling of GRN [20,18,19], has the double advantage of highlighting qualitative reasoning and enabling the application of

formal methods, especially model checking approaches [1]. After having formally specified a biological observation in form of a temporal logic property, it becomes possible to verify if a target dynamics satisfies the given property. However, the problem of parameter identification requires to investigate the entire set of possible dynamics, that is to consider each possible combination of parameter values. Unfortunately, the number of such dynamics rapidly grows with the size of the GRN and the key question becomes the design of effective techniques for analysing a family of models parameterised by unknown parameters.

Related work. The use of model checking techniques applied to verify whether a given discrete Thomas model fulfils some relevant biological temporal properties has already been widely advocated in several works. Especially, Bernot *et al.*[3] is a pioneering work in which biological knowledge is expressed using *Computation Tree Logic* (CTL) formulae [1]. In order to exhaustively search the parameters' space, the set of all possible models (defined by all possible parameters' values) is generated and a CTL model checking procedure is iterated, one model after one model. This approach is implemented in the SMBioNet tool [16] and has been illustrated in [8]. In [12], the approach has been extended to cope with the formation of complexes from proteins which allows modellers to express relationships between biological parameters leading to a reduced set of models to be investigated. This work prefigures the interest of using constraints on the parameters. [14,2] define an approach based on an encoding technique enabling the sharing of computations between different models. Sets of models are encoded by a binary vector, one bit (or colour) per model, and model checking algorithms for *Linear Temporal Logic* (LTL) [1] are extended with Boolean operations on vectors. Algorithms are optimised for the particular case of *time series*, that are sequences of states, made of one expression level per gene, that are observed one by one, possibly with some intermediate states. These time series can be expressed by means of LTL formulas with nested *Finally* (**F**) operators. In [6,5], the tool GNBox deals with the problem of parameter identification using Constraint Logic Programming (CLP) techniques. Once GRN dynamics and biological knowledge are described by means of declarative rules and constraints on parameters, target behaviours are expressed as some kind of finite paths that models have to verify. [9] uses also CLP techniques, for adapting CTL model checking algorithms, but the encoding introduces a lot of fresh logical variables that hamper to scale up the method.

Our contribution. Similarly to [14], our approach is based on LTL model-checking: we check the emptiness of the product between a modelling of the GRN and a Büchi automata built from formulae expressing the biological knowledge over the set of gene expression levels. The novelty is that our modelling of GRN is a parametric model, called a *Parametric GRN*, that allows us to encompass all models of a GRN in a unique representation, biological parameters being processed as symbols. Thus, dynamics are not enumerated but implicitly referenced as solutions of constraints defined over parameters. Indeed, we follow the

same creed as the one advocated in [6,5]: model sets are handled through some logical language both to avoid combinatorial explosion and to take benefit of some constraint solving techniques. A preliminary version of our approach has been described in [15] using AGATHA, a tool dedicated to the symbolic analysis of models for reactive systems. In the present version, algorithms combining symbolic execution and constraint solving techniques have been reengineered and tuned to cope with GRN features and to search for parametric accepting cycles. Thereby, we consider the full LTL language while [14] essentially focuses on time series and [6,5] focuses on properties carrying on finite paths.

Paper organisation. In Section 2, we reformulate the logical description of Thomas' modelling framework (as detailed in [2,3,6]) by encoding the set of dynamics of a GRN by means of a Parametric GRN. Section 3 presents our adaptation of LTL model checking algorithms and details the use of symbolic execution and constraint solving techniques to search for parametric accepting cycles. In the end, parameters that satisfy the constraints characterizing accepting cycles define dynamics that verify the considered LTL formula. We also briefly discuss the validity of our approach with our dedicated tool, called SPuTNIk. Finally, Section 4 contains some concluding remarks.

2 Parametric modelling of GRN

2.1 Interaction Graph

A GRN is a collection of regulatory inter-dependencies between genes. The dynamics of gene expression (the biological process by means of which proteins are synthesised) depends on the presence of activator/inhibitor proteins: if their concentration is sufficient, they may activate or inhibit the transcription of a gene, hence regulating the synthesis of the end proteins. GRN are classically represented by *interaction graphs*.

Definition 1 (Interaction graph). *An* interaction graph *is a labelled directed graph* $G = (V, E, S, T)$ *where* V *is a finite set whose elements are called* genes*,* $E \subseteq V \times V$ *is the set of* interactions*,* $S : E \to \{+, -\}$ *associates to each interaction its* role *("+" for* activation *and "−" for* inhibition*), and* $T : E \to \mathbb{N}^+$ *associates to each interaction its* threshold*.*

The threshold k of an interaction $i \to j$ indicates the minimal level of expression that i needs to be for influencing the expression of j. Note that all values lower than k must be used on at least another outgoing interaction from i.

Example 1 (Interaction graph). Figure 1 presents an interaction graph where gene α activates the expression of gene β when its level of expression is greater or equal to 1, and activates both α and β when its level of expression is equal to 2. β inhibits the expression of α when its level is equal to 1.

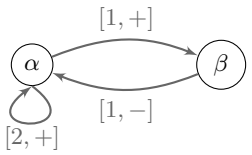


Fig. 1. An example of interaction graph.

2.2 Dynamics and Biological Parameters

We generically denote by x_i the level of expression of a gene i: to some extent, x_i abstracts the concentration level of the protein synthesised by i. The knowledge of x_i for all genes i defines a *dynamic state* or simply a *state*. We call *dynamics* of a GRN the evolution over time of the levels of expression of all genes. The evolution of x_i depends on the levels of expression of the genes regulating i, denoted $V^-(i) = \{j | j \in V, (j,i) \in E\}$. More precisely, it depends only on genes j among $V^-(i)$ whose x_j values are above the threshold $T(j,i)$, i.e. on genes in $\{j \in V^-(i) \mid x_j \geq T(j,i)\}$. Let $\omega \in 2^{V^-(i)}$ be a subset of regulators of i, then there exists a constant $K_i(\omega)$ in $\mathbb{N}$ which represents the target level towards which i will tend to from a given state $x = (x_1, \ldots, x_i, \ldots)$ verifying that $\forall j \in \omega, x_j \geq T(j,i)$ and $\forall j' \in V^-(i) \backslash \omega, x_{j'} < T(j',i)$. Thus, if $x_i < K_i(\omega)$ then x_i can increase, if $x_i > K_i(\omega)$ then it can decrease, and if $x_i = K_i(\omega)$, then x_i remains stable. Thus, a dynamics of a GRN can be described by a set of parameters $K_i(\omega)$:

Definition 2 (Biological parameters). *Let $G = (V, E, S, T)$ be an interaction graph. $\{K_i(\omega) \mid i \in V, \omega \subseteq V^-(i)\}$ is the set of* biological parameters *associated to G.*

Example 2 (Biological parameters). The biological parameters for the interaction graph of Figure 1 are: $K_\alpha(\{\})$, $K_\alpha(\{\alpha\})$, $K_\alpha(\{\beta\})$, $K_\alpha(\{\alpha, \beta\})$, $K_\beta(\{\})$, $K_\beta(\{\alpha\})$. In the dynamic state $(x_\alpha, x_\beta) = (1, 0)$, the set of effective predecessors of α is empty, thus x_α tends to evolve towards $K_\alpha(\{\})$, while, β having one effective predecessor, namely α, x_β tends to evolve towards $K_\beta(\{\alpha\})$.

2.3 Parametric GRN

Biological parameters depend on biological considerations and remain mainly unknown in practice. Each instantiation of these parameters defines a possible dynamics. In order to check all the dynamics simultaneously, we represent them all with one single (meta)model, called *Parametric GRN*. The Parametric GRN contains two families of symbols, the family of biological parameters $K_i(\omega)$ that represent unknown biological constants, and the family of state variables x_i representing the expression level of genes i in V, that can evolve, that is, decrease or increase of one unit at a time. For each gene i, a Parametric GRN contains a transition for each kind of variation (increase or decrease) of x_i. For $\omega \in 2^{V^-(i)}$, let

$P_i(\omega)$ denote the formula $(\bigwedge_{j\in\omega} x_j \geq T(j,i)) \wedge (\bigwedge_{j\in V^-(i)\setminus\omega} x_j < T(j,i))$ characterizing the set of dynamic states $(x_1,\ldots,x_i,\ldots)$, in which ω corresponds to the set of effective regulators of the gene i. The transition associated to the increase of x_i is conditioned by the guard $Increase(i) = \vee_{\omega\subset V^-(i)} Increase(i,\omega)$, where $Increase(i,\omega)$ is $(P_i(\omega) \wedge x_i < K_i(\omega))$. Similarly, the transition associated to the decrease of x_i is conditioned by the guard $Decrease(i) = \vee_{\omega\subset V^-(i)} Decrease(i,\omega)$, where $Decrease(i,\omega)$ is $(P_i(\omega) \wedge x_i > K_i(\omega))$. Finally, the expression level of gene i remains stable if the condition $Stable(i) = \vee_{\omega\subset V^-(i)} (P_i(\omega) \wedge x_i = K_i(\omega))$ is satisfied. We can now define Parametric GRN based on these expressions $Increase(i)$, $Decrease(i)$ and $Stable(i)$:

Definition 3 (Parametric GRN). *A Parametric GRN associated to a GRN $G = (V,E,S,T)$ is a couple (St,Tr) with $St = (I,F)$ a pair of states (I, the initial state and F, the stable state), and Tr a set of transitions. A transition of Tr is of the form (s,g,a,s') with s and s' states of St, g a guard and a an assignment. More precisely, Tr is the set of all instances of following transitions*[3]*:*

- $(I, Increase(i), x_i\uparrow, I)$ *with i in V,*
- $(I, Decrease(i), x_i\downarrow, I)$ *with i in V,*
- $(I, \wedge_{i\in V} Stable(i), id, F)$ *where id is the identity assignment,*
- $(F, \top, id, F)$ *where $\top$ indicates the guard always true.*

Example 3. Figure 2 sketches out the form of the Parametric GRN associated to the interaction graph of Figure 1: there are 4 (twice the number of genes) transitions from I to I, one transition from I to F, and one transition from F to F. In relation with the different possible subsets ω, one can explicit the different guards: for instance, for the two possibilities $\omega = \{\}$ and $\omega = \{\alpha\}$ for β, $Increase(\beta)$ is the formula $(x_\alpha < 1 \wedge x_\beta < K_\beta(\{\})) \vee (x_\alpha \geq 1 \wedge x_\beta < K_\beta(\{\alpha\})))$.

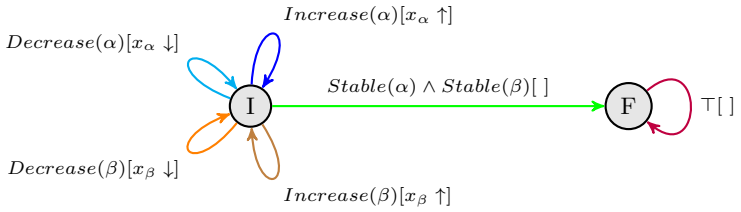


Fig. 2. The Parametric GRN associated to the interaction graph from Figure 1.

Example 4 (State transition graph). For the interaction graph of Figure 1, if the state $(x_\alpha, x_\beta) = (1,0)$ is such that the conditions $Increase(\alpha)$ and $Increase(\beta)$ are satisfied, then the model can evolve towards either $(2,0)$ or $(1,1)$. Figure 3

[3] $x_i\uparrow$ (resp. $x_i\downarrow$) denotes the assignment $x_i \mapsto x_i + 1$ (resp. $x_i \mapsto x_i - 1$).

presents the dynamics corresponding to the parameter instantiation, denoted pi, $K_\alpha(\{\}) = 2$, $K_\alpha(\{\alpha\}) = 2$, $K_\alpha(\{\beta\}) = 0$, $K_\alpha(\{\alpha,\beta\}) = 1$, $K_\beta(\{\}) = 0$ and $K_\beta(\{\alpha\}) = 1$ for the interaction graph from Figure 1 in the form of a *state transition graph*. In this oriented graph, there is a transition $(x_\alpha, x_\beta) \to (x'_\alpha, x'_\beta)$ if there exists a transition (s, g, a, s') in the Parametric GRN such that (x_α, x_β) satisfies pi and $(x'_\alpha, x'_\beta) = a(x_\alpha, x_\beta)$.

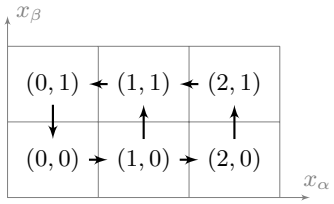


Fig. 3. A state transition graph for the interaction graph from Figure 1.

2.4 Static constraints over parameters

The Parametric GRN we obtain from a given GRN represents a template model which needs to be enriched with biologically relevant information. Such biological knowledge is expressed in terms of the following four constraints (on the model parameters) which reduce the parameter space by ruling out biological impossibilities:

Constraint 1 (Domain). $\forall i \in V, \forall \omega \subset V^-(i)$*:* $0 \leq K_i(\omega) \leq \max_{(i,j)\in E} T(i,j)$.

The *domain constraint* is obvious and defines the bounds of the finite domain of each gene.

Constraint 2 (Definition). $\forall i \in V, \forall j \in V^-(i), \forall \omega \subset V^-(i) \setminus \{j\}$*: if* $S(j,i) = +$ *then* $K_i(\omega) \leq K_i(\omega \cup \{j\})$*, if* $S(j,i) = -$ *then* $K_i(\omega) \geq K_i(\omega \cup \{j\})$.

The *Definition constraint* (or Snoussi constraint [17]) states that if the level of expression of a gene j which activates (resp. inhibits) a gene i becomes greater than the threshold of the corresponding interaction, then the expression level of i cannot decrease (resp. increase).

Constraint 3 (Observation). $\forall i \in V$, $\forall j \in V^-(i)$*, it exists* $\omega \subset V^-(i) \setminus \{j\}$*:* *if* $S(j,i) = +$ *then* $K_i(\omega) < K_i(\omega \cup \{j\})$*, if* $S(j,i) = -$ *then* $K_i(\omega) > K_i(\omega \cup \{j\})$.

The *Observation constraint* expresses how we identify that a predecessor is an activator or an inhibitor. If j is an activator (resp. inhibitor) of i, then it exists at least one dynamic state where the increase of the level of expression of j leads to an increase (resp. decrease) of the expression level of i.

Constraint 4 (Min/Max). $\forall i \in V$, $K_i(\{j | j \in V^-(i), S(j,i) = -\}) = 0$ *and* $K_i(\{j | j \in V^-(i), S(j,i) = +\}) = \max_{(i,j)\in E} T(i,j)$.

Finally, the *Min/Max constraint* states that in a dynamic state where all the activators (resp. inhibitors) of a gene are above the threshold and simultaneously none of the inhibitors (resp. activators) is, then the level of expression of the gene is maximum (resp. minimum, *i.e.* equal to 0).

Example 5 (Constraints on parameters).
By application of the above constraints to the biological parameters of the interaction graph from Figure 1 we obtain, after simplification, the following conditions : $K_\alpha(\{\alpha\}) = 2$, $K_\alpha(\{\beta\}) = 0$, $K_\beta(\{\}) = 0$, $K_\beta(\{\alpha\}) = 1$, $\big(K_\alpha(\{\}) < 2 \vee 0 < K_\alpha(\{\alpha,\beta\})\big)$ and $\big(K_\alpha(\{\}) > 0 \vee 2 > K_\alpha(\{\alpha,\beta\})\big)$. As a result, the guard $Increase(\beta) \equiv (x_\alpha < 1 \wedge x_\beta < K_\beta(\{\})) \vee (x_\alpha \geq 1 \wedge x_\beta < K_\beta(\{\alpha\}))$ of the associated Parametric GRN can be simplified in $(x_\alpha \geq 1 \wedge x_\beta = 0)$. Finally, only 7 sets of parameters remain consistent with all these conditions, over the 324 ones corresponding to the single Domain constraint. These 7 models correspond to 5 different dynamics. The simplification of the Parametric GRN of Figure 2 is given in the Figure 4:

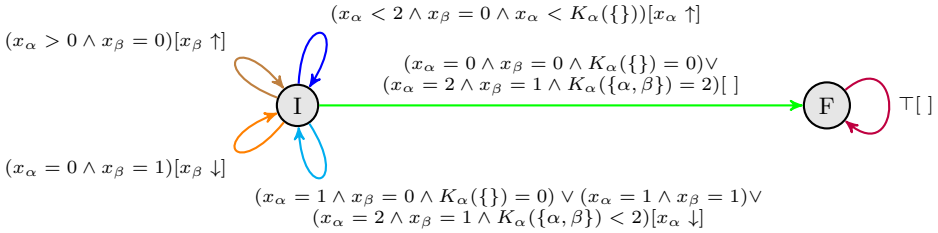


Fig. 4. Simplification of the Parametric GRN from Figure 2.

These constraints are not directly integrated inside the definition of Parametric GRN since there are not always considered by biologists (except for the Domain constraint). Thus, even if all these constraints seem to be well-founded, they can be relaxed on demand. In the sequel, by default, they will be considered.

3 Adapting LTL Model Checking

The classical approach of LTL model-checking consists in confronting one dynamics against observed (*in vivo* or *in vitro*) biological behaviours or against biological hypotheses expressed using temporal formulae in LTL over levels of expression of genes. To do this, the negation of the LTL formula is transformed into a so called Büchi automaton. Then the product between this automaton and the dynamics is computed, and we look for accepting paths in the product by checking the existence of reachable cycles containing at least an accepting state. Indeed if an accepting path exists, we have found a witness of a path of the

dynamics satisfying the negation of the biological property and we can conclude that the dynamics does not satisfy the property for all paths. Nevertheless, model checking is usually time consuming, and since the number of dynamics is large, this method is not applicable in practice. To avoid the combinatorial explosion, we check directly the Parametric GRN rather than each dynamic separately. Section 3.1 explains how we define the product between the Büchi automaton and the Parametric GRN. Since parameters remain symbolic, a path in this *Parametric Product* represents several paths in different dynamics, one for each different instantiation of parameters. Section 3.2 explains how we look for accepting paths based on symbolic execution techniques which are program analysis techniques initially developed for testing purposes [13]. The key point is the substitution of actual values by symbolic variables in order to symbolically perform computations. Each execution (or path) of the program associates to each variable a symbolic computation together with a *path condition* that expresses what are the conditions on input values to execute the given path. Symbolic execution techniques has been extended to symbolic transition systems [11] by unfolding transition systems as symbolic trees. As symbolic execution is only applicable for finite paths, selection criteria are used to cut infinite paths when considering testing. In the sequel, we will take particular care to cut infinite paths in identifying situations of return on a node already encountered. Indeed such situations reveal the presence of cycles.

3.1 LTL, Büchi automaton and Parametric Product

LTL. Biological properties on a sequence of (dynamic) states can be expressed using LTL formulae. These formulae are built from a set of atomic propositions using the usual logical operators in $\{\top, \bot, \neg, \wedge, \vee\}$ and the temporal operators **X** (for neXt time), **G** (Globally), **F** (Finally) and **U** (Until) [1]. Since we need to express biological knowledge on levels of expression of genes, atomic propositions will be of the form $x_i \approx c$ where x_i denotes the level of expression of a gene i, $\approx \in \{=, \neq, <, >, \leq, \geq\}$ and $c \in \mathbb{N}$.

Example 6. Considering the GRN from Figure 1, the existence of a steady state (*i.e.* a state which is itself its only own successor) in $(x_\alpha, x_\beta) = (2, 1)$ corresponds to the LTL formula $\mathbf{G}((x_\alpha = 2 \wedge x_\beta = 1) \rightarrow \mathbf{X}(x_\alpha = 2 \wedge x_\beta = 1))$.

Büchi automaton. Any LTL formula φ can be translated into a Büchi automaton $B(\varphi)$, that is a transition system such that an infinite sequence of states provided with truth values for all atomic propositions (a path) verifies φ iff this path is accepted by $B(\varphi)$, i.e. iff this path contains at least a so-called accepting state infinitely often.

Definition 4 (Biological Büchi Automaton). *A Biological Büchi Automaton for a GRN G is a tuple (Q, q_0, A, δ) where Q is the set of states, $q_0 \in Q$ is the initial state, $A \subseteq Q$ is the set of accepting states, and δ is the set of transitions which associates to $(q, q') \in Q^2$ a formula over the levels of expressions of genes from G.*

Example 7. The Büchi Automaton corresponding to the negation of the formula introduced in the example 7 is represented in Figure 5. This Büchi automaton has been computed with LTL2BA4J[4].

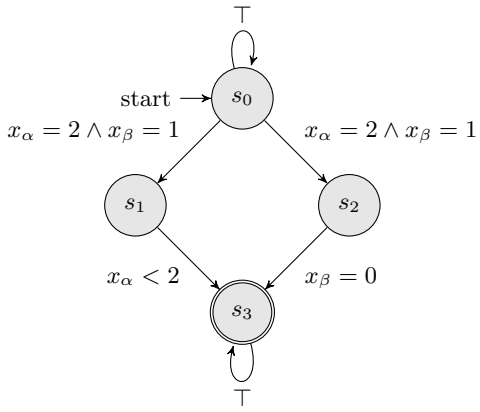


Fig. 5. $B(\neg\varphi)$ with $\varphi \equiv \mathbf{G}\big((x_\alpha = 2 \wedge x_\beta = 1) \Rightarrow \mathbf{X}(x_\alpha = 2 \wedge x_\beta = 1)\big)$.

Parametric product. From the Büchi Automaton $B(\neg\varphi)$ and the Parametric GRN P, we build the Parametric Product $\Pi = P \otimes B(\neg\varphi)$ to perform the model-checking of the meta-model. This product corresponds to the Cartesian product of P and $B(\neg\varphi)$.

Definition 5 (Parametric Product). *Let $P = (St, Tr)$ be a Parametric GRN and $B(\neg\varphi) = (Q_B, q_{0B}, A_B, \delta_B)$ a Büchi Automaton associated to the LTL formula $\neg\varphi$. The product $\Pi = P \otimes B(\neg\varphi)$ is the tuple $(Q_\Pi, q_{0\Pi}, A_\Pi, \delta_\Pi)$ with $Q_\Pi = St \times Q_B$ the set of vertices, $q_{0\Pi} = (I, q_{0B})$ the initial vertex, $A_\Pi = St \times A_B$ the set of accepting vertices, and δ_Π the set of transitions. A transition of δ_Π is of the form $(q_\Pi, g_\Pi, a, q'_\Pi)$ such that $q_\Pi = (s, q_B) \in Q_\Pi$, $q'_\Pi = (s', q'_B) \in Q_\Pi$, $g_\Pi = g \wedge \delta_B(q_B, q'_B)$ is satisfiable, with $(q_B, q'_B) \in {Q_B}^2$ and $(s, g, a, s') \in Tr$.*

Example 8. The product of the Parametric GRN from Figure 4 and the Büchi Automaton from Figure 5 is represented in the Figure 6. The product has been reduced by removing output transitions whose guard on expression levels is not satisfiable according to the guards and affectations of the input transitions of the same vertex; we also remove the transitions whose guard is not satisfiable according to the guards on parameters necessarily crossed (ϕ_{2_1} and ϕ_{2_2} here). Finally, we remove vertices which can not be reached and those belonging to a terminal cycle without accepting vertex.

3.2 Search of parametric accepting cycles

Symbolic execution. To find accepting cycles, we symbolically execute the product $\Pi \equiv P \otimes B(\neg\varphi)$: the parameters $K_i(\omega)$ are handled as symbolic variables

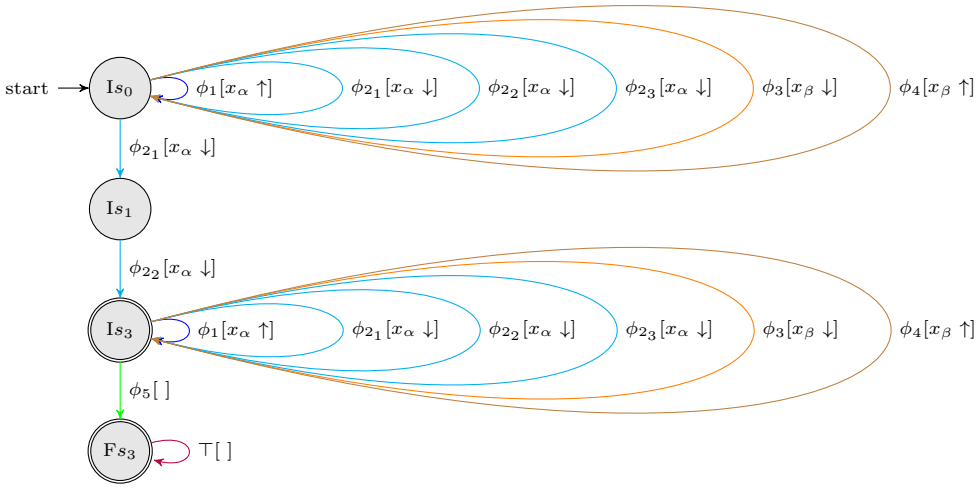


Fig. 6. Product $P \otimes B(\neg\varphi)$ associated to the *Parametric GRN* from Figure 4 and the Büchi Automaton from Figure 5 (after simplification), with:
$\phi_1 \equiv x_\alpha < 2 \wedge x_\beta = 0 \wedge x_\alpha < K_\alpha(\{\})$ $\quad \phi_3 \equiv x_\alpha = 0 \wedge x_\beta = 1$
$\phi_{2_1} \equiv x_\alpha = 2 \wedge x_\beta = 1 \wedge K_\alpha(\{\alpha, \beta\}) < 2$ $\quad \phi_4 \equiv x_\alpha > 0 \wedge x_\beta = 0$
$\phi_{2_2} \equiv x_\alpha = 1 \wedge x_\beta = 1$ $\quad \phi_5 \equiv x_\alpha = 0 \wedge x_\beta = 0 \wedge K_\alpha(\{\}) = 0.$
$\phi_{2_3} \equiv x_\alpha = 1 \wedge x_\beta = 0 \wedge K_\alpha(\{\}) = 0$

(i.e. not evaluated), and Π is unfolded leading to the construction of several *Symbolic Execution Trees* (SET).

Each SET is recursively built from a starting node (root node). Each node corresponds to a state of Π, a specific evaluation of all x_i and a path condition (in the form of a constraint over parameters $K_i(\omega)$) which defines the dynamics that can lead to reach this node.

From any node, by considering all transitions issued from its underlying state in $P \otimes B(\neg\varphi)$, we can compute the successor nodes: provided that the guard of the transition is satisfiable with regards to the x_i values of the current node, then a successor node is built, with an associated path condition equal to the conjunction of the path condition of the source node and the guard of the transition from Π. The values for x_i are computed with the assignment of the transition. Thus, by construction, path conditions expressed over parameters increase along paths of SET, and reduce the number of dynamics compatible with the path under construction.

Cut and termination. Biological properties are expressed along infinite sequences, and thus, paths of the product and paths of SET are also infinite. But, by disregarding path conditions, the number of possible nodes in a SET is finite [4], and some symbolic states need to appear infinitely often in a path under construction. So, when we are building a new node (child) that corresponds to an ancestor (the return node) (with the same product state and the same values for x_i), we stop the analysis of the path. By construction, the path condition of the child node is included in the path condition of the ancestor node, i.e. all parameter instantiations satisfying the child path condition also satisfy the ancestor path condition.

Thus, by performing a mixed symbolic and numerical execution (biological parameters $K_i(\omega)$ remain unchanged and the variables x_i are evaluated), we can stop the execution procedure of the product $P \otimes B(\neg\varphi)$ so that each path of the resulting SET is finite and contains a cycle (starting at the return node and ending with the transition leading to the child node). If there exists an accepting state between the return node and the child node, the path condition is said *accepting*.

Accepting path condition. Once the finite SET is built, it remains to compute for which parameter instantiations there exist accepting paths. For that, it suffices to consider every accepting path conditions of the SET associated to $P \otimes B(\neg\varphi)$. If the biological parameters verify (at least) one of the accepting path condition, it means that there exists a path in the product going infinitely often through the associated cycle, and thus passing infinitely often by an accepting state. And so there exists a path in the dynamics verifying $\neg\varphi$.

Thus, biological parameters verifying the conjunction of the negation of every accepting path condition of the SET associated to $P \otimes B(\neg\varphi)$ are such that there is no path verifying $\neg\varphi$, in another words, all paths verify φ. Note that the obtained dynamics verify φ along *all* paths; if the model must verify φ only on *at least one* path, our approach remains adequate with a small adaptation: for this we have to get the disjunction of all accepting path conditions of the SET associated to $P \otimes B(\varphi)$.

Example 9. For the Parametric Product from Figure 6, there are two solutions after computation; the corresponding values of parameters are: $K_\alpha(\{\}) = 1$ or 2, $K_\alpha(\{\alpha\}) = 2$, $K_\alpha(\{\beta\}) = 0$, $K_\alpha(\{\alpha, \beta\}) = 2$, $K_\beta(\{\}) = 0$ and $K_\beta(\{\alpha\}) = 1$.

Algorithm of traversal of SETs. The algorithm 1, based on a *Depth First Search* schema, gives an overview of how we practically compute all the accepting path conditions. We use three global variables: the product Π, the list of accepting path conditions *acceptingPC*, and the list *nodesList* of SET nodes which have already been analyzed. At least one root node is required to call for the first time the function $DFS()$. The root nodes are built from the initial

[4] the number is bounded by the product of all combinations of expression levels, the state number of the Büchi automaton and the state number (2) of P.

Algorithm 1 Overview of DFS()

```
Require: global Π, global acceptingPC, global nodesList
 1: function DFS(node)
 2:     for all transition of Π outgoing from the state of node do
 3:         if transition is VALID then
 4:             COMPUTE succ, the successor node reached by the transition
 5:             if the pc of succ is not INCLUDED in the acceptingPC then
 6:                 if succ is a RETURN NODE then
 7:                     if succ is an ACCEPTING RETURN NODE then
 8:                         Add the pc of succ to acceptingPC
 9:                     end if
10:                 else
11:                     if succ does not CORRESPOND to a node in nodesList then
12:                         DFS(succ)
13:                         Add succ to nodesList
14:                     end if
15:                 end if
16:             end if
17:         end if
18:     end for
19: end function
```

vertex of the Parametric Product, the initial values of levels of expression (by default we consider all the possibilities), and an initial path condition equal to $True$ (which means no constraint on parameters).

Starting with a SET node, line 2 to line 4 test and compute its successors, as explained in the "Symbolic Execution" part of section 3.2. Three tests are then performed successively. Firstly, if the path condition of the successor node is already known, it cannot provide additional information (the *pc* becomes more specific every depth call), and we stop the study of this successor (line 5). Secondly, line 6 tests if the successor is a return node back to one of its ancestors (ancestor with the same vertex and dynamic state). If it is the case, then there is an infinite cycle between them and, if there is an accepting node in that cycle, then the successor node is an accepting return node, and its path condition is added to the list $accepting\_PCs$ (lines 7 to 8). Thirdly, if the successor is not a return node, we check (line 11) if the node corresponds to a node in *nodesList* with the same vertex, the same dynamic state and a weaker path condition. If it is not the case, then the DFS function is recalled with the successor node in argument (line 12), which is then added to *nodesList* (line 13).

3.3 Case study

In order to demonstrate our methodology at work we present a simple case study, namely the analysis of the genetic network that controls the life cycle of the λ phage virus. We performed such study through the SPuTNIk tool.

The SPuTNIk tool. We have implemented a prototype software tool, called [5] SPuTNIk, which implements the reverse engineering procedure for parametric GRN models described above. SPuTNIk is written in Java and relies on the following tools: the Z3 constraint solver [7] (used for checking the satisfiability of path conditions during the construction of the symbolic execution tree for a given GRN problem) and the ltl2ba and LTL2BA4J libraries (used for generating a Büchi Automaton of minimal size from an LTL formula [10,4]).

λ phage. Aiming at validating our approach here we consider a previously studied λ phage model [14] consisting of four genes, denoted cI, cII, cro and N, and ten interactions (see graph G_λ in Figure 7). The bacteriophage λ is a virus that infects the *Esherichia Coli* (E. Coli) bacterium. It is characterised by a *temperate* life cycle, meaning that the virus can follow one of two destinies: either it integrates the genome of the host through a process called *lysogeny* or it enters a *lytic* phase through which it kills the residing cell to reproduce itself.

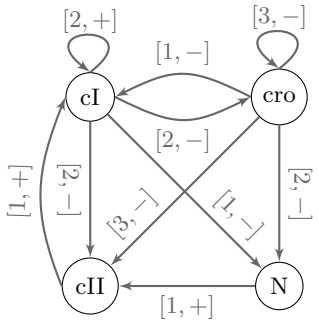


Fig. 7. The interaction graph G_λ for the λ phage.

The *lytic* and *lysogenic* phases of the λ phage correspond to specific states of G_λ (notice that a state of G_λ is a quadruple $(x_{cI}, x_{cII}, x_{cro}, x_N) \in \{0,1,2\} \times \{0,1\} \times \{0,1,2,3\} \times \{0,1\}$). In particular we distinguish between the following states $init := [0000]$, $lyt_1 := [0021]$, $lyt_2 := [0020]$, $lyt_3 := [0030]$, $lys_1 := [2101]$ and $lys_2 := [2000]$, where lyt_3 and lys_2 represent the completion of the lytic, respectively, lysogenic phase. In [14] such model has been studied (through *coloured model checking*) so to find instances that comply with *time series* given in the form $\theta : s_1, *, s_2, *, \ldots, *, s_n$ (i.e. where s_i is the i^{th} observed state while $*$ denotes a possibly empty sequence of unspecified states). Specifically here we consider the same pair of time series studied in [14], namely: θ_1 and θ_2 which are illustrated in Table 1 together with their equivalent LTL counterparts.

We then asked SPuTNIk to find out the instances of G_λ which are guaranteed to exhibit either a lytic or a lysogenic phenotype in compliance with series θ_1

[5] *Symbolic Parameters of Thomas' Networks Inference*

time series	LTL formula
$\theta_1 : init, *, lyt_1, *, lyt_2, *, lyt_3$	$\phi_1 := init \wedge F(lyt_1 \wedge F(lyt_2 \wedge F(lyt_3 \wedge F(lyt_2))))$
$\theta_1 : init, *, lys_1, *, lys_2$	$\phi_2 := init \wedge F(lys_1 \wedge F(lys_2))$

Table 1. The considered time series θ_1 and θ_2 and the corresponding LTL formulae.

and θ_2 (i.e. all models that contains at least one trajectory that satisfies ϕ_1 and at least one that satisfies ϕ_2). In order to compare the results computed with SPuTNIk with that in [14] we had to consider a specific setting for the SPuTNIk experiments: i.e. we had to discard the Min/Max constraint for all genes (as it is not supported in [14]), and we had to relax the Observation constraint for the specific case where cI is activator of itself (as done in [14], i.e. we do not impose that it exists at least one dynamic state where the increase of the level of expression of cI leads to an increase of the expression level of cI). The obtained results are in agreement: amongst the over 7 billions total model instances, we obtain the same 8759 valid models as in [14].

4 Conclusion

In this paper we introduced a new methodology for reverse-engineering of genetic network models, based on adaptation of classical LTL model-checking with symbolic execution. In order to find dynamics consistent with biological knowledge, we use the whole extent of LTL to express biological knowledge in terms of constraints over time. Instead of checking each dynamics of the GRN, we propose a method which performs checking with a novel formalism, the Parametric GRN, a compact (symbolic) representation of all the dynamics associated to an interaction graph within a single structure. From the Parametric GRN and LTL formulae, our algorithm processes parameters, defining the dynamics, as symbols in order to avoid combinatorial explosion. The solutions are in the form of a set of constraints that the parameters must fulfil. Such analysis has been carried out through the SPuTNIk tool, a prototype software implementation of the proposed method.

References

1. C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.
2. J. Barnat, L. Brim, A. Krejci, A. Streck, D. Safránek, M. Vejnar, and T. Vejpustek. On parameter synthesis by parallel model checking. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 9(3):693–705, 2012.
3. G. Bernot, J.-P. Comet, A. Richard, and J. Guespin. Application of formal methods to biological regulatory networks: extending Thomas' asynchronous logical approach with temporal logic. *Journal of Theoretical Biology*, 229(3):339–347, August 2004.
4. Eric Bodden. *LTL2BA4J Software, http://www.sable.mcgill.ca/ ebodde/rv/ltl2ba4j/*. RWTH Aachen University, 2011.

5. F. Corblin, E. Fanchon, and L. Trilling. Applications of a formal approach to decipher discrete genetic networks. *BMC Bioinformatics*, 11:385, 2010.
6. F. Corblin, S. Tripodi, E. Fanchon, D. Ropers, and L. Trilling. A declarative constraint-based method for analyzing discrete genetic regulatory networks. *BioSystems*, 98:91–104, 2009.
7. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
8. D. Filopon, A. Mérieau, G. Bernot, J.-P. Comet, R. Leberre, B. Guery, B. Polack, and J. Guespin. Epigenetic acquisition of inducibility of type III cytotoxicity in P. aeruginosa. *BMC Bioinformatics*, 7:272–282, 2006.
9. J. Fromentin, J.-P. Comet, P. Le Gall, and O. Roux. Analysing gene regulatory networks by both constraint programming and model-checking. In *EMBC'07, 29th IEEE Engineering in Medicine and Biology Society*, pages 4595–4598, 2007.
10. P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65, Paris, France, 2001. Springer.
11. C. Gaston, P. Le Gall, N. Rapin, and A. Touil. Symbolic execution techniques for test purpose definition. In *18th IFIP Int. Conf. TestCom*, volume 3964 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2006.
12. Z. Khalis, J.-P. Comet, A. Richard, and G. Bernot. The SMBioNet method for discovering models of gene regulatory networks. *Genes, Genomes and Genomics*, 3(special issue 1):15–22, 2009.
13. J.-C. King. A new approach to program testing. *Proceedings of the international conference on Reliable software, Los Angeles, California*, 21-23:228–233, April 1975.
14. H. Klarner, A. Streck, D. Šafránek, J. Kolčák, and H. Siebert. Parameter identification and model ranking of thomas networks. In *Proceedings of the 10th international conference on Computational Methods in Systems Biology*, CMSB'12, pages 207–226, Berlin, Heidelberg, 2012. Springer-Verlag.
15. D. Mateus, J.-P. Gallois, J.-P. Comet, and P. Le Gall. Symbolic modeling of genetic regulatory networks. *Journal of Bioinformatics and Computational Biology*, 5(2B):627–640, 2007.
16. A. Richard. *SMBioNet User manual, http://www.i3s.unice.fr/ richard/smbionet/*, 2010.
17. E. Snoussi and R. Thomas. Logical identification of all steady states: the concept of feedback loop characteristic states. *Bull. Math. Biol.*, (55(5)):973–991, 1993.
18. D. Thieffry, M. Colet, and R. Thomas. Formalisation of regulatory networks : a logical method and its automation. *Math. Modelling and Sci. Computing*, 2:144–151, 1993.
19. D. Thieffry and R. Thomas. Dynamical behaviour of biological regulatory networks - II. immunity control in bacteriophage lambda. *Bull. Math. Biol.*, 57(2):277–97, 1995.
20. R. Thomas and R. d'Ari. *Biological Feedback*. CRC Press, 1990.

Premières leçons sur la spécification d'un train d'atterrissage en B événementiel

Jean-Pierre Jacquot

LORIA – équipe DEDALE – Université de Lorraine
Vandoeuvre lès Nancy, France
Jean-Pierre.Jacquot@loria.fr

Résumé Ce papier présente les leçons préliminaires obtenues en traitant en B Événementiel l'étude de cas proposée par la conférence ABZ 2014. Le problème consiste à modéliser le logiciel de contrôle du train d'atterrissage d'un avion. L'utilisation de B Évémentiel sur cette étude pose des questions intéressantes quant à la nature des invariants, quant au moment de leur introduction, ainsi que quant à l'expression et la vérification des propriétés fonctionnelles. Le raffinement est organisé en niveaux d'observation structurés par la description du matériel. Le système est vu comme un automate assez simple piloté par des capteurs externes. La description d'un tel système en B Événementiel est simple mais sa validation est beaucoup plus difficile. Cette étape utilise JeB, un simulateur de B Événementiel en JavaScript. L'émulation des capteurs est un point crucial.

1 Introduction

Pour promouvoir l'usage des méthodes formelles, il est important d'analyser chacune en tant qu'outil. Dans cette perspective, une méthode est moins caractérisée par ses propriétés intrinsèques, par exemple le type de logique utilisée, que par la problématique de son emploi. Sur quel type de problèmes est-elle adaptée ? Peut-on l'adapter à d'autres contextes ? Quelles phases et activités du développement permet-elle de traiter ? Les environnements supports sont-ils adéquats ? En délimitant ainsi le périmètre de chaque méthode, nous facilitons le choix par les développeurs de la « meilleure » pour leur projet en cours.

Dans la veine de [7] qui explorait l'usage de B Événementiel [2] pour la modélisation de domaine, ce papier propose une première analyse de cette méthode sur l'étude de cas proposée par V. Wiels et F. Boniol [5]. Cette étude propose de modéliser le logiciel de contrôle de manœuvre du train d'atterrissage d'un avion. L'architecture du système comporte trois grands éléments :

- une interface de contrôle pour le pilote (levier et indicateurs lumineux),
- la partie mécanique et hydraulique actionnée via des électrovannes et observée par des capteurs, et
- le logiciel de commande.

B Événementiel n'est pas, *a priori,* le langage de choix pour ce type de système. De fait, trois caractéristiques introduisent des difficultés sensibles :

– Invariant faible. En première analyse, le système peut se concevoir comme un petit automate fini. B Événementiel permet d'écrire facilement un automate, états et transitions, mais ne permet pas de vérifier, c'est-à-dire, de prouver, que la spécification est bien celle de l'automate et qu'il a bien les bonnes propriétés. La raison tient principalement au fait que le parcours de l'automate s'exprime très mal à travers un invariant.
– Interface avec la partie matérielle. Le matériel possède un comportement propre et autonome dont doit tenir compte le logiciel de contrôle. La difficulté est moins dans l'expression des « communications » entre logiciel et matériel que dans la vérification/validation des bons comportements.
– Exigences temporelles. De nombreuses exigences sont exprimées comme des réponses à la (non) réalisation d'actions matérielles dans un délai donné. B ne possède pas de moyen d'expression natif pour ce type d'invariants qu'il est néanmoins possible de modéliser [11].

Ce papier traite uniquement des deux premières caractéristiques. Elles ont en commun le fait que si les preuves restent indispensables pour garantir le développement, il faut leur adjoindre des activités qui relèvent de la *validation.* Il convient en effet d'observer le comportement des modèles pour s'assurer que les états s'enchaînent bien, qu'il n'y a pas de blocage, etc. Il faut donc utiliser des outils d'animation et de *model-checking* comme AnimB [9] ou ProB[4]. Ce papier utilise JeB, un outil de simulation développé pour permettre la validation de modèle B Événementiels à tous les niveaux de raffinement [13].

La suite est structurée de la façon suivante. B Événementiel et JeB sont d'abord brièvement décrits. Puis, la stratégie générale de développement suivie sur l'étude de cas est présentée. La simulation des modèles est ensuite décrite. Enfin, une analyse des observations des différents outils est proposée.

2 Cadre formel et outils

2.1 B Événementiel

B Événementiel est un cadre de spécification basé sur trois idées :
– un système est décrit par un modèle formel constitué d'un état et d'un ensemble d'événements qui agissent sur l'état,
– un développement est une suite de modèles qui sont liés par une relation de raffinement formel,
– la sémantique de chaque modèle ainsi que celle des raffinements est donnée par un ensemble d'obligations de preuve.

La combinaison de ces trois éléments permet de définir une notion de correction pour chaque modèle qui garantit que le modèle final est une implantation

correcte du modèle initial lorsque toutes les obligations de preuves ont été démontrées. Ce cadre formel s'inscrit dans la démarche de production de logiciels « corrects par construction ».

Modèle Un état est une fonction associant des noms à des valeurs. Les valeurs sont des expressions ensemblistes construites à partir d'entiers, de symboles, des ensembles $\mathbb{N}$ et $\mathbb{Z}$, et d'ensembles de symboles (*carrier sets*). Des notations spécifiques permettent de construire facilement les relations–dont les fonctions– entre ensembles ainsi que les opérations telles que l'image d'un sous-ensemble par une relation ou l'inverse d'une relation. Le typage est défini comme l'appartenance à un ensemble. Syntaxiquement, B Événementiel distingue les constantes des variables qui décrivent respectivement les parties statiques et les parties dynamiques de l'état.

L'élément essentiel du modèle est *l'invariant* qui circonscrit l'ensemble des valeurs licites que peut prendre l'état. L'invariant est une formule logique du premier ordre portant sur les valeurs des variables et des constantes. Syntaxiquement, c'est une conjonction de prédicats. Un état licite est un état pour lequel l'invariant est vrai.

Un événement est une substitution gardée. La garde est un prédicat du premier ordre sur l'état. La substitution est une affectation parallèle de nouvelles valeurs à un sous-ensemble des variables. Un événement peut comporter des *paramètres* qui sont des variables libres susceptibles de prendre n'importe quelle valeur qui rend la garde vraie. Un événement est *déclenchable* si sa garde est vraie ; le déclenchement d'un événement provoque une évolution de l'état.

Raffinement Du point de vue du développeur, le raffinement est une opération qui consiste à transformer un modèle abstrait en un modèle plus concret par l'adjonction de nouveaux éléments qui rapprochent d'une implantation par un langage de programmation.

Concrètement, un raffinement en B Événementiel peut être vu comme une ou plusieurs opérations élémentaires.

- L'extension de l'état. Cette opération introduit de nouvelles variables et constantes sans relation avec l'état abstrait. Elle permet de décrire graduellement les différentes facettes d'un système.
- Le renforcement de l'invariant. Cette opération restreint l'espace dans lequel le modèle évolue à un espace plus proche de celui du système final.
- La réification de variables. Cette opération correspond au classique « raffinement de données » par lequel une variable abstraite est codée par une ou des variables plus proches d'une structure de données programmable.

– La décomposition d'un événement. Cette opération décompose un événement « global » en un ensemble d'événements « locaux ». Elle correspond à l'introduction d'un nouveau niveau d'observation du système.

Techniquement, chaque opération se traduit par une évolution de la plupart des composants du texte de la spécification : invariants, axiomes, gardes et actions. L'intérêt de la typologie des opérations est d'ordre méthodologique. Elle permet de mieux comprendre la différence entre le raffinement vu par la méthode B [1], la description plus précise d'un ensemble fixe de fonctions, et le raffinement vu par B Événementiel, l'enrichissement d'un modèle. Cette distinction n'a pas d'impact sur la définition formelle du raffinement. En revanche, elle induit la nécessité de nouveaux outils d'aide au développement.

Sémantiques La sémantique axiomatique de B Événementiel est structurée autour de trois grands principes. Un, le modèle doit être réalisable : l'ensemble des états licites n'est pas vide. Deux, l'invariant est préservé lorsqu'un événement est déclenché depuis un état licite. Une autre façon de penser cette propriété est que les événements ne permettent pas de rejoindre un état non licite. Trois, le raffinement maintient l'invariant abstrait : il existe une fonction d'abstraction reliant l'état du modèle concret au modèle abstrait et chaque événement concret maintient l'invariant abstrait.

Grâce à une syntaxe adaptée, ces principes peuvent être traduits en obligations de preuve qui, appliquées à une spécification, génèrent un ensemble de théorèmes dont la démonstration garantit la correction du modèle. Cette sémantique est particulièrement adaptée pour le développement de systèmes dont l'implantation doit garantir le respect de propriétés exprimables comme une invariance dans un espace abstrait.

Le développeur a également besoin d'une vision plus opérationnelle des modèles afin de définir ou analyser les comportements du système. La sémantique opérationnelle de B Événementiel est intuitivement décrite par :

déclencher l'événement INITIALISATION
calculer l'ensemble des événements déclenchables (Ed)
tant que $Ed \neq \emptyset$
    choisir un événement e dans Ed
    fixer les valeur des paramètres de e
    exécuter la substitution de e
    calculer Ed

L'arrêt peut correspondre à une anomalie dans le modèle (*deadlock*) ou au résultat souhaité (terminaison du calcul).

La caractéristique principale des sémantiques de B Événementiel concerne le non-déterminisme. Celui-ci est présent à trois niveaux : lors du choix des

paramètres des événements, lors du choix de l'événement à déclencher et lors du choix des valeurs à substituer. Hors extension de l'état, les opérations de raffinements sont généralement guidées par la diminution du non-déterminisme.

2.2 JeB

La conception du raffinement en B Événementiel comme un enrichissement du modèle formel implique que la question de la correction du développement doit être posée différemment. Si la preuve, c'est-à-dire la vérification, garantit toujours que les invariants fondamentaux sont maintenus, elle ne dit rien quant au fait que les modèles raffinés sont conformes aux exigences ou aux contraintes réelles. Assurer cette conformité est le rôle de la validation.

Parmi les techniques de validation, l'exécution du modèle formel est un outil de choix pour vérifier que les comportements spécifiés couvrent les scénarios prévus ou qu'il n'y a pas de comportement anormal (*deadlock* par exemple). L'exécution du modèle formel peut être réalisée de plusieurs façons. Le modèle peut être traduit en un programme rédigé dans un langage exécutable : E2B [12] ou EB2ALL [8] sont de tels compilateurs. Le modèle peut être directement interprété par un outil qui réalise la sémantique opérationnelle : AnimB[9] ou ProB [6] utilisent ce principe. Traduction et interprétation, encore appelée animation, sont efficaces sur des modèles suffisamment déterministes. En effet, les choix nécessitent d'énumérer des ensembles de valeurs. Même lorsque les ensembles sont suffisamment définis pour que leurs éléments soient énumérables, la technique bute souvent sur une explosion combinatoire.

JeB est un environnement conçu pour aider à la validation des modèles lorsque l'animation ou la traduction sont inopérantes. L'idée est de construire des *simulateurs* du modèle, c'est-à-dire des versions exécutables composées :

- d'une traduction automatique du modèle en JavaScript,
- d'un moteur d'exécution comportant la boucle de base, une bibliothèque des opérations ensemblistes en JavaScript et une interface HMTL de visualisation et de contrôle des simulations,
- de fragments de code en JavaScript fournis par l'utilisateur pour éliminer ou réduire les non-déterminismes massifs.

L'idée sous-jacente à JeB est qu'un développeur peut utiliser la démarche des « petits pas à partir d'un modèle très abstrait » défendue dans [3] tout en ayant une idée assez précise de la nature du résultat qu'il obtiendra. Il lui est ainsi possible de proposer très tôt des implantations raisonnables d'ensembles laissés indéfinis, ou d'imposer les trajectoires de comportement qui sont les plus pertinentes. L'implantation de JeB est décrite dans la thèse de Yang [13].

La question de la fidelité de la simulation au modèle se pose naturellement dès qu'une intervention humaine est nécessaire. Une définition formelle de la

notion de fidélité est proposée dans [14]. Elle donne lieu à la création d'obligations de preuves qui permettent de garantir que les comportements observés sur une simulation sont bien spécifiés par le modèle.

3 Stratégie de développement

3.1 Présentation de l'étude de cas

L'étude de cas traite du développement d'un système de contrôle du train d'atterrissage d'un avion. Le point de départ est un cahier des charge informel qui décrit l'architecture matérielle, le comportement des différents composants et les exigences pour le logiciel de commande.

Au niveau le plus externe, le train d'atterrissage se compose de trois atterrisseurs et d'une interface à usage du pilote. Chaque atterrisseur est contitué d'une trappe, d'une jambe et de verrous. L'interface comporte le levier de commande et des indicateurs lumineux informant de l'état du train complet. Au niveau le plus interne, chaque atterrisseur se compose des pistons hydrauliques de manœuvre de la jambe et de la trappe, ainsi que des micro-capteurs qui détectent la position de la trappe et de la jambe. Le niveau intermédiaire se compose des électro-vannes de commande des pistons, des capteurs virtuels agrégeant les micro-capteur et des capteurs globaux (pression, alimentation électrique, etc.).

Les comportements sont décrits à travers les séquences d'états observées en fonctionnement nominal et la caractérisation des états pour les capteurs et les indicateurs. Une importance particulière est donnée à la description du temps de latence des manœuvres élémentaires des équipements hydrauliques. Des contraintes temporelles, liées aux temps de latence, entre différents commandes sont décrites. La définition de la « santé » du système et des actions de traitement des fautes détectées est détaillée.

Les exigences sont données sous la forme de deux listes. La première liste concerne le fonctionnement nominal. Elle contient des items d'ordre fonctionnel, c'est-à-dire des états à atteindre. Par exemple (page 18 de [5]) :

> *(R_{11}bis) When the command line is working (normal mode), if the landing gear command handle has been pushed DOWN and stays DOWN, then eventually the gears will be locked down and the doors will be seen closed.*

Les items d'ordre non-fonctionnel sont des contraintes sur l'enchaînement de certaines actions (page 18 de [5]) :

> *(R_{31}) When the command line is working (normal mode), the stimulation of the gears outgoing or the retraction electro-valves can only happen when the three doors are locked open.*

La seconde liste contient la définition des états anormaux qui doivent être détectés sur de critères temporels, principalement le dépassement de délais (page 19 de [5]) :

(R_{61}) If one of the three doors is still seen locked in the closed position more than 0.5 second after simulating the opening electro-valves, then the boolean output normal_mode *is set to* false

Au final, le cahier des charges pose 19 exigences, dont 12 (ou 10 dans une version simplifiée des exigences fonctionnelles) concernent le respect de délais.

3.2 Développement et raffinements

L'approche « naturelle » des méthodes basées sur le raffinement est d'exprimer les exigences comme des invariants sur un état. La démarche débute alors par la définition d'une abstraction de l'espace où évoluera le système à définir. Si le choix est judicieux, les invariants et les fonctions peuvent être décrits de façon compacte et complète. Cette compacité rend possible la validation du modèle par des procédures de lecture attentive par des experts du domaine.

Dans le cas présent, les exigences et contraintes sont décrites sur des éléments de bas niveau impliqués dans le fonctionnement des composants physiques. En fait, celles-ci sont des contraintes sur la dynamique interne du système. Il y a peu de possibilité d'abstraction du système physique pour exprimer tous les invariants. En conséquence, la démarche usuelle a de fortes chances d'amener à la construction d'un modèle initial de grande taille, donc très difficile à valider. Par ailleurs, la modélisation explicite du temps en B Événementiel ne fait qu'ajouter à la difficulté.

Une approche différente, conforme à l'esprit d'introduction la plus progressive possible de la complexité, a été choisie. Elle repose sur une stratégie générale de description de « l'extérieur vers l'intérieur » correspondant à l'introduction progressive des composants matériels. Les exigences sont donc introduites parallèlement à la progression des raffinements.

Ce papier utilise les cinq premiers modèles du développement (modèle initial et 4 raffinements) [1].

Modèle initial Ce modèle décrit l'état macroscopique de l'avion : fonctionnement nominal ou non, train déployé ou non. Il possède trois événements : prepare_landing, prepare_flight et alert. La vérification et la validation de ce modèle sont triviales.

Raffinement 1 Le premier raffinement est un changement de niveau d'observation. Il est guidé par les composants matériels externes qui participent à l'action : trappe, jambe et verrous, levier de commande. Les mouvements se décrivent facilement par des automates, la spécification encode simplement ceux-ci par un

1. Le texte B Événementiel est disponible sous forme d'archive Rodin à l'adresse : http://dedale.loria.fr/?q=en/etude-train-atterrissage

état (au sens de B) comportant les états (au sens des automates) des composants et des événements modélisant les transitions. Ainsi, prepare_landing se décompose en

- switch_handle_down : commande du pilote,
- initiate_prepare_landing : mise en activité du train,
- landing_unlock : déverrouillage et ouverture des trappes,
- landing_start_lowering : descente de la jambe,
- landing_stop_lowering : jambe descendue,
- landing_lock_down : verrouillage et fermeture des trappes.

Le modèle comporte une vingtaine d'événements dont six correspondent à l'exigence (implicite) que les manœuvres puissent être interrompues et inversées à tout moment. La vérification est triviale puisque l'invariant ne concerne que le typage des nouvelles variables. La simulation avec JeB ne pose pas de difficulté et il est aisé de valider le modèle en exécutant tous les scénarios possibles.

Raffinement 2 Le deuxième raffinement est une extension de l'état pour introduire les indicateurs lumineux. Les contraintes sur les indicateurs sont décrites dans l'invariant, par exemple :

" no_light_failure " (operating_mode=normal) ⇒ (light_failure= light_off)
" light_failure " (operating_mode=emergency) ⇒ (light_failure=light_on)

Les gardes des événements sont inchangées. Comme les indicateurs lumineux changent de valeur à l'occasion de certaines transitions, la partie action des événements les codant est modifiée. La vérification du modèle n'est pas complexe mais se révèle fastidieuse du fait de la multiplication des analyse par cas. La validation est facile une fois développée une visualisation graphique et colorée des indicateurs et atterrisseurs.

Raffinement 3 Le troisième raffinement est également une extension de l'état qui introduit les capteurs. Un capteur est ici l'entité logique perçue par le système ; sa définition à partir des micro-capteurs sera réalisée dans un raffinement ultérieur. Comme lors du raffinement précédent, seules les parties action des événements sont concernées. Par exemple, pour l'événement prepare_landing, le capteur d'ouverture de trappe doit indiquer `faux` et l'interrupteur général `ouvert` :

"door_open" door_open := LANDING_SET ∗∗ {sfalse}
"switch" analog_switch := open

Il peut sembler surprenant que les capteurs se voient affectés des valeurs dans la partie action des événements. En fait, il faut interpréter ces affectations comme « les valeurs qui devront être lues après que l'événement ait eu lieu ». Ceci sera utilisé pour la vérification du raffinement qui introduira la lecture effective

des micro-capteurs. La vérification et la validation sont dans la continuité du raffinement précédent.

Raffinement 4 Le quatrième raffinement est un nouveau changement de niveau d'observation. Il est guidé par l'introduction des micro-capteurs et de leur lecture. Le cahier des charges indique que chaque capteur logique est le résultat de l'agrégation de trois micro-capteurs. À ce stade, les fonctions d'agrégation sont simplement postulées. La stratégie de modélisation suivie est d'introduire un seul événement de lecture par groupe de capteurs : un pour les capteurs liés aux atterrisseurs et aux circuit généraux, et un pour le capteur lié au levier de commande dans le cockpit. Ce dernier est exprimé comme :

event read_handle_sensor
any h_s
where
 "flipped−state" flipped = strue
 "handle_phase−in" handle_phase = reading
 "h_s−type" h_s ∈ 1..3 → HANDLE_STATES
then
 "μ_handle−out" μ_handle := h_s
 "handle_phase−out " handle_phase := read

La phase code l'état d'un protocole qui permet de décomposer les événements et de synchroniser les lectures. Le protocole peut se schématiser ainsi :

$$Evt_{R_3} \triangleq init\_Evt;\ do\_Evt;\ read\_sensors;\ Evt_{R_4}$$

où la gauche de $\triangleq$ est un événement du Raffinement 3 et la droite l'enchaînement des événements du Raffinement 4 qui le réalisera.

Concrètement, le Raffinement 3 est transformé de manière systématique par le méta-algorithme suivant :

```
Pour chaque évènement Evt
            introduire init_Evt et do_Evt
            mettre les actions sur les capteurs dans les gardes
            (les affectations devenant des égalités)
            remplacer les valeurs dans les actions par
            l'agrégation des valeurs lues des micro-capteurs
```

Cette structure pourrait être simplifiée, mais elle présente trois avantages : l'indépendance des mouvements des atterrisseurs est conservée, la synchronisation des événements clés est garantie, le terrain pour l'introduction des contraintes de temps de chaque mouvement est préparé. En particulier, l'introduction des événements *Init_Evt* servira à la mise en place des *timers.* Il faut noter que la synchronisation est ici une propriété émergente (implicite) qui modélise le fait qu'un seul circuit hydraulique physique assure les mouvements.

La vérification est fastidieuse (longues analyses par cas) mais ne présente pas de difficulté majeure. La validation a posé quelques dificultés qui seront exposées dans la section suivante.

4 Validation des comportements

En B Événementiel, la démonstration des obligations de preuve garantit la correction des modèles et du développement. Néanmoins, celles-ci traitent presque exclusivement de la partie fonctionnelle du système. Dans les cinq premiers modèles, seule la gestion des indicateurs du cockpit et des valeurs des capteurs relève de cette partie fonctionnelle. Les preuves ne garantissent pas que les comportements spécifiés sont bien ceux attendus. Ainsi, les premières versions du Raffinement 4 étaient prouvées bien qu'elles ne respectaient pas l'exigence de réversibilité de la manœuvre. Il faut donc des outils d'analyse des comportements.

4.1 La visualisation graphique

La figure 1 montre l'exécution d'une simulation du Raffinement 4 avec JeB. Le bas de la fenêtre visualise le modèle formel dans toutes ses composantes : état à gauche, événements au centre et historique du scénario à droite. Cette partie est générée automatiquement. La partie haute présente une vue graphique qui facilite l'analyse des comportements. La réalisation de cette partie est sous la responsabilité totale des utilisateurs.

La programmation du graphisme est réalisée en utilisant l'API `canvas` de HTML5. JeB fournit les points d'accès nécessaires pour observer les valeurs dans l'état. Il est également possible d'agir sur la simulation en « basculant » le levier ou en désignant certains micro-capteurs pour les rendre défectueux.

La réalisation du graphisme ne présente aucune difficulté autre que d'obtenir une interface visuellement plaisante, lisible et portant l'information nécessaire en utilisant des primitives graphiques de bas niveau.

La table 1 donne la répartition des lignes de code JavaScript pour la réalisation de la simulation du Raffinement 4. Elle permet de comparer le nombre de lignes de JavaScript selon leur provenance (générées par JeB ou programmées par l'utilisateur). La partie programmée est elle-même décomposée selon leur fonction dans la simulation. Il faut noter que le code utilisateur est développé incrémentalement : toutes les constantes introduites pour un raffinement sont nécessaires pour les suivants, une grande partie du code graphique et des fonction d'acquisitions des paramètres est conservée entre les raffinements.

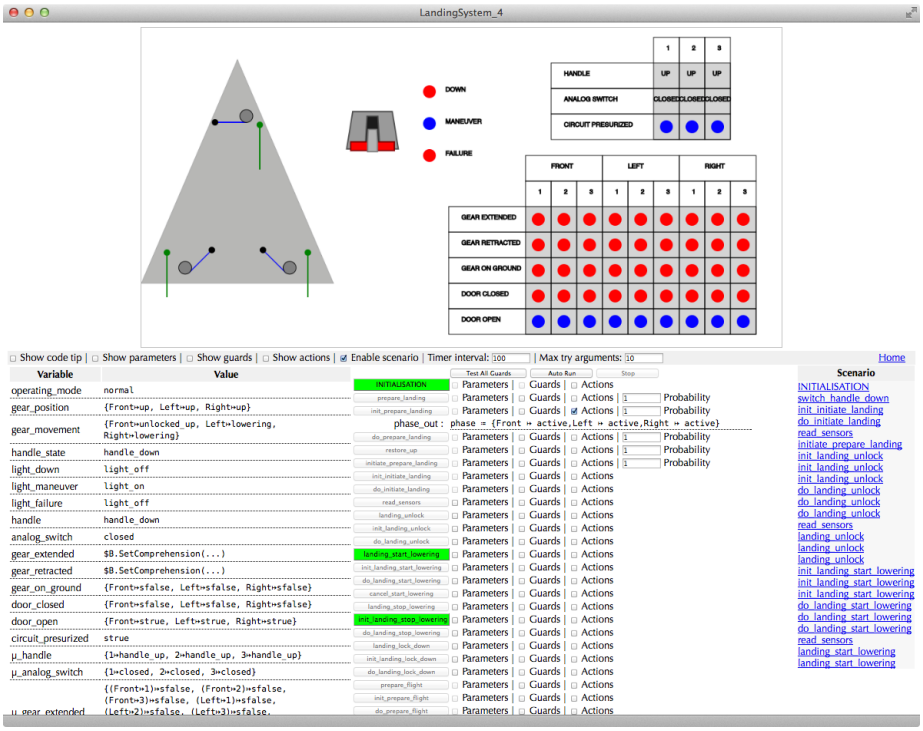


FIGURE 1. Simulation du Raffinement 4

Générées par JeB		10 100
Programmées à la main		
	Valuation des constantes	410
	Graphisme	410
	Émulation capteurs	630
Total manuel		1 450
Total simulation		11 550

TABLE 1. Répartition de l'effort de programmation

4.2 L'émulation du matériel

Les chiffres de la table 1 montrent que l'émulation du comportement du système matériel est ce qui a requis le plus d'effort. La difficulté est de convevoir un émulateur qui possède le comportement attendu tout en autorisant la simulation à être pilotée via l'interface générée par JeB. En effet, il faut que l'utilisateur puisse choisir librement l'événement à déclencher au cas ou il y en a plusieurs, il faut aussi qu'il y ait une synchronisation entre la boucle de la sémantique opérationnelle du modèle et la lecture des valeurs fournies par l'émulateur.

Techniquement, les micro-capteurs sont codés comme des objets. Leur état interne contient une table qui associe à chaque état du mouvement du train la valeur nomimale du capteur et l'état courant. Ils ont deux méthodes : la lecture qui donne la valeur associée à l'état *suivant* et l'avancement de l'état. L'événement read_sensor utilise ces deux méthodes : la lecture lors de la génération des valeurs des paramètre et l'avancement lors de la réalisation des actions.

Cet émulateur d'un système parfait a permis de valider la spécification qui décrit les comportements nominaux. Il a été indispensable pour mettre au point le comportement de réversibilité. C'est par l'exécution de différents scénarios avec JeB et l'émulateur que le petit protocole sur les capteurs a été augmenté d'une phase d'attente pour la re-synchronisation lors de l'interruption de la manœuvre. C'est également à travers ces exécutions que les événements de gestion de l'interruption ont été rétroactivement introduits dans le Raffinement 1.

5 Observations sur les outils

Au moment où ce papier est écrit, le modèle comporte 55 événements, environ 1 300 lignes pour le Raffinement 4 et 260 lignes pour les différents contextes. Bien que des parties importantes de l'étude de cas restent à aborder, notamment la modélisation des délais, certaines leçons peuvent être tirées dès maintenant.

5.1 B Événementiel

B Événementiel n'a pas été, *a priori,* conçu pour traiter ce type de problème, néanmoins, son usage n'a pas soulevé de difficulté particulière. L'absence de moyen d'exprimer facilement un automate (il faut le coder explicitement) est compensée par la simplicité du modèle opérationnel.

La plus grosse difficulté d'usage de B Événementiel tient surtout à la gestion des remises en cause. La démarche méthodologique associée au raffinement formel est un modèle en cascade : d'abord on fixe les exigences, puis on les formalise abstraitement, puis on construit en garantissant la conformité aux

exigences. Ce modèle de développement ne prévoit pas de retour en arrière et suppose que le modèle initial est complet. Or, peu de développements respectent ces contraintes : les exigences évoluent [10] et il est rare de trouver tout de suite la « bonne » solution.

Dans le cadre de ce développement, la gestion de l'interruption des manœuvres est un cas typique. La solution initiale, introduite au Raffinement 1, fonctionnait jusqu'au Raffinement 3. Ensuite, il est apparu qu'elle ne pouvait pas être raffinée pour intégrer la re-synchronisation de la lecture des capteurs. D'un point de vue théorique, il faut effacer et refaire les raffinement depuis le point où est introduite la nouvelle solution. D'un point de vue pratique, il est plus facile de penser en terme de modification, c'est-à-dire en terme d'une opération qui fait quelques petits changement et garde le reste. Circonscrire le changement, évaluer les conséquences, propager la modification : ces opérations ne sont pas prises en compte aujourd'hui. Ceci ralentit beaucoup le développement.

5.2 Rodin

Le Raffinement 4 génère environ 500 obligations de preuves visibles, c'est-à-dire, qui ne sont pas trivialement vérifiées. De l'ordre d'un tiers requièrent une intervention du développeur, souvent limitée au choix d'un démonstrateur. Un quart environ nécessite de guider la preuve, principalement en organisant des analyses par cas et des factorisations d'expressions. Rodin et les démonstrateurs associés sont assez efficaces.

Deux points techniques méritent une amélioration. Lors des modifications rétroactives, le système complet est re-vérifié : toutes les obligations de preuves sont considérées comme non prouvées et l'ancienne preuve est rejouée. Trop souvent, les outils laissent des obligations dans un état *à prouver* alors que la simple ouverture de l'obligation dans le démonstrateur interactif conclut la démonstration. L'outil a donc conservé assez d'information mais ne l'a pas exploitée. Cette situation fait perdre énormément de temps. Sauf dans les cas où on fait de la « mise au point » de gardes, d'invariants ou de comportement compliqués, il est nécessaire de s'assurer que le modèle est vérifié, c'est-à-dire prouvé, avant de générer les simulateurs. Il est inutile de valider un modèle incorrect.

Le second point concerne la fragilité de Rodin devant les spécifications de grande taille. Les blocages de l'environnement ne sont pas rares. Il arrive, malheureusement, que les fichiers soient corrompus au point de nécessiter une re-création complète du raffinement.

Enfin, la longueur du texte de la spécification commence à poser des difficultés pratiques. Les vues proposées soit totalement structurelles, soit totalement aplaties, ne sont plus suffisantes pour une lecture et une édition efficaces.

5.3 La validation et JeB

La proportion du code programmé à la main est d'environ 13% du total (en nombre de lignes de JavaScript). C'est un peu plus important que dans les cas que nous avions traités précédemment : environ 4%. Il est possible d'analyser ces chiffres en fonction du découpage par domaine de la table 1.

La valuation des constantes correspond à une partie de code qu'il est possible de générer. Ceci nécessiterait de faire une analyse des valeurs pour détecter les cas fréquents tels que la création d'ensembles de symboles par exemple. Ce cas se répète plusieurs fois dans l'étude et le code JavaScript reproduit directement la spécification.

Le graphisme occupe une part importante. La leçon, peu originale, est qu'il y a besoin de bibliothèques de formes, d'images et d'interacteurs simples tels que des boutons pour faciliter la réalisation des interfaces. Un des objectifs de l'interface était d'étudier la possibilité de contrôler le déroulement d'une simulation à partir du graphisme. La faisabilité, avec un coût raisonnable, de cette idée confirme la pertinence du choix HTML/JavaScript pour les simulations.

L'émulation des capteurs est la partie la plus novatrice. Vis-à-vis de la sémantique du modèle, l'émulateur produit des valeurs de paramètres pour deux événements. L'affectation de valeurs aux paramètres étant une des principales causes de non-determinisme des simulations, JeB propose l'infrastructure pour créer et utiliser des fonctions de génération d'arguments. Dans les modèles précédemment traités avec JeB, les générateurs d'arguments étaient pour l'essentiel des algorithmes tirant aléatoirement des valeurs. Pour ce modèle, la démarche est différente : les valeurs proviennent d'un mécanisme déterministe qui est totalement extérieur au modèle.

La configuration simulateur et émulateur repose la question de la fidélité des simulations par rapport à la spécification. Lorsque l'émulateur est produit, comme ici, en regard de la spécification, il est légitime d'interroger la correction de l'implantation vis-à-vis du mécanisme réel. Lorsque l'émulateur provient d'une autre source, il est possible d'admettre sa conformité, mais il faudra l'interfacer avec le simulateur. Concrètement, une couche logicielle doit être développée. Cette couche a pour fonction d'abstraire la vue du comportement concret de l'émulateur au niveau abstrait attendu par le simulateur. L'interrogation porte alors sur la transparence de l'interface vis-à-vis des comportements.

6 Conclusion

Trois leçons se dégagent à ce stade du développement de l'étude de cas :

– Le choix de B Événementiel est raisonnable pour développer ce type de système, à condition d'avoir de bons outils de validation.

– JeB, malgré quelques défauts de jeunesse, confirme son potentiel pour construire rapidement des simulations fidèles.
– L'émulation et, plus généralement, la connexion de système concrets externes pour la validation de modèles abstraits posent des questions auxquelles la communauté devra apporter des réponses.

Le travail sur l'étude doit se poursuivre pour aborder deux questions. Les patrons d'introduction du temps « tiendront-ils la charge » dans une spécification de grande taille ? Quelle part de l'investissement mis dans la validation peut être conservée dans les raffinements à venir ?

Références

1. Abrial, J.R. : The B Book. Cambridge University Press (1996)
2. Abrial, J.R. : Modeling in Event-B : System and Software Engineering. Cambridge University Press (2010)
3. Abrial, J.R. : Formal methods in industry : achievements, problems, future. In : Proc. of the 28th int. conf. on Software engineering. pp. 761–768. ICSE '06, ACM, New York, USA (2006).
4. Bendisposto, J., Leuschel, M., Ligot, O., Samia, M. : La validation de modèles Event-B avec le plug-in ProB pour RODIN. Technique et Science Informatiques 27(8), 1065–1084 (2008)
5. Boniol, F., Wiels, V. : Landing gear system. http://www.irit.fr/ABZ2014/landing_system.pdf (2013), case-study for ABZ'2014
6. Hallerstede, S., Leuschel, M., Plagge, D. : Refinement-animation for event-b – towards a method of validation. In : Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) Abstract State Machines, Alloy, B and Z, LNCS, vol. 5977, pp. 287–301. Springer Berlin Heidelberg (2010),
7. Mashkoor, A., Jacquot, J.P. : Utilizing Event-B for Domain Engineering : A Critical Analysis. Requirements Engineering 16(3), 191–207 (2011)
8. Méry, D., Singh, N. : Automatic Code Generation from Event-B Models. In : Proc. Symposium on Information and Communication Technology. ACM, Hanoi, Vietnam (2011)
9. Métayer, C. : B model animator. Website (2012), http ://www.animb.org
10. Nakatani, T., Tsumaki, T., Tsuda, M., Inoki, M., Hori, S., Katamine, K. : Requirements Maturation Analysis by Accessibility and Stability. In : Software Engineering Conference (APSEC), 18th Asia Pacific. pp. 357–364 (2011)
11. Rehm, J. : Gestion du temps par le raffinement. Ph.D. thesis, Nancy-Université — Université Henri Poincaré (2009)
12. Wright, S. : Automatic Generation of C from Event-B. In : IM_FMT, Workshop on Integration of Model-based Formal Methods and Tools. Dusseldorf, Germany (2009)
13. Yang, F. : A Simulation Framework for the Validation of Event-B Specifications. Ph.D. thesis, Université de Lorraine (November 2013)
14. Yang, F., Jacquot, J.P., Souquières, J. : Proving the Fidelity of Simulations of Event-B Models. In : 15th IEEE International Symposium on High Assurance Systems Engineering (HASE), Miami, USA (2014)

Modélisation formelle d'IHM multi-modales en sortie avec B Événementiel

Linda Mohand-Oussaïd[1], Idir Aït-Sadoune[2], Yamine Aït-Ameur[3] and Mohamed Ahmed-Nacer [4]

CERIST, Algiers, Algeria, LIAS - ENSMA, Poitiers, France, mohandl@ensma.fr[1]
E3S - SUPELEC, Gif-Sur-Yvette, France, idir.aitsadoune@supelec.fr[2]
IRIT-ENSEEIHT, Toulouse, France, yamine@enseeiht.fr[3]
LSI - USTHB, Algiers, Algeria, anacer@cerist.dz[4]

Résumé Les interfaces homme-machine (IHM) multi-modales offrent à l'utilisateur la possibilité de combiner les modalités d'interaction afin d'augmenter la robustesse et l'utilisabilité de l'interface utilisateur d'un système. Plus particulièrement, en sortie, les IHM multi-modales permettent au système de restituer à l'utilisateur, l'information produite par le noyau fonctionnel en combinant sémantiquement plusieurs modalités. Dans l'optique de concevoir de telles interfaces pour des systèmes critiques, nous avons proposé un modèle formel de conception des interfaces multi-modales en sortie. Le modèle proposé se décompose en deux modèles : le modèle de fission sémantique qui décrit la décomposition de l'information à restituer en informations élémentaires, et le modèle d'allocation qui spécifie l'allocation des modalités et médias aux informations élémentaires. Nous avons également développé une formalisation B Événementiel détaillée des deux modèles : fission sémantique et allocation. Cet article est dédié à la présentation d'une démarche formelle et générique relative au modèle proposé, il décrit la démarche générale de développement B Événementiel ainsi que le modèle générique B Événementiel correspondant au modèle de conception générique.

Keywords: Interaction Homme Machine, multi-modalité en sortie, développement formel, B Événementiel

1 Introduction

Plusieurs modèles de conception ont été proposés pour maîtriser le processus de développement des IHM multi-modales. Lorsque ces dernières sont implantées dans des systèmes interactifs critiques, les approches de conception existantes (voir section 2) s'avèrent insuffisantes pour le développement des interfaces qui requièrent la même rigueur employée pour le développement du noyau fonctionnel. Notre travail s'intéresse à la conception formelle des IHM multi-modales en sortie. Il s'inspire des travaux de Rousseau [19] présentant un cadre de conception semi-formel. Dans [13], nous avons proposé un modèle formel pour la spécification des IHM multi-modales en sortie. Il spécifie la construction de l'interface multi-modale en sortie (présentation multi-modale) à partir de l'information générée par le noyau fonctionnel. D'abord, il modélise la fission sémantique de l'information en informations élémentaires, ensuite, il modélise l'allocation des modalités et des médias à ces informations élémentaires. Cette formalisation a été prolongée dans la méthode dans B Événementiel, donnant lieu au modèle B Événementiel de la fission sémantique dans [14] et au modèle B Événementiel de l'allocation

dans [15]. Des modèles B Événementiel développés sur des études de cas ont permis de construire un ensemble de modèles B Événementiel, liés par un raffinement qui formalisent le processus de fission sémantique et d'allocation. Les modèles ont ensuite été instanciés sur des études de cas spécifiques. Dans cet article, nous présentons une démarche générique et formelle basée sur l'utilisation de la méthode B Événementiel. Ainsi, après un survol des principaux modèles de conception existants en section 2, nous présentons dans la section 3 le modèle conceptuel que nous proposons. En section 4, nous présentons la formalisation du modèle conceptuel dans B Événementiel. Pour cela, nous introduisons la démarche de modélisation, suivie du processus de développement faisant intervenir les quatre modèles génériques développés. Enfin, nous concluons et donnons des perspectives à ce travail.

2 Travaux existants

Plusieurs travaux se sont intéressés à la conception des systèmes interactifs multimodaux, les approches proposées traitent aussi bien la multi-modalité en entrée que la multi-modalité en sortie. Parmi ces travaux, nous citons les approches orientées objet [10] et les approches ICO (Interactive Cooperative Objects) [17] et [16]. Des approches formelles ont été proposées pour la modélisation des IHM multi-modales, on cite les travaux basés sur : les réseaux de Petri de [2], la méthode Z de [9] , une combinaison de Z et de CSP dans [12], les machines à états dans [8] et la méthode B Événementiel [3]. Des travaux se sont également penchés sur la vérification des IHM multi-modales, comme les travaux basés sur le model checking dans [11], et sur le test avec Lutess dans [7]. De manière plus ciblée, la multi-modalité en sortie a fait l'objet de deux modèles semi-formels, le modèle SRM (Standard Reference Model) [6] orienté but où le but représente l'information à présenter à l'utilisateur, le modèle consiste à affiner le but en sous-but, à déterminer les attributs morphologiques et spatio-temporels des sous-but, puis à générer l'interface multi-modale en sortie (présentation) et à la distribuer sur les différents médias. Le modèle WWHT (What, When, How, Then) [19] construit la présentation multi-modale en s'adaptant au contexte d'interaction au moyen de quatre étapes : (1) la fission sémantique (What ?) correspond à une décomposition sémantique de l'information produite par le noyau fonctionnel en informations élémentaires à présenter à l'utilisateur, (2) l'allocation (When ?) consiste à sélectionner pour chaque unité d'information élémentaire, la présentation multimodale adaptée à l'état courant du contexte d'interaction, (3) l'instanciation (How ?) détermine selon l'état du contexte, les contenus lexico syntaxiques et des attributs morphologiques des modalités de la présentation et (4) l'évolution (Then ?) de la présentation multi-modale construite suivant le changement du contexte. Ces deux modèles modélisent le fonctionnement de l'interface multi-modale en sortie à partir de la génération de l'information par le système jusqu'à sa restitution à l'utilisateur. Cependant, ces modèles demeurent semi-formels ce qui limite leur utilisation pour la conception formelle d'un système interactif multi-modal en sortie. Afin d'y remédier, nous avons proposé un modèle formel générique dédié à la multi-modalité en sortie [13] basé sur le modèle WWHT.

3 Le modèle formel de conception des IHM multi-modale en sortie

Le modèle de conception (Figure 1) décrit formellement moyennant une syntaxe, une sémantique statique et dynamique, la construction de la présentation multi-modale

à présenter à l'utilisateur à partir de l'information générée par le noyau fonctionnel conformément aux choix du concepteur et du contexte d'interaction. Il se compose du modèle de fission sémantique et du modèle d'allocation.

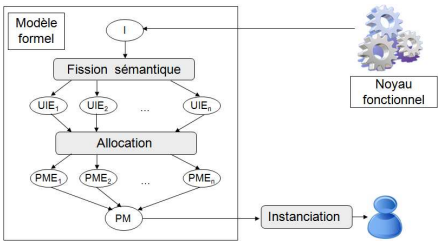


FIGURE 1. Le modèle formel de conception des IHM multi-modale en sortie

3.1 Modèle de fission sémantique

Il exprime la fission (décomposition) sémantique de l'information (*i*) générée par le noyau fonctionnel, en unités d'information élémentaires (*uie*) combinées au moyen d'opérateurs temporels et sémantiques suivant la règle syntaxique (1).

$$I ::= UIE \mid (op_{temp}, op_{sem})(I, I) \mid It(n, I) \; avec \; n \in \mathbb{N} \quad (1)$$

I représente l'ensemble des informations intervenant dans la construction de l'interface multi-modale en sortie, UIE l'ensemble des unités d'informations élémentaires, op_{temp} et op_{sem} les opérateurs temporel et sémantique combinant les informations et It l'opérateur exprimant l'itération sur une information. La sémantique statique décrit les propriétés statiques de l'information à fissionner, elle définit la fonction d'interprétation qui associe à chaque information son interprétation (sens) à travers un domaine sémantique et définit la durée de restitution de l'information par l'utilisation de bornes temporelles. La sémantique dynamique décrit les relations temporelles et sémantiques des informations fissionnées en utilisant la fonction d'interprétation sémantique et la fonction durée définies en sémantique statique. Une formalisation des différents cas de fission sémantique dans B Événementiel a été proposée dans [14].

3.2 Modèle d'allocation

Il formalise la construction de la présentation multi-modale (*pm*) relative à l'information (*i*). Il combine sémantiquement et temporellement les présentations multi-modales élémentaires (*pme*) correspondantes aux informations fissionnées (*uie*) suivant la règle syntaxique (2) en utilisant les règles de collage (3) et (4) qui relient les informations aux présentations qui les restituent. Les présentations (*pme*) sont ensuite décomposées en unités de présentations élémentaires (*upe*) selon la règle syntaxique (5), les unités de présentation (*upe*) ainsi obtenues sont allouées avec une modalité présentée sur un média (*mod*, *med*) par la règle syntaxique (6).

$$PM ::= PME \mid (op'_{temp}, op'_{sem})(PM, PM) \mid It'(n, PM) \; avec \; n \in \mathbb{N} \quad (2)$$

$$PM ::= ALL(I) \quad (3)$$

$$PME ::= ALL(UIE) \quad (4)$$

$$PME ::= UPE \mid compl(UPE, PME) \mid redun(UPE, PME) \mid choice(UPE, PME) \mid iter(n, PME) \text{ avec } n \in N \quad (5)$$

$$AFF\ (UPE) ::= (MOD, MED) \quad (6)$$

PM est l'ensemble des présentations multi-modales intervenant dans la construction de l'interface multi-modale en sortie, PME l'ensemble des présentations multi-modales élémentaires, UPE l'ensemble des unités de présentations élémentaires, op'_{temp} et op'_{sem} les opérateurs temporel et sémantique combinant les présentations et It' l'opérateur exprimant l'itération sur une présentation. ALL représente la fonction d'allocation, AFF la fonction d'affectation, *compl*, *redun*, *choice* et *iter* les opérateurs qui expriment respectivement les combinaisons complémentaire, redondante, par choix et itérative. Enfin, MOD est l'ensemble des modalités en sortie et MED l'ensemble des média en sortie. La sémantique statique décrit les propriétés statiques de la présentation durant l'allocation, elle décrit l'interprétation représentationnelle des présentations, la durée de leur restitution ainsi qu'un ensemble de propriétés définies sur le modèle syntaxique pour la description de la robustesse et de l'utilisabilité de l'interface. La sémantique dynamique décrit les relations temporelles et sémantiques entre les présentations combinées durant l'allocation moyennant les fonctions définies en sémantique statique. Une formalisation des différents cas d'allocation dans B Événementiel a été proposée dans [15].

3.3 Étude de cas

Nous illustrons le modèle formel que nous proposons par un scénario d'interaction multi-modale en sortie *CityMap* inspiré du système *SmartKom* [18]. Il est issu d'un dialogue entre l'utilisateur et *Smartakus,* un agent conversationnel composant de l'interface en sortie. L'interaction en sortie survient en réaction à une demande de l'utilisateur pour afficher la carte de la ville de Heidelberg. *Smartakus* y répond par synthèse vocale : "Here you can see the map of the city", la carte de la ville de Heidelberg est affichée en même temps. Aussi, nous considérons les ensembles suivants :

$$I = \{emptyI, infoCityMap, infoSee, infoMap\}$$
$$D = \{emptyD, CityMap, See, Map\}$$
$$UIE = \{infoSee, infoMap\}$$
$$PM = \{emptyP, presentCityMap, presentSee, presentMap, presentSeeSpeech, presentSeeExpression\}$$
$$PME = \{presentSee, presentMap\}$$
$$UPE = \{presentSeeSpeech, presentSeeExpression, presentMap\}$$
$$MOD = \{parole, expression, image\}$$
$$MED = \{haut-parleur, écran\}$$

Où : $CityMap$ = *voici la carte de la ville de Heidelberg* ; See = *voici la carte de la ville* ; Map = *carte de Heidelberg*

La réponse délivrée par *Smartkom* est restituée à l'utilisateur par l'interface multi-modale en sortie générée par le processus illustré en Figure. 2. Il se compose de :

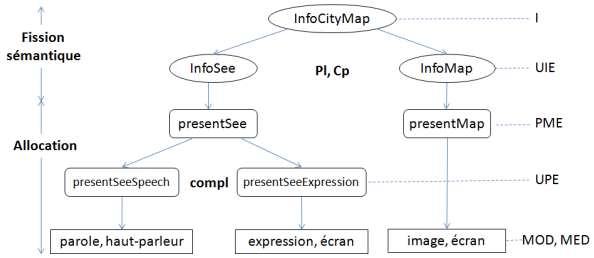


FIGURE 2. Processus de modélisation de *CityMap*

1. **La fission sémantique** de l'information $infoCityMap$ qui exprime "voici la carte de la ville de Heidelberg" en deux informations parallèles et complémentaires : $infoSee$ exprimant l'information "voici la carte de la ville " et $infoMap$ décrivant la carte de Heidelberg (7). La complémentarité signifie que lorsque des informations complémentaires sont considérées de manière disjointes, leur sémantique n'est pas significative. On écrit :

$$infoCityMap = (Pl, Cp)(infoSee, infoMap) \tag{7}$$

2. **L'allocation.** $presentCityMap$ qui restitue l'information $infoCityMap$ est obtenue en combinant parallèlement les présentations complémentaires $presentSee$ et $presentMap$ correspondant respectivement aux informations $infoSee$ et $infoMap$ (8). La présentation $presentSee$ est décomposée en deux présentations complémentaires $presentSeeSpeech$ et $presentSeeExpression$ (9). Lors de l'affectation, la présentation $presentSeeSpeech$ est produite par la parole sur le haut parleur (10) et $presentSeeExpression$ est produite par des expressions faciales sur l'écran (11). La combinaison complémentaire de $presentSeeSpeech$ et $presentSeeExpression$ reproduit le discours de *Smartakus*. La présentation $presentMap$ est produite par une image sur l'écran (12). On obtient :

$$presentCityMap = (Pl', Cp')(presentSee, presentMap) \tag{8}$$
$$presentSee = compl(presentSeeSpeech, presentSeeExpression) \tag{9}$$
$$AFF(presentSeeSpeech) = (parole, haut - parleur) \tag{10}$$
$$AFF(presentSeeExpression) = (expression, écran) \tag{11}$$
$$AFF(presentMap) = (image, écran) \tag{12}$$

4 Modélisation B Évènementiel des IHM multi-modales en sortie

La formalisation B Évènementiel des IHM multi-modales en sortie décrit les développements B Évènementiel à entreprendre pour modéliser une IHM multi-modale en sortie, elle prolonge la formalisation du modèle de conception proposé.

4.1 La méthode B Événementiel

B Événementiel [1] est une méthode formelle basée sur la logique du premier ordre et la théorie des ensembles. Un modèle B Événementiel décrit un système états-transitions où les variables représentent l'état du système et les évènements représentent les transitions d'un état à un autre. Le processus de modélisation de B Événementiel est incrémental. Il débute par le développement du modèle abstrait du système qui évolue progressivement vers un modèle concret par l'ajout de détails de conception à travers les étapes successives de raffinement. La description du modèle B Événementiel est accompagnée de la génération automatique d'obligations de preuves qui doivent être déchargées afin d'assurer la preuve de correction du modèle.

4.2 Démarche de modélisation

La démarche de formalisation B Évènementiel des IHM multi-modales en sortie formalise la construction de la présentation multi-modale qui restitue l'information à l'utilisateur. Elle repose sur le modèle de conception présenté en section 3 dont elle utilise les règles syntaxiques et elle exploite les mécanismes et éléments du langage de la méthode B Évènementiel comme suit :

1. **La modélisation progressive par raffinements successifs.** Elle permet de modéliser l'IHM multi-modale en sortie tout au long du processus de conception (voir Figure. 3). Nous utilisons le principe introduit dans [4], les auteurs formalisent les opérateurs d'une algèbre de processus dans la méthode B événementiel par raffinements en exploitant des règles BNF. Ainsi, la partie gauche d'une règle BNF est formalisée par un modèle abstrait et la partie droite de cette règle est formalisée par le modèle raffiné. Nous utilisons ce même principe pour le développement B Évènementiel de notre interface en s'appuyant sur les règles syntaxiques BNF définies dans le processus de conception de la section 3. L'information produite par le noyau fonctionnel est formalisée par le modèle abstrait de l'interface qui est raffiné pour chaque application d'une règle syntaxique de la fission sémantique ($n - 1$ raffinements). Le dernier raffinement de la fission sémantique est raffiné pour chaque application d'une règle syntaxique de l'allocation ($m - 1$ raffinements). Ainsi, le dernier raffinement de l'allocation formalise la présentation multi-modale obtenue.
2. **La dichotomie statique / dynamique.** La sémantique statique du modèle, qui exprime les propriétés statiques définies sur l'interface est formalisée par la partie CONTEXT du modèle B Évènementiel, par l'utilisation de la théorie des ensembles tandis que la sémantique dynamique exprimant les changements d'état de l'interface, est formalisée dans la partie MACHINE du modèle B Évènementiel par le biais des systèmes états/transitions.

5 Processus générique de développement B Évènementiel

Le processus générique de développement B Évènementiel de l'IHM multi-modale en sortie construit successivement plusieurs modèles B Évènementiel correspondant aux différentes étapes de transformation de l'IHM multi-modale en sortie. Les modèles ainsi obtenus sont : modèle abstrait, raffinements de la fission sémantique, raffinements de l'allocation. La construction de ces modèles est basée sur les règles syntaxiques

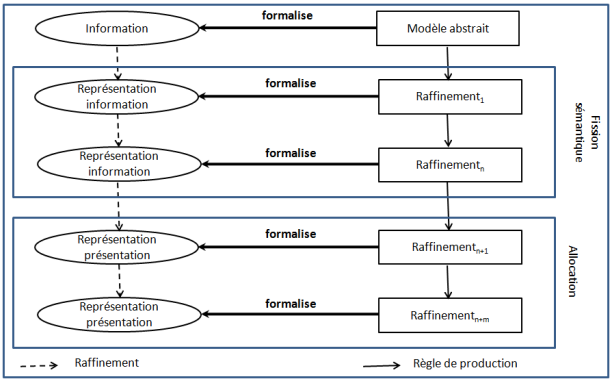


FIGURE 3. Processus de modélisation B Évènementiel de l'IHM multi-modale en sortie

développées dans la section 3. Ainsi, chaque règle syntaxique est formalisée par un modèle générique cadre à partir duquel sont dérivés les modèles spécifiques à chaque cas précis des règles syntaxiques qu'ils spécifient. Les modèles spécifiques à la fission sémantique et à l'allocation sont détaillés respectivement dans [14] et [15].

5.1 Modèle générique abstrait

Le modèle abstrait (voir Figure 4) modélise la production de l'information *i* par le noyau fonctionnel, il formalise la partie gauche de la règle syntaxique 1. Il contient :

1. **Le CONTEXT** $Informations$ qui décrit l'environnement statique de l'interface par la déclaration des ensembles : $Information$ qui regroupe les informations intervenant lors du processus de génération de l'interface, $InterpretationDomain$ qui définit les interprétations sémantiques de ces informations et $Presentation$ qui regroupe les présentations intervenant lors du processus de génération de l'interface. Il définit également les fonctions : $interpretation$ qui affecte à chaque information son interprétation sémantique et $allocation$ qui alloue à chaque information la présentation qui la restitue à l'utilisateur.
2. **La MACHINE** $Information$**.** Elle modélise la production de l'information à fissionner. Elle fait intervenir deux évènements en séquence : l'évènement $information$ qui produit l'information *i* à fissionner suivi de l'évènement $interpretation$ qui calcule *d* l'interprétation de l'information *i* ainsi que *p* la présentation qui la restitue à l'utilisateur. L'ordonnancement séquentiel des deux évènements $information$ et $interpretation$ est garanti par l'introduction du variant $varSeq$.

5.2 Modèle générique de la fission sémantique

Le modèle générique de la fission sémantique (voir Figure 5) décrit la production de l'information *i* par la production des informations fissionnées *i*1 et *i*2. Il formalise la partie gauche de la règle syntaxique 1 et contient :

1. **le CONTEXT** $SemanticFission$ qui étend le CONTEXT $Informations$ par l'introduction de l'opérateur sémantique générique $semanticOperator$ qui relie les informations fissionnées.

CONTEXT Informations
SETS
Information InterpretationDomain Presentation
CONSTANTS
interpretation allocation
AXIOMS
axm1 : $interpretation \in Information \rightarrow InterpretationDomain$
axm2 : $allocation \in Information \rightarrow Presentation$
END

MACHINE Information
SEES Informations
VARIABLES
i d p varSeq
INVARIANTS
inv1 : $i \in Information$
inv2 : $d \in InterpretationDomain$
inv3 : $p \in Presentation$
inv4 : $varSeq \in \{0,1\}$
VARIANT
varSeq
EVENTS
Initialisation
begin
act1 : $i :\in Information$
act2 : $d :\in InterpretationDomain$
act3 : $p :\in Presentation$
act4 : $varSeq := 1$
end

Event *information* $\widehat{=}$
Status convergent
any
x
where
grd1 : $x \in Information$
grd2 : $varSeq = 1$
then
act1 : $i := x$
act2 : $varSeq := varSeq - 1$
end
Event *interpretation* $\widehat{=}$
when
grd1 : $varSeq = 0$
then
act1 : $d := interpretation(i)$
act2 : $p := allocation(i)$
end
END

FIGURE 4. Le modèle générique abstrait

2. **la MACHINE** $FissionedInformation$ raffine la MACHINE $Information$ afin de produire l'information i en termes des informations fissionnées $i1$ et $i2$. En effet, l'évènement $information$ est raffiné par la production de deux informations supplémentaires $i1$ et $i2$, dont la combinaison sémantique des interprétations est égale à l'interprétation de i ($grd2$), deux nouveaux évènements $interpretation1$ et $interpretation2$ sont introduits, ils produisent respectivement $d1$ et $d2$ les interprétations sémantiques de $i1$ et $i2$. Enfin, l'évènement $interpretation$ est raffiné, il produit d au moyen de la combinaison sémantique de $d1$ et $d2$. L'ordonnancement temporel des deux évènements $interpretation1$ et $interpretation2$ n'étant pas précisé, il est assuré par l'introduction d'un variant générique var qui décroît dans les deux évènements et qui devient nul au déclenchement de l'évènement $interpretation$. Ce variant formalise l'opérateur temporel de la règle syntaxique qui définit la fission d'information en se basant sur les modèles définis dans [4]. L'invariant de collage est exprimé par les invariants : $inv2$ qui définit la relation entre les interprétations de i, $i1$ et $i2$ lors de leur production et $inv3$ qui détermine les valeurs de $d1$ et de $d2$ après l'enclenchement des évènements $interpretation1$ et $interpretation2$.

CONTEXT SemanticFission
EXTENDS Informations
CONSTANTS

semanticOperator

AXIOMS

axm1 : $semanticOperator \in InterpretationDomain \times InterpretationDomain \rightarrow InterpretationDomain$

END

MACHINE FissionedInformation
REFINES Information
SEES SemanticFission
VARIABLES

i d p varSeq var i1 i2 d1 d2

INVARIANTS

inv1 : $var \in \mathbb{N}$
inv2 : $varSeq < 1 \Rightarrow interpretation(i) = semanticOperator(interpretation(i1) \mapsto interpretation(i2))$
inv3 : $var = 0 \Rightarrow d1 = interpretation(i1) \wedge d2 = interpretation(i2)$ ...

VARIANT

var

EVENTS
Initialisation

begin

act1 : $var :\in \mathbb{N}_1$ ...

end

Event *information* $\widehat{=}$
Status convergent
refines *information*

any

$x, x1, x2$

where

grd1 : $x \in Information \wedge x1 \in Information \wedge x2 \in Information$
grd2 : $interpretation(x) = semanticOperator(interpretation(x1) \mapsto interpretation(x2))$
grd3 : $varSeq = 1$
grd4 : $var \in \mathbb{N}_1$

then

act1 : $i, i1, i2 := x, x1, x2$
act2 : $varSeq := varSeq - 1$

end

Event *interpretation* $\widehat{=}$
refines *interpretation*

when

grd1 : $varSeq = 0$
grd2 : $var = 0$

then

act1 : $d := semanticOperator(d1 \mapsto d2)$
act2 : $p := allocation(i)$

end

Event *interpretation1* $\widehat{=}$
Status convergent

when

grd1 : $varSeq = 0$
grd2 : $var \in \mathbb{N}_1$

then

act1 : $d1 := interpretation(i1)$
act2 : $var :| (var' \in \mathbb{N} \wedge var' < var)$

end

Event *interpretation2* $\widehat{=}$
Status convergent

when

grd1 : $varSeq = 0$
grd2 : $var \in \mathbb{N}_1$

then

act1 : $d2 := interpretation(i2)$
act2 : $var :| (var' \in \mathbb{N} \wedge var' < var)$

end

END

FIGURE 5. Le modèle générique de la fission sémantique

5.3 Modèle générique de l'allocation

Le modèle générique de l'allocation (voir Figure 6) raffine le modèle générique de la fission sémantique, il modélise la combinaison temporelle et/ou sémantique de deux présentations multi-modales pour constituer une présentation résultante. Il formalise, d'une part, la partie gauche de la règle syntaxique 2 pour décrire la combinaison temporelle et sémantique des présentations multi-modales élémentaires *pme* correspondant aux *uie*, pour reconstituer la présentation multi-modale *pm* correspondant à *i* en utilisant la règle de collage 4, la règle de collage 3 est employée depuis le modèle abstrait.

D'autre part, il formalise la partie gauche de la règle syntaxique 3 pour décrire la décomposition des *pme* en *upe*. Le modèle générique de l'allocation contient :

CONTEXT Allocation
EXTENDS SemanticFission
CONSTANTS
combinationOperator
AXIOMS
axm1 : $combinationOperator \in Presentation \times Presentation \rightarrow Presentation$
END

MACHINE Presentation
REFINES FissionedInformation
SEES Allocation
VARIABLES
i d p i1 i2 d1 d2 p1 p2 varSeq var
INVARIANTS
inv1 : $p1 \in Presentation$
inv2 : $p2 \in Presentation$
inv3 : $varSeq < 1 \Rightarrow allocation(i) = combinationOperator(allocation(i1) \mapsto allocation(i2))$
inv4 : $var = 0 \Rightarrow p1 = allocation(i1) \wedge p2 = allocation(i2)$
EVENTS
Initialisation
begin
act1 : $var :\in \mathbb{N}_1$ ...
end
Event *information* $\widehat{=}$
Status convergent
any
$x, x1, x2$
where
grd1 : $allocation(x) = combinationOperator(allocation(x1) \mapsto allocation(x2))$
...
then
act1 : ...
end

Event *presentation* $\widehat{=}$
refines *interpretation*
when
grd1 : $varSeq = 0$
grd2 : $var = 0$
then
act1 : $p :=$ $combinationOperator(p1 \mapsto p2)$
act2 : ...
end
Event *presentation1* $\widehat{=}$
Status convergent
when
grd1 : $varSeq = 0$
grd2 : $var \in \mathbb{N}_1$
then
act1 : $p1 := allocation(i1)$
act2 : ...
end
Event *presentation2* $\widehat{=}$
Status convergent
when
grd1 : $varSeq = 0$
grd2 : $var \in \mathbb{N}_1$
then
act1 : $p2 := allocation(i2)$
act2 : ...
end
END

FIGURE 6. Le modèle générique de l'allocation

1. **le CONTEXT** *Allocation* étend le CONTEXT $SemanticFission$ par l'introduction de l'opérateur de composition *combinationOperator* qui combine les présentations mutli-modales. Cet opérateur générique couvre aussi bien les opérateurs sémantiques op'_{sem} que les opérateurs *compl* et *redon*.
2. **la MACHINE** *Presentation* raffine la MACHINE $FissionedInformation$. Ainsi, elle raffine l'évènement $information$ par l'introduction d'une garde supplémentaire ($grd1$), qui exprime que la combinaison sémantique des présentations relatives aux informations fissionnées est égale à la présentation qui restitue *i*. Les évènements $interpretation1$ et $interpretation2$ sont raffinés par les évènements $presentation1$

et *presentation*2 qui calculent en plus des interprétations, les présentations $p1$ et $p2$ correspondantes respectivement à $i1$ et $i2$. L'ordonnancement temporel de *presentation*1 et *presentation*2 est garanti par le variant générique *var*, il formalise l'opérateur temporel de la règle syntaxique qui définit le modèle d'allocation en se basant sur les modèles définis dans [4] et assure que les présentations sont produites dans le même ordre temporel que les informations qu'elles restituent. L'évènement *interpretation* est raffiné par l'évènement *presentation* qui produit p au moyen de la combinaison de $p1$ et $p2$. L'invariant de collage est exprimé par les invariants : *inv*3 qui définit la relation entre les présentations relatives à i, $i1$ et $i2$ lors de leur production et *inv*4 qui détermine les valeurs de $p1$ et de $p2$ après le déclenchement des évènements *presentation*1 et *presentation*2.

5.4 Modèle générique de l'affectation

Le modèle générique de l'affectation (voir Figure 7) raffine le modèle générique de l'allocation, il modélise l'affectation des modalités et des médias aux *upe*. Il formalise la partie gauche de la règle syntaxique 6. Le modèle générique de l'affectation contient :

CONTEXT Affectation
EXTENDS Allocation
SETS
Modality Media
CONSTANTS
presentationUnit affectation linkModalityMedia
AXIOMS
axm1 : $presentationUnit \subseteq Presentation$
axm2 : $affectation \in presentationUnit \rightarrow Modality$
axm3 : $linkModalityMedia \in Modality \rightarrow \mathbb{P}(Media)$
END

MACHINE AffectedPresentation
REFINES Presentation
SEES Affectation
VARIABLES
item i i1 i2 d d1 d2 p p1 p2 varSeq var
INVARIANTS
inv1 : $item \in presentationUnit \rightarrow Modality \times Media$
EVENTS
Initialisation
begin
act1 : $item :\in presentationUnit \rightarrow Modality \times Media$
act2 : ...
end
Event *information* $\widehat{=}$
...
Event *presentation* $\widehat{=}$
...

Event *presentation1* $\widehat{=}$
...
Event *presentation2* $\widehat{=}$
...
Event *affectation1* $\widehat{=}$
any
m
where
grd1 : $p1 \in presentationUnit$
grd2 : $m \in linkModalityMedia(affectation(p1))$
grd3 : $var = 0$
then
act1 : $item(p1) := (affectation(p1) \mapsto m)$
end
Event *affectation2* $\widehat{=}$
...
END

FIGURE 7. Le modèle générique de l'affectation

1. **le CONTEXT** $Affectation$**.** Il étend le CONTEXT *Allocation* par l'introduction des ensembles : *Modality* pour les modalités, *Media* pour les médias et *PresentUnit* pour l'ensemble des unités de présentations multi-modales, il s'agit des présentations qui sont allouées avec un couple (*modality*, *media*). L'allocation des couples (*modality*, *media*) se fait de manière séparée : d'une part, la modalité est affectée à la présentation, ainsi, le CONTEXT $Affectation$ définit la fonction $affectation$ qui affecte une modalité à une présentation unitaire. D'autre part, un média est sélectionné dans l'ensemble des médias qui peuvent restituer la modalité pour la véhiculer. Par conséquent, la fonction *linkModalityMedia* est introduite, elle affecte à une modalité l'ensemble des médias qui peuvent la restituer.
2. **la MACHINE** $AffectedPresentation$**.** Elle raffine la MACHINE *Presentation* par l'introduction d'une variable de type fonction *item* qui affecte à une unité de présentation un couple (*modality*, *media*). La MACHINE $AffectedPresentation$ introduit deux nouveaux évènements $affectation1$ et $affectation2$ qui permettent d'affecter respectivement à $p1$ et $p2$, sous la condition qu'elles appartiennent à *presentationUnit* ($grd1$), les couples (*modality*, *media*) qui les restituent. La modalité est allouée à la présentation au moyen de la fonction $affectation$, le média, par contre, est sélectionné au moment de l'enclenchement des évènements $affectation1$ et $affectation2$ en utilisant la fonction *linkModalityMedia*. L'ordonnancement temporel des évènements $affectation1$ et $affectation2$ par rapport aux évènements $Presentation1$ et $Presentation2$ est assuré par la garde $grd3$ dans $affectation1$ et $affectation2$, elle assure que les évènements $affectation1$ et $affectation2$ ne sont enclenchés qu'une fois que les présentations $p1$ et $p2$ sont calculées respectivement dans $Presentation1$ et $Presentation2$.

Les modèles décrits ci-dessous ont généré 56 obligations de preuves (OP) dont 52 (voir Table 1) ont été prouvées de manière automatique par le prouveur de la plateforme *Rodin*, 4 ont nécessité l'intervention du concepteur dans une preuve interactive.

Composant	OP automatiques	OP manuelles	OP totales	OP non prouvées
Information	9	0	9	0
FissionedInformation	22	2	24	0
Presentation	13	2	15	0
AffectedPresentation	8	0	8	0
Total	52	4	56	0

Table 1. Les obligations de preuves

6 Application à l'étude de cas

Le processus de développement B Évènementiel de l'interface *CityMap* est obtenu par l'instanciation des ensembles du modèle générique par les ensembles définis dans la section 3.3 ainsi que la définition des valeurs des fonctions correspondant aux règles (7, 8, 9, 10, 11 et 12). Les modèles génériques sont instanciés par l'extension des

contextes génériques par des contextes spécifiques et par l'enrichissement des Machines génériques par des modèles d'opérateurs temporels définis dans [4] en utilisant le raffinement. A cause des limitations en nombre de pages, les modèles B Evénementiel ne sont pas présentés dans cet article. Néanmoins nous présentons ci-dessous le processus de développement B Évènementiel de l'interface *CityMap*, il consiste à :

1. Développer le modèle abstrait relatif à la partie gauche de la règle 7. Il se compose : (1) du contexte instancié $InformationCityMap$ (voir Figure 8) qui étend le contexte générique $Information$ par la définition par extension des ensembles qui constituent l'environnement de l'interface : $Information$, $interpretationDomain$ et $Presentation$ et des fonctions *allocation* et $interpretation$, (2) de la machine $CityMap$ qui correspond à la machine $Information$.
2. Développer le modèle raffiné de fission sémantique relatif à la partie droite de la règle 7. Il se compose : (1) du contexte instancié $ComplementaryFission$ qui étend le contexte abstrait $InformationCityMap$ ainsi que le contexte générique $SemanticFission$ en instanciant l'opérateur générique *semanticOperator* par l'opérateur spécifique *complementaryOperator*. (2) de la machine $ParallelCityMap$ qui enrichit la machine $FissionedInformation$ par le modèle de l'opérateur temporel qui permet de déclencher de manière parallèle les évènements $interpretation1$ et $interpretation2$ qui produisent respectivement See et Map, les interprétations relatives aux informations $infoSee$ et $infoMap$. Ils sont suivis de $interpretation$ qui produit $CityMap$ relative à l'information fissionnée $infoCityMap$.
3. Développer le modèle raffiné d'allocation relatif à la partie droite de la règle 8. Il se compose : (1) du contexte instancié $AllocationCityMap1$ qui étend le contexte précédent $ComplementaryFission$ mais également le contexte générique *Allocation* en instanciant l'opérateur générique *combinationOperator* par l'opérateur spécifique *PcomplementaryOperator*. (2) de la machine *CityMapPresentation* qui enrichit la machine $Presentation$ en précisant le modèle de l'opérateur temporal qui permet de déclencher en parallèle $presentation1$, $presentation2$ et *presentation* qui produisent respectivement les présentations correspondantes $presentSee$, $presentMap$ et $presentCityMap$ dans le même ordre temporel que les évènements $interpretation1$, $interpretation2$ et $interpretation$.
4. Développer le modèle raffiné d'allocation relatif à la partie droite de la règle 9. Il se compose : (1) du contexte instancié $AllocationCityMap2$ qui étend le contexte précédent $AllocationCityMap1$, et le contexte générique *Allocation* en instanciant l'opérateur *combinationOperator* par l'opérateur *PcomplementaryOperator*. (2) de la machine *ComplementaryCityMapPresentation* qui enrichit la machine générique *Presentation* en précisant le modèle de l'opérateur temporel qui permet de déclencher en parallèle les événements $presentation11$ et $presentation12$ qui produisent respectivement $presentSeeSpeech$ et $presentSeeExpression$. Ils sont suivis de l'évènement $presentation1$ qui produit $presentSee$ en combinant de manière complémentaire $presentSeeSpeech$ et $presentSeeExpression$.
5. Développer le modèle raffiné d'affectation relatif à la partie droite des règles 10, 11 et 12. Il se compose : (1) du contexte instancié $AffectationCityMap$ qui étend le contexte précédent $AllocationCityMap2$, et le contexte générique $Affectation$

par la définition des ensembles *presentationUnit*, *modality* et *media* et de la fonction $affectation$. (2) de la machine $AffectedCityMapPresentation$ qui enrichit la machine $AffectedPresentation$ en précisant le modèle de l'opérateur temporel qui permet de déclencher : $affectation1$, $affectation2$ et $affectation3$ affectant respectivement $presentSeeSpeech$, $presentSeeExpression$ et $presentMap$, une fois leur calcul opéré, dans les évènements $presentation11$, $presentation12$ et $presentation2$.

CONTEXT InformationCityMap
EXTENDS Informations
SETS
$Information = \{emptyI, infoCityMap, infoSee, infoMap\}$
$interpretationDomain = \{emptyD, CityMap, See, Map\}$
$Presentation = \{emptyP, presentCityMap, presentSee, presentMap\}$
AXIOMS
axm1 : $interpretation(infoCityMap) = CityMap$
axm2 : $allocation(infoCityMap) = presentCityMap$ ...
END

FIGURE 8. Le context $InformationCityMap$

Les différents enrichissements des machines génériques ont engendré des raffinements. Ainsi, l'instanciation de l'étude de cas selon le processus ci-dessus a généré 100 obligations de preuves dont 85 ont été prouvées de manière automatique et 15 ont nécessité l'intervention du concepteur dans une preuve interactive.

7 Conclusion

Cet article traite de la modélisation des IHM multi-modales, il étend des travaux antérieurs proposant un modèle formel pour la conception des IHM multi-modales en sortie pour lesquels il propose une formalisation B Évènementiel. Cette formalisation s'articule autour d'une démarche de modélisation qui précise l'enchaînement des développements B Évènementiel à effectuer et d'un ensemble de modèles génériques qui formalisent les différentes étapes de construction de l'interface. Ces modèles génériques représentent des modèles cadres à partir desquels les modèles B Évènementiel détaillés de la fission sémantique et de l'allocation sont dérivés. Nous avons montré au travers d'une étude de cas que ces modèles génériques peuvent être instanciés pour formaliser une interface concrète par la définition concrète des ensembles décrivant le contexte de l'interface et par la spécialisation des variants génériques en fonction de l'ordonnancement temporel intervenant dans la génération de l'interface. Ce travail se poursuit actuellement par le développement de raffinements spécifiques pour la vérification des propriétés d'utilisabilité et de robustesse de ces interfaces. Il est également envisagé d'automatiser le processus de génération des modèles B Évènementiel à partir d'une description de l'interface dans le modèle de conception proposé. Ces mécanismes automatisés permettrait d'aborder la modélisation d'interfaces plus complexes.

Références

1. J-R Abrial. *Modeling in Event-B : system and software engineering*, Cambridge University Press, 2010.

2. J. Accot, S. Chatty, and P. Palanque. A Formal Description of Low Level Interaction and its Application to Multimodal Interactive Systems, Design, Specification and Verification of Interactive Systems'96, 92-104, 1996.
3. Y. Ait-Ameur, I. Ait-Sadoune , M. Baron. Etude et comparaison de scénarios de développements formels d'interfaces multi-modales fondés sur la preuve et le raffinement. MOSIM'06 « Modélisation, Optimisation et Simulation des Systèmes : Défis et Opportunités », 2006.
4. Y. Ait-Ameur, M. Baron, N. Kamel and J. Mota. Encoding a process algebra using the Event-B method, International Journal on Software Tools for Technology Transfer, Volume 11(Number 3), 2009.
5. Y. Ait-Ameur, I. Ait-Sadoune, M. Baron and J. Mota. *Vérification et validation formelles de systèmes interactifs fondées sur la preuve : application aux systèmes Multi-Modaux*, Journal d'Interaction Personne-Système, vol. 1(1), 2010.
6. M. Bordegoni, G. Faconti, M.T. Maybury, T. Rist, S. Ruggieri, P. Trahanias and M. Wilson. *A Standard Reference Model for Intelligent Multi-media Presentation Systems*, Computer Standards and Interfaces 18 (6-7), 1997.
7. J. Bouchet, L. Madani, L. Nigay, C. Oriat, and I. Parissis. Formal testing of multimodal interactive systems. In Engineering Interactive Systems, 36-52, Springer Berlin Heidelberg., 2008.
8. M.L. Bourguet. Designing and prototyping multimodal commands. In INTERACT'03, pages 717-720, 2003.
9. D.J. Duke, M.D. Harrison. Mapping user requirements to implementations, Software Engineering Journal(Volume :10, Issue : 1), 13 - 20, 1997.
10. F. Flippo, A. Krebs and I. Marsic. A Framework for Rapid Development of Multimodal Interfaces, ICMI '03, 109–116, 2003.
11. N. Kamel, Y. Ait-Ameur. A Formal Model for CARE Usability Properties Verification in Multimodal HCI. A formal model for care usability properties verification in multimodal HCI. IEEE International Conference on Pervasive Services, 341-348, 2007.
12. I. Maccoll and D. Carrington. Testing MATIS : a case study on specification-based testing of interactive systems, In Formal Aspects of HCI (FAHCI98), 57-69, 1998.
13. L. Mohand-Oussaid, Y. Ait-Ameur, M. Ahmed-Nacer. *A generic formal model for fission of modalities in output multi-modal interactive systems*, VECOS'09, 2009.
14. L. Mohand-Oussaid, I. Ait-Sadoune, and Y. Ait-Ameur. *Modelling information fission in output multi-modal interactive systems using Event-B*, MEDI 2011, 2011, pp 200-213.
15. L. Mohand-Oussaid, I. Ait-Sadoune, Y. Ait-Ameur, M. Ahmed-Nacer. Formal modelling of output multi-modal HCI in Event-B. Modalities and media allocation. In : AAAI Symposium : modeling in human-machine systems : challenges for formal verification (2014).
16. D. Navarre and P. Palanque and R. Bastide and A. Schyn and M. Winckler and L.P. Nedel and C.M.D.S. Freitas. "A Formal Description of Multimodal Interaction Techniques for Immersive Virtual Reality Applications", INTERACT 2005, Lecture Notes in Computer Science, Springer Verlag, September, 25-28, 2005.
17. P. Palanque and A. Schyn. "A Model-based for Engineering Multimodal Interactive Systems ", 9th IFIP TC13 International Conference on Human Computer Interaction (Interact'2003).
18. N. Reithinger and J. Alexandersson and T. Becker and A. Blocher and R. Engel and M. Löckelt and J. Müller and N. Pfleger and P. Poller and M. Streit and V. Tschernomas. SmartKom : Adaptive and Flexible Multimodal Access to Multiple Applications, ICMI'03, 101-108, 2003.
19. C. Rousseau, Y. Bellik, and F. Vernier. WWHT : Un modèle conceptuel pour la présentation multimodale d'information. In IHM2005, 59-66, 2005.

Vers une approche de construction de virus pour cartes à puce basée sur la résolution de contraintes

Samiya Hamadouche [1,3], Mohamed Mezghiche[1], Arnaud Gotlieb[2]
Jean-Louis Lanet[3]

[1] Laboratoire LIMOSE, département d'informatique, Faculté des Sciences, Université M'Hamed Bougara, Boumerdès, Avenue de l'Indépendance, 35000 Algérie
{hamadouche-samiya, mohamed-mezghiche}@umbb.dz

[2]Certus V&V Center, SIMULA RESEARCH LAB., Lysaker, Norvège
arnaud@simula.no

[3]Université de Limoges, 123 rue Albert Thomas, 87000 Limoges, France,
jean-louis.lanet@unilim.fr

Résumé. En tant que supports sécurisés pour l'exécution d'applications et le stockage d'informations, les cartes à puce détiennent et manipulent des informations hautement sensibles. De ce fait, elles sont devenues la cible d'attaques, visant à contourner leurs mécanismes de sécurité afin de s'approprier des données sensibles qu'elles contiennent. Dans notre travail, nous nous intéressons spécifiquement aux *virus activables par attaque en faute*, c'est à dire aux programmes malveillants, pouvant être chargés dans la carte, sans être détectés par les mécanismes de sécurité et qui sont activés uniquement lorsqu'une faute est injectée. Notre objectif est, d'une part, de trouver une méthodologie pour construire de tels programmes, et d'autre part de développer les contre-mesures permettant de se prémunir contre ces virus. La difficulté de notre projet réside dans la construction d'un programme correct vis-à-vis de la spécification initiale, tant de par sa structure que de par sa sémantique, tout en respectant un ensemble précis de contraintes pour le choix de la séquence d'instructions à exécuter. Pour cette construction, nous adoptons une approche fondée sur la Programmation par Contraintes en développant un modèle de satisfaction de contraintes qui caractérise le comportement attendu. Ce papier présente les enjeux de notre projet et détaille les premiers éléments de notre approche.

1 Introduction

Les cartes à puce appartiennent à un domaine très sensible dans lequel l'apport des méthodes de développement formelles peut apporter des garanties, hors de portée des méthodes de développement traditionnelles. Un aspect particulier de la sécurité des cartes à puce concerne la sensibilité à de nombreuses attaques. Par exemple, dans le contexte de la sécurité des cartes Java Card, il a été récemment montré la possibilité de charger du code illégal, au sens où ce code ne respecte pas la sémantique de Java [9]. En principe, ce type de programmes ne doit pas franchir la phase initiale de l'opérateur en charge du déploiement des applications. En effet, ce dernier doit s'assurer que tout programme devant être chargé, respecte les règles de typage Java et de plus, que des règles ad-hoc de programmation sont respectées (e.g., non utilisation de certaines APIs, valeur interdite de certains paramètres, etc.). L'ensemble de ces vérifications font qu'il est, en principe, impossible de charger un code malveillant dans une carte à puce dans un contexte industriel. Cependant, une forme d'attaque dite *attaque combinée,* où l'attaquant modifie l'environnement de fonctionnement de la carte par une perturbation très ciblée, a récemment été mise au point [10]. Ce type

d'attaque permet de charger un code sain, mais capable très subtilement d'exécuter un code hostile, ce dernier étant, soit modifié de manière permanente après son chargement, soit de manière transitoire durant son exécution.

Jusqu'à présent, ces attaques utilisaient un code donné et tentaient de le muter de telle sorte à ne pas exécuter un test de sécurité. La problématique que nous adressons dans ce papier est relative à la conception d'un code malveillant embarqué dans un autre code sain et pouvant muter dynamiquement afin de réaliser des actions malveillantes. Autrement dit, peut-on concevoir un code malveillant caché dans un code sain et ne pouvant être activé que suite à une attaque ?

Notre papier est organisé de la manière suivante : dans la section ci-après (section 2) nous donnons un aperçu des modèles de faute ainsi que des attaques combinées. Ensuite, nous présentons un exemple de construction d'un virus activable par attaque en faute (section 3), qui constitue notre preuve de concept. Puis, nous évoquons les premiers éléments de notre approche (section 4) qui s'appuie sur la Programmation par Contraintes. Enfin, nous concluons par les travaux futurs (section 5).

2 Attaques combinées sur des cartes à puces

Les *attaques par perturbation* sont aujourd'hui considérées comme étant les plus puissantes [3]. Elles peuvent prendre la forme simple de perturbation sur l'alimentation, l'horloge, un ajout d'énergie par sonde électro magnétique ou par effet photo électrique. Ces attaques ont été utilisées essentiellement pour fauter le comportement d'algorithmes cryptographiques mais se sont répandues au système d'exploitation de la carte, à l'algorithme de chargement, à la machine virtuelle et même à l'application elle-même. Tous les éléments de la carte sont attaquables, en fonction de l'intérêt particulier de l'attaquant. L'attaque la mieux maîtrisée est l'attaque par impulsion laser, où les photons accèdent à la couche dopée du silicium et font basculer les transistors. Dès lors, les registres ou bien les cellules mémoire se saturent et peuvent prendre des valeurs précises. Ce type d'attaque est actuellement l'un des moyens les plus efficaces pour obtenir de l'information.

2.1 Modèle de fautes

De nombreux modèles de faute ont été introduits dans la littérature, comme en témoigne le Tableau1. Actuellement, le modèle *Precise Byte Error* est communément admis comme étant à la portée d'un attaquant. Il est donc capable de viser une seule cellule mémoire choisie, à un instant, synchronisé généralement avec le début de la commande. L'effet de l'ajout d'énergie sera la mise à un ou à zéro de tous les éléments de la cellule (bsr : bit set or reset) ou un résultat aléatoire si la mémoire est cryptée.

Tableau1. Les modèles de fautes existants

Modèle de faute	Précision	Position	Timing	Type de faute	Difficulté
Precise bit errors	Bit	full control	full control	bsr	++
Precise byte errors	Byte	full control	full control	bsr, random	+
Unknown byte errors	Byte	lose control	full control	bsr, random	-
Random errors	Variable	no control	partial control	random	--

Jusqu'à récemment on considérait qu'une seule faute pouvait être réalisée durant une commande. Auquel cas, une simple redondance temporelle permettait de détecter

une attaque. Avec l'usage des diodes laser, de multiples fautes synchronisées peuvent avoir lieu dans la même commande annihilant l'effet d'une telle redondance.

2.2 Attaques combinées

À partir de ce mécanisme primaire (saturation des cellules ou registres) l'effet de la faute se propage dans le système et se transforme en erreur, dès son activation. Barbu *et al.* ont proposé [1] un moyen de parvenir à utiliser une arithmétique de pointeur sur une plate-forme Java Card. Le principe de l'attaque repose sur la perturbation du code natif exécuté lors d'une conversion de type entre une instance d'une classe et une instance d'une autre classe n'appartenant pas à la même branche du treillis de types. Au final, l'attaquant pourra accéder au même objet avec deux références de type différent. Dans [2], les auteurs décrivent plusieurs attaques basées sur la perturbation de la pile d'opérande, en particulier, si la valeur est une variable booléenne précédent l'exécution d'un saut conditionnel. Bouffard *et al.* ont proposé dans [4] de tirer parti de la rupture du flot d'exécution à la fin d'une boucle *for* en mettant à profit le sens du déplacement d'un offset pouvant amener jusqu'à exécuter des opérandes. Dans sa thèse, Kauffmann-Tourkestanky [8] montre que la probabilité d'avoir un code désynchronisé long, *i.e.* un décalage dans le flot d'instructions interprétées avant de retrouver le flot initial, est faible. Mais son approche nommée *durée de vol* est basée sur une distribution uniforme des byte codes et sur la distribution équiprobable des arguments sur le domaine d'un octet. Il montre que la désynchronisation ne peut excéder 1,53 instruction et donc l'exploitation est faible. Cependant pour un attaquant, le code est choisi et ne suit donc pas une distribution uniforme comme nous le montrons dans la section suivante.

3 Conception expérimentale

3.1 Principe général

Notre objectif est de pouvoir cacher un code hostile dans un code sain tel que ce dernier ait une sémantique correcte, vis-à-vis de la machine virtuelle (*i.e.* considéré correct par le vérifieur de byte code), même après l'injection de la faute. Considérons un modèle de faute *Precise Byte Error* et une mémoire non cryptée. Lorsque la faute arrive, l'instruction stockée dans la case mémoire-cible se transforme en une instruction NOP (`0x00`) et son opérande devient potentiellement une instruction valide. Ainsi, pour cacher le code du virus, il faut insérer une/plusieurs instruction(s) (qui fera l'objet de l'injection de la faute) juste avant son début de telle sorte que certaines contraintes soient respectées afin de contourner les contre-mesures de la carte (elles ne détecteront pas le virus lors de l'exécution du programme).

3.2 Preuve de concept : exemple de virus

Nous avons dans un premier temps invalidé l'hypothèse de Kauffmann-Tourkestanky [8] en montrant qu'avec un code choisi la désynchronisation peut être longue amenant l'exécution d'un code exploitable.

Nous avons considéré une application dans laquelle le programme utilise une clé cryptographique initialisée lors de la phase de personnalisation et donc inconnue pour le concepteur de l'application. Le but du virus est d'envoyer cette clé en clair à l'extérieur de la carte. Le code java de l'exemple considéré est le suivant :

```
public void process(APDU apdu) {
     ... // variables locales
     byte[] apduBuffer= apdu.getBuffer();                                  B1
     if (selectingApplet()) {return;}
     byte readByte = (byte) apdu.setIncomingAndReceive();
-------------------------------------------------------------------------
     DES_keys.getKey(apduBuffer, (short) 0);                               B2
-------------------------------------------------------------------------
     apdu.setOutgoingAndSend((short) 0, DES_keys.getSize());               B3
}
```

Ce code peut être décomposé en trois blocs :

- B1 et B3 le code sain qui doit être exécuté.
- B2 la commande qui décrypte la clé et la dépose en clair dans le buffer de sortie de l'application. C'est le code à cacher et qui sera exécuté suite à l'injection de la faute uniquement.

Nous avons pu montrer dans [7] comment cacher manuellement un tel code. Nous avons commencé par la résolution statique des liens à l'extérieur de la carte afin de récupérer la référence de la méthode `getKey` (à travers l'attaque de Hamadouche *et al*, [6]). Après cela, une instruction supplémentaire a été insérée juste avant le code à cacher (on opère au niveau byte code). Le code final obtenu est un code valide, il vérifie les règles de typage d'où il peut être chargé dans la carte sans être détecté par le vérifieur de byte code. De plus, il satisfait les règles de programmation Java Card donc il n'est pas considéré comme étant dangereux. Après un tir laser modifiant le byte code exécuté, la transformation de la sémantique permet à l'attaquant de renvoyer la clé à l'extérieur de la carte.

4 Approche proposée

4.1 Éléments de modélisation

Cacher un code hostile dans un code sain revient à trouver, parmi l'ensemble des instructions possibles, une séquence d'instructions qui permette, à partir d'un état donné de la mémoire (*i.e.*, le début du code à cacher) de rejoindre un fragment de code sain comme illustré par la figure ci-dessous (Fig.1). Nous devons garantir que l'insertion de ces instructions se fasse en respectant les contraintes qui seront vérifiées ultérieurement par le vérifieur de byte code. Par exemple, la pile ne doit pas dépasser la valeur stockée dans le header de la méthode, elle ne doit pas créer de dépassement par le bas, le nombre de paramètres locaux est fixé, les types produits et consommés doivent être compatibles avec l'état courant de la pile, la pile doit être vide au début et à fin de l'exécution, *etc*.

L'objectif de notre approche est donc, à partir d'un état partiellement connu de la mémoire, représenté par la pile et les paramètres locaux, d'insérer des instructions et de recalculer l'état mémoire précédent afin de converger vers l'état mémoire à rejoindre. L'état connu est partiel puisqu'il résulte de l'exécution des instructions antérieures qui ont pu produire des effets sur la pile. La construction de la séquence doit résoudre deux problèmes : le choix d'une instruction parmi celles qui existent dans le langage Java Card et le calcul de l'état mémoire précédent cette instruction. L'objectif du choix de l'instruction est de se rapprocher de cet état mémoire tout en respectant les contraintes Java. La fonction de décision du choix de l'instruction à insérer doit être accompagnée d'un mécanisme de retour arrière si la séquence choisie ne permet pas de rejoindre l'état désiré.

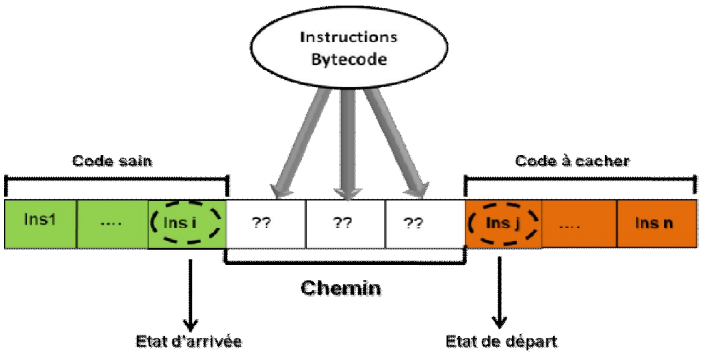


Fig. 1. Principe général de l'approche

4.2 Utilisation de la Programmation par Contraintes

La construction d'un code malveillant, qui sera chargé dans la carte à puce, activable par une attaque en faute se modélise ainsi comme un Problème de Satisfaction de Contraintes (CSP). Bien entendu, les contraintes envisagées sont d'une grande complexité, puisqu'il s'agit de considérer chacune des instructions byte code Java Card comme étant une relation entre deux états mémoire : l'état mémoire avant et l'état mémoire après l'exécution de l'instruction. Dans notre travail, nous avons pris le parti de réutiliser les travaux existants de Charreteur et Gotlieb [5] qui ont défini un modèle relationnel d'un sous-ensemble conséquent des instructions byte code Java dans le but de générer des données de test pour des programmes écrits dans ce langage. Ce modèle, écrit en Prolog avec contraintes, établit pour chaque instruction byte code, une relation entre deux états mémoire, nommées *EMC (Etat Mémoire sous Contrainte)*, et implémente des règles de déduction fonctionnant dans le sens de l'exécution, mais aussi dans le sens inverse. C'est très précisément cette capacité qui nous intéresse afin de calculer l'état mémoire précédant une instruction choisie lors de la construction de notre séquence.

4.3 Calcul des états mémoire

La création des EMCs nécessite la modélisation des données manipulées par la machine virtuelle Java et des zones de stockage de ces données à savoir les registres, la pile d'opérande et le tas. Un état mémoire est défini comme étant un triplet *(f, s, H)* où *f* est une fonction pour les registres, *s* la séquence des variables (de type primitif ou référence) qui modélisent la pile, et *H* une fonction représentant le tas.
L'effet de l'exécution de chaque instruction byte code est exprimé sous forme d'une relation entre deux états de la mémoire. La relation entre les états de la mémoire avant et après l'exécution d'une instruction se traduit notamment par des contraintes qui portent sur les éléments composant les EMCs.

La méthode de génération des données de test (états mémoire) proposée dans [5] raisonne dans le sens inverse à celui de l'exécution : partant d'un objectif (une instruction à couvrir) elle essaie de trouver un chemin menant vers le point d'entrée du programme en explorant progressivement et à l'envers le graphe de flot de contrôle, puis de déterminer une donnée d'entrée qui puisse activer un tel chemin. Au cours de la construction de ce dernier, les instructions parcourues sont modélisées par des contraintes. Pour ce faire, le modèle relationnel à contraintes est exploité et permet de raisonner automatiquement sur les EMCs. Ces derniers contiennent les variables sous contraintes du CSP et sont progressivement instanciées jusqu'à obtenir

un chemin final complet permettant d'atteindre l'instruction sélectionnée. Afin de mettre en application ce modèle et la méthode de génération, l'outil JAUT (Java Automatic Unit Testing) a été développé [5]. Il prend en entrée un fichier byte code traduit dans un langage intermédiaire, l'objectif de test (instruction à couvrir), ainsi que certains paramètres de génération (pour le parcours du graphe) et restitue des états mémoire en entrée permettant de couvrir l'instruction cible (objectif de test). La réutilisation de cette approche pour déterminer les instructions nécessaires à l'établissement de l'attaque en faute qui nous intéresse reste encore à valider, mais cette piste semble très prometteuse.

5 Conclusions et travaux futurs

Dans ce travail en cours, nous raisonnons sur la possibilité de cacher un code hostile dans un code sain et qui sera activé par une attaque en faute une fois chargé dans la carte à puce. Nous avons montré qu'on pouvait ramener ce problème à un problème de satisfaction de contraintes par la recherche automatique de séquence d'instructions. Notre approche s'appuie sur un modèle à contraintes qui caractérise chaque état mémoire par une structure partiellement connue dont la reconstruction fait l'objet du problème de satisfaction de contraintes. Cependant, le travail antérieur présenté dans [5] va être adapté et complété afin de prendre en compte les spécificités du langage Java Card (notre plateforme cible). De plus, cette approche reste à évaluer précisément, pour en démontrer complètement la faisabilité et l'efficacité. Néanmoins, nous pensons qu'elle est très prometteuse.

Références

1. Barbu, G., Thiebeauld, H., Guerin, V., *Attacks on Java Card 3.0 Combining Fault and Logical Attacks*, Gollmann D., Lanet J.-L., Iguchi-Cartigny J., Eds., Smart Card Research and Advanced Application, vol. 6035 de Lecture Notes in Computer Science, p. 148-163, Springer Berlin / Heidelberg, 2010.
2. Barbu, G., Duc, G., Hoogvorst, P., *Java Card Operand Stack : Fault Attacks, Combined Attacks and Countermeasures* , PROUFF E., Ed., Smart Card Research and Advanced Applications, vol. 7079 de Lecture Notes in Computer Science, p. 297-313, Springer, 2011.
3. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The Sorcerer's Apprentice Guide to Fault Attacks. *Proceedings of the IEEE, 94*(2), 370-382, Février 2006.
4. Bouffard, G., Lanet, J.-L., Iguchy-Cartigny, J., *Combined Software and Hardware Attacks on the Java Card Control Flow*, PROUFF E., Ed., Smart Card Research and Advanced Applications, vol. 7079 de Lecture Notes in Computer Science, Berlin Heidelberg, Springer, p. 283–296, September 2011.
5. Charreteur, F., Gotlieb. A.: *Constraint-based test input generation for java bytecode*. In Proc. of the 21st IEEE Int. Symp. on Softw. Reliability Engineering (ISSRE'10),San Jose, CA, USA, November 2010.
6. Hamadouche, S., Bouffard, G., Lanet, J.-L., Dorsemaine, B., Nouhant, B., Magloire, A., Reygnaud, A.: *Subverting Byte Code Linker service to characterize Java Card API*, S. SARSSI 2012, Cabourg, France, Mai 2012
7. Hamadouche, S., Lanet, J-L.: *Virus in a smart card: Myth or reality?*, Journal of Information Security and Applications, Septembre 2013.
8. Kauffmann-Tourkestansky, X. : *Analyses sécuritaires de code de carte à puce sous attaques physiques simulées*, Thèse de Doctorat, Université d'Orléans, 2012.
9. Oracle. Java Card 3.0.1 Specification, 2009.
10. Vetillard, E., Ferrari, A.: Combined Attacks and Countermeasures. In *Smart Card Research and Advanced Application*, vol. 6035 of *Lecture Notes in Computer Science*, pages 133–147. Springer, 2010.

Modélisation et validation formelle de systèmes globalement asynchrones et localement synchrones

Fatma Jebali[2,1,3], Mouna Tka[1,4], Christophe Deleuze[1,4],
Frédéric Lang[2,1,3], Radu Mateescu[2,1,3], Ioannis Parissis[1,4]

[1] Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France
[2] Inria
[3] CNRS, LIG, F-38000 Grenoble, France
[4] Univ. Grenoble Alpes, LCIS, F-26000 Valence, France

Résumé

Les automatismes industriels et domestiques sont fréquemment mis en oeuvre au moyen de contrôleurs logiques programmables (CLP), qui exécutent en mode synchrone des applications embarquées interagissant avec leur environnement. En combinant plusieurs CLP qui opèrent indépendamment et communiquent à travers un réseau, il est possible de réaliser des automatismes plus élaborés, de type GALS (Globally Asynchronous, Locally Synchronous). Pour assurer une conception correcte des systèmes GALS, qui est difficile à cause de la présence simultanée des aspects synchrones et asynchrones, nous proposons dans cet article une méthodologie rigoureuse, basée sur des méthodes formelles et des techniques de validation automatique (test et vérification) issues des paradigmes synchrone et asynchrone.

1 Introduction

La mise en oeuvre classique des automatismes industriels et domestiques repose sur l'utilisation de contrôleurs logiques programmables (CLP), qui exécutent en mode synchrone des applications embarquées interagissant avec leur environnement (capteurs et actionneurs). Des automatismes plus élaborés peuvent être réalisés au moyen de plusieurs CLP qui opèrent indépendamment et communiquent de manière asynchrone à travers un réseau. Ces systèmes, qui appartiennent à la classe des GALS (*Globally Asynchronous, Locally Synchronous*) [2], combinent des aspects synchrones et asynchrones, ce qui rend leur conception difficile à cause du non-déterminisme des communications.

Dans cet article, nous proposons une méthodologie de conception rigoureuse des systèmes GALS, à base de méthodes formelles et de techniques de validation

automatique issues des paradigmes synchrone et asynchrone. Au niveau synchrone, la validation des CLP individuels est effectuée par génération de tests destinés à couvrir le comportement des applications embarquées. Au niveau asynchrone, les réseaux de CLP sont modélisés au moyen d'un langage pivot dédié à la description des GALS, qui est traduit ensuite vers un langage formel asynchrone équipé d'outils de vérification par équivalences et logiques temporelles. Les deux types de validation sont interdépendants : les scénarios de tests décrits au niveau synchrone peuvent être rejoués sur le modèle asynchrone (projeté sur les CLP individuels), et des séquences peuvent être générées automatiquement à partir du modèle asynchrone afin d'alimenter la génération de tests au niveau synchrone.

2 Validation synchrone

Dans le cadre du test d'un système synchrone, nous souhaitons offrir la possibilité aux concepteurs de spécifier des tests de telle sorte que leur génération soit automatisée. De manière similaire à des approches antérieures sur le test synchrone [5, 6], un générateur de données de test est branché au système et, en s'appuyant sur les spécifications de test fournies, génère des valeurs pour les entrées du système, puis observe les sorties produites. En fonction de ces dernières, il procède à une nouvelle génération de valeurs d'entrée et ainsi de suite. Un oracle peut vérifier la conformité des sorties par rapport aux spécifications du système.

Nous proposons un nouveau langage, SPTL (*Synchronous Programs Testing Language*), pour spécifier les tests. SPTL permet de définir des modèles de test décrivant des *contraintes* sur le comportement de l'environnement et des *scénarios* de test conformes à ces contraintes. SPTL a pour objectif d'être utilisable par des non spécialistes du test. Il adopte le paradigme *flot de données*, comme beaucoup de langages de programmation d'automates. Les entrées et sorties du système sont représentées par des variables typées. Un ensemble de contraintes relient ces variables : elles définissent les valeurs possibles pour les variables d'entrée en fonction des valeurs passées des variables d'entrée et de sortie.

Une spécification SPTL contient des déclarations de variables, des contraintes réparties en *groupes* et *catégories* (que nous discutons plus loin) et des scénarios. La figure 1 montre un exemple de spécification pour un système de régulation de climatisation. Les contraintes peuvent utiliser les opérateurs arithmétiques et logiques, ainsi que deux opérateurs spécifiques :

- `prob(`e`,`p`)` est une expression dont la valeur est e avec la probabilité p, et une autre valeur (parmi les valeurs possibles pour le type) sinon,
- `pre(`v`)` est la valeur de la variable (ou plus généralement expression) v au cycle précédent. La valeur utilisée lors du premier cycle est donnée dans la déclaration de la variable (cas de `Tamb` ici).

Des sous-programmes (`CompTemp` dans la figure) permettent de factoriser des expressions complexes pouvant apparaître dans les contraintes.

```
var input bool OnOff;      // bouton marche arrêt
    input int Tamb = 10;   // température ambiante
    input int Tuser = 7;   // consigne de température
    output bool IsOn;      // souffle ?
    output int Tout;       // température de l'air soufflé

categ CountrySeason
{  group FranceSummer
    { 20 < Tamb ; Tamb < 44 ; Tuser = CompTemp(Tamb) }  // contraintes

   group TunisiaWinter { 6 <= Tamb ; Tamb <= 14 }

sp CompTemp(int Tamb) returns(int Tuser) { Tuser = pre(Tamb) - 3 }
}
scenario UserIsWarm
time t
begin
  { Tuser = 8; Tamb = 30; t.start }             // point de passage
                                                        // chemin
| [ Tamb = pre(Tamb) - (if t % 20s = 0 then 1 else 0) // - contrainte
                            ( Tamb = 8 ) ]            // - transition
end
```

FIGURE 1 – Exemple de spécification SPTL (extrait d'un système de climatisation)

Profil Un système peut être utilisé dans des environnements variés. Décrire l'environnement le plus général reviendrait à fortement sous-spécifier le test. SPTL introduit la notion de *profil*, qui permet de décliner l'environnement en plusieurs variantes. Pour le système de climatisation, on pourrait avoir un profil correspondant à une habitation individuelle l'hiver en Tunisie, un autre correspondant à un bâtiment public l'été en France, chacun exprimant des contraintes différentes. Pour simplifier le travail d'élaboration des profils, SPTL permet de découper l'environnement en un petit nombre de *catégories*. Pour chaque catégorie, on définit un ensemble de *groupes*, chacun contenant des contraintes déterminées par cet aspect de l'environnement. Dans notre exemple, on a défini une catégorie `CountrySeason` avec deux groupes `FranceSummer` et `TunisiaWinter`. Une autre catégorie pourrait être le lieu d'utilisation (batiment public ou résidence individuelle).

Scénario Un scénario permet de spécifier une évolution de l'environnement dans le temps, de façon à amener le système dans un état précis. Il peut par exemple représenter une séquence d'actions d'un utilisateur. Un scénario s'exécute dans le cadre d'un profil et spécifie un ensemble de contraintes supplémentaires qui varient au cours de l'exécution. Un scénario est une séquence comprenant deux types d'éléments : des points de passage et des chemins. Un point de passage indique un ensemble de contraintes qui doivent s'appliquer sur un cycle de l'exécution. Un chemin comprend un ensemble de contraintes et une condition de transition.

Les contraintes s'appliquent à chaque cycle jusqu'à ce que la condition de transition, calculée à la fin de chaque cycle, soit vraie. À ce moment le scénario passe à l'étape suivante. Les scénarios permettent aussi de prendre en compte l'écoulement du temps avec des variables de type `time`. Ces variables sont des chronomètres qui peuvent être déclenchés dans une étape ponctuelle, et utilisés dans contraintes. Ceci permet d'utiliser le temps dans le déroulement du scénario (dans notre exemple, `Tamb` est décrémentée toutes les 20 secondes).

3 Validation asynchrone

Nous proposons un langage textuel formel, nommé GRL (*GALS Representation Language*) [4], pour la description des systèmes GALS composés de plusieurs composants synchrones en interaction permanente avec leur environnement et communiquant via des médiums de communication asynchrones. Un programme GRL est structuré en plusieurs types d'entités :

- les *types*, prédéfinis (entiers, booléens, etc.) ou définis par l'utilisateur (énumérés, intervalles, structures)
- les *blocs* (figure 2), qui constituent les briques d'exécution synchrones
- les *médiums* (figure 3) et les *environnements*, qui modélisent respectivement le réseau et des contraintes physiques ou logiques sur les blocs, et dont l'exécution est guidée par l'interaction avec les blocs
- les *systèmes* (figure 4), qui décrivent l'assemblage et les interactions entre blocs, médiums et environnements
- les *modules*, qui structurent l'ensemble et permettent de construire des bibliothèques réutilisables

Pour les blocs synchrones, GRL s'inspire des diagrammes de flot de données. Chaque bloc est décrit sous la forme d'un programme déterministe qui combine les structures de contrôle classiques des langages algorithmiques (affectations, if-then-else, case, etc.) et les instanciations hiérarchiques de blocs. Ce programme déterministe définit une boucle synchrone dont l'exécution est atomique : (1) lecture des entrées (2) réception de données du réseau (3) calcul des sorties (code impératif déterministe) (4) envoi de données au réseau (5) écriture des sorties.

Chaque instance de bloc dispose de sa propre mémoire, constituée de variables temporaires (dont les valeurs sont locales à un cycle d'exécution de la boucle synchrone) ou permanentes (dont les valeurs, qui persistent entre les cycles d'exécution de la boucle synchrone, définissent l'état courant du bloc).

Par rapport aux blocs, les médiums et les environnements s'exécutent de manière "passive", c'est-à-dire que leur comportement est guidé par l'occurrence de signaux induits (implicitement) par la réception ou l'envoi de données par un bloc (médiums) ou par la lecture ou l'écriture d'une entrée/sortie (environnements). De plus, ce comportement est non-déterministe, ce qui permet de modéliser des comportements complexes au bon niveau d'abstraction.

```
block AileronSystem (in spo : bool; out cpo : nat)
                    {receive lock, up, down : bool; send apo : nat} is
  perm pos : nat := 0

  if not(lock) and spo then
      if up then pos := pos + 1 elsif down then pos := pos − 1 end if
  end if;
  cpo := pos; apo := pos
end block
```

FIGURE 2 – Exemple de bloc GRL (extrait d'un système de contrôle de vol)

```
medium AccessScheduler {receive apo : nat | send lock, up, down : bool |
            receive lp, up, dp : bool | send app : nat |
            receive ls, us, ds : bool | send aps:nat } is

  perm lock_buff : bool := true; up_buff, down_buff : bool := false, apo_buff : nat := 0
  select
      on lp, up, dp → lock_buff := lp; up_buff := up; down_buff := dp
  []
      on ls, us, ds → lock_buff := ls; up_buff := us; down_buff := ds
  []
      on apo → apo_buff := apo
  []
      on ?app → app := apo_buff
  []
      on ?aps → aps := apo_buff
  []
      on ?lock, ?up, ?down → lock := lock_buff; up := up_buff; down := down_buff
  end select
end medium
```

FIGURE 3 – Exemple de médium GRL (extrait d'un système de contrôle de vol)

Syntaxiquement, GRL est inspiré du langage LNT [1], lui-même issu de la lignée des algèbres de processus, étendu avec des données et les structures de contrôle classiques des langages de programmation algorithmiques. D'une part, ce choix nous offre expressivité et concision pour la description des aspects synchrones et asynchrones et, d'autre part, il permet d'envisager à moindre coût des passerelles entre les modèles écrits en GRL et les outils de vérification de systèmes concurrents asynchrones diffusés au sein de la boîte à outils CADP [3], qui utilisent LNT comme langage d'entrée.

4 Conclusion et travaux en cours

La méthodologie proposée pour modéliser et analyser formellement des systèmes GALS est destinée à être intégrée dans un flot de conception industriel des automatismes à base de réseaux de PLC. Le langage GRL vise à établir une passerelle entre les diagrammes de flot de données décrivant les applications em-

```
system FlightControlSystem (in op, os : nat, sp : bool; out alrm : bool) is

  allocate FBWComputer as Primary, FBWComputer as Secundary,
           AileronSystem as Aileron, FCDConcentrator as Concentrator,
           AileronControl[10] as Control, AccessScheduler as Scheduler

  temp tp : bool, app : int, lp, up, dp : bool, ts : bool, aps : int, ls, us, ds : bool,
       spo, cpo, apo : nat, lock, up, down : bool

  network -- blocs synchrones
      Primary (op, tp) {app; ?lp, ?up, ?dp},
      Secondary (os, ts) {aps; ?ls, ?us, ?ds},
      Aileron (spo; ?cpo) {lock, up, down; ?apo}
  constrainedby -- environnements
      Concentrator (sp | ?tp | safeSecundary | ?ts | ?alrm),
      Control (cpo | ?spo)
  connectedby -- mediums
      Scheduler {lp, up, dp | ?app | ls, us, ds | ?aps | apo | ?lock, ?up, ?down}

end system
```

FIGURE 4 – Exemple de système GRL (extrait d'un système de contrôle de vol)

barquées sur les PLC individuels et la description formelle du comportement des PLC connectés en réseau. Une première connexion avec des outils de validation formelle sera obtenue par traduction du langage GRL vers LNT. A plus long terme, les fonctionnalités de validation seront déployées comme services sur des nuages de calcul, afin de promouvoir une approche de conception rigoureuse d'automatismes dans l'internet des objets.

Références

[1] D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, C. McKinty, V. Powazny, F. Lang, W. Serwe, and G. Smeding. Reference manual of the LOTOS NT to LOTOS translator (version 5.4). INRIA/VASY, 149 pages, September 2011.

[2] D. M. Chapiro. *Globally-asynchronous Locally-synchronous Systems*. PhD thesis, Stanford, CA, USA, 1985.

[3] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2011 : A toolbox for the construction and analysis of distributed processes. *Springer Int. Journal on Software Tools for Technology Transfer (STTT)*, 15(2) :89–107, 2013.

[4] F. Jebali, F. Lang, and R. Mateescu. GRL : A specification language for globally asynchronous locally synchronous systems. Research Report RR-8527, Inria, 2014. `http://hal.inria.fr/hal-00983711`.

[5] L. Madani, V. Papailiopoulou, and I. Parissis. Towards a testing methodology for reactive systems : A case study of a landing gear controller. In *ICST*, pages 489–497, 2010.

[6] P. Raymond, X. Nicollin, N. Halbwachs, and D. Weber. Automatic testing of reactive systems. In *RTSS*, pages 200–209, 1998.

Refactoring Graph for Reference Architecture Design Process

Francisca Losavio, Oscar Ordaz
MoST, Escuela de Computación, Facultad de Ciencias, Universidad Central de Venezuela, Caracas
francislosavio@gmail.com, oscarordaz55@gmail.com

Nicole Levy
CEDRIC, CNAM, Paris, France
nicole.levy@cnam.fr

ABSTRACT
Reference Architecture (RA) is the main artefact shared by all products of a *Software Product Line (SPL)*; it covers commonality and variability of *SPL* products, and it is used as a template to produce new products in an industrial software production context. Responding to industrial practice, in previous works we have proposed a semi-automatic bottom-up *refactoring process* to build *RA* from existing products; their architectures are represented by a connected graph or *valid architectural configuration (P, R)*, where *P* and *R* represent components and connectors. In this paper, on one hand we formalize the existence of the *Refactoring Graph (RG)* automatically constructed for each product, as a main artefact of the process. On the other hand, this process is extended with a new sub-process to manually complete the *candidate architecture (CA)* obtained automatically, combining the *ISO/IEC 25010 standard* quality model and goal-oriented techniques to model also non-functional variability, which is still an open research issue in *SPL*. Finally, *RA* is manually constructed from the completed *CA* by generalizing the variants obtained. The complete extended refactoring process is applied to obtain *RA* for the *Integrated Healthcare Systems (IHS)* domain. The process is partially tool supported.

1. INTRODUCTION

A *Software Product Line (SPL)* is a set of software-intensive systems, sharing a common, managed set of features that satisfy the specific needs of a particular market segment or domain. These features are developed from a common set of core assets, which are reused in different products that form a family [1]. The *SPL* approach favours reusability and claims to decrease costs and time-to-market. The key issue in *SPL* development is the construction of a common architecture from which new products can be derived. A *Reference Architecture (RA)* is a generic architecture for high-level design of *SPL* products constructed in the *SPL Domain Engineering* phase [2]; it considers a *core* or set of components common to all products of the *SPL* family and a *variability model* to indicate and document how a variant component can be customized for each product [2]. *Software Architecture* is defined in [3] as "a collection of computational components - or simple components - together with a description of the interactions between these components, the connectors". In this work the bottom-up approach is followed to construct *RA*, because in practice many industrial organizations dispose only of isolated products. In this case existing similar products must be examined, using reverse engineering techniques, to identify commonalty and variants. The *RA* design is a complex process that is in general poorly described in the literature and left to incomplete case studies, and details of methods and approaches are difficult to follow because there are no design standards. Moreover, existing traditional architectural methods and evaluation techniques for single-systems are reengineered and not specifically designed for *SPL*. In the reactive approach architectural knowledge is recovered from existing products [4] [5]. An important question that commonly appears in the literature is what needs to be done to ensure a suitable choice of architecture for the *SPL* family of products. To answer this question, this work provides a bottom-up refactoring process [6] modelled by a graph structure *(RG)* that generates automatically an initial architecture or *Candidate Architecture (CA)* from the *RG* of each considered product, taking into account commonalty and functional and non-functional variability. The goal of this work is two-folded: on one hand the mathematical foundations of our refactoring process are improved with the justification of the existence of *RG*, that allows to find all possible ways to assemble one by one the architectural valid configurations of a product; this fact can be seen as an intrinsic property of the product architectural connectivity. On the other hand, the process is extended with a new sub-process to complete the first *CA* obtained automatically from all *RGs*, considering a better documented and justified variability modelling of functional and non-functional requirements by combining the *Softgoals Interdependency Graph (SIG)* [7] [8] and the *ISO/IEC 25010* quality model [9].

One of the problems with the *SIG* is the lack of standard notations and the handling of complexity of the graph configuration; however, even if automatic tools are available to support its graphical construction [10], not many guidelines are provided on how to make a "good" decomposition or refinement of non-functional requirements [11]. This work applies the extended refactoring process to a case study in the *Integrated Healthcare Systems (IHS)* domain, focusing on free-use software of the Open Source Foundation (OSI).

This paper is structured as follows, besides this introduction: the second section formalizes the graph model supporting the refactoring process. The third section is dedicated to present the case study in the context of the *IHS* domain and the

standards involved in our approach. The fourth section describes and applies the refactoring process to build the *RA* to the case study. The fifth section discusses the related works. Finally, conclusions and perspectives are presented.

2. FORMAL MODELING OF THE REFACTORING PROCESS

The product architecture is represented by a *connected graph* or *valid architectural configuration (P, R)*, where *P* and *R* represent components and connectors of the product. The *cardinality of P* is defined as the *order* of the configuration; to avoid trivial cases, *order* ≥ *2*. Note that the assets, which are configurations of order 1, constitute a trivial case. A component provides/requires interface to/from another component; we will neither consider other relation types among components, nor ports. Only the logical view of the architecture expressing static aspects will be considered in this work and it will be specified using UML 2.0 [12]. The model supporting our process is a graph or *Refactoring Graph*, denoted by *RG* [6]. Given a certain product, the nodes of *RG* are all connected or valid intermediate architectural configurations of the product, i.e., *connected induced sub-graph of (P, R)*, the nodes are distributed into levels, where L_i is referred as level *i* containing all intermediate configurations of order *i*. There are as many levels in *RG* as many components in the product; if *n* is the number of components, the last level L_n has only one node corresponding to the product architecture. *RG* is constructed by a bottom-up process from the last level. Each node of L_i, $i \geq 2$, originates as many nodes in L_{i-1}, as valid configurations exist from $C_{i,i-1}$ combinations. There is an *arc* between two nodes in consecutive levels, if one of them is obtained from the other by adding a new component. That is to say, all precedent nodes of L_i are placed in L_{i-1}. Given two nodes c_{i-1} *and* c_i, the arc represents the *transformation* to obtain c_i from c_{i-1}, by adding a new component to c_{i-1}. Figure 1 shows an example of *RG* for a product of four components. In *RG*, paths from the first to the last level represent different ways to assemble the product as a sequence of intermediate valid configurations; components are added one by one, preserving connectivity, to obtain the product from its different assets. The connectivity of *(P, R)* ensure the existence of *RG*, from the following facts:

- *If (P, R) is a valid configuration of order n, n ≥ 3, then (P, R) contains at least one intermediate valid configuration of order 2, of order 3,…, of order n, respectively.*
- *Let (P, R) be a valid configuration of order n, n≥3 and (Q, R') be a valid configuration of order i, 2≤i<n of (P, R). Then from (Q, R') it is possible to iteratively construct intermediate valid configurations of (P, R) of order s, i+1 ≤ s ≤ n, each one of them including (Q, R').*

Note that the fact of constructing *RG* from only valid configuration reduces the possible combinatorial explosion when products have a huge number of components.

The *Product Commonality (PC)* is a valid configuration of maximum order, common to each product.

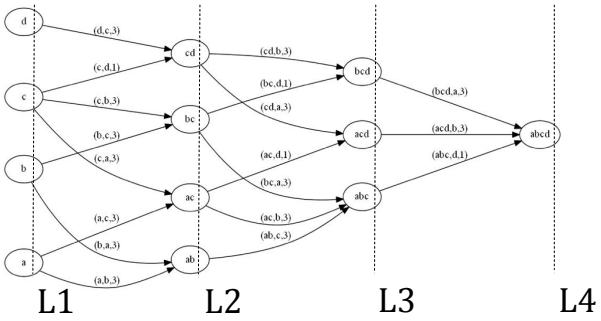


Figure 1. *RG for a product of four components, a, b, c, d, relations aRb, aRc, bRc, cRd, and levels L1 (assets), L2, L3, and L4 (whole product); the triplet on the arc has information on the transformation: names of component origin, of component added and the relevance of component.*

3. CASE STUDY: INTEGRATED HEALTHCARE SYSTEMS (IHS)

A clinical record or *Health Record (HR)* is the set of documents containing data, values and information on the situation and clinical evolution of a patient over the whole assistance process. Privacy, availability and persistency are priority quality requirements associated to systems supporting the management of these kinds of documents. *HR* evolves into the *Electronic Health Record (EHR)* with the introduction of information and communication technology. *HER* management is now the major functionality of *IHS*. A modern *IHS* makes sense only if interoperability of *EHR* among regional, national and international healthcare institutions can be guaranteed [13] [14]. The *IHS-RA* has to meet main quality requirements, namely, *security*, *reliability* (*availability-persistency)*, *portability* (*adaptability-scalability)*, *maintainability* (*modularity*, *modifiability)*, *efficiency* (*time behaviour*), and *functional suitability* (*correctness* or *"precision"*). The architecture is in general responsible for these non-functional requirements. However, *interoperability* of *EHR* depends on the standards used for the internal structure of the document and on the way it is transmitted through

the network. The main functionalities of *IHS* are: *management of EHR, queries of medical information for correct diagnosis, edition of orders for medical treatment, access and consult of patient demographic data*.

3.1 Standards used

In the *IHS* case the use of standards is mandatory to guarantee interoperability of medical data. Work has been going on from about fifteen years on standards, nevertheless the community has not yet reached an agreement [15]. In what follows, to abridge the presentation, we only mention those standards directly involved in the *IHS* products considered, and the standard used to specify software product quality.

HL7 (*Health Level 7) version 3 (2003)* messaging standard supports communication between the healthcare institution and medical records (*Health Care Records (HCR)* or *Electronic Medical Records (EMR)*) produced by different *IHS* [13]. **HL7 CDA** (*Clinical Document Architecture*), is proposed as an ANSI standard in 2005; it offers 4-layers: GUI, service, logic, and persistency. It contains specifications of message formats, electronic document structure and vocabulary for the health domain. It has been approved as an ISO standard [13]. **HXP** (*Healthcare eXchange Protocol*) is a standard data exchange protocol used for transparent communication among institutions, independently of the platform. It consists of an XML message format and a Procedure Call Dictionary (PCD) [16]. **OpenEHR**: *Open standard to create normalized HER with knowledge management issues.* It offers two separated models (dual models) to favour flexibility; it is now part of the ISO 13606 standard; the specification is written in UML [17]. **ISO/IEC 25010** provides a hierarchical model of eight high-level characteristics that are refined in multiple levels to define the product quality, until the measurable elements, called attributes, are reached. A software product can also be an intermediate artefact produced during the development process. In this work we will use this standard considering only product quality [9].

3.2 Description of three IHSs.

Many commercial and open source *IHS* are commonly available for main informatics platforms, however the majority are not compatible or do not handle *EHR*. Moreover, their complete adoption by healthcare staff finds cultural, social and organizational obstacles [18]. According to a study in [14], the open source systems *OpenEMR* and *PatientOS* [18] present a 90 and 92% usage respectively, according to 2010 data. Another open source system used in recent concrete national projects was *Care2x* [14]. The architectures of these three systems are shown in Figure 2; they were studied on the basis of the general available documentation. We observed that two of them are Web-based systems, under SOA (Service Oriented Architecture) [25]; all of them follow a client/server model for distribution and the classic layered style of information systems: Presentation Layer (**a**), Process or Business Logic Layer (**b**) and Data Layer (**c**), plus the Communication/transmission Layer (**d**) including all the networked information transport (**d1, d2**) (see Table 1); this layer ensures *portability (adaptability-scalability)* and *maintainability (modularity, modifiability)* with respect to Web Services, as well as the distribution to geographically distant locations by Internet, and locally within a healthcare institution by Intranet; *Security (authenticity, confidentiality* or*"privacy"* and *integrity*) of *HER* document transmission are also assured by **d**. The main differences are on how to solve *interoperability* (components **b4**, **c2** y **c3**) and *adaptability* with respect to different databases; the separated GUI (**a4-b5**) components ensure the global system *modularity* and *modifiability*, not limited to Web Services maintainability, offering greater *availability* because it does not depend on the availability of Web Servers. The dynamic Web-pages user interface (**a1**) common to two of the products, is of low cost and fast development, offering however less availability, for the above discussion. Data *persistency* and *integrity* are assured by the database (**c1**). The main *IHS* functionalities found are: - *the basic services for patient attention: - the EHR management, scheduling, demographic data, etc. through a "Patient Portal"* **(a2-b1, b2, b3)**, and - "*Reports"* for medical and administrative purposes (**a3-b3**).

4. REFACTORING PROCESS TO BUILD RA

It is composed by three main sub-processes: - *Specification of the products' architectures,* - *Construction of the Candidate Architecture,* and - *Construction of the Reference Architecture*. The process details, algorithms, etc. can be found in [6]. A prototype computational tool to build *RG* supports this process.

4.1 Specification of the products' architectures.

The three *IHS* (see Figure 2) considered have been described in Section 3.2. The components names were unified according to their semantic similarity w.r.t functionality.

4.2 Construction of the Candidate Architecture

1. RG construction

An *RG* for each product (see the example pf *RG* shown in Figure 1), OpenEMR, PatientOS and Care2X, is automatically built; the tool output is not shown here to abridge the presentation; a complete output on a different case study on the robotics industry domain is provided in [6]. Each *RG* contains 10 levels; the complete configuration of each product is

located on level 10, for each *RG*; the valid configuration *PC* generated by *{b1, b2, b3, c1}* is found on level 4; on level 1, the assets are located.

2. Automatic generation of the Candidate Architecture

A first *CA* is automatically produced from all *RGs*, considering the nearest level to *PC*, which involves the heuristic of containing at least one of the products' configurations to maintain the *SPL* architectural style; in this particular case it is level 10 containing all the products. *CA* constitutes an initial version of the *IHS-RA*; it captures the *PC* functional commonality and the *SPL* variability.

All components respect the connections they had in the original product are shown in Figure 3. Since in this case the initial *CA* configuration contains all the products' components, each product could be reconstructed. *CA* follows the 3-tiers style of the three products considered: *a. Presentation Layer*, *b. Process Layer*, and *c. Data Layer*, plus the *d. Transmission Layer*. Notice that two variants for the user interface are present in this valid configuration automatically generated; however a constraint must be placed on the fact that both solutions cannot run in the presentation layer at the same time. *CA* is now completed taking into account variability issues.

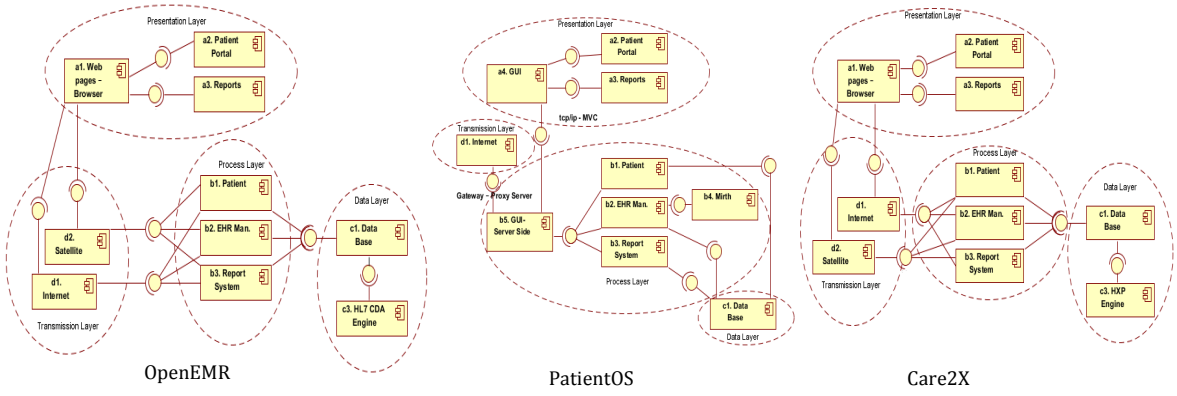


Figure 2. *Three products' architectures of the HIS domain*

3. Completion of the Candidate Architecture

- *Construction of the EQM*

The domain *Extended Quality Model (EQM)* (construction details can be seen in [6]) contains information on the component, the set of quality characteristics related to the component with assigned priority, and the possible architectural solutions for each quality characteristic. The information on quality requirements is extracted by the architect from his experience, catalogues, products' architectures, and domain documentation.

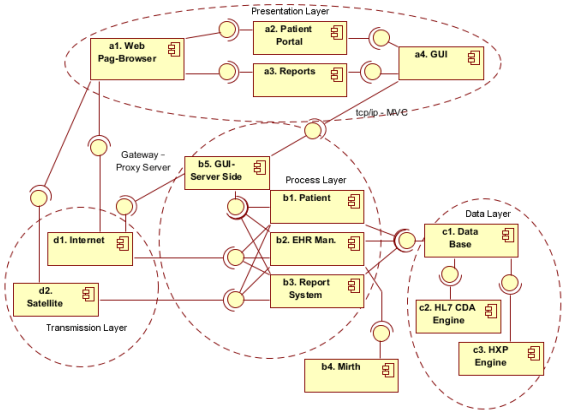


Figure 3. *The Candidate Architecture automatically generated*

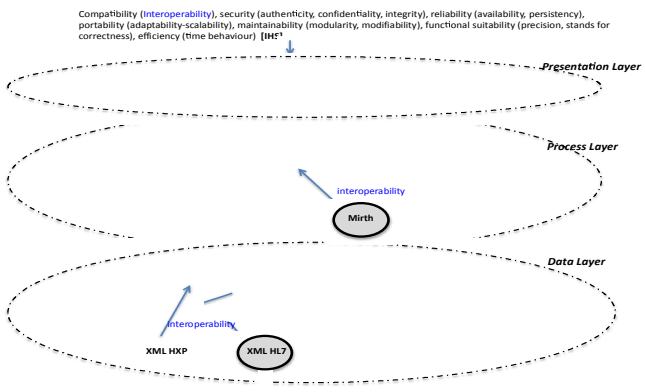


Figure 4. *SIG, example for interoperability*

- *Construction of the SIG*

EQM can be constructed in parallel with *SIG*; our *SIG* node is composed by characteristics of the quality model and their associated components: ***type*** (list of softgoals or quality characteristics) and ***[name]*** or *context of action* represents the architectural component affected by the listed quality characteristics. Many graphical notations have been proposed for

SIG; we choose to show together all the softgoals related to a certain context of action (component). Softgoals, specified by this quality model are refined into either new components or architectural solutions of *EQM*, which are represented by the SIG leaves (operationalizations) that can be variants to be "customized" into the different products of the *SPL*. *SIG* helps to locate variants and components that may eventually be missing for the heuristic used of including at least one of the products; since other components can appear during the refinement process to satisfy quality properties, these can be completely traced and evaluated on the *SIG*, to help corroborate or justify the architectural solutions extracted from the products' documentation by the architect. Layers have been marked in Figure 2, 3, 4, and 5 as ovals with dashed lines.

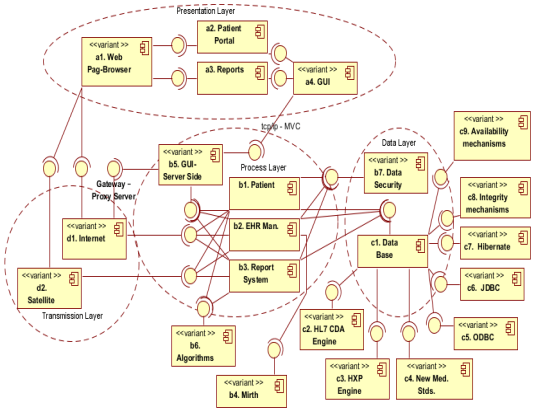


Figure 5. *The completed Candidate Architecture*

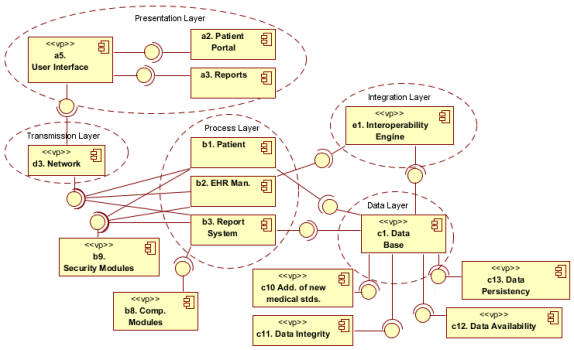


Figure 6. *IHS-RA*

1. Variability modelling

SIG is used to model variability; the terminology of Pohl et al. [2] is followed, where *variant* or *variability object* is the "instance" of the *variation point* or *variability subject*; a *context* can be associated to the variation point for documentation purposes. For example, "internet" and "satellite" are objects of the "network" variation point and its context could be the "information transmitted". The SIG operationalizations or architectural solutions for interoperability are obtained by refinement in Figure 4; three alternative solutions to translate HL7 are proposed: two engines, XML HXP and XML HL7 (marked with two slashes, OR) contribute to the DB solution and the Mirth system contributes to the HER Management solution; in this case, no new components are added in the refinement. After analysing AND/OR decompositions on the *SIG*, the final UML logic view of the *CA* architecture with variant components, signalled as UML stereotypes, is produced, see Figure 5. All these variants will be generalized into variation points to obtain *IHS-RA* as the final artefact.

4.3 Construction of the Reference Architecture

This is also a manual process based on the final *CA* configuration and on the architect expertise. On the *CA* variability model, variants are studied; the architect could provide other alternatives that could be added to enrich these variants and discuss more trade-offs. On the basis of this analysis, variant solutions are "generalized" into instantiable components, the variation points of the *IHS-RA*, see Figure 6.

5. RELATED WORKS

Refactoring techniques, used in bottom-up design have been traditionally used to reconstruct legacy code and reverse engineering to recover or reconstruct documentation [19]. Works in [20] [21] have similarity with our approach, however none of them use a graph model for the *RA* construction, nor the ISO/IEC 25010 standard to specify quality requirements. Many works are found in the context of functional variability modelling with feature models [10] for *SPL*. However, the variability modelling of non-functional requirements has not been worked as much [11] [22]; several of these works use the Goal-oriented approach combining *SIG* and feature models, namely, F-SIG (Feature-SIG) [23] and EFM (Extended Feature Model) [24]; we found only one method [10] that uses EFM, F-SIG and the specification of quality requirements by ISO/IEC 9126-1 (older version of ISO/IEC 25010); however, they follow a top-down approach and construct the feature model; in our bottom-up approach, features are captured from the refactoring of existing products and we do not need to construct the feature model. In the context of *SPL* and *RA* for *IHS* handling of *EHR*, work has been going on, considering SOA for data exchange and communication [25], [26], and focusing on model-driven architecture applied to the HL7 standard [27]. In [28] a general quality-driven *RA* architectural design process is proposed and applied to the *IHS* domain, not directly related to *SPL*. In view of the variety of *IHS* products and the absence of common adopted reference architectures, we propose the semi-automatic bottom-up approach based on the refactoring of existing products, described in this paper to build *HIS-RA*.

6. CONCLUSION

A semi-automatic graph modelled refactoring process to design a *RA* for *SPL* has been extended to handle non-functional variability modelling using ISO/IEC 25010 and the *SIG* approach. The graph model mathematical foundations have been justified. In case of a great number of products, the number of variant components may increase dramatically and the handling of combinatorial complexity by this automatic process considering valid configurations can be a great advantage. The process has been applied to the *IHS* domain with three refactored open-source *IHS* as input to the process. However, as in all architectural design methods, the architect's experience remains irreplaceable. A computational tool for the refactoring process is under construction, from a prototype built in [6]. A process for products derivation from the *RA*, inspired in [11] is an on-going work.

7. ACKNOWLEDGMENT

This work has received partial support from projects PEII DISoft 2011001343 of Fonacit, Venezuela and CDCH DARGRAF PG 03-8730-2013-1, and the Postgraduate Studies of Computer Science, Faculty of Science, Venezuela Central University.

REFERENCES

[1] P. Clements and L. Northrop. 2001. *Software product lines: practices and patterns*, 3rd edn. Readings, MA, Addison Wesley.

[2] K. Pohl, G. Böckle, F. van der Linden. 2005. *SPL engineering - foundations, principles, and techniques*. Springer IXXVI, 1-467

[3] M. Shaw, D. Garlan. 1996. *Software Architecture. Perspectives of an emerging discipline*, Prentice-Hall,

[4] A. Rashid, JC. Royer, A. Rummler (Eds). 2011. *Aspect-Oriented Model-Driven Software Product Lines. The AMPLE Way*. Chapter 1 and 2, Cambridge University Press, Cambridge.

[5] M. Matinlassi. 2004. Comparison of software product line architecture design Methods: COPA, FAST, FORM, KobrA and QADA, *Proc. of the 26th. Inter. Conference on Software Engineering (ICSE'04).*

[6] Losavio F. Ordaz O., Levy N., Baiotto. A. 2012. Graph Modelling of a Refactoring Process for Product Line Architecture Design, *Journée Lignes de Produits (JLDP), Lille, 47-58, 7-11 Novembre.*

[7] Chung L., Nixon B. and Yu E., Leite J, y Mylopoulos J. 2000. *NFR in Software Engineering.* Springer, Reading, Massachusetts.

[8] Supakkul S., Chung L., Integrating FRs and NFRs: A Use Case and Goal Driven Approach. 2004. *2nd ICSE, pp 30-37.*

[9] ISO/IEC 25010. 2011. SQuaRE -- System and software quality models, *ISO/IEC JTC1/SC7/WG6.*

[10] Gurses O. (2010). *Non-functional variability management by complementary quality modeling in a software product line*, MSc thesis, Graduate School of Natural and Applied Science of Middle East Technical University.

[11] Siegmund N., Rosenmuller M., Kuhlemann M., Kastner C., Apel S., Saake G. 2012. SPL Conqueror: Toword Optimization of Non-functional Properties in Software Product Line, *SQJ*, Vol. 20, No. 3-4, pp. 487-517(31).

[12] Object Management Group (OMG). 2005. Unified Modelling Language Superstructure, version 2.0 (formal/05-07-04), *August. www.omg.org/spec/UML/2.0.*

[13] Open Clinical Knowledge Management for Medical Care of EMR. 2011, http://www.openclinical.org/emr.html

[14] Samilovich S.. 2010. OpenEMR – Historia Clínica Electrónica de codigo abierto y distribuicón gratuita, apta para su uso en el sistema de salud Argentina, *JAIIO CAIS*. http://www.39jaiio.org.ar/sites/default/files/Programa_CAIS_39AIIO_v8.pdf

[15] De la Torre I., Castaño Y, Diaz Pernaz F., Diaz J., Antón M., Martínez M., Gonzalez D., Bota D., López F. 2010. *Categorización de los estándares de las HCE*. Universidad de Valladolid.
http://www.revistaesalud.com/index.php/revistaesalud/article/view/395/778

[16] LATORRILLA E., *Healthcare eXchange Protocol*, draft proposal 0.1, March 2004
http://books.google.co.ve/books?id=PrrUMPo5JzAC&printsec=frontcover&hl=es#v=onepage&q&f=false

[17] OpenEHR Foundation, Open domain-driven platform for developing e-health systems, http://www.openehr.org

[18] Holzinger A., Burgsteiner H., Maresch H., Experiences with the practical use of Care2x in Medical Informatics Education. 2009. *Medical Informatics: Concepts, Methodologies, Tools, and Applic.* http://www.igi.tugraz.at/harry/psfiles/MedInfoEducation.pdf

[19] M. Fowler. 1999. *Refactoring. Improving the design of existing code*. Addison-Wesley.

[20] Koziolek H., Weiss R., Doppelhamer J.. Evolving industrial software architectures into a software product line: A case study. 2009. *R. Mirandola, J. Gorton, and C. Hofmeister (Eds): QoSA 2009, LNCS 5581, pp. 177-193.*

[21] Y. Wu, Y. Yang, X. Peng, C. Qiu and W. Zhao. 2011. Recovering object-oriented framework for software product line reengineering. *12th Inter. Conf. on Software Reuse, ICSR'11, Pohang, South Korea, June 13-17. LNCS 6727, Springer, p. 119-134.*

[22] Perrouin G. 2011. A Metamodel-based Classification of Variability Modeling and Software Product Lines,
http://www.academia.edu/1021260/A_Metamodelbased_Classification_of_Variability_Modeling_Approaches.

[23] Jarzabek, S., Yang, B. y Yoeun, S. 2006. Addressing quality attributes in domain analysis for product lines. *IEEE Proceedings Software, 153(2), pp 61-736.*

[24] Benavides, D., Segura, S. and Ruiz, A. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. University of Seville.

[25] Cohen S., *SOA in Electronic Health Record Product Line*, 2009. SEI, Carnegie Mellon University, Pittsburg, USA, June.

[26] Raka D. *2010. A Product Line and Model Driven Approach for Interoperable EMR Messages Generation*. Master Report, Dep. of Computer Science, California State University, Fresno, USA, December.

[27] Pazos P., Aplicación de Estándares en la HCE. 2010. *JAIIO CAIS*.
http://www.slideshare.net/pablitox/aplicacion-de-estandares-en-sistemas-de-historia-clinica-electronica

[28] Levy N., Losavio F., Pollet Y. 2014. *Architectures logicielles: principes, techniques et outils.* Chap. 7. Architecture et qualité de systèmes logiciel, Hermes.

Certification de l'assemblage de composants dans le développement de logiciels critiques

Philippe Baufreton[1], Emmanuel Chailloux[2], Jean-Louis Dufour[1], Grégoire Henry[3]*, Pascal Manoury[3], Etienne Millon[2], Etienne Prun[4], Florian Thibord[3], Philippe Wang[2]†

[1] Sagem Défense Sécurité - [2] LIP6/UPMC - [3] PPS/Paris Diderot - [4] ClearSY

Résumé

La certification des logiciels des systèmes embarqués critiques est une opération industrielle très difficile et fastidieuse. Elle doit respecter un processus long, rigoureux et extrêmement normé. Le développement logiciel moderne repose sur la mise en œuvre de composants et de leur assemblage. Le problème à résoudre est celui de l'existence de méthodes et outils d'assemblage peu coûteux et industrialisables qui permettent de construire l'architecture du système final en induisant sa certification. À partir du contexte industriel avionique utlisant Simulink/Scade comme environnement de développement, on introduit une méthodologie innovante de certification de la composition s'appuyant sur le raffinement introduit par la méthode B.

1 Introduction

La norme DO178C de l'aviation civile [9], décrit les objectifs que l'ensemble des processus de spécification, conception, codage et vérification du logiciel doit satisfaire pour obtenir sa certification. Le développement logiciel moderne repose sur la mise en œuvre de composants et de leur assemblage. La norme de certification, en particulier dans son exigence de vérification reposant sur des tests, conduit concrètement à avoir à faire deux fois le même contrôle : une fois lors du développement des composants et une seconde fois lors de leur assemblage. Cela induit un coût. L'idée pour tenter de diminuer ce coût est de capitaliser la certification des composants. En conséquence, le problème à résoudre est celui de l'existence de méthodes et outils d'assemblages moins coûteux et industrialisables qui permettent de construire l'architecture du système final en induisant sa certification.

De telles voies ont déjà été explorées dans le cadre du développement des langages de programmation (typage, objets, modules, contrats) proposant des modèles originaux de conception ou de codage. Mais elles n'abordent pas le domaine du logiciel critique embarqué pour lequel l'exigence de certification est un préalable incontournable (normes de certification) qui repose principalement sur des tests. Obtenir que l'on dispose du moyen de vérifier l'assemblage de composants A et B qui ont été préalablement certifiés impose que l'on sache écrire correctement :

— les unités A et B en tant que «composants» (par opposition à une application complète) offrant certaines garanties de fonctionnement et possibilités d'assemblage ;

— la composition A-B en tant qu'assemblage préservant les garanties établies pour ses composants et réalisant ses propres exigences de fonctionnement.

Sur cette base, on peut espérer établir et la préservation de la correction des composants assemblés et la correction de l'assemblage par rapport à ses propres exigences par l'utilisation de *vérifications formelles*.

Pour résoudre ce problème, CERCLES^2 [1] propose d'intégrer la discipline de «programmation par contrats» à la démarche industrielle de construction de composants modélisés dans un langage graphique *à la Simulink*. Plus exactement, nous concentrons notre effort sur un fragment «logiciel» de Simulink homothétique du langage Scade pour lequel existent une sémantique et un processus de génération de

*Ce travail avait commencé quand l'auteur, maintenant chez ocamlpro, était à PPS/Paris-Diderot.

†Ce travail avait commencé quand l'auteur, maintenant à l'université de Cambridge, était au LIP6/UPMC.

1. CERtification Compositionnelle des Logiciels Embarqués critiques et Sûrs, projet ANR-10-SEGI-017. `http://www.algo-prog.info/cercles`

code. Pour cela, nous adjoignons aux définitions graphiques de Scade, des *annotations* (i.e. des *assertions*) donnant leurs pré, post-conditions et leurs invariants. Ces contraintes forment leurs *contrats*. Les définitions des composants Scade sont pris en charge par une *traducteur automatique* vers des composants de la méthode B, et celle-ci permet la vérification formelle, d'une part des propriétés d'implantation d'un composant et d'autre part de la correction de leurs assemblages dans d'autres composants.

Notre démarche se démarque ainsi, par exemple, de l'approche prise par l'utilisation de la chaîne FIACRE/TINA [4] orientée vers une vérification formelle à base de *model checking* : en effet, à cette technique que limite l'explosion combinatoire, nous avons préféré le choix d'une validation par *preuves formelles*. De plus les travaux de Lanoix Colin et Souquieres [7] ouvre l'utilisation de la méthode B pour ce type d'approche.

On présente en section 2 la méthodologie proposée en précisant la notion de composants, de leurs annotations et comment on se sert de ces dernières pour certifier leur assemblage. Le schéma de la figure 2 résume l'ensemble de la démarche. La section 3 présente les outils utilisés (extraction du source Scade, traduction de Scade vers B, vérification en B) en les illustrant par une application modélisant et implantant un gyroscope. La section 4 montre sur un exemple d'intégration les impacts de la vérification formelle dans le développement des composants. La dernière section conclut cet article en discutant de cette approche dans le cadre de logiciels critiques et en dressant les futurs travaux en vue d'industrialiser cette démarche.

2 Méthodologie

Le processus de développement classique suit grossièrement les étapes suivantes :

1. Fournir la spécification du logiciel, dite exigences de haut niveau (HLR pour *High Level Requirements*).
2. Préciser les exigences en termes de structures de données et procédures de calcul.
3. Les implanter dans un langage de programmation.
4. Vérifier que le programme exécutable respecte bien sa spécification.

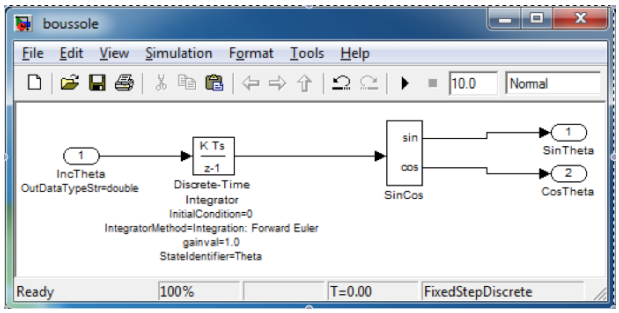
À l'étape 2 la capacité de réutilisation de composants peut réduire fortement la complexité de ce qu'il est nécessaire de préciser : de grandes parties de procédures de calcul ont pu déjà être implantées et l'approche *divide-and-conquer* s'adapte bien avec des parties déjà réalisées. La stratégie *top-down* peut alors se transformer en une stratégie *bottom-up* en agglutinant des composants pour réaliser une nouvelle fonctionnalité. Avec cette approche, nous devons être sûrs que la composition est correctement implantée. Avec le développement classique décrit ci-dessus, la vérification arrive tardivement dans le processus de développement. L'utilisation de vérifications formelles autorise une vérification précoce. Un processus de type Model Based Design est proche de nos besoins mais doit être complété.

Model Based Design (MBD) Dans l'approche conventionnelle avionique, MBD est utilisé pour le développement et la vérification est effectuée par des tests, des revues et des analyses (de code). Dans l'approche CERCLES^2, MBD est un moyen pour une conception basée sur réutilisation de composants existants enrichie d'un développement formel basé sur les contrats. Ceux-ci permettent que l'on puisse effectuer les vérifications formelles des propriétés du système au niveau de l'intégration des composants logiciels. Et c'est ainsi que l'on peut vérifier formellement la correction de la composition.

Langages synchrones à flots de données Le processus de développement de logiciels critiques avioniques a fortement évolué sur les dix dernières années pour la conception du code de contrôle (*transfer function*). En particulier l'utilisation de langages impératifs classiques (à la C ou Ada) a fortement chuté au profit de langages de modèles à flots de données. Le langage C n'est alors plus utilisé que comme langage intermédiaire entre les modèles à flots de données et la génération de code.

Les composants logiciels visés dans le cadre du projet CERCLES^2 sont issus de planches Simulink, logiciel de modélisation de systèmes physiques développé par MathWorks qui offre un langage de programmation graphique à flots de données. Les éléments de constructions utilisés dans les planches sont transposables dans le langage graphique Scade, développé par Esterel-Technologies, dont la version textuelle est LUSTRE (V3+), langage de programmation synchrone, déclaratif et par flots. Et, à la différence

de Simulink, Scade possède une sémantique formelle [8]. La figure 1 montre une planche Simulink et son équivalent textuel en Scade.

<table>
<tr><th>planche Simulink</th><th>code Scade de la planche</th></tr>
<tr><td> 
</td><td>

```
function compass(IncTheta : real)
  returns (SinTheta : real;
           CosTheta : real)
var
  _L1 : real;
  _L2 : real;
  _L3 : real;
  _L4 : real;
let
  _L1= IncTheta;
  _L2= #1 integrator(_L1);
  _L3, _L4= #1 SinCos(_L2);
  SinTheta= _L3;
  CosTheta= _L4;
tel
```

</td></tr>
<tr><td>

```
function SinCos(X : real)
  returns (SinX : real; CosX : real)
let
  assume A1 : X >= -3.14 and X <= 3.14;
  [...]
tel
```

</td><td>

```
node integrator(Inc : real)
  returns (Integr : real)
let
  [...]
tel
```

</td></tr>
</table>

FIGURE 1 – Gyroscope

Méthode B La méthode B est un système de développement logiciel introduit par J.R. Abrial [5] dans la continuation des travaux de C.A.R. Hoare, R. Floyd et E. Dijkstra. Elle est basée sur le raffinement de spécifications formelles des besoins vers une spécification implantable que l'on transcrit en code pour un langage cible. Chaque étape de raffinement donne lieu à des obligations de preuves. Leur validation par une démonstration formelle garantit l'adéquation du raffinement vis-à-vis de la spécification originale. Le principe de développement par raffinements de la méthode B donne aux modules logiciels ainsi développés une double nature :

- abstraite : le composant machine qui exprime la spécification des besoins.
- concrète : le raffinement du ou des composants qui ré-exprime la spécification en introduisant du détail et de l'implantation. Le dernier niveau est directement transposable vers un code cible.

Approche industrielle En supposant d'une part que les composants ont été développés à partir de modèles Simulink ou Scade, et que ceux-ci ont déjà été validés dans des contextes précédents, et en supposant d'autre part qu'en utilisant les assertions de Scade (`assume`/`guarantee`), les modèles embarquent leurs contrats, pour pouvoir les utiliser dans un processus formel basé sur les contrats nous avons besoin de leur expression formelle dans un environnement formel. Pour l'apport de ses obligations de preuves et les outils de preuves qu'il fournit, le projet CERCLES^2 s'appuie sur le cadre formel de la méthode B. La représentation d'un composant Simulink est traduite vers Scade dans le cycle actuel de développement. Il s'agit donc d'un sous ensemble de Simulink et Scade qui est utilisé ici. Le formalisme plus strict de Scade nous permet de construite une passerelle de Scade vers B qui traduit automatiquement les modèles Scade dans des modules B (voir figure 2).

Chaque planche (ou «nœud)» Scade est traduite en modules B. Il y a donc à ce niveau un contrôle de consistance et de bon raffinement des assume/guarantee. Le module traduisant la planche de composition oblige la preuve des préconditions des unités composées, c'est-à-dire, la vérification formelle du *control coupling* et du *data coupling* au sens de la norme DO-178C. L'avantage de cette approche est de faciliter la maintenance et l'évolution des applications logicielles ; les rendant plus modulaires. Le point principal pour la mise en œuvre de ces techniques est de maîtriser le couplage des composants.

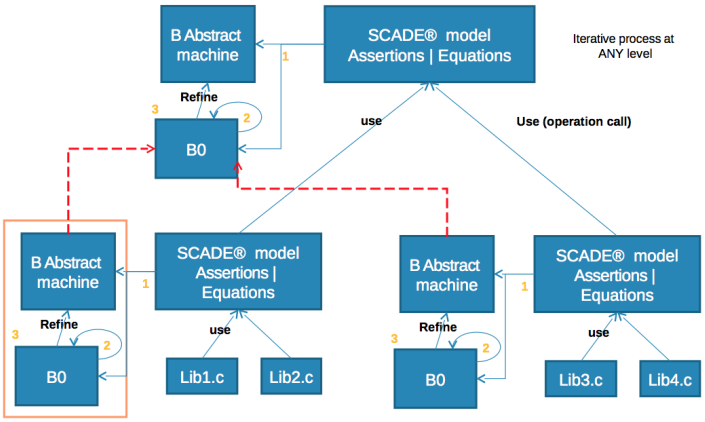


FIGURE 2 – Processus de vérification de la composition

3 La passerelle de Scade vers B

Scade On considère les programmes Scade syntaxiquement corrects, bien typés et dont la causalité a été établie. En l'état du projet CERCLES^2, les programmes obéissent de surcroît aux restrictions suivantes :

- pas de `when` explicite, ainsi, toutes les expressions sont synchronisées sur l'horloge de base ;
- l'utilisation de `fby` et de `->` est restreinte à celle permettant de traduire le bloc Simulink `1/z` ; c'est-à-dire à la construction `C -> (pre X)`, ou son équivalent `fby(C,1,X)`. Avec quelques restrictions supplémentaires sur l'initialisation `C`. Cette construction sera dénommée «registre» par la suite ;
- pas d'automate d'états ;
- les types admis sont `int`, `float` et `bool` et les vecteurs ou matrices de valeurs de tels types.

Ces restrictions proviennent en partie de la traduction automatique de Scade vers B mais aussi du contexte industriel et des restrictions liées à la transcription de Simulink vers Scade.

Principe général de traduction Chaque nœud Scade est traduit par un module B ayant un couple de composants B : une machine spécifiant le contrat et une machine réalisant son implantation (voir figure 3). Ces deux composants correspondent respectivement à la signature du nœud enrichie des annotations de pré et post-conditions et à sa définition en termes d'équations entre variables de flots. Chaque module ne déclare qu'une seule opération réalisant la fonctionnalité décrite par le nœud.

contrat

```
MACHINE M_compass

OPERATIONS
SinTheta, CosTheta <-- compass(IncTheta) =
 PRE
   IncTheta : REAL
 THEN
   CosTheta :: { ii | ii : REAL }||
   SinTheta :: { ii | ii : REAL }
 END
END
```

implantation

```
IMPLEMENTATION M_compass_i
REFINES M_compass
IMPORTS cS1.M_SinCos, ci1.M_integrator
OPERATIONS
SinTheta, CosTheta <-- compass(IncTheta) =
 VAR L1, L2, L3, L4 IN
   L1 := IncTheta;
   L2 <-- ci1.integrator(L1);
   L3, L4 <-- cS1.SinCos(L2);
   SinTheta := L3;
   CosTheta := L4
 END
END
```

FIGURE 3 – Couple de composants B (contrat et implantation) pour le Gyroscope (figure 1)

Machine B : le contrat Les sorties de l'opération sont celles du nœud. Les entrées du nœud sont soit celles de l'opération, soit un paramètre de la machine, dans le cas où elles servent à l'initialisation d'un «registre». La substitution qui définit l'opération est une substitution non déterministe qui donne *a minima* les types attendus des entrées et sorties et qui peut être enrichie par traduction des annotations de pré-conditions et post-conditions du nœud ; typiquement, des contraintes d'intervalle.

Implantation B L'implantation raffine le contrat. Elle contient autant de variables d'état qu'il y a de «registres» dans le nœud. La machine ne contient pas d'autre variable d'état. L'initialisation des «registres» est réalisée par la clause `INITIALISATION` de l'implantation. L'opération déclare autant de variables locales qu'il y a de fils reliant les divers opérateurs du nœud traduit ; à l'exception des fils de sortie des «registres» qui portent le nom de la variable d'état correspondante.

Le corps de l'opération est une séquence de substitutions simples et d'appels d'opérations (au sens de B) qui se termine par les substitutions de mise-à-jour des variables d'état. Les paramètres des opérations sont les noms de ses fils d'entrées, les variables affectées ont les noms des fils de sortie.

Pour obtenir un séquencement fidèle on considère le graphe de définition du nœud privé de ses «registres». Ce graphe est acyclique. On séquence les substitutions et les appels d'opérations selon un tri topologique de ce graphe ; et on achève la séquence par la mise à jour des «registres» : la variable d'état prend la valeur de son fil d'entrée.

Composant Dans l'approche faite ici on entend alors par composant le module B constitué d'une machine B et d'une implantation B. Un module B correspond donc a un nœud du langage synchrone à flots de données. De même qu'en langage source il est possible de réutiliser un nœud pour définir un nouveau logiciel, nous réutilisons les modules B pour vérifier nos nouveaux composants.

4 Exemple d'intégration simple avec vérification formelle

On reprend l'exemple du gyroscope de la figure 1 qui calcule une valeur d'état `Theta` et la passe à la fonction `SinCos`. La vérification formelle produit un échec à remplir les obligations de preuves (voir figure 4). En effet le composant `integrator` est insuffisamment spécifié, il n'y a aucune borne sur sa valeur de sortie alors que le composant `SinCos` attend une valeur comprise entre $-\pi$ et π.

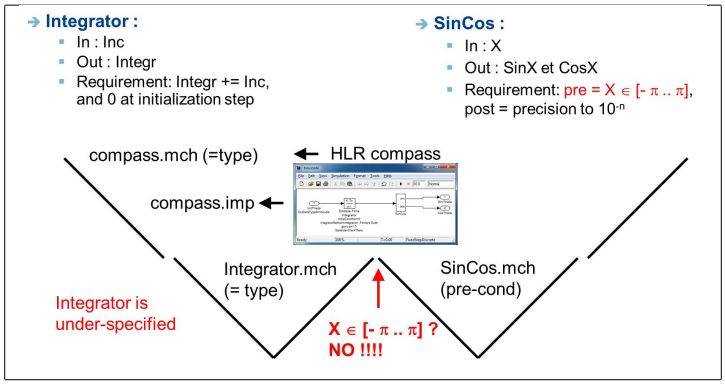


FIGURE 4 – Echec d'intégration

À ce point, deux interventions sont possibles. Soit modifier le code afin de remplir la partie du contrat non respectée : en changeant, par exemple, l'intégrateur pour un calcul plus précis effectuant un modulo 2π sur sa sortie. Soit modifier la conception de la composition pour répondre à la partie du contrat à respecter : ici, ajouter le modulo 2π au niveau du code de composition permet de respecter le contrat.

5 Conclusion

Cette expérience cherche à améliorer le processus de développement de logiciels critiques en introduisant une vérification formelle de la composition. Pour cela il est nécessaire d'introduire des contrats sur les composants écrits dans des langages à flots de données. Ces contrats sont vérifiés formellement en utilisant la méthode B tant au niveau des composants qu'au niveau de leurs compositions. Après une formalisation des contrats il est possible d'utiliser des composants ayant suivi un processus de qualification plus classique à base principale de tests.

Des planches Scade plus complexes, en particulier sur l'utilisation de «registres» et de plusieurs instances d'un composant, ont été traduites et automatiquement vérifiées [2] dans l'atelier B. On utilise ici le générateur d'obligation de preuve et le prouveur de l'AtelierB afin de suivre les «nouvelles» recommandations du *Formal Methods Supplement* [10] de la DO-178C. Les preuves générées actuellement dans le Générateur d'Obligations de Preuve (GOP) de l'atelier B (consistance, raffinement, garde...) couvrent le besoin actuel de composition. Toutefois il peut s'avérer utile une fois cette approche industrialisée d'implanter des nouvelles obligations de preuve spécifiques ce qui permettrait de couvrir plus de besoins de la norme.

Pour que notre approche puisse s'industrialiser il est nécessaire de pouvoir combiner des composants prouvés et des composants testés dans un processus de certification. On retrouve une telle démarche dans le projet Hi-lite [6] qui permet de prouver statiquement quand cela est possible et sinon de tester dynamiquement des propriétés décrites dans un langage d'annotation.

La méthode B est utilisée de longue date dans un cadre du développement de logiciels critiques, même si ce n'était pas dans l'avionique civile [3]. L'intégrer dans ce domaine d'activités, encore peu habitué aux méthodes formelles, demande de bien séparer la partie spécification des contrats que l'on intègre dans les planches Scade de la partie vérification qui se déroule dans un environnement B. En cas d'échec il convient alors de modifier l'implantation de la composition ou de préciser les annotations de la planche Scade. On ne change pas les environnements de développement industriel, même s'il est enrichi par les spécifications des contrats.

Il reste un point important à réaliser, même si le fragment Scade utilisé paraît simple, c'est de prouver l'équivalence d'un modèle Scade vers un modèle B. Une fois ce point réalisé il sera possible, de prétendre faire de la *preuve unitaire* comme le faisait [11] pour des vérifications unitaires de code C. En effet la méthodologie industrielle impose déjà l'utilisation d'un compilateur «qualifié» (au sens DO-178 comme KCG le générateur de code de Scade vers C) ou un compilateur prouvé (à la manière de [1]).

Références

[1] Auger, C. *Compilation certifiée de SCADE/LUSTRE.* Thèse, Université Paris Sud - Paris XI, Feb. 2013.

[2] Baufreton, P., Chailloux, E., Dufour, J.-L., Henry, G., Manoury, P., Prun, E., Thibord, F., and Wang, P. Compositional certification : the CERCLES2 project. In *ERTSS 2014 - Embedded Real-Time Software and Systems* (Feb. 2014), pp. 582–591.

[3] Behm, P., Benoit, P., Faivre, A., and Meynadier, J.-M. MéTéOR : A Successful Application of B in a Large Project. In *Proceedings of World Congress on Formal Methods (FM) - LNCS1709* (1999).

[4] Berthomieu, B., Bodeveix, J.-P., Dal Zilio, S., Dissaux, P., Filali, M., Gaufillet, P., Heim, S., and Vernadat, F. Formal Verification of AADL models with Fiacre and Tina. In *ERTSS 2010 - Embedded Real-Time Software and Systems* (May 2010).

[5] J-R. Abrial. *The B-book - assigning programs to meanings.* Cambridge University Press, 2005.

[6] Kanig, J., Guitton, J., and Moy, Y. Hi-Lite - Verification by Contract. *Softwaretechnik-Trends 31*, 3 (2011).

[7] Lanoix, A., Colin, S., and Souquières, J. Développement formel par composants : assemblage et vérification à l'aide de B. *Technique et Science Informatique 28* (2008).

[8] Raymond, P. *Compilation efficace d'un langage déclaratif synchrone : le générateur de code Lustre-V3.* Thèse, Institut National Polytechnique de Grenoble - INPG, Nov. 1991. 141 pages.

[9] RTCA SC205, and EUROCAE WG71. DO-178C/ED-12C – Software Considerations in Airborne Systems and Equipment Certification. *Radio Technical Commission for Aeronautics* (Dec. 2012).

[10] RTCA SC205, and EUROCAE WG71. DO-333/ED-216 – Formal Methods Supplement to DO-178C and DO-278A. *Radio Technical Commission for Aeronautics* (Dec. 2012).

[11] Sourys, J., Wiels, V., Delmas, D., and Delseny, H. Formal verification of avionics software products. In *FM'09 Proceedings of World Congress on Formal Methods* (2009), pp. 532–546.

Réflexions sur les liens possibles entre Argumentation et V&V pour le Logiciel

Thomas Polacsek

ONERA, Département Traitement de l'Information et Modélisation
2, avenue Edouard Belin BP74025, 31055 TOULOUSE Cedex 4

RÉSUMÉ. *Le but de cet article est d'ouvrir une piste de réflexion concernant la possibilité d'utiliser les travaux existants dans le domaine de l'argumentation et, plus précisément, dans l'argumentation légale, pour les appliquer dans le cadre de l'acceptation d'un logiciel par une autorité, que cela soit une autorité de certification ou un client. A terme, l'objectif de cette approche est de définir un cadre pour la construction d'une argumentation, que l'on pourrait qualifier d'acceptable, cadre qui structurerait l'ensemble des documents composant le dossier de Vérification et Validation (V&V).*

MOTS-CLÉS : *argumentation, V&V, VV&A*

1. Introduction

Dans le cadre de l'ingénierie de la simulation est apparue depuis une dizaine d'année le terme de VV&A pour *Verification, Validation and Accreditation*[1]. Ici, aux opérations de vérification et de validation vient s'ajouter une activité dite d'accréditation qui consiste à ce qu'une autorité accepte l'usage d'une simulation ou d'un modèle pour une utilisation dans un contexte précis[2]. Nous pouvons faire un parallèle entre cette activité dans le domaine de la simulation et la tâche consistant, pour une autorité, telle qu'un client ou une autorité de certification, à accepter un logiciel, qui peut correspondre à sa validation ou sa certification. Pour réaliser cela, il faut disposer d'une documentation exhaustive à ce sujet, expliquant non seulement les résultats, mais aussi les données d'entrées, les hypothèses faites, les techniques appliquées, etc. La tâche d'accréditation consiste donc à collecter cette documentation, mais surtout à l'évaluer. Cette documentation n'étant pas formelle, ou du moins pas dans son intégralité, il paraît vain de chercher à en établir uniquement de façon formelle sa validité.

Face à de tels énoncés, nous devons substituer à la notion de validité celle d'acceptabilité. Il n'est plus question ici de la vérité d'un énoncé mais d'étudier si un énoncé est acceptable ou pas. Nous allons donc chercher à modéliser des énoncés (la documentation) que des logiciens jugent non scientifiques. D'ailleurs, (Carnap, 1962) qualifie de tels énoncés de préscientifiques, il les considère comme vagues et incorrects. En ce sens, nous pouvons opérer un rapprochement avec les travaux de (Hamblin, 1970) qui remet en question l'utilisation de la logique formelle face à l'étude de l'argumentation. Son but étant de comprendre ce qui

1. Définition donnée par le Département de la Défense des États-Unis, on trouve aussi dans la littérature le terme d'*acceptation* (*acceptance*).
2. DoD directive 5000.59 : *"the official certification that a model or simulation is acceptable for a specific purpose"*.

rend une argumentation acceptable, il donne un nouveau modèle de la validité d'un raisonnement où la validité ne dépend pas de critères logiques relatifs à la vérité des prémisses mais à des critères dialectiques. Pour lui, la relation entre les prémisses et la conclusion n'est plus de l'ordre de l'implication logique mais d'une dialectique qui autorise, ou interdit, des comportements discursifs. Notons que l'auteur y expose sa préférence pour le terme acceptabilité plutôt que celui de validité. Au même moment, les travaux de (Perelman et Olbrechts-Tyteca, 2008) définissent une nouvelle théorie de l'argumentation basée sur une approche dialectique. Pour eux, les logiciens n'admettent comme rationalité que la démonstration logique. Dès lors, il devient impossible d'établir des raisonnements autres que purement formels ce qui est *"une limitation indue et parfaitement injustifiée du domaine où intervient notre faculté de raisonner et de prouver"*.

L'étude de l'argumentation s'intéresse aux liens entre hypothèses et conclusions, à la structuration du raisonnement. La notion d'argumentation renvoie bien évidemment à la notion de preuve qui a largement évoluée dans l'histoire des sciences et qui ne revêt pas le même sens suivant que l'on se trouve dans les disciplines formelles et axiomatiques, les sciences expérimentales ou les sciences humaines et sociales. Aujourd'hui, l'étude de la validité d'une argumentation et des mécanismes sous-jacents est étudiée par un ensemble de disciplines telles que : l'informatique, la linguistique, l'épistémologie et les sciences légales.

Le but de cet article est donc d'ouvrir une piste de réflexion concernant la possibilité d'utiliser les travaux existants dans le domaine de l'argumentation pour les appliquer dans le cadre de l'acceptation d'un logiciel par une autorité. Notons que l'idée de structurer l'ensemble des éléments servant à établir un fait sous la forme d'un arbre d'argumentation semble aujourd'hui faire son chemin, notamment au travers de ce qui se nomme *Assurance Case*. De plus, un groupe au sein de l'Object Management Group[3] cherche à définir un métamodel de l'argumentation. Cependant, il nous semble primordiale que tous les travaux cherchant à modéliser une argumentation ne se bornent pas à définir des diagrammes de boites et de flèches, mais s'inscrivent plutôt dans la lignée des travaux menés en linguistique et en droit.

2. Schéma de Toulmin

C'est en 1958 que (Toulmin, 2003) présente son schéma de l'argumentation. Ce modèle, bien que quasiment inexistant dans la littérature scientifique française, est enseigné dans de nombreuses universités américaines afin d'expliquer les mécanismes de l'argumentation. Il sert, par exemple, dans l'enseignement et la compréhension de disciplines telles que l'argumentation légale ou encore, dans ce que les anglophones nomment *critical thinking*.

Dans le modèle de Toulmin, toute argumentation est composée d'une *conclusion*, notée (C) sur la Figure 1, et des *données* (D). Pour justifier le passage des données à la conclusion, on utilise des données supplémentaires appelées *garanties* (W). Distinguer données et garanties n'est pas toujours chose aisée, les garanties sont générales, elles attestent de la solidité de l'argumentation. Une conclusion n'étant pas toujours absolue, Toulmin ajoute la possibilité d'exprimer des réserves à l'aide de *qualificateurs modaux* (Q). Ces qualificateurs correspondent à des notions telles que possiblement, probablement. A cela se rajoute des *conditions de réfutation* (R) qui expriment les circonstances dans lesquelles la conclusion n'est pas vraie. Pour finir, Toulmin ajoute les *fondements* (B) qui sont les justifications des garanties.

A titre d'exemple considérons le cas suivant : nous avons (C) *"des données qui sont cryptées avec OpenSSL et une clé de 128 bits"* et nous savons, en nous appuyant sur l'Advanced Encryption Standard,

3. http ://www.omg.org/spec/ARM/

(D) les données sont cryptées avec OpenSSL et une clé de 128 bits	$\longrightarrow$	*Donc* (Q) nécessairement (C) les informations contenues dans le fichier ne peuvent pas être accédée par quelqu'un non autorisé à avoir la clé
	Vu que (W) une clé de 128 bits suffit pour protéger les données d'après l'Advanced Encryption Standard.	
	En vertu de (B) l'Advanced Encryption Standard publié en 2001 préconise une taille de clé de 128 bits (FIPS PUB 197)	
		Sauf si (R) quelqu'un vole la clé

Figure 1. *Un exemple du schéma d'argumentation de Toulmin*

qu'une clé de 128 bits suffit pour protéger les données. Nous nous intéressons à l'argumentation, au pas de raisonnement, qui consiste à dire si (D) est vraie alors nous pouvons conclure que (C) *"les informations contenues dans le fichier ne peuvent pas être accédée par quelqu'un non autorisé à avoir la clé"*. Comme nous ne sommes pas dans un cadre formel, nous ne disposons pas d'une théorie axiomatique qui puisse nous donner la valeur de validité de la formule logique $D \rightarrow C$. Nous sommes ici dans un cadre purement rhétorique où nous devons essayer de définir si nous sommes face à une *"bonne"* argumentation ou pas. Dans cet exemple, la conclusion n'est pas toujours vraie. En effet, la confidentialité des informations contenues dans le fichier n'est pas établie dans l'absolue : un attaquant peut voler la clé. Nous devons donc ajouter une condition de réfutation, ce qui nous donne le schéma Figure 1.

3. Un schéma d'argumentation simplifié

Dans le cadre d'une argumentation visant à structurer les éléments permettant de statuer sur la validité d'une propriété logicielle, certaines subtilités développées par Toulmin ne nous semblent pas nécessaires. En effet, dans notre cas, nous cherchons à établir des propriétés qui sont toujours vraies, dès lors les *qualificateurs modaux* et les *conditions de réfutation* sont superflus. De plus, nous voulons simplifier le travail d'acceptation sans compliquer celui de vérification. Par conséquent, nous recherchons un schéma d'argumentation qui soit à la foi simple et utile. Nous proposons donc un schéma simplifié, structuré autour de deux notions clés : l'élément de preuve et la stratégie, ces deux éléments nous permettant d'établir une conclusion.

3.1. *Données ou éléments de preuve*

Toute argumentation, démonstration, se fonde sur des vérités préétablies. Par exemple, une démonstration dans un système formel postulera toujours qu'un ensemble d'axiomes sont vrais. Ces axiomes sont vrais par nature, il n'existe pas de démonstration qui les prouve, ils sont l'élément de base de tout raisonnement dans ce système. De façon analogue, toute argumentation repose sur un ensemble de postulats, acceptées par celui qui énonce la démonstration ainsi que son auditoire. Nous appellerons ces vérités préétablies : *éléments de preuve*[4].

Dans le cadre de l'argumentation légale, (Rodney A. Reynolds, 2002) définissent les éléments de preuve comme : "les données (faits et opinions) présentées comme des preuves pour une affirmation[5]". Les éléments de preuve ont la particularité de reposer sur l'autorité de celui qui les énonce. La validité, ou plutôt

4. traduction du mot anglais *"evidence"*.
5. *"data (facts or opinions) presented as proof for an assertion"*.

l'acceptation par l'auditoire, d'un fait ne repose plus sur le fait lui-même, mais sur la confiance que l'on accorde à celui qui l'énonce. Des exemples simples peuvent être : un résultat donné dans un article scientifique, une information donnée par un expert ou une pratique définie dans une norme. Ainsi, ce n'est pas la validité de l'information donnée qui est à démontrer, c'est la crédibilité de celui qui l'énonce, la source, qui est à prouver. Nous sommes typiquement ici dans un problème de confiance. Notons que la confiance n'est pas une valeur absolue, la confiance est relative à un domaine. On a confiance en quelqu'un dans un certain cadre.

3.2. *La garantie ou stratégie*

La *garantie* chez Toulmin est la pierre angulaire du raisonnement. C'est la garantie qui explicite clairement comment, à partir de données, il est possible d'inférer une conclusion. Remarquons que ce que Toulmin appelle *garantie* correspond exactement à ce que l'ISO 15026[6] appelle *Arguments* et que le Goal Structured Notation (GSN) (Kelly et Weaver, 2004) appelle *Strategy*. Afin d'homogénéiser les différents termes, nous avons décidé de ne plus utiliser le terme de garantie de Toulmin, mais le terme *stratégie*.

Parce que nous sommes dans un cadre où l'argumentation vise à convaincre une autorité (et doit être acceptée par elle), en plus de la stratégie, nous gardons le concept de Toulmin qui lui est directement associé : le *fondement*. Pour nous, les fondements sont les justifications sur le pourquoi une stratégie est acceptable. Prenons le cas de l'usage d'un outil d'analyse statique, l'utilisation de ce logiciel doit être motivée : il faut indiquer, par exemple, si le logiciel est acceptable dans le cadre de cette étude (cette acceptation pouvant être possiblement une certification) et, pourquoi pas, renvoyer à l'argumentation du logiciel qui elle-même explicitera son algorithme, son implémentation, etc.

4. Un exemple d'arbre d'argumentation

Dans (Pires *et al.*, 2013) les auteurs proposent une méthode pour vérifier automatiquement la conformité d'un code C, conforme à un certain patron de code, par rapport à sa spécification exprimée à l'aide d'une machine à états UML. Pour ce faire, ils utilisent des techniques d'analyse statique qui permettent la vérification de propriétés sur un programme sans avoir à l'exécuter. De façon pratique, ils génèrent automatiquement des annotations de preuve en langage ACSL[7] dans un code C à partir du modèle lui servant de spécification et vérifient ces annotations à l'aide d'un outil d'analyse statique, ici Frama-C. Si nous cherchons à établir l'argumentation, s'appuyant sur cette méthode, visant à établir la propriété qu'un code est conforme à sa spécification, nous devons éviter deux écueils.

Le premier consiste à assembler tous les documents (ici la spécification UML, les résultats donnés par Frama-C, etc.), considérer que ce sont autant d'éléments de preuve et que, suivant la stratégie qui consiste à suivre la méthode, la propriété est valide. Bien évidemment, ceci n'est pas une argumentation, cela ne structure pas le raisonnement sous-jacent à l'ensemble des documents. Dans la pratique, une telle approche a pour seul effet de noyer l'autorité en charge de l'acceptabilité sous un flot de documents, sans aucune indication sur l'articulation de l'ensemble.

---

6. La norme ISO/IEC 15026 est une norme applicable aussi bien à des systèmes qu'à des logiciels. Elle permet de définir des niveaux d'intégrités.
7. ANSI/ISO C Specification Language.

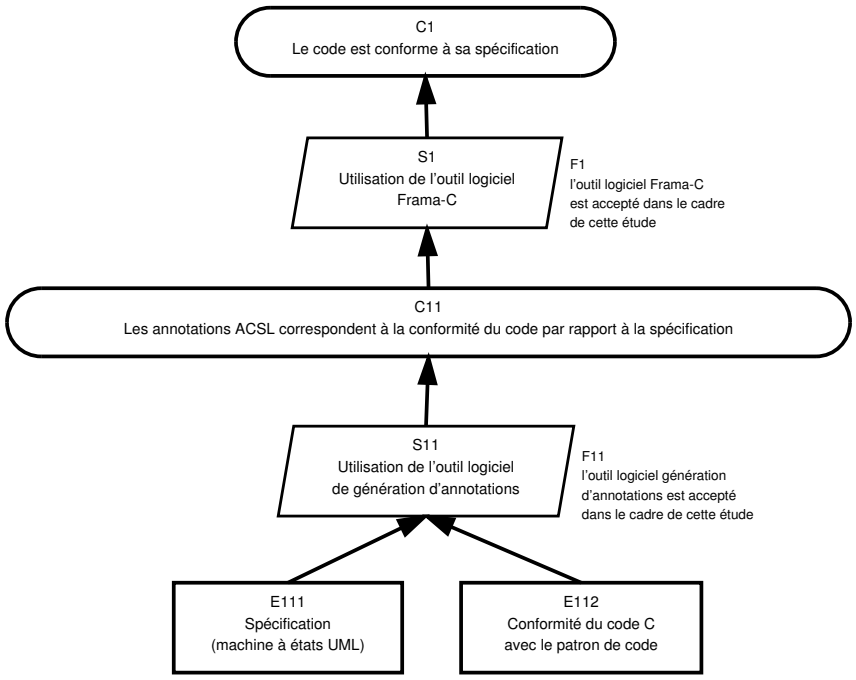


Figure 2. *Exemple d'arbre d'argumentation*

Le deuxième écueil consiste à confondre argumentation et description du processus. Le but d'une argumentation n'est pas d'expliquer les étapes qui ont permis de vérifier une propriété mais d'expliciter, sans ambiguïté, quels documents permettent d'établir la validité d'une propriété.

Finalement, une *bonne* argumentation, pour notre exemple, se compose de deux pas de raisonnement : Figure 2. Les stratégies employées dans cet arbre sont toujours de même nature, elles relèvent de résultats donnés par l'utilisation d'un outil logiciel s'appuyant sur des méthodes formelles. Attention, n'oublions pas que cette argumentation n'est qu'une représentation de l'articulation de l'ensemble des éléments qui permettent d'atteindre la conclusion, chaque élément doit renvoyer à un document (voir à un ensemble de documents) qui l'explicite.

Le premier pas de raisonnement dans notre arbre repose sur deux éléments de preuve qui renvoient (E111) à la spécification du système en UML et (E112) à un document qui établit la conformité du code C avec le patron de code de la méthode. De là, en appliquant la stratégie (S11) d'utilisation de l'outil logiciel de génération d'annotations, nous pouvons conclure que (C11) nous avons des annotations ACSL qui correspondent à la conformité du code par rapport à la spécification. Les fondements associés à cette stratégie sont que (F11) l'outil logiciel est accepté dans le cadre de cette étude (ce qui peut renvoyer, par exemple, au cahier des charges de l'étude). La conclusion (C11) du premier pas de raisonnement est l'élément de preuve du deuxième pas de notre argumentation qui, s'appuyant sur la stratégie (S1) d'utilisation de l'outil logiciel Frama-C, conclue que notre propriété de haut niveau est valide. Notons que des documents sont aussi associés aux stratégies, ainsi, est associé à (S1) les documents établissant que le solveur utilisé par Frama-C a bien prouvé les annotations.

5. Perspectives

Maintenant que nous disposons d'une argumentation qui motive l'acceptation d'une propriété, nous pourrions nous interroger sur le bien fondé de cette argumentation. Plus précisément, est-il pas possible de fournir à l'autorité en charge de la tâche d'acceptation du logiciel des éléments supplémentaires lui permettant de statuer sur le bien fondé de l'application d'une stratégie ? Pour cela, nous pouvons faire un parallèle entre les résultats donnés par de tels logiciels et la parole d'un expert. En effet, ces résultats relèvent d'un niveau d'expertise dont ne dispose pas forcément l'autorité en charge de l'acceptation, mais qui doit pour autant statuer sur l'acceptation, ou pas, des résultats donnés par un outil logiciel ou un expert. Pour l'aider dans cette tâche, nous proposons d'ajouter à notre schéma d'argumentation la notion de questions critiques. Depuis de nombreuses années, Douglas Walton mène des travaux où il cherche à définir un schéma représentant la parole d'un expert et à analyser comment une parole d'expert peut être réfutée ou affaiblie (Walton, 1996 ; Godden et Walton, 2006). Pour cela, il a défini ce qu'il appelle des *questions critiques*. Le but de cet ensemble de questions est d'évaluer et d'analyser de manière simple la parole d'un expert. Sur ce modèle, nous pourrions attacher à notre schéma d'argumentation dans le cadre de l'utilisation d'un outil logiciel des questions critiques. Le but de ces questions est, d'une part, d'obliger le créateur de l'argumentation à vérifier que son pas d'argumentation est bien construit, d'autres part, quand c'est nécessaire, d'ajouter à l'argumentation des documents répondant aux questions que peut légitimement se poser un relecteur.

Dans notre proposition, nous sommes restés à un stade très informel. Il pourrait être intéressant, dans l'avenir, de définir un cadre plus formel d'arbre d'argumentation, cadre qui nous permettrait de réaliser des opérations automatiques telles que : l'analyse des éléments de preuves, détecter les manques, les stratégies incomplètes, etc. Pour finir, une telle approche ne peut à terme se concevoir sans outil. Les arbres d'argumentations étant potentiellement immenses et gérant de l'hypertextualité, la question d'outil informatique de visualisation et de navigation reste cruciale pour leur utilisation sur des cas réels.

6. Bibliographie

Carnap R., *Logical Foundations of Probability*, University of Chicago Press, 1962.

Godden D. M., Walton D., « Argument from Expert Opinion as Legal Evidence : Critical Questions and Admissibility Criteria of Expert Testimony in the American Legal System », *Ratio Juris*, vol. 19, n° 3, 2006, p. 261–286, Blackwell Publishing.

Hamblin C., *Fallacies*, University paperback, Methuen, 1970.

Kelly T., Weaver R., « The Goal Structuring Notation /- A Safety Argument Notation », *Proc. of Dependable Systems and Networks 2004 Workshop on Assurance Cases*, 2004.

Perelman C., Olbrechts-Tyteca L., *Traité de l'argumentation : La nouvelle rhétorique*, UBlire - Fondamentaux, Éditions de l' Université de Bruxelles, 2008.

Pires A. F., Polacsek T., Wiels V., Duprat S., « Behavioural Verification in Embedded Software, from Model to Source Code », Moreira A., Schätz B., Gray J., Vallecillo A., Clarke P. J., Eds., *MoDELS*, vol. 8107 de *Lecture Notes in Computer Science*, Springer, 2013, p. 320-335.

Rodney A. Reynolds J. L. R., « *Evidence* », p. 427-446, SAGE Publications, 2002.

Toulmin S. E., *The Uses of Argument*, Cambridge University Press, Cambridge, UK, 2003, Updated Edition, first published in 1958.

Walton D. N., « Practical Reasoning and the Structure of Fear Appeal Arguments », *Philosophy and Rhetoric*, vol. 29, n° 4, 1996, p. 301–313.

Formula Negator, Outil de négation de formule.

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

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

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

1 Introduction

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

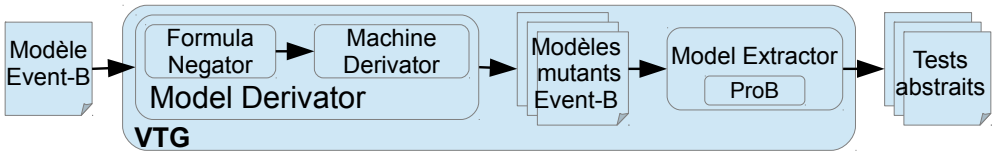


Fig. 1. Décomposition du VTG en sous-processus

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

2 Choix d'implémentation

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

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

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

3 *Formula Negator*

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

Algorithm 1 formulaExtractor

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

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

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

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

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

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

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

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

4 Fonctionnement de TOM

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

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

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

5 *Model Extractor*

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

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

Fig. 2. Grammaire Rodin définie en TOM

```
%match(formula) {
    ...
    Equal(left,right)->
    {
    Formula<?> f1,f2;
    f1=ff.makeRelationalPredicate(Formula.GT,`left,`right,null);
    f2=ff.makeRelationalPredicate(Formula.LT,`left,`right,null);
    Set<Formula> s = new Set<>();
    s.add(f1);
    s.add(f2);
    return  s;
    }
    ...
}
```

Fig. 3. Analyseur du *Formula Negator*

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

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

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

6 Conclusion

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

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

References

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

Le projet BWare : une plate-forme pour la vérification automatique d'obligations de preuve B

David Delahaye[1], Claude Marché[2] et David Mentré[3]
(pour le consortium du projet BWare)

[1] Cedric/Cnam/Inria, Paris, France
[2] Inria Saclay - Île-de-France & LRI, CNRS, Univ. Paris-Sud, Orsay, France
[3] Mitsubishi Electric R&D Centre Europe, Rennes, France

Le projet de recherche industrielle BWare (ANR-12-INSE-0010) est financé pour 4 ans par le programme « Ingénierie Numérique & Sécurité » (INS) de l'Agence Nationale de la Recherche (ANR) et a débuté en septembre 2012 (voir le site web du projet : `http://bware.lri.fr`). Le consortium du projet BWare associe les partenaires académiques Cedric, LRI, et Inria, ainsi que les partenaires industriels Mitsubishi Electric R&D Centre Europe (MERCE), ClearSy, et OCamlPro.

1 Présentation

Le projet BWare vise à produire un environnement permettant la vérification automatique d'obligations de preuve (OP) provenant du développement d'applications industrielles à haute intégrité utilisant la méthode B. Son cœur est la plate-forme générique de vérification déductive de programmes Why3 [2] intégrant différents outils de démonstration automatique tels que des systèmes au premier ordre et des solveurs SMT (« Satisfiability Modulo Theories »). Les outils au premier ordre considérés sont Zenon [4] et iProver Modulo [5], tandis que nous avons choisi Alt-Ergo [1] comme solveur SMT. Au-delà de l'aspect multi-outils, l'originalité du projet BWare réside également dans la production d'objets preuves par les outils de vérification. Pour vérifier indépendemment ces objets preuves, nous considérons deux vérificateurs : l'outil d'aide à la preuve Coq et le vérificateur de preuve universel Dedukti [3]. Pour évaluer notre méthodologie et tester notre plate-forme, une large collection d'OP est fournie par les partenaires industriels du projet qui développent des outils ou applications autour de la méthode B.

2 Résultats préliminaires

La plate-forme BWare est déjà opérationnelle. Les OP initialement produites par l'Atelier B sont traduites par un outil spécifique en buts Why3, qui sont fondés sur un encodage de la théorie des ensembles de B en Why3 [8]. La plate-forme Why3 permet alors d'envoyer ces OP aux outils de démonstration automatique, utilisant le format TPTP pour Zenon et iProver Modulo, et un format natif pour Alt-Ergo. Enfin, une fois que les preuves ont été trouvées par ces outils, certains peuvent générer des objets preuves pouvant être vérifiés : Zenon peut produire des objets preuves pour Coq et Dedukti [4, 7], et iProver Modulo des objets preuves pour Dedukti [6].

Pour évaluer la plate-forme BWare, deux partenaires industriels du projet ont fourni un banc d'essais initial de plus de 10 500 OP provenant de différentes applications industrielles : pour MERCE, il s'agit d'un cas d'utilisation complet d'un passage à niveau, et pour ClearSy, de trois projets industriels déployés. Les résultats obtenus au début du projet sont les suivants (obtenus sur une machine Intel Xeon X5650 2.67GHz, avec un temps limite de 30s) : le « main prover » (mp) de l'Atelier B (4.0) est capable de prouver 84% de ces OP, tandis qu'Alt-Ergo (0.95.1) obtient un taux de 58%, iProver Modulo (basé sur iProver 0.7) 19%, et Zenon (0.7.2) moins de 1%. Les

outils au premier ordre (iProver Modulo et surtout Zenon) rencontrent des difficultés parce que ces systèmes ne connaissent pas la théorie des ensembles de B. Concernant le solveur SMT Alt-Ergo, un ensemble intermédiaire de résultats obtenus avec des versions améliorées est publié sur le blog d'OCamlPro[1]. Ces résultats sont très prometteurs puisque la version de développement est maintenant capable de décharger automatiquement plus de 98% des OP.

3 Axes de travail actuels

Faute de place, nous ne pouvons pas décrire toutes les tâches du projet mais seulement deux axes de travail majeurs actuels. Le premier axe consiste à compléter en amont l'axiomatisation de la théorie des ensembles de B en Why3 pour pouvoir considérer toutes les OP fournies. Ce travail suit l'approche [8] en ajoutant des constructions B à l'axiomatisation et en modifiant le traducteur d'OP de l'Atelier B vers Why3 en conséquence. Ce travail nous permettra de considérer un large spectre d'OP et de tester également le passage à l'échelle de notre plate-forme.

Le second axe de travail se focalise sur les outils au premier ordre pour qu'ils puissent raisonner modulo la théorie des ensembles de B. Pour ce faire, nous utilisons la déduction modulo, une extension du calcul des prédicats permettant de réécrire des termes ainsi que des propositions, et qui est bien adaptée pour la recherche de preuve dans les théories axiomatiques, puisqu'elle transforme les axiomes en règles de réécriture. Nous avons ainsi étendu les deux outils au premier ordre pour obtenir Zenon Modulo [7] et iProver Modulo [5], deux extensions basées sur la déduction modulo et également capables de produire des preuves Dedukti [7, 6] (reposant aussi sur la déduction modulo). Actuellement, nos efforts sur cet axe consistent à construire une théorie des ensembles de B modulo adaptée à la démonstration automatique.

À plus long terme, nous prévoyons de faire une étude comparative plus complète des outils de vérification de manière à déterminer quel taux de couverture nous pouvons obtenir automatiquement avec notre plate-forme (notamment avec les outils de vérification étendus). Enfin, nous avons l'intention d'exploiter les résultats de notre projet en intégrant notre plate-forme à l'Atelier B, le transformant ainsi en un système multi-outils de vérification.

Références

[1] F. Bobot, S. Conchon, É. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout. *Alt-Ergo, version 0.95.2*. CNRS, Inria, and Université Paris-Sud, 2013. `http://alt-ergo.lri.fr`.

[2] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3 : Shepherd Your Herd of Provers. In *International Workshop on Intermediate Verification Languages (Boogie)*, 2011.

[3] M. Boespflug, Q. Carbonneaux, and O. Hermant. The λΠ-Calculus Modulo as a Universal Proof Language. In *Proof Exchange for Theorem Proving (PxTP)*, 2012.

[4] R. Bonichon, D. Delahaye, and D. Doligez. Zenon : An Extensible Automated Theorem Prover Producing Checkable Proofs. In *LPAR – Springer LNCS/LNAI 4790*, 2007.

[5] G. Burel. Experimenting with Deduction Modulo. In *CADE – Springer LNCS/LNAI 6803*, 2011.

[6] G. Burel. A Shallow Embedding of Resolution and Superposition Proofs into the λΠ-Calculus Modulo. In *Proof Exchange for Theorem Proving (PxTP) – EasyChair EPiC 14*, 2013.

[7] D. Delahaye, D. Doligez, F. Gilbert, P. Halmagrand, and O. Hermant. Zenon Modulo : When Achilles Outruns the Tortoise using Deduction Modulo. In *LPAR – Springer LNCS/ARCoSS 8312*, 2013.

[8] D. Mentré, C. Marché, J.-C. Filliâtre, and M. Asuka. Discharging Proof Obligations from Atelier B Using Multiple Automated Provers. In *ABZ – Springer LNCS 7316*, 2012.

1. Voir : `http://www.ocamlpro.com/blog/2013/10/22/alt-ergo-evaluation-october-2013.html`.

Exécution symbolique et critères de test avancés ⋆

Sébastien Bardin, Nikolai Kosmatov et François Cheynier

CEA, LIST, Laboratoire pour la Sûreté des Logiciels, PC 174, 91191 Gif-sur-Yvette, France
prenom.nom@cea.fr

Nous nous intéressons à la génération automatique de tests à partir du code source d'un programme. L'exécution symbolique dynamique (DSE) est une approche récente et particulièrement prometteuse [3,4]. Cependant, cette technique ne supporte pas nativement la plupart des critères de test classiques [1], comme les conditions multiples ou les mutations.

Notre objectif est de combler le fossé séparant la DSE des critères de test usuels. Nous définissons un nouveau critère, la couverture de labels, qui peut être vu comme un mécanisme de spécification pour décrire d'autres objectifs de tests (par ex. : décisions, conditions multiples, mutations faibles). Nous montrons que ce critère est expressif et peut être intégré efficacement dans la DSE. Ces résultats généralisent des travaux antérieurs [5,6]. Notamment, nous définissons des optimisations spécifiques à la DSE permettant de gérer les labels tout en évitant complètement l'explosion du nombre de chemins du programme, ce qui était la principale limitation des approches existantes. Les résultats expérimentaux montrent que ces optimisations permettent des gains particulièrement significatifs, aboutissant à une intégration des labels dans la DSE pour un surcoût marginal.

Il apparaît donc que les labels ont toutes les qualités requises pour être au centre d'un environnement générique de test automatisé : un mécanisme puissant de spécification de critères de tests, un calcul efficace de la couverture et enfin une intégration à moindre coût dans une des approches de génération automatique de tests les plus récentes.

Références

1. P. Ammann, A. J. Offutt : Introduction to software testing. Cambridge University Press (2008)
2. S. Bardin, N. Kosmatov, F. Cheynier. : Efficient Leveraging of Symbolic Execution to Advanced Coverage Criteria. In : ICST 2014. IEEE, Los Alamitos (2014)
3. P. Godefroid, N. Klarlund, K. Sen : DART : Directed Automated Random Testing. In : PLDI 2005. ACM, New York (2005)
4. P. Godefroid, M. Y. Levin, D. Molnar : Automated Whitebox Fuzz Testing. In : NDSS 2008.
5. K. Jamrozik, G. Fraser, N. Tillmann, J. de Halleux : Generating Test Suites with Augmented Dynamic Symbolic Execution. In : TAP 2013. Springer, Heidelberg (2013)
6. M. Papadakis, N. Malevris, M. Kallia : Towards Automating the Generation of Mutation Tests. In : AST 2010 (with ICSE 2010).

⋆. Ces résultats ont été présentés à ICST 2014 [2]. Les auteurs ont été partiellement financés par le programme EU-FP7 (projet STANCE, bourse 317753) et l'ANR (projet BINSEC, bourse ANR-12-INSE-0002).

A Compositional Automata-based Semantics for Property Patterns *

K. C. Castillos[1], F. Dadeau[1], J. Julliand[1], B. Kanso[2] and S. Taha[2]

1. FEMTO-ST/DISC - 16 route de Gray F-25030 Besançon cedex and

2. SUPELEC - 3 rue Joliot-Curie F-91192 Gif-sur-Yvette cedex.

Dwyer et al. [3, 2] (DAC) ont défini un langage pour spécifier des propriétés temporelles en combinant des schémas de propriétés (patterns) et des portées (scopes). Pour spécifier une propriété, un utilisateur choisit un pattern et un scope parmi une liste de patterns et de scopes proposée. DAC définissent une sémantique à ces propriétés en donnant pour chaque composition une formule de logique temporelle linéaire (LTL) Cette sémantique n'est pas compositionnelle et de ce fait, elle est difficilement extensible dans la mesure où l'ajout d'un pattern ou d'un scope induit la définition de toutes ses compositions par une formule LTL. De plus l'utilisateur peut avoir des difficultés à s'approprier les formules données. Par exemple, avec le langage de DAC, on peut exprimer la propriété suivante : "entre les états vérifiant les propriétés Q et R à un état satisfaisant P, il faut répondre par la propriété d'état P'" composant le pattern "P' responds to P" avec le scope "between Q and R" (voir Fig. 1). La formule LTL donnée par DAC est la suivante $\square((Q \wedge \neg R \wedge \Diamond R) \Rightarrow (P \Rightarrow (\neg R \ \mathsf{U} \ (P' \wedge \neg R))) \ \mathsf{U} \ R)$. On peut se rendre compte, que même si l'utilisateur est familier de la LTL, il aura des difficultés à appréhender le sens de cette formule.

Dans ce papier [1], nous proposons une sémantique compositionnelle à base d'automates. La sémantique de chaque pattern (voir Fig. 1 en haut à gauche) et de chaque scope est définie par un automate (voir Fig. 1 en haut à droite). Puis la sémantique de chaque propriété combinant les deux est définie par une opération de composition de leurs automates (voir Fig. 1 en bas). Ainsi la sémantique est compositionnelle et plus facilement extensible. Pour l'illustrer, nous donnons des exemples et des exemples d'extension du langage. Nous comparons cette sémantique compositionnelle avec celle des formules LTL des articles DAC. Pour cela, nous comparons les automates que nous proposons avec les automates de Büchi sous-jacents aux formules LTL définies dans les articles DAC. Cette comparaison a été effectuée systématiquement avec l'outil GOAL (Graphical Tool for Omega-Automata and Logics) [4]. Nous montrons que dans certains cas la sémantique que nous proposons est légèrement différente. Cela révèle un manque d'homogénéité de la sémantique par traduction en formules de logique temporelle.

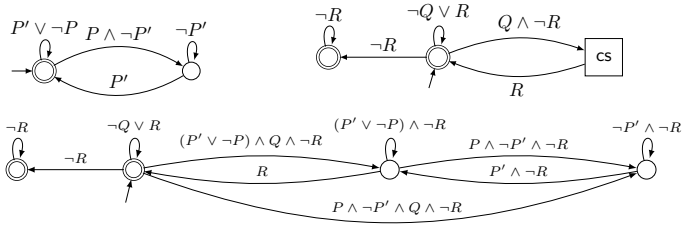


Figure 1: Automate de P′ responds to P between Q and R Obtenu par Composition

References

[1] Kalou Cabrera Castillos, Frédéric Dadeau, Jacques Julliand, Bilal Kanso, and Safouan Taha. A compositional automata-based semantics for property patterns. In *IFM*, pages 316–330, 2013.

[2] M.B. Dwyer, H. Alavi, G. Avrunin, J. Corbett, L. Dillon, and C. Pasareanu. Specification Patterns. *http://patterns.projects.cis.ksu.edu/*.

[3] M.B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *21st ICSE*, pages 411–420, 1999.

[4] Y.K. Tsay et al. Graphical Tool for Omega-Automata and Logics. `http://goal.im.ntu.edu.tw/wiki/doku.php`.

*Cet article est paru dans les actes du congrès iFM 2013, 10th International Conference on integrated Formal Methods, volume 7940 of LNCS, pages 316–330.

Conception de Diagrammes de Séquences Robustes aux Attaques *

Boutheina Bannour[1], Jose Escobedo[2], Christophe Gaston[1],
Pascale Le Gall[2], Gabriel Pedroza[2]
[1]CEA, LIST, laboratoire LISE
Point Courrier 174, 91191, Gif-sur-Yvette, France
{boutheina.bannour, christophe.gaston}@cea.fr
[2]ECP, laboratoire MAS
Grande Voie des Vignes, 92295, Châtenay-Malabry, France
{jose.escobedo, pascale.legall, gabriel.pedroza}@ecp.fr

Les systèmes logiciels largement distribués offrent des opportunités pour des attaquants, en partie à cause des nombreuses communications en jeu entre des composants distants. La vulnérabilité de tels systèmes tient ainsi en grande partie à la grande dispersion géographique des composants, qui peuvent ainsi être hors de contrôle des gestionnaires des systèmes.

Nous nous proposons de définir une méthodologie qui vise à analyser l'impact des attaques dès la phase de conception du système afin d'anticiper les effets des attaques et de réviser la conception des systèmes au plus tôt pour en renforcer les éléments de sécurité. L'idée directrice est l'identification d'un composant sécurisé, appelé *chien de garde*, qui sera en charge de l'analyse des communications du système et qui aura pour objectif de signaler (via un message d'alerte) la suspicion de la présence d'une attaque. Par des techniques d'exécution symbolique, les traces des scénarios nominaux et des scénarios perturbés par les attaques sont comparées. Un critère de *robustesse du chien de garde* est proposé, qui requiert que le chien de garde doit être capable de notifier le succès de la détection de l'attaque, pour peu que l'attaquant soit intervenu un certain nombre de fois dans le système. Si le critère de robustesse est mis en défaut pour une attaque jugée critique, i.e. représentant un trop grand risque, le concepteur est invité à réviser le diagramme de séquences des comportements nominaux, pour y inclure explicitement des envois des messages d'alerte pour la configuration particulière de l'attaque en question.

Les Smart Grids, dédiés à la gestion optimisée de la production et consommation énergétique, sont des systèmes dont leurs dispositifs sont grandement dispersés et que par construction se composent d'une partie sous le controle du gestionaire, et une autre en dehors du périmètre de confiance du gestionaire sous le controle des utilisateurs (notament les compteurs intélligents). C'est à cause de cette nature qu'ils se prêtent bien pour être analysés avec notre méthodologie.

Notre approche est outillée sur la base de l'environnement Papyrus d'édition de diagrammes de séquence, d'un plugin `sdToTIOSTS` en charge de la traduction diagrammes de séquence vers des automates temporisés ; et de l'outil Diversity d'exécution symbolique.

*Ce travail a été financé partiellement par le bureau français DGCIS, dans le contexte du projet SESAM Grids, dédié à l'étude de la sécurité et sûreté des Smart Grids. Cette publication a été acceptée dans l'atelier "5th International Workshop on Security Testing (SECTEST 2014)", événenment satellite de la conférence ICST 2014.

Flower : réduction optimale de suites de test en utilisant la programmation par contraintes

Arnaud Gotlieb et Dusica Marijan
Certus Software Validation & Verification Center,
Simula Research Laboratory, Norvège
{arnaud,dusica}@simula.no

Préserver la qualité des logiciels tout en réduisant l'effort de test est un des objectifs essentiels des ingénieurs en charge de la validation des logiciels. Étant donnée une suite de test, et un ensemble d'exigences couvertes par ces tests, *réduire la suite de test* est un processus qui vise à sélectionner un sous-ensemble des tests qui préserve la couverture des exigences. Bien que ce problème ait reçu un intérêt considérable, trouver un sous-ensemble de cas de test de taille minimale dans un temps raisonnable est toujours un sujet d'un grand intérêt pratique. Dans la mesure où ce problème est connu pour être NP-complet, les approches existantes se contentent ainsi de solutions approchées peu satisfaisantes. En effet, lorsque l'exécution d'un cas de test demande des heures de préparation, trouver un plus petit sous-ensemble de cas de test devient alors crucial.

Dans cet article, nous introduisons une approche radicalement nouvelle pour le problème de la réduction de suite de test. Notre approche, baptisée Flower, s'appuie sur la recherche de flot maximum dans un réseau de flots en s'inspirant d'algorithmes de filtrage introduits en programmation par contraintes pour le traitement de certaines contraintes globales. À partir d'une suite de test et d'un ensemble d'exigences, Flower forme un réseau de flot, qui est ensuite systématiquement exploré afin de découvrir un flot maximum bien particulier. En utilisant l'algorithme de Ford-Fulkerson pour la recherche de flot maximum, Flower est une méthode exacte qui calcule une suite de test de taille minimale qui préserve la couverture des exigences, même si un mécanisme permet d'interrompre la recherche à tout moment avec une solution approchée. De manière intéressante, nos résultats expérimentaux montrent que cette approche est plus perfomante qu'un modèle de programmation linéaire en nombres entiers de 15 à 3000 fois en terme de temps requis pour trouver une solution optimale, et plus performante qu'une approche gloutonne de 5 à 15% en terme de taille obtenue de la suite réduite.

Cet article sera présenté lors de la conférence ISSTA (*International Symposium on Software Testing and Analysis*) qui se déroulera du 21 au 25 Juillet 2014 en Californie, et son contenu sera publié dans les actes de cette conference.

Dérivation formelle et extraction d'un programme data-parallèle pour le problème des valeurs inférieures les plus proches *

F. Loulergue[1], S. Robillard[1], J. Tesson[2], J. Légaux[1] and Z. Hu[3]

[1] Univ. Orléans, INSA Centre Val de Loire, LIFO EA 4022, France
{Frederic.Loulergue,Simon.Robillard,Joeffrey.Legaux}@univ-orleans.fr
[2] Université Paris Est Créteil, LACL, Créteil, France, Julien.Tesson@lacl.fr
[3] National Institute of Informatics, Tokyo, Japan, hu@nii.ac.jp

Le problème des valeurs inférieures les plus proches (*All Nearest Smaller Values* ou ANSV) est un problème important pour la programmation parallèle [1] car il peut être utilisé pour résoudre plusieurs problèmes plus spécifiques et il est également l'une des phases d'algorithmes plus complexes. Le problème est le suivant: soit $xs = [x_1; x_2; \ldots; x_n]$ une liste d'éléments d'un domaine totalement ordonné, pour chaque x_i, trouver la valeur la plus proche à gauche de x_i et la valeur la plus proche à droite de x_i qui sont inférieures à x_i. S'il n'y a pas de telle valeur, indiquer $\perp$ à la place.

Partant d'une spécification naïve, sous forme d'un programme fonctionnel inefficace, nous dérivons pas-à-pas un programme fonctionnel séquentiel plus efficace. Cette dernière formulation utilise une fonction d'ordre supérieure appelée homomorphisme quasi synchrone, dont nous fournissons une implantation parallèle vérifiée. La dérivation et la vérification de l'implantation parallèle sont conduites à l'aide de l'assistant de preuve Coq [4].

Nous utilisons finalement la fonctionnalité d'extraction de code de Coq pour obtenir un programme OCaml avec des appels à la bibliothèque de programmation parallèle *Bulk Synchronous Parallel ML*. Les performances du programme parallèle fonctionnel obtenu par extraction sont comparées avec une version implantée avec la bibliothèque C++ de squelettes algorithmiques OSL [3], et avec une implantation d'un algorithme non vérifié dû à He et Huang [2].

Références

[1] Omer Berkman, Baruch Schieber, and Uzi Vishkin. Optimal Doubly Logarithmic Parallel Algorithms Based on Finding All Nearest Smaller Values. *Journal of Algorithms*, 14(3): 344–370, 1993. doi:10.1006/jagm.1993.1018.

[2] Xin He and Chun-Hsi Huang. Communication Efficient BSP Algorithm for All Nearest Smaller Values Problem. *Journal of Parallel and Distributed Computing*, 61(10):1425–1438, 2001. doi:10.1006/jpdc.2001.1741.

[3] Joeffrey Légaux. *Squelettes algorithmiques pour la programmation et l'exécution efficaces de codes parallèles*. Thèse de doctorat, LIFO, Université d'Orléans, Décembre 2013.

[4] Julien Tesson. *Environnement pour le développement et la preuve de correction systématiques de programmes parallèles fonctionnels*. Thèse de doctorat, LIFO, Université d'Orléans, Novembre 2011. URL http://hal.archives-ouvertes.fr/tel-00660554.

*Résumé de Frédéric Loulergue, Simon Robillard, Julien Tesson, Joeffrey Légaux, and Zhenjiang Hu. Formal Derivation and Extraction of a Parallel Program for the All Nearest Smaller Values Problem. In *ACM Symposium on Applied Computing*, pages 1577–1584. ACM Press, 2014.

Comment la génération de tests facilite la spécification et la vérification déductive des programmes dans Frama-C*

Guillaume Petiot[1,2] Nikolai Kosmatov[1] Alain Giorgetti[2,3] Jacques Julliand[2]

[1] CEA, LIST, Software Reliability Laboratory, PC 174, F-91191 Gif-sur-Yvette
firstname.lastname@cea.fr
[2] FEMTO-ST/DISC, Université of Franche-Comté, F-25030 Besançon Cedex
firstname.lastname@femto-st.fr
[3] INRIA Nancy - Grand Est, projet CASSIS, F-54600 Villers-lès-Nancy

Pour la validation des systèmes critiques, l'analyse statique du code source, sans exécution, et l'analyse dynamique, par exécution du programme, sont deux techniques complémentaires. Parmi les techniques d'analyse statique, la vérification déductive fournit une preuve mathématique rigoureuse qu'un programme respecte sa spécification. Les prouveurs de théorèmes automatisent certaines de ces preuves, mais écrire des spécifications appropriées pour la preuve automatique est en pratique un travail délicat, qui nécessite souvent une analyse manuelle fastidieuse des échecs de preuve. Nous montrons comment la génération automatique de tests peut faciliter l'écriture de spécifications formelles correctes et leur vérification déductive.

Nos contributions sont (a) une présentation rapide d'un nouvel outil de combinaison d'analyse statique et dynamique, nommé StaDy ; (b) une méthodologie de vérification déductive incrémentale, profitant des résultats fournis par la génération de tests. Ses avantages sont illustrés par un ensemble de scénarios de vérification fréquents ; (c) une synthèse d'expérimentations montrant les capacités de détection de bogues de StaDy.

Dans la plateforme Frama-C d'analyse de programmes C, l'outil StaDy combine le générateur de tests concolique PathCrawler avec le greffon de vérification déductive Wp. Il traite tous les aspects de la spécification formelle (pre-/postconditions, assertions, invariants de boucle et variants) lors de la génération de tests avec PathCrawler. A partir d'un programme C annoté dans le langage de spécification exécutable e-acsl, StaDy traduit la spécification en code C exécutable, instrumente le programme pour la détection d'erreurs de non-conformité avec la spécification, utilise PathCrawler pour générer des tests du code instrumenté, et enfin retourne les résultats dans Frama-C. Pour détecter les erreurs, la traduction génère des branches supplémentaires, afin que la génération de tests explore aussi les cas d'erreur et génère des données d'entrée activant les erreurs, si de telles données existent. Ces erreurs sont forcées dans les assertions, les postconditions, les invariants de boucles, les variants, et aussi dans la pré- et la postcondition des fonctions appelées. Contrairement à d'autres outils concoliques, PathCrawler est complet et n'approxime pas les conditions de chemin. Ainsi, chaque fois que la génération de tests selon le critère de couverture de tous les chemins termine sans trouver d'erreur, l'ingénieur validation est sûr que le programme respecte sa spécification e-acsl. Si la couverture n'est que partielle, mais qu'aucune erreur n'est survenue, la génération de tests ne peut pas garantir que le programme est conforme à sa spécification, mais cette analyse dynamique augmente la confiance en cette conformité. Enfin, lorsqu'un test a échoué, ses données d'entrée et de sortie aident à localiser l'erreur dans la spécification ou dans le programme, à comprendre la nature de l'erreur, et ainsi à la corriger plus facilement.

Du point de vue méthodologique, nous suggérons d'utiliser la génération de tests dès le début de l'écriture des spécifications (*Early validation*), même si ces dernières ne sont pas complètes pour la vérification déductive. Le test permet d'éliminer bon nombre d'erreurs de spécification, avant que ces erreurs soient dupliquées dans plusieurs annotations du code. Pour les boucles, ceci permet d'augmenter la confiance dans la correction des invariants avant d'avoir réussi à les compléter pour la preuve (*Incremental loop validation*). Pour réduire le nombre de chemins à explorer par le test, nous introduisons un nouveau mot-clé (`typically`) permettant de définir une précondition supplémentaire e.g. bornant la taille des données d'entrée. Cette précondition est exploitée par le générateur de tests et ignorée par les outils d'analyse statique.

*Cet article est un résumé étendu de l'article "How Test Generation Helps Software Specification and Deductive Verification in Frama-C" accepté à la conférence "Tests and Proofs 2014". Ce travail a été partiellement financé par le programme EU-FP7 (projet STANCE, bourse 317753).

Lazart: a symbolic approach for evaluating the robustness of secured codes against control flow fault injections

Marie-Laure Potet, Laurent Mounier, Maxime Puys and Louis Dureuil

Laboratoire Vérimag, Université de Grenoble Alpes
prénom.nom@imag.fr

Article présenté à ICST 2014[1]. Les cartes à puce sont un vecteur important pour les applications exigeant un haut niveau de sécurité, par exemple dans le domaine bancaire, de l'identité numérique ou du médical. Au vu de la sensibilité des données qu'elles protègent, le matériel et les applications embarquées doivent être conçues pour résister à des attaques de haut niveau [CCD09]. Nous nous intéressons ici aux attaques par perturbation, consistant en une attaque physique provoquant une modification du code à exécuter. Nous proposons une approche, appelée Lazart, permettant d'évaluer la robustesse d'un code contre les injections de fautes visant des modifications du flot de contrôle. L'approche Lazart est basée sur une analyse statique du code, dont l'exécution symbolique. On peut ainsi produire des attaques mais aussi établir des verdicts d'absence d'attaques. De plus le choix d'une analyse statique nous permet de nous intéresser au multi-fautes (plusieurs injections de faute) tout en maîtrisant la combinatoire que cela introduit.

L'approche proposée est implémentée dans une chaîne d'outils, basée sur le format LLVM [Inf]. Une première étape permet de calculer les points d'injection de fautes en fonction d'un objectif d'attaque. Ceux-ci sont embarqués dans un unique mutant dont l'exécution symbolique (basée sur le générateur de test concolique Klee [CDE08]), permet de calculer les attaques d'ordre inférieur à n, pour un n fixé. Nous présentons les résultats obtenus sur deux implémentations plus ou moins robustes de la vérification de PIN. Nous proposons aussi un critère permettant de comparer certaines attaques.

References

[CCD09] Application of Attack Potential to Smartcards. Technical Report CCDB-2009-03-001, Commun Criteria, march 2009.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[Inf] The LLVM Compiler Infrastructure. http://llvm.org.

[1] IEEE International Conference on Software Testing, Verification, and Validation, Cleveland, Ohio, USA, March 31- April 4, 2014