Actes de la 8e conférence AFADL

Approches Formelles dans l'Assistance au Développement de Logiciels

Éditeurs
Marie-Laure Potet
Pierre-Yves Schobbens
Hubert Toussaint
Germain Saval

Préface

Pour son dixième anniversaire, la huitième conférence francophone sur les Approches Formelles dans l'Assistance au Développement de Logiciels, AFADL, se tient pour la première fois hors de France : elle prend place à Namur, capitale de la Région Wallonne.

Depuis sa création en 1997 à Toulouse, cette conférence constitue un moment privilégié de rencontre et d'échanges conviviaux pour les chercheurs francophones du domaine, qu'ils soient académiques ou industriels, débutants ou confirmés.

Cette année, la conférence est précédée d'une journée de tutoriels sur l'utilisation de méthodes formelles appliquées dans le domaine spatial. M. Guillaume Brat (NASA Ames) traitera de *Software Verification for Space Applications.* M. Marc Niézette (Vega Group), nous présentera son travail sur le *Planning and Scheduling for Space Applications.*

Elle s'ouvre par une conférence invitée de M. Guillaume Brat, sur son *Expérience en analyse statique de logiciels embarqués.* Le professeur Jean-Raymond Abrial, de l'ETH de Zürich, nous parlera de *l'utilisation du B événementiel avec la plateforme Rodin.* La méthode B est en effet un thème important d'AFADL depuis sa création. Mme Odile Laurent, d'Airbus, nous présentera *EICOSE, the European Institute for Complex & Safety Critical Embedded Systems Engineering* récemment créé. Le professeur Pierre Wolper de l'Université de Liège nous présentera le *calcul de fermetures par automates*, un des derniers développements de sa ligne de recherche sur les automates.

Nous adressons tous nos remerciements les plus sincères à toutes les personnes qui ont contribué au succès de cette édition, et en particulier : nos orateurs invités, les auteurs des articles soumis, le comité de programme, et le comité d'organisation.

Nous tenons également à remercier ici nos mécènes, les Facultés Universitaires Notre-Dame de la Paix à Namur, le Fonds de la Recherche Scientifique et IBM Belgique.

Marie-Laure Potet et Pierre-Yves Schobbens

Présidents

Marie-Laure Potet
Pierre-Yves Schobbens

Comité de Programme

Yamine Ait Ameur
Philippe Ayrault
Lilian Burdy
Dominique Cansell
Catherine Dubois
Mamoun Filali Amine
Pascal Gribomont
Patrick Heymans
Jacques Julliand
Jean-Marc Jézéquel
Olga Kouchnarenko
Regine Laleau
Jean-Louis Lanet
Yves Ledru
Nicole Levy
Bruno Marre
Pierre Michel
Charles Pecheur
Jean-François Raskin
Christel Seguin
Jeanine Souquières
Sylvie Vignes

Comité d'organisation

Pascal Gribomont
Pierre-Yves Schobbens
Cedric Aerts
Radu Cotet
Isabelle Daelman
Anne de Baenst
Babette Di Guardia
Maxime Lamury
Annick Massart
Germain Saval
Gabriel Schwanen
Hubert Toussaint
Wim Vanhoof

Relecteurs

Benjamin Blanc
Bernard Botella
Sylvain Boulmé
Michel Charpentier
Fabrice Derepas
Pierre-Cyrille Héam
Arnaud Lanoix
Gérard Padiou
Philippe Quéinnec
Marianne Simonot

Table des matières

Conférences invitées

Expériences en analyse statique de logiciels embarqués 11
Guillaume Brat

Le projet EICOSE 13
Odile Laurent

Le B événementiel (Event-B) et son utilisation avec la plateforme Rodin . 15
Jean-Raymond Abrial

Calcul de fermetures par automates 17
Pierre Wolper

Articles

Contribution à la validation formelle des systèmes interactifs 21
Alexandre Cortier, Bruno d'Ausbourg, Yamine Aït-Ameur

Une approche incrémentale combinant test et extraction de modèles 39
Roland Groz, Keqin Li, M. Muzammil Shahbaz

Déclinaison d'exigences de sécurité du système vers le logiciel, assistée par des modèles formels 57
Jean-Marc Bosc, Charles Castel, Pierre Darfeuil, Yves Dutuit, Eric Focone, Sophie Humbert, Christel Seguin

On the Design and the Implementation of a Game-based Model for Open Systems : Current Status and Perspectives 75
Marco Faella, Axel Legay

Schémas de développement d'adaptateurs à l'aide de B 91
Arnaud Lanoix, Samuel Colin, Jeanine Souquières

Intégration de propriétés temporelles dans des applications à base de composants 109
Sébastien Saudrais, Olivier Barais, Laurence Duchien, Noël Plouzeau

Vers la génération automatique de tests à partir d'arbres de tâches 125
Laya Madani, Ioannis Parissis

Test de logiciels synchrones avec Lutess : apports de la programmation par contraintes 143
Besnik Seljimi, Ioannis Parissis

Test fonctionnel de conformité vis-à-vis d'une politique de contrôle d'accès 161
Frédéric Dadeau, Amal Haddad, Thierry Moutet

Développement formel de circuits électroniques par la méthode B 181
Yann Zimmermann

Debugging Event B Models using the ProB Disprover Plug-In 199
Olivier Ligot, Jens Bendisposto, Michael Leuschel

Dérivation d'algorithmes sans verrou à partir d'une spécification atomique 213
Loïc Fejoz, Stephan Merz

Contrôler le contrôle d'accès : approches formelles 227
Mathieu Jaume, Charles Morisset

A System to Check Operational Properties of Logic Programs 245
François Gobert, Baudouin Le Charlier

Différentiation automatique et formes de Taylor en analyse statique de programmes numériques 261
Alexandre Chapoutot, Matthieu Martel

Présentations d'outils

Formal Requirements Modelling and Early Verification & Validation of Critical Systems 281
C. Ponsard, P. Massonet, J.F. Molderez, G. Dallons

Intégration du support OCL dans Kermeta 283
J-M Mottu, O. Barais, M. Skipper, D. Vojtisek, J-M Jézéquel

haRVey : satisfaisabilité et théorie 287
D. Caminha B. de Oliveira, D. Déharbe, P. Fontaine

VeSTA : Vérification de la préservation des propriétés d'un composant lors de son intégration dans un système temporisé 289
J. Julliand, H. Mountassir, E. Oudot

Test fonctionnel pour Focal 291
M. Carlier, C. Dubois

Tobias-2 : un outil pour la maîtrise des tests combinatoires 293
Y. Ledru, S. Ville, E. Rose, L. du Bousquet, F. Dadeau

B2EXPRESS : Un animateur de modèles B événementiels 295
Idir Ait-Sadoune, Yamine Ait-Ameur

Application de critères structurels en présence d'appels de fonctions pour la sélection et génération de tests 297
Patricia Mouy, Nicky Williams, Pascale Le Gall

Utilisation des outils IBM Rational pour le développement orienté modèle 301
Éric Cattoir

Index des auteurs 303

Conférences invitées

Expériences en analyse statique de logiciels embarqués

Guillaume Brat

RIACS / NASA Ames, USA

Résumé

Dans cette présentation, je vais décrire comment nous avons essayé d'appliquer des techniques d'analyse statique à des systèmes de vol embarqués de la NASA. Nos expériences avec des analyseurs commerciaux se sont revélées très décevantes. Nous avons donc dû concevoir notre propre analyseur, en nous servant de la théorie de l'interprétation abstraite. Cette présentation portera plus sur les expériences elles-mêmes que sur les détails théoriques des techniques des outils utilisés. J'essaierai de montrer, en me basant sur des études de cas réels, pourquoi l'analyse statique est importante pour la NASA.

Le projet EICOSE

Odile Laurent

Airbus

Résumé

Le projet EICOSE (European Institute for Complex & Safety Critical Embedded Systems Engineering) devient officiellement le premier pôle d'innovation d'ARTEMIS (Advanced Research & Technology for EMbedded Intelligence and Systems), la plate-forme technologique européenne spécialisée dans le domaine des systèmes embarqués. Il inclut les pôles de compétitivité System@tic Paris-Region, Aerospace Valley, ainsi que le cluster allemand SafeTrans. Quatre des cinq groupes de travail du pôle Eicose sont particulièrement intéressants pour les chercheurs AFADL. À savoir : les Environnements de développement des systèmes embarqués critiques ; la réduction des coûts de développement de systèmes certifiables ; la conception pour la robustesse et la fiabilité ; l'interaction système pour les assistances homme-machine.

Le B événementiel (Event-B) et son utilisation avec la plateforme Rodin

Jean-Raymond Abrial

École polytechnique fédérale de Zurich (ETH)

Résumé

Le B événementiel a maintenant atteint sa maturité. L'exposé présente donc les principes et concepts de ce formalisme utilisé pour la modélisation de programmes séquentiels, distribués, concurrents ou parallèles ainsi que celle de systèmes complets incluant aussi bien un logiciel que son environnement.

L'usage de ce formalisme est toujours basé, comme l'était le B « classique », sur la pratique du raffinement et de la preuve.

Les outils de la plateforme Rodin permettent maintenant d'utiliser le B événementiel dans de bonnes conditions. On présentera donc aussi succintement cette plateforme ouverte et d'accès totalement libre.

Le calcul de fermetures par automates

François Cantin, Axel Legay, Pierre Wolper

Université de Liège

Résumé

Dans le cadre de la vérification de systèmes à espace d'états infini, les automates finis s'avèrent être une représentation for commode des ensembles d'états. En particulier, les ensembles de vecteurs d'entiers ou de réels se représentent très naturellement par des automates sur lesquels on peut directement appliquer des procédures de calcul. Dans ce contexte, nous considérons le problème de calculer l'enveloppe convexe d'un ensemble fini de vecteurs d'entiers à n dimensions représenté par une automate fini. La méthode consiste à calculer une séquence d'approximations de l'enveloppe convexe et à utiliser des techniques d'extrapolation pour obtenir la limite de cette séquence. La fermeture convexe peut alors être directement calculée à partir de cette limite sous la forme d'une représentation par automate de l'ensemble correspondant de vecteurs de réels. La technique est fort générale et a été implémentée avec succès.

Articles

Contribution à la Validation Formelle des Systèmes Interactifs

Alexandre Cortier[1], Bruno d'Ausbourg[1], and Yamine Aït-Ameur[2]

[1]Centre d'Études et de Recherches de Toulouse - ONERA
2 Avenue E. Belin - BP 4025
310555 TOULOUSE, France
{cortier,ausbourg}@cert.fr, http://www.cert.fr/

[2]LISI / ENSMA
Téléport 2, 1 avenue Clément Ader - BP 40109 -
86961 Futuroscope Chasseneuil
{yamine}@ensma.fr, http://www.lisi.ensma.fr/

Résumé Les Interfaces Utilisateurs (IU) sont des systèmes de plus en plus complexes qui assistent à présent des activités critiques. Le développement de ces interfaces rend nécessaire l'utilisation de méthodologies de validation permettant d'assurer la correction du système. Ce papier propose d'exploiter les techniques de rétro-ingénierie et les méthodes formelles pour valider la correction des IU en terme d'utilisabilité du système. L'idée de cette approche est de dériver un modèle comportemental abstrait d'applications Java/Swing par analyse de leur code source. Le modèle formel extrait est alors utilisé pour prouver que le système interactif développé respecte les exigences relatives à l'usage du système telles qu'elles peuvent être exprimées par des modèles de tâches CTT.

Mots clés. Interfaces Utilisateurs, Interaction Homme-Machine, Analyse Statique, Méthode B événementielle, Modèles de tâches, CTT.

1 Introduction

Depuis une vingtaine d'années, les Interfaces Utilisateurs (IU) sont devenues de plus en plus complexes, d'une part du fait de la taille sans cesse croissante des systèmes à interfacer, et d'autre part du fait des évolutions technologiques et des nouvelles possibilités d'interaction qu'elles induisent. En outre ces interfaces assistent des systèmes critiques tels que, par exemple, les systèmes de pilotage d'un avion.

Les exigences attendues des applications interactives s'expriment souvent en termes d'*utilisabilité*. L'utilisabilité dénote l'efficacité et la satisfaction avec lesquelles les utilisateurs peuvent employer le système pour réaliser leur but [15]. En d'autres termes, le système peut être considéré comme correct s'il permet aux utilisateurs d'effectuer les tâches pour lesquelles il a été conçu. La validation

de tels systèmes nécessite donc d'avoir à sa disposition à la fois une représentation de leur comportement et une représentation des tâches, qu'ils doivent permettre de réaliser, et de leurs enchaînements possibles. Ces deux représentations peuvent être décrites par deux modèles : un modèle concret (le comportement du système) et un modèle abstrait (tâches utilisateurs) du même système.

De nombreux modèles ont été décrits, analysés et étudiés dans la littérature. Chacun d'eux possède un point de vue particulier sur le système final. Parmi eux, certains modèles centrés utilisateur décrivent les tâches que l'utilisateur doit pouvoir effectuer sur le système [11,13]. D'autres modèles, tels que SEEHEIM ou ARCH se concentrent sur la description de l'architecture logicielle du système [6]. Enfin certains modèles décrivent formellement le comportement d'un système interactif en représentant la structure des interactions qu'il admet. On peut citer parmi eux, et de manière non exhaustive les travaux de [8,14,5,12,4].

Des travaux relativement récents s'intéressent à la construction de ces modèles par analyses du code source de l'application développée. Ces analyses, statiques ou dynamiques suivant l'objectif recherché (validation formelle ou test), permettent l'extraction de modèles d'exécution du système [9,17,10]. Les travaux de Silva et al. [16] ont notamment proposé l'extraction d'un modèle abstrait comportemental (modèle d'interaction et machine à états) et d'un modèle structurel (graphe d'événements) par analyse statique de codes sources Java/Swing.

L'approche présentée dans ce papier s'inscrit dans cette même perspective et a pour objectif de contribuer à la validation formelle des IU. Techniquement, cette approche combine deux modèles qui font l'objet d'une formalisation dans le langage B événementiel (Event-B). Le premier est un modèle de tâches considéré comme une spécification possible du système. On fait l'hypothèse que ce modèle de tâche est formulé dans le langage CTT (Concur Task Tree) [13]. Il doit donc faire l'objet d'une formalisation pour produire un modèle B événementiel M_{Sp}. Une analyse statique de programme Java/Swing permet l'extraction d'un modèle abstrait M_{Sy}, également formulé dans le langage B, qui capture les aspects comportemental et structurel du système. Dans ce contexte, démontrer la conformité du système à sa spécification revient à démontrer que M_{Sy} est un raffinement correct de M_{Sp}.

Le papier est construit comme suit. La section 2 expose brièvement la méthode Event-B et la notation CTT. Les principes de la programmation Java/Swing ainsi qu'un exemple *jouet* sont présentés en section 3. Les principes généraux de l'approche sont présentés en section 4. Les sections 5 et 6 illustrent les deux étapes principales de l'approche (extraction et validation). Enfin, une conclusion est proposée en section 7.

2 Méthode B et notations CTT

2.1 La Méthode Event-B

Modèle B événementiel. La méthode Event-B, adaptée à la représentation des systèmes interactifs [4] a été développée par J.R Abrial [2,1]. Cette méthode est

```
MODEL nameM
REFINES nameR
    Nom du modèle abstrait
  SETS
    Noms de type et noms des ensembles
  CONSTANTS
    Déclaration du nom des constantes
  PROPERTIES
    Définition des propriétés logiques des constantes
  VARIABLES
    Déclaration du nom des variables du modèle
  INVARIANT
    Définition des propriétés statiques par des formules logiques
    (typage, invariants de collage, propriétés de sûreté)
  ASSERTIONS
    Définition des propriétés sur les variables et les constantes.
    A prouver à partir des invariants.
  INITIALISATION
    Evénement d'initialisation du modèle
  EVENTS
    Définition des événements associés au modèle
END
```

Fig. 1. Structure d'un modèle Event-B.

basée sur la notion de modèle. Un modèle Event-B est constitué d'un ensemble de variables définies dans une clause **VARIABLES**, qui évoluent grâce à des événements décrits dans une clause **EVENTS**. Un modèle B encode un système d'états-transitions où les variables représentent l'état du système et où les événements décrivent les transitions du système. La figure 1 présente la structure d'un modèle B événementiel.

Les événements. Un événement B possède un *nom*, une *garde* et une *action* communément appelée corps de l'événement. La garde représente la condition nécessaire au déclenchement de l'événement. Cette garde est un prédicat G (logique du premier ordre) construit sur les constantes et les variables du système. L'action est définie par une substitution qui décrit la manière dont l'événement modifie les variables. Le langage B dispose d'une sémantique à base de trace avec entrelacement. Ainsi, si deux événements ont leurs gardes vraies au même instant, ils ne sont pas déclenchés en même temps : il y a entrelacement des événements dans un ordre indéterminé. Nous utiliserons deux types d'événement dans cet article, que nous présentons ci-dessous. Dans ce qui suit, x représente un ensemble de variables définies dans la clause **VARIABLES** du modèle :

1. *nomEvt*= **SELECT** $G(x)$ **THEN** $S(x)$ **END** : l'événement gardé est une substitution $S(x)$ gardée par l'expression $G(x)$. L'événement se déclenche lorsque $G(x)$ est vraie ;
2. *nomEvt*= **ANY** l **WHERE** $G(x, l)$ **THEN** $S(x, l)$ **END** : événement indéterministe. Cet événement est gardé par $\exists l.G(l, x)$. Cet événement ne peut se déclencher que s'il existe des valeurs pour les variables locales l qui satisfont la condition $G(x, l)$.

Raffinements. Un modèle B peut être raffiné. Le processus de raffinement consiste à remplacer les structures de données abstraites par des structures de

données concrètes et à remplacer les substitutions abstraites par des substitutions concrètes. En outre de nouveaux événements peuvent être introduits afin de décrire plus finement le comportement du système. Un raffinement est un modèle dont les comportements sont des comportements du modèle abstrait. Il satisfait donc l'invariant du modèle abstrait. Un nouvel invariant est ajouté au modèle du raffinement. Il consiste à lier les variables abstraites à celle du raffinement. Cet invariant est appelé invariant de *collage*. Un raffinement est donc correct si : l'initialisation de même que chaque événement du système préservent l'invariant. Ces Obligations de Preuves (OPs) sont générées automatiquement par les outils B.

En outre, précisons que les modèles B événementiels sont représentés en B classique du fait du manque d'outil implémentant le B événementiel. Lors de la traduction, une propriété doit être ajoutée dans la clause **ASSERTIONS** afin d'assurer la réactivité du système. Cette propriété est exprimée en indiquant que la disjonction des gardes abstraites implique la disjonction des gardes concrètes [7]. En d'autres termes, ceci assure que le modèle concret ne se bloque pas plus que son abstraction (propriété de vivacité).

2.2 La notation CTT

CTT (Concur Task Tree) [13] est une notation permettant de décrire des modèles de tâches. Une tâche peut être définie comme "*un ensemble d'actions permettant à l'utilisateur de l'interface d'atteindre un but*". Il existe souvent plusieurs chemins (décrivant plusieurs *scénarii* d'interactions) permettant d'atteindre un même but. Un modèle de tâches CTT décrit un ensemble de scénarii d'interactions permettant de décrire les chemins que l'utilisateur peut ou doit suivre pour atteindre un but en utilisant l'interface. Un modèle CTT est basé sur une structure hiérarchique de tâches représentée sous forme d'arbre. Cet arbre permet la description de tâches en combinant un ensemble de tâches atomiques par le biais d'opérateurs temporels. La figure 2 présente une grammaire du langage CTT. On y retrouve les opérateurs temporels classiques d'une algèbre de processus.

$T ::= T >> T$ `-- Activation (séquence)`
$\mid T \; [] \; T$ `-- Choix`
$\mid T \; ||| \; T$ `-- Concurrence`
$\mid T \; |=| \; T$ `-- Ordre indépendant`
$\mid [T]$ `-- Tâche optionnelle`
$\mid T \; [> T$ `-- Désactivation`
$\mid T \; | > T$ `-- Interruption`
$\mid T^* \; [> T$ `-- Désactivation d'un processus infini`
$\mid T^N$ `-- Itération fini d'un processus`
$\mid T_{At}$ `-- Tâche atomique`

FIG. 2. Grammaire du langage CTT.

A titre d'exemple :

- $T_1 >> (T_2[\,]T_3)$ exprime que la tâche T_1 est activée et suivie par l'exécution de T_2 ou de T_3 ;
- $T_1^*[> (T_2|||T_3)$ exprime que l'exécution concurrente de T_2 et T_3 désactive l'exécution infinie de la tâche T_1 ;

De nombreuses expressions peuvent être construites sur la base de cette syntaxe. Ces expressions permettent la description correcte des tâches réalisables par un utilisateur et permettent ainsi de spécifier l'usage qui peut être fait de l'interface.

3 Comportement d'un programme Java-Swing

3.1 Programmes Java-Swing

Le langage Java-Swing. Le langage Java, associé à la bibliothèque Swing, permet la programmation d'applications interactives. La boite à outils Swing définit un ensemble de composants graphiques appelés *widgets* permettant de composer l'apparence et le comportement graphiques de l'IHM : boutons, champs de saisie, barres de défilement... Les widgets sont caractérisés par un ensemble d'attributs. Notamment, tous les widgets définis par la librairie Swing disposent des attributs booléens `enabled` et `visible`. Un widget visible apparaît sur l'interface et l'action de l'utilisateur sur un widget est possible si et seulement si ce widget est visible et actif. L'action d'un utilisateur sur un widget, par exemple un clic sur un bouton, génère un événement au sein du système. Des objets *listeners* peuvent être associés aux *widgets* pour traiter des événements d'une classe donnée. Par exemple, un `ActionListener` écoute des événements du type `ActionEvent`, un `MouseListener` écoute des événements du type `MouseEvent`. Un listener déclare un ensemble de méthodes Java réalisant les traitements pouvant être associés à chaque événement de cette classe. Selon la nature de l'événement produit, l'une ou l'autre de ces méthodes est invoquée. Par exemple, il est possible d'associer un listener `KeyListener` à un widget `JTextField`. Ce `KeyListener` capture des événements de classe `KeyEvent` et déclare plusieurs méthodes dont : `KeyPressed()`, méthode invoquée lors d'une pression sur une touche du clavier, `KeyReleased()` méthode invoquée lorsque l'utilisateur relâche la touche pressée. Le lien entre l'événement généré par une action utilisateur et la méthode de listener invoquée est effectué à l'exécution par la Machine Virtuelle Java (JVM) suivant les attributs de l'événement émis. Parmi ces attributs on trouve : la source de l'événement (le widget sur lequel l'utilisateur agit), le type de cet événement (`MouseEvent`, `KeyEvent`, `ActionEvent`, ...) et un identifiant représentant la nature de cet événement (`KEY_PRESSED`, `KEY_RELEASED`, `MOUSE_CLICK`, ...).

Comportement d'une application Java-Swing. Les événements Java émis à l'exécution lors d'une action utilisateur sont stockés dans une file par la JVM. Le distributeur d'événements (*event dispatcher*) extrait les événements de la file et les distribue au listener associé à la source et au type de l'événement considéré. La méthode du listener à exécuter est alors déterminée en fonction de la nature

de l'événement. Dans ce papier, on considère uniquement des applications Java sans condition de synchronisation entre *threads*. Les événements sont ainsi traités séquentiellement par la JVM. Le comportement d'une application interactive Java-Swing peut donc être vu comme l'enchaînement séquentiel des traitements opérés par les méthodes dont l'invocation est déclenchée par les actions de l'utilisateur sur l'interface. Le choix de la méthode Event-B est motivé par sa capacité à représenter simplement sous forme d'événements ces enchaînements de traitements.

3.2 Exemple : convertisseur Euros/Francs

Un exemple simple, celui d'un convertisseur Euros/Francs, permet d'illustrer ces principes. La figure 3 présente l'interface de cette application et un modèle de tâches CTT vis-à-vis duquel nous souhaitons valider le système.

Description de l'application. L'interface est constituée de 4 widgets. L'utilisateur entre la valeur à convertir dans le champ texte de gauche (Fig.3, tag 1), effectue la conversion en appuyant soit sur le bouton `Franc->Euro` (tag 2) pour convertir en euros, soit sur le bouton `Euro->Franc` (tag 3) pour convertir en francs. Le résultat de la conversion est affiché dans le champ texte de droite (tag 4).

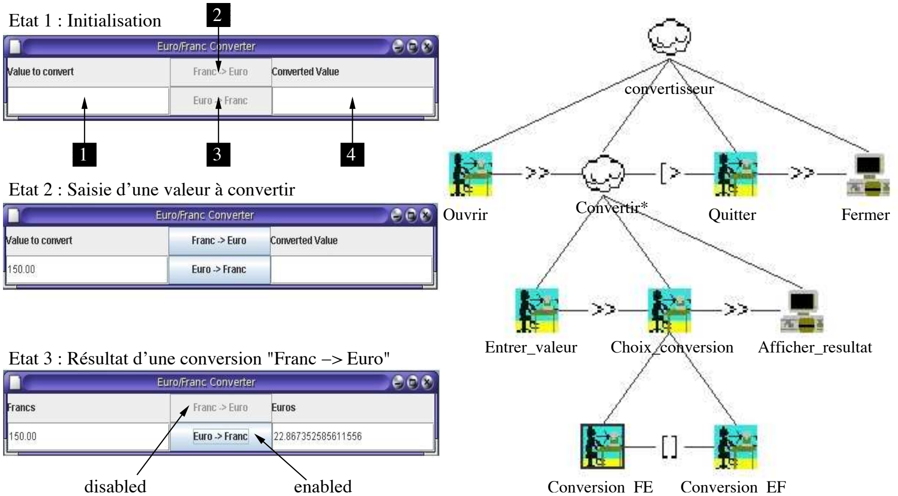


Fig. 3. Exemple : un convertisseur Euros/Francs.

A l'état initial (État 1), le champ de saisie gauche est le seul widget actif (`enabled=true`). Lorsque l'utilisateur modifie la valeur de ce champ texte en utilisant le clavier, les deux boutons de conversion sont activés (État 2) : l'utilisateur peut alors choisir entre les deux conversions autorisées. Quand l'utilisateur appuie sur l'un des deux boutons (État 3) celui-ci est désactivé et le second bouton

devient actif. Le résultat de la conversion est alors affiché sur le champ texte droit.

Description du modèle de tâches. Le modèle de tâches servant de référence à l'application convertisseur est donné par l'arbre CTT de la figure 3. On y remarque trois types de tâche distincts. Le premier, matérialisé par un utilisateur, correspond à une tâche *action utilisateur*. Le second, matérialisé par un ordinateur, correspond à une réaction du système. Enfin, une tâche abstraite de haut niveau est matérialisée par un nuage. En omettant les tâches de type réaction système, le modèle de tâches présenté peut être textuellement formulé par :

$$Ouvrir >> \Big(Entrer_Valeur >> (Convertir_FE[\,]Convertir_EF)\Big)^{*}[> Quitter$$

Il faut noter l'itération infinie de la séquence d'action consistant à entrer une valeur puis à effectuer la conversion. Cette itération est désactivée lorsque l'utilisateur décide de quitter l'application.

4 Vue globale de l'approche : principes et techniques

La figure 4 met en évidence les différentes étapes mises en œuvre. Ces différentes étapes sont matérialisées par les étiquettes noires numérotées. On identifie deux phases principales. La première (Fig.4, tag A) concerne l'extraction d'un modèle formel B à partir du code source de l'application. La seconde (Fig.4, tag B) concerne la phase de validation de l'application exploitant d'une part le modèle formel obtenu, et d'autre part le modèle de tâches CTT exprimant les exigences attendues de l'application.

4.1 Extraction d'un modèle B événementiel

Principes. La phase d'extraction du modèle formel consiste en une forme de projection du code source de l'application sur son comportement (Fig.4, tag 2). Cette projection, obtenue par le biais d'une analyse statique, capture les aspects comportementaux de la partie interactive du système. Cette phase d'analyse statique exploite une représentation B des composants d'interactions (widgets) utilisés pour le développement de l'application. Cette représentation ne peut pas être construite directement à partir du code source de l'application puisque ces composants sont importés dans le code depuis des librairies ou des paquetages externes permettant ainsi de les rendre communs à toutes les applications. On construit donc un modèle B (B_{Swing}) définissant une abstraction des composants de la bibliothèque Swing (Fig.4, tag 1). B_{Swing} représente une ressource qui peut être utilisée par chaque opération de projection. Le modèle B_{ApplM} obtenu capture les aspects comportemental et structurel de l'application. Ce modèle est directement exploitable (Fig.4, tag 6) pour vérifier certaines propriétés de sûreté (cf. exemple section 5).

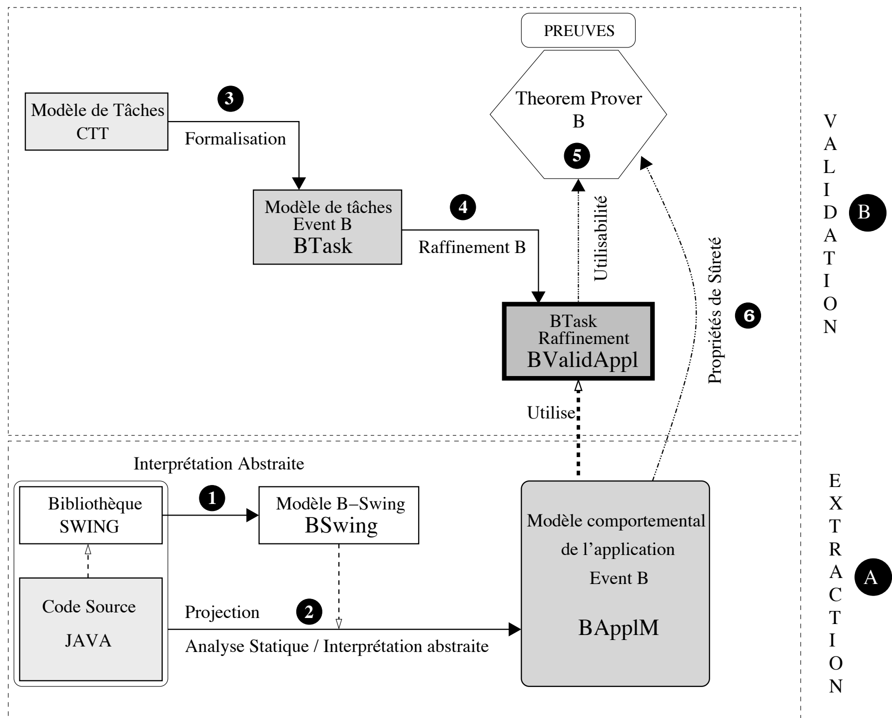


Fig. 4. Principes de l'approche.

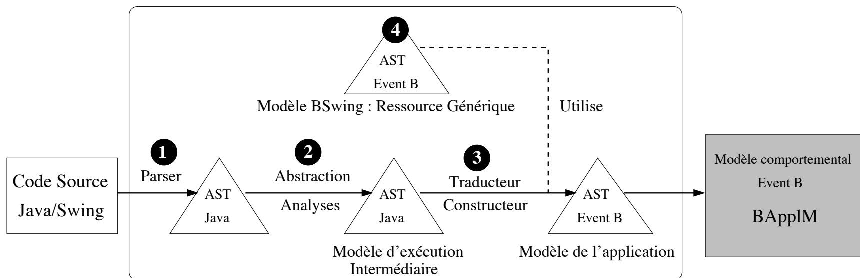


Fig. 5. Analyse statique de programme : technique.

Technique. La figure 5 présente la technique d'extraction utilisée. Une passe d'analyse syntaxique construit un arbre de syntaxe abstrait (AST) Java (Fig.5, tag 1). Cet arbre est exploré et analysé afin d'identifier et capturer uniquement la partie interactive du programme Java complet et d'en construire une abs-

traction prenant la forme d'un nouvel AST Java (Fig.5, tag 2). Des règles de transformation et de traduction prédéfinies permettent de traduire l'AST abstrait obtenu en un AST Event-B (Fig.5, tag 3). Cette étape de traduction fait appel à la ressource B_{Swing} permettant la représentation abstraite des composants d'interactions. Enfin, une dernière étape de *pretty-print* permet d'obtenir le modèle comportemental B_{ApplM} de l'application.

4.2 Validation de l'application vis à vis d'un modèle de tâches CTT

L'objectif est de confronter le comportement de l'application au modèle de tâches reflétant les exigences attendues de l'application. On cherche à montrer que les structures d'interaction encodées dans le programme s'inscrivent bien dans les scénarii d'usage représentés en compréhension dans un modèle de tâches CTT [13]. Pour cela, on formalise le modèle de tâche en un modèle Event-B B_{Task} (Fig.4, tag 3). Ce modèle est raffiné en ajoutant de nouveaux événements et de nouvelles variables (Fig.4,tag 4). Les événements introduits sont les événements issus du modèle extrait B_{ApplM} qui reflètent le comportement de l'application (déclenchement de méthodes Java) lorsqu'elle répond aux actions de l'usager sur l'interface. La preuve de correction du raffinement ainsi que la preuve de vivacité du système, établies à l'aide d'un prouveur de théorème B (Fig.4, tag 5), assurent la correction de l'application vis à vis du modèle de tâche.

5 Extraction d'un modèle B événementiel

L'objectif est d'extraire un modèle formel abstrait capturant l'aspect comportemental de l'interface. Cette étape de projection nécessite de disposer d'une représentation abstraite de la bibliothèque Java-Swing. Après avoir présenté brièvement le modèle *BSwing*, abstraction B de la bibliothèque Java-Swing, nous présentons la mise en application pratique de cette étape d'extraction et de projection.

5.1 Abstraction de la bibliothèque Swing

La figure 6 présente une partie du modèle B_{Swing}. Les types abstraits sont définis dans la clause **SETS** afin de typer les variables du système. Les variables `widgets` et `listeners` dénotent respectivement l'ensemble des instances de widgets et l'ensemble des instances de listeners créées au lancement de l'application (initialisation) et utilisées en cours d'exécution. Les sous-classes (`JFrames`, `JButtons`, `JTextfields`) sont définies comme des sous-ensembles de `widgets`.

Les attributs des instances de widgets sont représentés de manière abstraite par un ensemble de fonctions totales : les fonctions `enabled` et `visible` associent à chaque instance de widgets une valeur booléenne représentant respectivement l'état d'activation et la visibilité courante du widget sur l'interface. La fonction `lists` associe à chaque instance de widget l'ensemble des instances de listeners associées au widget considéré. Les attributs spécifiques à chaque sous-classe de

```
MODEL BSwing
SETS
WIDGETS, LISTENERS, LISTENERS_IDs, EVENTS_IDs, EVENTS_Mods
...
INVARIANTS
widgets ⊆ WIDGETS ∧ JFrames ⊆ widgets ∧ JTextFields ⊆ widgets
∧ JButtons ⊆ widgets...

/% Définition des attributs des widgets %/
∧ enabled ∈ widgets → BOOL ∧ visible ∈ widgets → BOOL
∧ lists ∈ widgets → ℙ(listeners) ∧ value ∈ JTextFields → string∧ ...

∧ listeners ⊆ LISTENERS ∧ lis_ID ∈ listeners → LISTENERS_IDs

∧ UA_source  ∈ widgets ∧ UA_id  ∈  EVENTS_IDs
∧ structural_model_coherence
```

Fig. 6. Une partie du modèle B_{Swing}.

widget peuvent être représentés de la même manière. C'est le cas de l'attribut `value`, représentant la valeur du champ de saisie d'un widget. La fonction totale `lis_ID` associe à chaque instance de listener un attribut représentant l'identifiant du listener (`ActionListener`, `KeyListener`,...).

Les variables préfixées par `UA` modélisent le dernier événement émis en réponse à la dernière action utilisateur sur l'interface : la `source` de l'événement (le widget sur lequel l'utilisateur a agi) et l'identifiant de l'événement (`ActionListener`, `KeyListener`,...). Enfin, un ensemble d'invariants (non décrit dans ce modèle) définit des contraintes structurelles sur les variables. Notamment, nous trouvons dans cet ensemble un invariant exprimant qu'une instance de widget appartient nécessairement à une des sous-classes définies (`JFrames`, `JButtons`, `JTextfields`).

5.2 Capturer le comportement de l'interface.

Les éléments d'information présents dans le code source et capturant le comportement de la partie interactive du système sont : **(1)** les instances de widgets et de listeners créées lors de l'initialisation de l'application ainsi que les liens qui les unissent. L'analyse statique de la méthode `main()` du programme ainsi que des méthodes appelées par celle-ci permet de retrouver et d'extraire ces informations ; **(2)** les méthodes de listeners qui codent la réaction de l'interface en réponse à une action utilisateur. L'exécution d'une méthode listener modifie l'état interne du système. Or du point de vue de l'interaction, ce sont ces modifications, affectant les attributs et donc l'état d'activité des widgets, qui sont pertinentes et méritent d'être retenues. La partie purement fonctionnelle de l'application, qui ne structure pas directement l'interaction, est abstraite. Par exemple, dans le cas du convertisseur, la méthode listener associée à l'événement "presser bouton `Franc->Euro`" fait nécessairement appel à une méthode calculant le résultat de la conversion. La méthode de calcul du résultat n'influant pas directement sur l'aspect interactif du système, elle est abstraite.

Représentation des méthodes listeners. La méthode Event-B permet la représentation de l'exécution d'un code linéaire, constitué d'instructions d'affectations dont l'ordre d'exécution est dirigé par des structures de contrôle du langage : séquencement, conditionnelles, itérations. Le corps d'une méthode Java peut faire appel à des d'instructions plus complexes comme les appels de méthodes. Afin de réaliser la traduction du corps d'une méthode listener en un système d'événements B une étape de linéarisation est effectuée en remplaçant les invocations de méthodes dans ce corps par le code des méthodes invoquées (*in-lining*). Une fois cette opération effectuée, chaque méthode listener est représentée par un ensemble d'événements B. Concrètement, chacune des instructions constituant le code de la méthode est traduite par un événement B. Des variables de contrôle permettent d'ordonnancer l'exécution des événements afin de refléter l'ordre d'exécution des instructions du code. Cette étape de traduction exploite les règles de transformation définies par J.R Abrial [1] dans un mode ascendant. Dans la pratique, une analyse de dépendances des variables est effectuée pour regrouper de façon sûre plusieurs instructions dans le corps d'un même événement B. Ces instructions sont alors représentées par un ensemble de substitutions mises en parallèle : l'ensemble des affectations présentes dans le corps de l'événement sont alors effectuées simultanément.

```
public void
 actionPerformed(ActionEvent e)
  {
   output.setVisible(true);

  if (e.getSource()==EF){
     EF.setEnabled(false);
     FE.setEnabled(true);
     result=convert(
        input.getText(),true)
     output.setText(result);}
   else {
     FE.setEnabled(false);
     EF.setEnabled(true);
     result=convert(
        input.getText(),false);
     output.setText(result);}
  }
}
```

FIG. 7. Code Java/Swing de l'application convertisseur.

```
actionPerformed_list1_1=
SELECT
  UA_source=EF ∧
  list1 ∈ lists(UA_source) ∧
  list_ID(list1)=ActionListener ∧
  UA_id=actionPerformed ∧
THEN
  visible(output):=true ||
  enabled(EF) := true ||
  enabled(FE) := false ||
  value(output) := result
END ;

actionPerformed_list1_2 =
SELECT
  not(UA_source=EF) ∧
  list1 ∈ lists(UA_source) ∧
  list_ID(list1)=ActionListener ∧
  UA_id=actionPerformed ∧
THEN
  visible(output):=true ||
  enabled(EF) := false ||
  enabled(FE) := true ||
  value(output) := result
END
```

FIG. 8. Modèle comportemental de l'application.

5.3 Exemple

La figure 7 propose une partie du code définissant l'application convertisseur exposée en section 3.2 et plus particulièrement la méthode listener invoquée lorsque l'utilisateur presse l'un des deux boutons de conversion `FE` ou `EF`. Les deux boutons sont associés à un listener `list1` de type `ActionListener` écoutant les événements de type `ActionEvent` émis lorsque l'utilisateur presse l'un des deux boutons. `output` représente le champ de saisie permettant d'afficher le résultat de la conversion.

La figure 8 présente le modèle B extrait du code présenté en figure 7. La méthode `actionPerformed`, après abstraction et analyse de dépendance des variables, est traduite par deux événements B dans la clause **EVENTS**. Ces deux événements ont une partie de leur garde en commun. Cette garde commune G_{aP} exprime que les événements sont déclenchés si et seulement si :

- le listener `list1` définissant la méthode est bien associé au widget source de l'événement ;
- le listener `list1` est du type `ActionListener` ;
- et enfin, l'action effectuée par l'utilisateur est bien du type `actionPerformed`.

La conditionnelle exprimée dans le code source de la méthode `actionPerformed` est traduite par la partie non commune des gardes des deux événements : le premier est déclenché si `UA'source=EF` et le second lorsque `not(UA'source=EF)`. La méthode `convert`, appartenant au noyau fonctionnel de l'application, est abstraite : dans le code B, la valeur du champ `output` prend la valeur constante `result` définie dans la clause **CONSTANTS** du modèle (non représentée), indiquant ainsi qu'elle a restitué un résultat.

Validation : propriétés de sûreté. Sur l'exemple du convertisseur, l'extraction du modèle B_{ApplM} conduit à la définition de 7 événements. Plusieurs propriétés de sûreté, définies dans la clause **INAVRIANT**, ont pu être prouvées sur cet exemple :

1. `enabled(output)=true` $\wedge$ `visible(input)=true`
2. `value(input)=no_text` $\Rightarrow$ `(enabled(EF)=false` $\wedge$ `enabled(FE)=false)`
3. `value(input)=text` $\Longleftrightarrow$ `(enabled(EF)=true` $\vee$ `enabled(FE)=true)`
4. `value(output)=result` $\Rightarrow$ `(enabled(EF)=true` $\wedge$ `enabled(EF)=false)`
   $\vee$ `(enabled(EF)=true` $\wedge$ `enabled(FE)=false)`

Les invariants précédents énoncent respectivement les propriétés de sûreté suivantes : **(1)** le champ de saisie `input` est toujours actif et visible ; **(2)** si le champ de saisie est vide, alors les deux boutons `EF` et `FE` sont toujours inactifs ; **(3)** lorsque le champ de saisie est non vide (i.e. l'utilisateur indique une valeur à convertir) alors l'un des deux boutons `EF` ou `FE` est actif ; **(4)** enfin, lorsque le champ de saisie `output` est non vide (i.e. un résultat est affiché) alors nécessairement l'un des deux boutons est actif et le second est inactif.

Pour prouver les invariants et les assertions du modèle B_{ApplM} (propriétés de sûreté et propriété de vivacité) 69 Obligations de Preuves (OPs) ont du être

déchargées. Les outils B4free et l'interface Click'n'Prove [3] ont été utilisés à cet effet. 20 OPs ont du être déchargées interactivement. Ces OPs ne présentent aucune difficulté et peuvent être prises en charge par un utilisateur peu expérimenté.

6 Validation d'applications Java/Swing

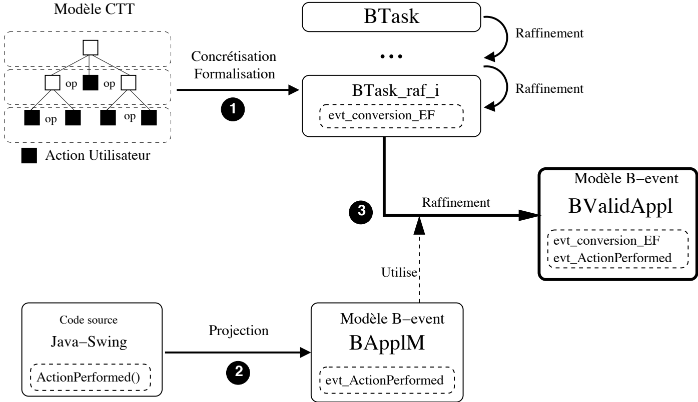


Fig. 9. Validation de l'application : technique.

La figure 9 présente les principes de la validation opérée. Celle-ci comporte deux étapes principales : une étape de formalisation et de concrétisation du modèle de tâches (Fig.9, tag 1), et une étape de couplage des modèles B_{Task} et B_{ApplM} (Fig.9, tag 3).

6.1 Concrétisation et Formalisation du modèle CTT

Dans un premier temps, le modèle CTT est concrétisé puis encodé en un modèle Event-B B_{Task}. La concrétisation est à la charge du développeur et consiste à ajouter certaines informations concrètes aux feuilles de l'arbre CTT qui représentent les actions utilisateur. Deux informations sont ajoutées : le widget sur lequel l'utilisateur agit et le type de l'action effectuée. Cette étape permet par la suite d'automatiser le couplage entre les modèles B_{ApplM} et B_{Task}. Afin de décomposer l'effort de preuve, la formalisation du modèle CTT s'effectue en plusieurs étapes de raffinement. La figure 10 présente un exemple de concrétisation sur la tâche `ChoixConversion`, sous-tâche extraite du modèle de tâches complet présenté en figure 3.

La figure 11 présente la formalisation B de la tâche `ChoixConversion`. L'opérateur choix `[]` du modèle CTT est traduit par trois événements B. Le premier

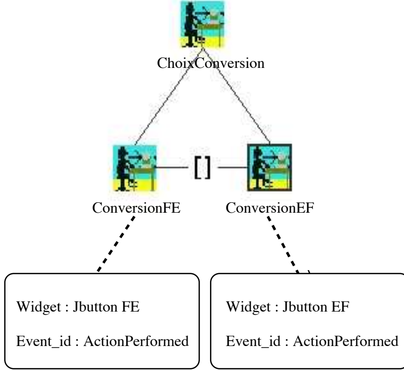


Fig. 10. Concrétisation de la tâche *ChoixConversion.*

```
ChoixInit=
ANY p
WHERE
p ∈ {1,2}
∧ V_Choix=3 ∧ G_Init
THEN
V_Choix := p
END;

ChoixConversion=
SELECT
V_Choix=0
∧ G_ChoixConv
THEN
S0
END;

ConversionEF=
SELECT V_Choix=1
∧ G_ConvEF
THEN
V_Choix := 0 ||
UA_source :=EF ||
UA_id :=actionPerformed
END;

ConversionFE=
SELECT V_Choix=2
∧ G_ConvFE
THEN
V_Choix := 0 ||
UA_source :=FE ||
UA_id :=actionPerformed
END;
```

Fig. 11. Formalisation de la tâche *ChoixConversion.*

événement `ChoixInit` est constitué d'une substitution indéterministe assignant à la variable d'ordonnancement V_{Choix} la valeur 1 ou 2. L'événement `ConversionFE` ou `ConversionFE`, suivant la valeur de la variable d'ordonnancement, peut alors être déclenché. Chacun de ces deux événements affectent au variant V_{Choix} la valeur 0, redonnant ainsi la main à l'événement abstrait `ChoixConversion`. La substitution de ce dernier événement S_0 dépend du contexte dans lequel est plongé la tâche correspondante. Notamment cette substitution fait décroître une variable de contrôle permettant de redonner la main à d'autres événements caractérisant des tâches du modèle CTT global. Enfin les gardes G_{ConvFE} et G_{ConvEF} déterminent une condition nécessaire stipulant qu'une action utilisateur sur le bouton `FE` (respectivement `EF`) est possible si et seulement si ce widget est actif et visible sur l'interface. Les règles de formalisation d'un modèle CTT en système d'événements B ont été étudiées par exemple en [4].

6.2 Validation : couplage des modèles concret et abstrait

Prouver que l'application Java/Swing se conforme au modèle de tâches CTT revient à prouver que le modèle concret B_{ApplM} est un raffinement correct du modèle abstrait B_{Task}. Pour ce faire, le modèle B_{Task} est raffiné. Ce raffinement introduit de nouveaux événements issus du modèle B_{ApplM} qui reflètent l'exécution opérée par le système en réponse à une action utilisateur. L'événement `ConversionFE` issu du modèle de tâches CTT B_{Task} est ainsi raffiné par l'adjonction de l'événement `Action_Performed_lis1_1` qui représente l'exécution de la méthode listener déclenchée par une action de l'utilisateur sur le bouton `FE`.

La figure 12 présente le modèle raffiné obtenu $B_{ValidAppl}$. Seule la sous-tâche `ChoixConversion` est ici validée. Le couplage est effectué par l'introduction de

```
REFINEMENT BValidAppl
REFINES BTask
...
ASSERTIONS

G_BTask ⟹ (G_ConversionFE ∨ G_ConversionFE ∨ G_ChoixInit ∨ G_ChoixConversion ∨
           G_actionPerformed_list1_1 ∨ G_actionPerformed_list1_2 ∨ ...)
∧ ...

INITIALISATION

V_Choix := 3 || V_EF := 2 || V_FE := 2 ...

EVENTS

ChoixInit=
 ANY p
 WHERE
   p ∈ {1,2} ∧ V_Choix=3 ∧ G_Init
 THEN
   V_Choix := p
 END ;

ChoixConversion=
 SELECT
   V_Choix = 0 ∧ G_ChoixConv ∧
   (V_FE=0 ∨ V_EF=0)
 THEN
   S_0 || V_EF := 2 || V_FE := 2
 END ;

ConversionFE=
 SELECT
   V_Choix=1 ∧ V_FE=2
   enabled(FE)=true ∧
   visible(FE)=true
 THEN
   V_FE :=1 || V_Choix :=0 ||
   UA_source := FE ||
    UA_id := actionPerformed
 END ;

ConversionEF=
 SELECT
   V_Choix=2 ∧ V_EF=2
   enabled(EF)=true ∧
   visible(EF)=true
 THEN
   V_EF :=1 || V_Choix :=0 ||
   UA_source := EF ||
   UA_id := actionPerformed
 END ;

actionPerformed_list1_1=
 SELECT
   UA_source=EF ∧
    V_EF=1 ∧ V_Choix=0 ∧ G_aP
 THEN
   V_EF :=0
   visible(output):=true ||
   enabled(EF) := false ||
   enabled(FE) := true ||
   value(output) := result
 END;

actionPerformed_list1_2=
 SELECT
   not(UA_source=EF) ∧
   V_FE=1 ∧ V_Choix=0 ∧ G_aP
 THEN
   V_FE :=0
   visible(output):=true ||
   enabled(FE) := false ||
   enabled(EF) := true ||
   value(output) := result
 END;

END ;
```

Fig. 12. Modèle de validation

deux nouveaux variants V_{EF} et V_{FE}. Du fait de l'utilisation du B classique pour encoder des modèles B événementiels, une assertion est ajoutée au modèle afin de s'assurer de la vivacité du modèle concret vis à vis du modèle abstrait. Cette assertion définie dans la clause **ASSERTIONS** du modèle stipule que la disjonction des gardes abstraites (représentée par G_{BTask}) implique la disjonction des gardes du modèle concret, assurant ainsi que le modèle concret ne se bloque pas plus que son abstraction.

6.3 Résultats expérimentaux

Cette technique de validation a été utilisée avec succès sur l'exemple du convertisseur en utilisant le modèle de tâches complet présenté en figure 3. L'ensemble des étapes de formalisation et de couplage a conduit à la réalisation de 4 modèles (2 raffinements pour la formalisation du modèle de tâches, 1 raffinement pour l'obtention du modèle $B_{ValidAppl}$). Au total 136 OPs ont été générées par les outils B parmi lesquelles 124 OPs ont été déchargées automatiquement.

Le couplage des modèles B_{Task} et B_{ApplM} constitue la principale difficulté de l'approche. Si la concrétisation du modèle de tâches peut permettre une semi-automatisation de la démarche, l'intervention explicite d'un développeur reste nécessaire. Notamment, la preuve de vivacité du modèle $B_{ValidAppl}$ nécessite l'introduction d'invariants de collage permettant de lier le modèle concret à son abstraction. Dans l'exemple du convertisseur, 5 invariants de collage ont du être ajoutés au modèle $B_{ValidAppl}$. L'adjonction de ces invariants est fortement guidée par les OPs générées, et par conséquent la définition de ces invariants ne peut pas être automatisée. Enfin, notons que la validation de l'application concerne uniquement l'ensemble des scénarios décrit par le modèle de tâches. Dans le cas du convertisseur, certains comportements de l'interface ne sont pas pris en compte par le modèle de tâches : nous ne pouvons alors rien affirmer quant à la validité de ces comportements. Par exemple le scénario de conversions `(ConversionFE>>ConversionEF)*` est réalisable sur l'interface mais non mentionné dans le modèle de tâches CTT.

7 Conclusion

Ce papier a présenté une approche exploitant la méthode B événementielle pour la validation formelle de systèmes interactifs. L'objectif est de démontrer que le comportement du système implémenté est conforme aux scénarii d'interaction spécifiés en compréhension par un modèle de tâches CTT. On extrait pour cela un modèle comportemental B de la partie interactive du système par analyse statique de son code source, et on prouve que ce modèle est un raffinement correct du modèle de tâche préalablement formalisé en B. On permet ainsi de conserver les techniques de développement habituellement utilisées (réutilisation de code, outils de développement graphique, génération de code) et on peut exploiter cette approche tout au long du cycle de développement du système.

Cette approche de validation a été utilisée avec succès sur un exemple simple programmé en Java/Swing. La technique présentée peut cependant être utilisée avec d'autres langages de programmation. A l'exception de la concrétisation du modèle de tâches, du raffinement $B_{ValidAppl}$ et des éventuelles preuves manuelles, la démarche proposée peut être menée à l'aide d'outils automatiques. A l'heure actuelle seules des applications Java/Swing non-multimodales et sans contraintes de synchronisation entre threads ont été traitées.

Des outils permettant d'automatiser les passes d'analyse et de traduction sont en cours de développement. Un enrichissement de la méthode pour la validation

d'applications multi-modales est également envisagé dans la suite de ces travaux. Un élargissement du spectre des exigences à valider est également envisagé : il permettrait de prendre d'autres modèles de références que les seuls arbres CTT. Il est alors nécessaire de définir un ensemble de projections permettant d'obtenir différentes vues du système, chaque vue permettant la validation d'un aspect particulier du système au regard de la modélisation des exigences afférentes au point de vue adopté.

Références

1. J. R. Abrial. Event Based Sequential Program Development : Application to Constructing a Pointer Program. In *FME*, pages 51–74, 2003.
2. J.R. Abrial. *The B-Book : Assigning Programs to Meanings.* Cambridge University Press, aug 1996.
3. J.R. Abrial and D. Cansell. *Click'n Prove : Interactive Proofs within Set Theory.* Springer, lecture notes in computer science : theorem proving in higher order logics edition, 2003.
4. Y. Aït-Ameur, M. Baron, and N. Kamel. Encoding a Process Algebra using the Event B Method. Application to the Validation of User Interfaces. In *ISOLA 2005*, Columbia, USA, September 2005. Springer-Verlag.
5. B. Ausbourg(d'). Using Model Checking for the Automatic Validation of User Interfaces Systems. In P. Markopoulos and P. Johnson, editors, *Proceedings of Design, Specification and Verification of Interactive Systems '98*, Abingdon, UK, June 1998. Eurographics, Springer-Verlag.
6. L. Bass, R. Pellegrino, S. Reed, R. Seacord, S. Sheppard, and M. R. Szcezur. The Arch Model : Seeheim Revisited. In *CHI 91 User Interface Developper's Workshop*, 1991.
7. D. Cansell. Assistance au développement incrémental et à sa preuve. Habilitation à diriger des recherches, Université Poincaré, Avril 2003.
8. D.J. Duke and M.D. Harrison. Event Model of Human-System Interaction. Software Engineering Journal,pp. 3-12, January, 1995.
9. A. Memon, I. Banerjee, and A. Nagarajan. GUI Ripping : Reverse Engineering of Graphical User Interfaces for Testing. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE '03)*, volume 0, page 260, Los Alamitos, CA, USA, 2003. IEEE Computer Society.
10. M. Moore. Rule-Based Detection for Reverse Engineering User Interfaces. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96)*, page 42, Washington, DC, USA, 1996. IEEE Computer Society.
11. D. Norman. *User Centered System Design.* Lawrence Erlbaum Associates, 1986.
12. P. Palanque, R Bastide, and V. Sengès. Validating Interactive System Design through the Verification of Formal Task and System Models. In Leonard J. Bass and Claus Unger, editors, *Working Conference on Engineering for Human-Computer Interaction (EHCI'95)*, pages 189–212. Chapman & Hall, USA, 1995.
13. F. Paternò. *Model-Based Design and Evaluation of Interactive Applications.* Springer-Verlag, London, UK, 1999.

14. P. Roché. *Modélisation et Validation d'Interface Homme-Machine*. PhD thesis, Ecole Nationale Supérieure de l'Aéronautique et de l'Espace (ENSAE - SUPAERO), 2002.

15. ISO/TC159 Sub-Commitee SC4. Draft International ISO DIS 9241-11 Standard, September 1995.

16. J.C. Silva, J.C. Campos, and J. Saraiva. Models for the Reverse Engineering of Java/Swing Applications. In J. M. Favre, D. Gasevic, R. Lämmel, and A. Winter, editors, *3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies (ateM 2006) for Reverse Engineering*, number 1/2006 in Informatik-Bericht series. Johannes Gutenberg-Universität Mainz, Institut für Informatik – FB 8, October 2006. ISSN : 0931-9972.

17. T. Systa. Dynamic Reverse Engineering of Java Software. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 174–175, London, UK, 1999. Springer-Verlag.

Une approche incrémentale combinant test et extraction de modèles

Roland Groz[2], Muzammil Shahbaz[1], and Keqin Li[2]

[1] France Telecom R&D
Meylan, France.
muhammad.muzammilshahbaz@orange-ftgroup.com
[2] LIG, Computer Science Lab
Grenoble Universités, France
{Keqin.Li,Roland.Groz}@imag.fr

Un des principaux obstacles à l'utilisation d'approches formelles est l'absence de modèles. Nous nous intéressons ici au problème de l'intégration et du test de composants logiciels. Dans la pratique industrielle actuelle, il est de plus en plus fréquent de travailler en intégrant des composants logiciels externes dont aucun modèle n'est disponible. La documentation fournit en général une spécification insuffisante.

Afin de guider l'intégration, nous proposons d'utiliser des algorithmes d'inférence de machines pour extraire des modèles des composants. L'interaction entre ces modèles dans une composition permet de déduire d'autres tests qui peuvent être confrontés au système. L'ingénieur chargé de l'intégration dispose ainsi d'outils lui permettant de découvrir les interactions effectives entre les composants et de guider son processus d'intégration et de test pour valider les combinaisons de comportements. Nous travaillons sur des modèles d'automates étendus avec des paramètres et des prédicats, mais sans variables internes.

1 Introduction

Alors que le développement logiciel était pendant longtemps une activité intégrée au sein d'une entreprise qui fabriquait l'ensemble de ses produits logiciels, on procède de plus en plus par assemblage de composants provenant de sources externes.

Cette évolution implique de nouveaux défis pour intégrer les approches formelles dans le développement logiciel du côté de l'intégrateur. D'abord, les composants externes sont rarement accompagnés de spécifications formelles. Plus largement, on est confronté à l'absence ou à la non-disponibilité des modèles ou des éléments qui ont permis de construire ces composants. En général, on dispose d'une documentation qui n'est pas forcément suffisamment précise pour répondre aux questions que se pose l'intégrateur, et qui ne peut pas être intégrée directement dans les processus de développement de l'assemblage.

Nous proposons une approche formelle assistant l'intégrateur de composants dans sa tâche d'expérimentation et de test de composants dans une architecture définie a priori. Nous voulons permettre une approche de *test* basée sur

des *modèles* (avec génération automatique de tests), même en l'absence de modèles initiaux pour les composants. Pour cela, nous combinons une approche descendante de génération de tests (pour l'assemblage) à partir de modèles avec une approche ascendante de reconstruction de modèles (de chaque composant) à partir des observations issues du test.

L'objectif est de permettre un test suffisant pour mettre en évidence les interactions des composants dans les divers cas d'usage de l'assemblage. On ne cherche donc pas à dériver un modèle complet des composants, mais une *approximation* en phase avec l'objectif. En général, dans un assemblage, seule une petite partie du fonctionnement d'un composant est sollicitée. Comme le modèle correspondant à cette partie sera dérivé des tests, la génération de tests sur un seul composant ne serait pas pertinente pour découvrir des erreurs (puisque les tests seraient déduits du composant lui-même, donc corrects par construction). C'est pourquoi notre approche prend tout son sens dans un contexte *d'intégration*, où les modèles permettent de déduire de nouvelles interactions à tester. En outre, l'expérience montre que les problèmes d'interfonctionnement entre composants proviennent souvent du traitement de certaines valeurs échangées, alors qu'en général les interfaces sont cohérentes puisqu'elles ont bien été étudiées dans la définition de l'architecture d'intégration. C'est pourquoi nous accordons une importance particulière à l'extraction de modèles paramétrés : chaque interaction entre composants porte également des valeurs.

1.1 Une approche globale

Nous travaillons sur une représentation des composants par des automates communicants. La communication se fait par des symboles d'entrées-sorties, représentant un type d'interaction, enrichis par des paramètres représentant les valeurs échangées lors de l'interaction. Les symboles correspondent par exemple aux primitives décrites dans la documentation du composant.

Nous faisons les hypothèses suivantes.

- Les composants sont des *boîtes noires*. On connaît leurs interfaces, c'est à dire les types des entrées, et accessoirement de leurs sorties. Il est possible qu'on n'en connaisse qu'un sous-ensemble. On connaît par exemple les primitives de services offertes par le composant et le type de leurs paramètres. C'est ce dont disposerait un ingénieur avec la documentation du composant.
- Nous nous plaçons dans une hypothèse de fonctionnement *réactif* du système. Les entrées globales ne seront fournies au système que dans un état stable de celui-ci, où il n'y a plus d'interaction interne possible. On supposera aussi que même si la communication entre composants du système est asynchrone, il n'y a qu'un flot de traitement actif (un seul message en transit dans le système).
- On dispose de *scénarios d'usage* du système permettant d'avoir un ensemble de séquences d'entrées et des valeurs significatives des paramètres associés. L'ingénieur qui a conçu l'architecture pour composer un service à partir des composants aura aussi élaboré des scénarios d'usage typiques,

et il s'agit, maintenant que l'architecture est définie, de voir si les composants interagiront bien selon les attentes qu'on en a. Un apport clé de notre approche va être de permettre l'enrichissement des tests, et la déduction automatique de tests systématiques à partir d'un nombre restreint de scénarios de test.

- Les composants sont intégrés, mais peuvent être aussi testés isolément. Lorsqu'ils sont intégrés, certaines de leurs interfaces sont connectées entre elles (interfaces internes), d'autres restent ouvertes pour communiquer avec l'environnement. On suppose que les interfaces internes restent *observables* (mais pas forcément contrôlables).
- Accessoirement, il se peut qu'on dispose de modélisations très partielles, sous forme d'automates à entrées et sorties (paramétrées), de certains composants, représentant par exemple leur flot de contrôle pour certains scénarios. Notre approche saurait intégrer de tels modèles, mais en leur absence, on peut reconstruire des modèles à partir de rien.

Notre approche va consister à tester d'abord chaque composant selon un *algorithme d'inférence d'automate* qui permettra d'en dériver un premier modèle cohérent avec les observations sur un nombre restreint d'entrée sorties correspondant à celles des scénarios d'usage fournis initialement. Ensuite, nous assemblerons les modèles, et dériverons des tests d'intégration nous permettant de confronter le fonctionnement du système réel aux modèles préliminaires. Ceci nous permettra d'enrichir ces modèles et de déduire de nouveaux test correspondants aux cas "croisés", d'interactions entre composants qui n'avaient pas été testées dans les scénarios antérieurs. Nous avons donc une approche incrémentale et itérative alternant les phases de test avec les phases d'enrichissement des modèles. Le processus peut être arrêté lorsqu'on ne détecte plus d'incohérence entre les modèles et le système ou lorsqu'on a atteint un certain objectif de couverture du système.

Dans la suite de cet article, nous présentons d'abord un exemple simple permettant d'illustrer les algorithmes d'inférence d'automates. Ensuite, nous illustrons l'algorithme lorsqu'on se restreint à des automates sans paramètres (machines de Mealy). Nous étendons le modèle pour prendre en compte les paramètres et expliquons comment l'algorithme peut être étendu. Enfin, nous présentons la méthode d'intégration qui s'appuie sur les algorithmes d'inférence.

2 Un exemple de composant à identifier

Nous donnons un exemple de contrôleur de climatiseur qui régule un système de chauffage et de ventilation. La structure interne du contrôleur et le fonctionnement du système encapsulé par ce composant ne sont pas connus. Il s'agit donc d'un composant considéré comme une boîte noire. Par contre, on peut connaître un ensemble d'entrées en analysant ses interfaces externes. La figure 1 présente un diagramme global du contrôleur.

Le contrôleur de climatiseur accepte des entrées venant de l'environnement pour contrôler le système. Il reçoit les signaux *Marche* et *Arrêt* pour allumer

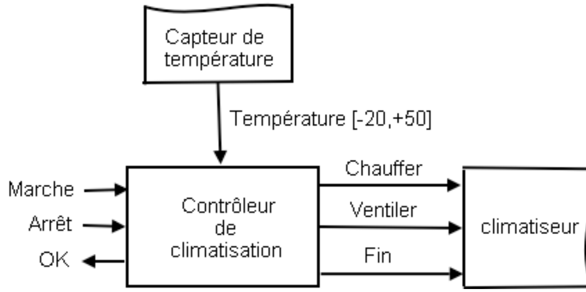


FIG. 1. Un diagramme global de climatiseur

et éteindre le système et Température T qui change le mode du système. Il accepte des valeurs de température entre -20 Celsius et +50 Celsius, qui sont des paramètres de l'entrée T. Ensuite, il donne une commande *Chauffer* pour mettre en route le chauffage s'il fait froid (entre -20°Cet 11°C) ou *Ventiler* pour la ventilation s'il fait chaud (entre 16°Cet 50°C). La commande de sortie porte des paramètres pour régler la vitesse de système, par exemple, s'il ne fait pas très froid, le contrôleur envoie une commande pour ralentir le chauffage. De même, il envoie une commande pour relancer la climatisation s'il fait très chaud, sinon, il peut suffire de ventiler.

Cet exemple classique va nous servir à illustrer le fonctionnement des algorithmes d'inférence de modèles. Dans un premier temps, nous montrerons comment on peut en apprendre un modèle d'automate simple, en faisant une abstraction des paramètres. Puis nous étendons l'algorithme pour en apprendre un modèle paramétré.

3 Apprentissage d'automates simples

Le modèle d'automate que nous considèrerons sera celui des automates à entrées et sorties (appelés aussi machines de Mealy). Un automate est un sextuplet $M = (Q, I, O, \delta, \lambda, q_0)$, où Q est l'ensemble des états, $q_0 \in Q$ est l'état initial, I l'alphabet d'entrée, O celui des sorties, $\delta : Q \times I \longrightarrow Q$ est la fonction de transition d'états et $\lambda : Q \times I \longrightarrow O$ est la fonction de sortie. Les ensembles Q, I, O sont bien sûr finis. On s'intéressera à des automates complètement définis, c'est à dire tels que $dom(\delta) = dom(\lambda) = Q \times I$. Au besoin, on complètera pour cela l'automate avec un symbole supplémentaire Ω qui sera une abstraction pour la réponse d'un système à des entrées non valides dans un état donné, sur lequel on rajoutera une boucle étiquetée par Ω en sortie.

Nous cherchons à déduire la structure de contrôle d'un composant qu'on peut tester en boîte noire. Pour cela, nous nous sommes intéressés aux travaux similaires menés soit dans le domaine de l'ingénierie à partir de scénarios [13] [15] [4], soit en vérification [5] [6] ou en test [8] [9]. Dans ce domaine, l'algorithme de base le plus efficace dans ce contexte de test actif est l'algorithme d'Angluin [1]. C'est

un algorithme de complexité polynomiale (en nombre de requêtes au système boîte noire) conçu pour des automates accepteurs déterministes. Notre travail actuel sur l'inférence de machines se situe dans la lignée d'adaptations et d'extensions à l'algorithme d'Angluin pour des modèles comportant des entrées et des sorties avec des paramètres et des prédicats.

Dans la plupart des travaux antérieurs, l'algorithme d'Angluin a été utilisé sans modification avec une simple correspondance entre les symboles d'entrées et de sortie et l'alphabet A d'un automate accepteur. Par exemple, en prenant $A = I \cup O$ [7] [9], ou en prenant des couples (entrée,sortie) $A = I \times O$ [13] comme lettres de l'alphabet. Nous avons proposé une adaptation de l'algorithme dans laquelle on remplace la notion d'acceptation (décision binaire, codée par 0 et 1 dans l'algorithme d'Angluin) par les sorties correspondant aux derniers symboles entrés [11]. Par ailleurs, l'algorithme d'Angluin a été initialement proposé dans le contexte d'un apprentissage dans lequel un oracle connaissant le contenu de la boîte noire peut être interrogé de deux façons.

- par une requête d'appartenance, qui permet de savoir si une séquence d'entrées est un mot accepté (reconnu) par l'automate ; dans notre approche, c'est le composant qui servira d'oracle et fournira les sorties correspondant à la séquence d'entrée
- par une requête d'équivalence, qui permet de savoir si l'automate minimal cohérent avec les observations enregistrées qu'on a pu reconstruire est équivalent à l'automate caché dans la boîte noire ; s'il ne l'est pas, l'oracle fournit un contre-exemple, i.e. une séquence reconnue par la boîte noire mais pas par l'automate construit ; dans notre approche, il n'y a pas de tel oracle omniscient, c'est le test d'intégration qui fournira un éventuel contre-exemple, ou le critère d'arrêt du test qui terminera l'algorithme sur un modèle approché.

Noter qu'on supposera toujours que le composant est réinitialisable ce qui permet de reprendre toutes les séquences de test à partir de l'état initial.

3.1 Algorithme pour machine de Mealy

Nous ne décrivons pas ici le détail de l'algorithme qui a été présenté ailleurs [11], mais nous en donnons les principes et une illustration sur l'exemple du climatiseur. Conformément à l'algorithme d'Angluin, on note les observations faites au cours du test dans une table appelée table d'observation, qui sera étendue progressivement au fur et à mesure des observations. Nous notons cette table (S, E, TO). Les lignes de cette table sont étiquetées par des séquences d'entrées, qui correspondent à des préfixes pour atteindre les états de l'automate. Soit S l'ensemble de ces séquences. On impose à S d'être clos par préfixe. Les colonnes sont elles aussi étiquetées par des séquences, d'un ensemble E auquel on imposera d'être clos par suffixe.

A l'intersection d'une ligne s de S et d'une colonne e de E on portera dans la table la valeur $TO(s, e)$ qui sera la séquence des sorties produites par le composant en réponse à la séquence e, lorsqu'il a été préalablement positionné dans un état par la séquence s.

On commence l'algorithme avec $S = \{\epsilon\}$ (le mot vide) et $E = I$ (l'ensemble des symboles d'entrées du composant). On remplit la première (et seule) ligne avec les résultats des tests consistant à appliquer chacune des entrées dans l'état initial : dans chaque case, on trouve donc la (ou les) sortie(s) correspondant à cette entrée (celle de la colonne de cette case).

On vérifie ensuite la "complétude" de la table en rajoutant toutes les lignes correspondant à $S \cdot I$. A tout moment, dans l'algorithme, il y aura ainsi deux parties dans la table : la partie haute, des lignes étiquetées par S et la partie basse qui étend celles-ci d'une entrée, pour toutes les entrées possibles. Un exemple de table d'observation est donné en figure 2.

		E	
		e1	e2
S	ε	o1	o2
S•I	s1	o1	o3
	s2	o2	o1

Fig. 2. Structure d'une table d'observation

La table sera dite *close* si toute ligne t de $S \cdot I$ est le doublon d'une ligne de S, i.e ; s'il existe s de S tel que $ligne(s) = ligne(t)$. Elle sera dite *cohérente* si lorsque deux éléments de S, s_1 et s_2 ont des lignes identiques, alors pour toute entrée a de I, $ligne(s_1 \cdot a) = ligne(s_2 \cdot a)$.

Lorsqu'on arrive à un état où la table est *close* et *cohérente* , comme illustré sur la table 1, alors on construit un modèle minimal d'automate (la conjecture) qui permet d'expliquer toutes ces observations de la façon suivante.

- $Q = \{ligne(s) : s \in S\}$
- $q_0 = ligne(\epsilon)$
- $\delta(ligne(s), a) = \{ligne(s \cdot a), a \in I\}$
- $\lambda(ligne(s), a) = \{TO(s \cdot a), a \in I\}$

Chaque séquence préfixe $s \in S$ est un chemin d'accès qu'on associe à un état. Deux séquences peuvent conduire au même état, et la propriété de cohérence garantit qu'alors les observations ultérieures restent cohérentes (pour toutes les entrées et suffixes discriminants déjà recensés). La propriété de clôture garantit qu'on n'atteint pas de nouveaux états, différents des précédents, en prolongeant la séquence. Il est assez facile de montrer que l'automate construit est compatible avec la table d'observation si elle est *close* et *cohérente* , c'est à dire que pour

tout s de $S \cup S \cdot I$ et pour tout e de E, $\lambda(q_0, s \cdot e) = \lambda(q_0, s) \cdot TO(s \cdot e)$ (en ayant étendu λ aux séquences d'entrée). On peut par ailleurs montrer comme dans [1] que l'automate ainsi obtenu est l'automate minimal compatible avec la table d'observation.

Lorsque la table n'est pas *close* , cela signifie qu'on a découvert une nouvelle séquence du type $s' = s \cdot a$ menant à un état inéquivalent aux états précédemment identifiés. On rajoute alors cette séquence s' à S (la ligne correspondante passe de la partie basse à la partie haute de la table), et on complète la table en partie basse par les lignes correspondant à $s' \cdot I$. Lorsque la table n'est pas *cohérente* , on a trouvé deux éléments s_1 et s_2 tels que $ligne(s_1) = ligne(s_2)$ alors qu'il existe a de I et e de E tels que $ligne(s_1 \cdot a \cdot e) \neq ligne(s_2 \cdot a \cdot e)$. Ce qui signifie que $a \cdot e$ est une séquence discriminant les deux états atteints par s_1 et s_2. On rajoute $a \cdot e$ à E et on complète la table (entre autres, pour vérifier si cette séquence pourrait discriminer d'autres préfixes). On continue ainsi jusqu'à obtenir une table *close* et *cohérente* , qui permet de construire un automate conjecturé. Les préfixes présents dans S représentent les chemins d'accès à des états "candidats". Les suffixes présents dans E représentent les séquences dont on a pu établir jusqu'à ce stade qu'elles permettent de distinguer les états 2 à 2.

Une conjecture, qui est compatible avec toutes les observations faites jusqu'à son établissement, peut être remise en cause si des observations ultérieures ne sont pas conformes aux prédictions de l'automate. Dans l'algorithme original d'Angluin, c'est un expert qui fournit un contre-exemple. Dans notre approche, le contre-exemple viendra d'un test ultérieur dans une phase d'intégration. On va alors enrichir la table en étendant S avec le contre-exemple et tous ses préfixes (non déjà présents dans la table). On complète alors la table avec les observations requises jusqu'à aboutir à une nouvelle conjecture.

L'intérêt de l'algorithme est que si le composant a n états, l'algorithme terminera en un temps polynomial en n. Il en sera de même si le composant peut être abstrait en un système à n états en ne distinguant pas les valeurs des paramètres.

On peut noter que l'algorithme est incrémental puisque l'insertion d'un contre-exemple ne fait que rajouter des lignes à la table précédemment construite. Cette caractéristique est également utile si on dispose d'un modèle initial du composant sous forme d'un automate. On peut associer à cet automate une table pour laquelle S est construit à partir des plus courtes séquences d'accès à chaque état et $E = I$. Il faut bien sûr que l'automate fourni initialement soit conforme au fonctionnement du système au moins sur les séquences de $S \cup S \cdot I \cdot I$.

3.2 Apprentissage du contrôleur de climatiseur comme machine de Mealy

Dans une machine de Mealy, il ne peut y avoir de paramètres sur les symboles d'entrée et de sortie. On pourrait certes associer un symbole différent à chaque valeur du paramètre. Par exemple, $T(12)$ pourrait être un symbole d'entrée qu'on pourrait noter $T12$. Mais ceci aurait plusieurs inconvénients.

- On devrait discrétiser l'ensemble du domaine des paramètres d'entrée. Pour les températures de notre exemple (de -20°C à +50°C), il y aurait 71 valeurs,

	M	A	BT	MT	HT
ϵ	OK	Ω	Ω	Ω	Ω
M	Ω	Ω	C	F	V
$M-BT$	Ω	F	C	F	Ω
$M-HT$	Ω	F	Ω	F	V
$M-MT$	Ω	Ω	C	F	V
$M-BT-A$	OK	Ω	Ω	Ω	Ω
$M-BT-BT$	Ω	F	C	F	Ω
$M-BT-MT$	Ω	Ω	C	F	V
$M-HT-A$	OK	Ω	Ω	Ω	Ω
$M-HT-MT$	Ω	Ω	C	F	V
$M-HT-HT$	Ω	F	Ω	F	V

Tab. 1. Une table *close* et *cohérente* pour apprendre le contrôleur de climatiseur

si on s'en tient à des valeurs entières, or il pourrait être souhaitable d'avoir un grain plus fin. Pour des valeurs flottantes, on pourrait donc avoir un grand nombre de valeurs.
- Ceci conduirait à une complexité inutile de l'automate, rédhibitoire pour l'apprentissage et l'utilisation.
- On perdrait la structure de contrôle de l'automate, en la mélangeant avec les données. Or l'intérêt de la modélisation est de pouvoir extraire le contrôle pour en déduire une abstraction représentative et pertinente pour l'utilisation de techniques de génération de tests.

Pour apprendre un modèle de Mealy du contrôleur de climatiseur, nous avons besoin de réarranger son ensemble d'entrée pour que l'algorithme puisse être utilisé commodément. Ceci ne donnera pas un fonctionnement détaillé du contrôleur mais une abstraction de son comportement face à un changement de température.

Par exemple, nous distinguons trois sous-domaines de température, auxquels nous associons donc 3 entrées. Nous remplaçons T par MT (la température moyenne, entre 12°C et 15°C), BT (température basse, entre -20°C et 11°C) et HT (température haute, entre 16°C et 50°C). Donc, nous construisons $I = \{M, A, BT, MT, HT\}$ pour l'algorithme de machine de Mealy. La table 1 est la table d'observation obtenue en fin d'exécution de l'algorithme, quand la table est *close* et *cohérente* . La figure 3 est la conjecture pour le contrôleur comme machine de Mealy déduite de la table. Pour simplifier, les lignes qui ne correspondent à aucune sortie ne sont pas affichées dans la table. Aussi, nous utilisons les abréviations à la place des noms complets des entrées et sorties dans la table et la figure : M (Marche), A (Arrête), C (Chauffer), V (Ventiler), F (Fin).

Il faut noter que plus on raffinera les domaines d'entrée, plus on augmentera la taille de la machine de Mealy calculée, il faut donc éviter une explosion du nombre d'états.

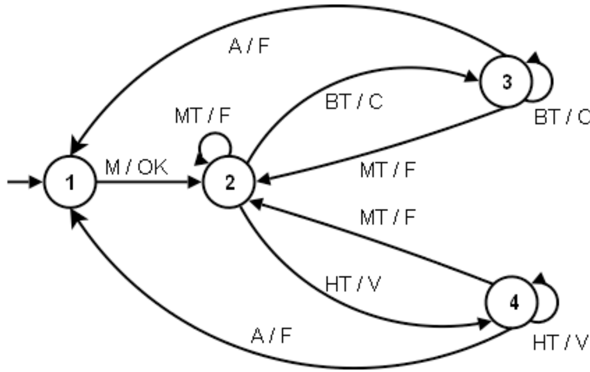


Fig. 3. Conjecture de Mealy

4 Modèle d'automates paramétrés

Comme on l'a noté, le modèle d'automates sans paramètres nécessite des abstractions de modélisation délicates. En outre, dans le contexte du test, il présente l'inconvénient majeur de ne pas instancier les valeurs. Les modèles d'automates obtenus pour les composants ne permettraient donc pas de produire des séquences de test complètes. C'est pourquoi nous proposons d'étendre le modèle pris en compte à des automates dans lesquels les entrées et les sorties sont paramétrés. Afin de prendre en compte les calculs différents auxquels peuvent conduire des valeurs différentes des paramètres pour une même entrée, on considèrera également que les transitions de l'automate peuvent être accompagnées d'une garde (prédicat) portant sur les valeurs des paramètres d'entrée. En cela, nous étendons le modèle paramétré que nous avions proposé dans [12]

En revanche, nous considérons des automates sans variables internes. En effet, autant on peut observer les valeurs des entrées et sorties de la boîte noire, autant on n'a pas accès aux états internes et à la structure de la machine. Il n'est donc pas possible de distinguer entre une mémorisation par des structures de contrôle ou dans des mémoires. Nous discutons cette restriction plus bas.

Le modèle que nous proposons est donc le suivant. Un automate paramétré (PFSM) est un septuplet : $M = (Q, I, O, D_I, D_O, T, q_0)$.

- Q est un ensemble fini d'états
- $q_0 \in Q$ est l'état initial.
- I est un ensemble fini de symboles d'entrée.
- O est un ensemble fini de symboles de sortie.
- D_I est un ensemble de valeurs du paramètre d'entrée.
- D_O est un ensemble de valeurs du paramètre de sortie.
- T est l'ensemble des transitions.

Une transition $t \in T$ est décrite par : $t = (q, q', i, o, p, f)$, où $q \in Q$ est l'état de départ, $q' \in Q$ est l'état d'arrivée, $i \in I$ le symbole d'entrée, $o \in O$ celui de sortie, $p \subseteq D_I$ est un prédicat sur la valeur du paramètre et $f : p \longrightarrow D_O$ est la

fonction de calcul du paramètre de sortie pour cette transition : elle associe la valeur du paramètre de sortie correspondant à celle du paramètre d'entrée.

Noter que les ensembles D_I et D_O ne sont pas nécessairement finis. Cependant, les modèles que nous construirons par apprentissage resteront des approximations finies, dans lesquels les valeurs explorées resteront dans des sous-domaines finis, avec dans tous les cas un ensemble fini de transitions.

Nous imposons néanmoins les restrictions suivantes aux machines que nous considèrerons : nous supposerons qu'elles seront complètes, déterministes, et observables.

Propriété 1 (Complet) *Un automate PFSM est complet si* $\forall q \in Q$, $\forall i \in I$ *et* $\forall x \in D_I$, $\exists t \in T$ *tel que* $t = (q, q', i, o, p, f)$, *avec* $x \in p$.

Comme pour les machines de Mealy simples, on peut compléter un automate paramétré en ajoutant des transitions bouclant sur l'état pour toutes les entrées non acceptées dans cet état. Ces transitions contiennent un symbole de sortie spécial Ω.

Propriété 2 (Déterministe) *Un automate paramétré est déterministe pour les entrées si pour* $t_1, t_2 \in T$ *tels que* $t_1 = (q_1, q_1', i_1, o_1, p_1, f_1)$ *et* $t_2 = (q_2, q_2', i_2, o_2, p_2, f_2)$ *et* $t_1 \neq t_2$, *si* $q_1 = q_2 \wedge i_1 = i_2$ *alors* $p_1 \cap p_2 = \phi$.

En d'autres termes, un automate est déterministe si pour une entrée donnée et une valeur du paramètre donnée, une seule transition lui est applicable dans un état donné.

Propriété 3 (Observable) *Un automate paramétré est observable si pour* $t_1, t_2 \in T$ *tels que* $t_1 = (q_1, q_1', i_1, o_1, p_1, f_1)$ *et* $t_2 = (q_2, q_2', i_2, o_2, p_2, f_2)$ *et* $t_1 \neq t_2$, *alors si* $q_1 = q_2 \wedge i_1 = i_2$ *alors* $o_1 \neq o_2$.

Dans un automate observable, deux transitions différentes partant du même état pour le même symbole d'entrée conduisent à deux symboles de sorties différents, ce qui permet de les différencier et de savoir que l'automate a pris un chemin différent. On peut aussi parler de non-déterminisme (au sens où il y a deux transitions) observable.

Quand M est dans l'état $q \in Q$ et reçoit l'entrée $i \in I$ portant la valeur de paramètre $x \in D_I$, alors l'état d'arrivée q', le symbole de sortie o et la valeur associée par f sont déterminés par les fonctions δ, λ and σ respectivement, définies comme suite :

- $\delta : Q \times I \times D_I \longrightarrow Q$ est la fonction d'état d'arrivée ; i.e. $\delta(q, i, x) = q'$ t.q. $(q, q', i, o, p, f) \in T$ et $x \in p$
- $\lambda : Q \times I \times D_I \longrightarrow O$ est la fonction de sortie
- $\sigma : Q \times I \longrightarrow {D_O}^{D_I}$ est la fonction définissant le paramètre de sortie.

Pour une séquence d'entrée $\gamma = i_1, ..., i_k$ et une séquence de valeurs des paramètres $\alpha = x_1, ..., x_k$, où chaque $i_j \in I, x_j \in D_I, 1 \leq j \leq k$, nous définissons l'association de γ à α comme $\gamma \otimes \alpha = i_1(x_1), ..., i_k(x_k)$, où chaque x_j est associé

à i_j. On procède de même pour associer les valeurs des paramètres de sortie aux symboles de sortie.

Notre modèle présente les extensions suivantes par rapport aux modèles d'automates habituellement pris en compte dans l'inférence de machines.

- Des entrées et sorties paramétrées.
- Des domaines arbitraires (non nécessairement finis) pour les paramètres.
- Des gardes p sur les paramètres d'entrée ; ces gardes sont associées aux transitions.
- Des fonctions arbitraires f pour le calcul des valeurs des paramètres en sortie.
- Ces fonctions peuvent être partielles.

Il y a cependant quelques restrictions dans ce modèle si on le compare aux modèles d'automates étendus du genre EFSM, tels qu'on les trouve dans des formalismes comme Estelle, SDL ou les Statecharts.

- Un paramètre unique pour les entrées et les sorties, et un domaine unique commun à tous les symboles.
- Pas de variables (dite parfois variables d'états). Toute la mémorisation doit être codée dans les états de Q.

Le premier point n'est pas une restriction sévère, c'est une commodité de notation. En effet, comme nous permettons des domaines arbitraires, il suffit d'introduire un codage entre les produits des domaines typés de paramètres des interactions réelles avec le composant et un domaine unique de représentation.

La seconde restriction est plus gênante, mais vient de l'impossibilité d'observer la structure interne de la boîte noire. Si un automate avait à la fois un état et une ou des variables internes, un même comportement pourrait être décrit par un automate à un seul état dans lequel toute la structure de contrôle serait codée dans une manipulation des variables, ou qui pourrait avoir un nombre arbitraire d'états. Il n'y aurait donc pas unicité de la solution calculée, ce qui remettrait en cause toute la démarche algorithmique.

Cette seconde restriction est donc sérieuse, puisqu'elle ne permet d'inférer le modèle que pour des composants ayant des variables dans des domaines finis, et de faible taille, car elle conduit à énumérer les états. Cependant, grâce au modèle paramétré que nous proposons, une forte réduction de la taille des automates inférés est déjà acquise par rapport aux algorithmes existants qui n'infèrent que des automates accepteurs déterministes, puisque nous pouvons factoriser un grand nombre de transitions. Pour aller plus loin et factoriser des états, nous pensons qu'on pourrait recourir à des heuristiques.

Comme nous n'infèrerons qu'une approximation des composants, nous ne prendrons en compte qu'un nombre fini d'états. Nous devrons supposer par ailleurs que les composants étudiés ont un modèle PFSM.

Dans un travail appuyé sur le code source (à des fins de compréhension des programmes), [16] proposent une technique permettant de reconstituer des structures d'automates enrichis avec une mémoire structurée. Un modèle et un algorithme d'inférence d'automates proposant certaines formes de paramètres a été proposé par [2]. Mais ce modèle ne prend en compte que des paramètres

booléens (et donc des domaines finis), et ne comporte pas de sorties (c'est une extension des automates accepteurs traités par Angluin).

5 Apprentissage de systèmes paramétrés

Cette partie explique comment on peut reconstituer un modèle d'automate paramétré de type PFSM présenté en 4. On présente les principes de l'algorithme, sans rentrer dans les détails. Comme le modèle PFSM est assez général pour représenter un composant réactif ayant un nombre fini d'états, on considère qu'on cherche à apprendre (ou identifier) un automate de référence inconnu $M = (Q, I, O, D_I, D_O, T, q_0)$, dont on connaît l'alphabet d'entrée I et le domaine du paramètre d'entrée D_I. Puisque nous pouvons soumettre n'importe quelle séquence d'entrées paramétrées au composant et observer les sorties paramétrées correspondantes, alors pour toute séquence d'entrée $\gamma \otimes \alpha (\gamma \in I^*, \alpha \in D_I^*, |\gamma| = |\alpha|)$, $\lambda(q_0, \gamma, \alpha)$ peut être trouvé par le test. On suppose là encore que tout composant peut être réinitialisé avant chaque test.

5.1 Table d'observation étendue

Dans l'algorithme présenté en 3 pour les automates de Mealy, on a utilisé la structure de table d'observation pour enregistrer les sorties du composant. Pour les PFSM, il faut enregistrer non seulement les symboles de sortie, mais aussi les valeurs des paramètres. Nous notons cette table (S, E, R, TO).

S est un ensemble fini non-vide de *séquences d'accès*, qui permettront effectivement d'accéder aux différents états du modèle. Dans un automate paramétré, l'état d'arrivée n'est pas déterminé seulement par la séquence de symboles d'entrée, mais aussi par la séquence des paramètres d'entrée. Nous appellerons l'association de ces deux séquences une *séquence d'entrée composée* ; ce sont ces séquences composées qui constitueront nos séquences d'accès.

E est un ensemble fini non-vide et clos par suffixe de séquences de symboles d'entrée. Ce sont les *séquences de séparation*, car elles servent à discriminer les états de la conjecture.

R est un sur-ensemble de S. Chaque fois qu'on ajoute une séquence d'accès à S, on obtient un groupe de séquences d'entrées composées en étendant la séquence d'entrée avec tous les $i \in I$ et en choisissant des $x \in D_I$ étendant l'ensemble des valeurs des paramètres des séquences d'entrée ; les séquences d'entrées composées sont ajoutées à R.

La fonction TO est définie sur $R \times E$. Dans le cas des PFSM, partant d'une même séquence d'entrée, on peut observer différentes séquences de sortie selon les valeurs qu'on donne aux paramètres d'entrée. Pour $r \in R$ et $e \in E$, la case $TO(r, e)$ contient l'association entre les paramètres des séquences d'entrée et ceux des séquences de sortie. La table 2 est un exemple d'une telle table d'observation, où la première colonne contient les séquences de R, avec S dans la première partie de la colonne (séparée par la barre horizontale).

5.2 Propriétés des tables d'observation

Dans le cas des automates simples de Mealy, on pouvait construire une conjecture dès que la table d'observation était *close* et *cohérente* . Pour les PFSM, la table doit avoir des propriétés supplémentaires pour construire une machine paramétrée en accord avec les observations des valeurs. Ces propriétés doivent permettre de comparer les lignes des tables, afin que la conjecture puisse être bien définie et minimale.

Les séquences de S représentent des états potentiels. Pour $r_1, r_2 \in R$, si l'on obtient, en partant des états atteints par r_1 et r_2, des séquences de sortie paramétrées différentes lorsqu'on a fourni en entrée les mêmes séquences composées, alors les lignes r_1 et r_2 sont *dissemblables.* Dans ce cas, on sait que r_1 et r_2 correspondent à des états différents.

Si deux lignes ne sont pas dissemblables, il faut, pour pouvoir les comparer, avoir exécuté les mêmes groupes de séquences composées à partir des états correspondants. Une table d'observation dont toutes les paires de lignes satisfont cette propriété sera dite *équilibrée.*

Pour tout $s \in S$ et $e \in E$, si $TO(s \cdot e)$ contient plus d'un symbole de sortie, alors la ligne s sera dite *contestée.* Une ligne s peut être contestée si $s \cdot e$ a été testé avec des valeurs différentes pour les paramètres d'entrée. Dans ce cas, on ajoutera de nouvelles lignes à R, qu'on testera avec les valeurs de paramètres qui sont dans $TO(s \cdot e)$.

Lorsqu'une table est équilibrée, on peut utiliser la relation d'égalité simple "=" pour comparer les lignes. On peut alors utiliser les concepts de table *close* et *cohérente* .

Quand la table d'observation est équilibrée, *close* et *cohérente* , on peut proposer une PFSM de conjecture d'une façon analogue à celle utilisée pour les automates de Mealy, en prenant soin d'associer les correspondances entre valeurs des paramètres en entrée et en sortie.

5.3 Algorithme d'apprentissage pour PFSM

L'algorithme d'apprentissage d'un modèle PFSM est défini par les étapes suivantes.

1. Au départ $R = S = \emptyset$ et $E = I$. Les cellules de la table sont toutes remplies avec l'ensemble vide.
2. Ajouter $\epsilon \otimes \epsilon$ à S, ainsi que dans R.
3. Construire des cas de test paramétrés pour les cellules non encore remplies de la table, et exécuter ces séquences. Pour $r \in R$ et $e \in E$, le cas de test correspondant a pour préfixe r, pour suffixe e et comme séquence de paramètres d'entrée α de ${D_I}^+$ et le test $r \cdot e \otimes \alpha$. [3]
4. Enregistrer le résultat du test $r \cdot e \otimes \alpha$ dans la table. Seule la dernière partie de la séquence de sortie sera enregistrée, celle qui correspond à la longueur

[3] On choisit la séquence de valeurs α parmi les valeurs extraites des scénarios d'usage.

de e (comme dans le cas des machines de Mealy). Le résultat du cas de test $\eta \otimes \beta$, où $\eta \in O^+$ est la séquence de sortie et $\beta \in {D_O}^+$ est la séquence de paramètres de sortie correspondante, est enregistré comme $(\alpha', \eta' \otimes \beta')$, où α' est la partie finale de α, η' celle de η, β' celle de β et $|\alpha'| = |\eta'| = |\beta'|$.

5. Équilibrer la table. Lorsqu'elle ne l'est pas, c'est à dire lorsque pour certaine séquences d'entrée les comportements du système pour différentes valeurs des paramètres d'entrée ne sont pas connus, construire les cas de test correspondants, les exécuter et enregistrer les résultats dans la table.
6. Pour chaque cas de test $r \cdot e \otimes \alpha$, s'il existe $r' \in R, e' \in E$ tel que $r' \cdot e'$ constitue un préfixe de $r \cdot e$, alors le préfixe correspondant des valeurs des paramètres sera rajouté à $TO(r' \cdot e')$. Si r' devient une ligne contestée, alors on ajoute de nouvelles lignes dans R, que l'on teste avec les valeurs de paramètres enregistrées dans $TO(r' \cdot e')$.
7. Fermer la table. Chaque fois qu'elle n'est pas *close* , ajouter la séquence composée correspondante à S et revenir à l'étape 3 pour remplir les cases ajoutées.
8. Rendre la table *cohérente* . Lorsqu'elle ne l'est pas, ajouter la séquence d'entrée correspondante à E et revenir à l'étape 3 pour remplir les cases vides.
9. Lorsque la table est *équilibrée*, *close* et *cohérente* , on peut établir la conjecture M'.

5.4 Apprentissage du contrôleur de climatiseur comme système paramétré

Pour identifier le modèle PFSM du contrôleur de climatiseur, on peut utiliser les entrées indiquées sur la figure 1, sans adaptation particulière, comme dans le cas de l'apprentissage du modèle de Mealy. Ainsi, nous construisons $I = \{M, A, T\}$ avec $D_I = [-20, 50]$ comme domaine pour le paramètre d'entrée. On utilise l'algorithme présenté en 5. La table 2 est *équilibrée*, *close* et *cohérente* . La conjecture est illustrée sur la figure 4 avec les valeurs des paramètres déterminées par l'apprentissage.

Par exemple, les valeurs 4, 12, 20 et 35 correspondraient à des valeurs dans des plages dont la spécification (sous forme de scénarios d'usage) nous apprendrait qu'elles pourraient donner lieu à des comportements distincts. Noter que ces valeurs ne correspondent pas forcément aux valeurs limites de ces plages, et que sauf dans l'état 2 où les 4 valeurs donnent lieu à 4 transitions différentes, pour les autres états, seule une des (plages de) valeur introduit un comportement spécifique.

On voit bien qu'on a appris plus de détails que sur le modèle de Mealy. Grâce à la structure paramétrée de PFSM, nous sommes capables de tester des valeurs différentes pendant l'algorithme d'inférence. Nous venons de comprendre que quand la valeur de la température est 35°C, le contrôleur envoie une commande pour lancer la climatisation CM à moyenne vitesse mv. Par ailleurs, la vitesse de chauffage C est régulée quand la valeur de la température est 4°C. En revanche,

si la température est 20°C, il envoie hv pour mettre en route le ventilateur V à vitesse haute. La conjecture ne recouvre pas tout le domaine du paramètre, mais ne connaît que les valeurs qui ont été effectivement testées. En effet, on ne peut parcourir dans le test tout le domaine des paramètres. Même s'il est fini, la combinaison des valeurs demanderait trop de cas de test, on ferait du test exhaustif.

R	M	A	T
$\epsilon \otimes \epsilon$	$(\bot, OK \otimes \bot)$	$(\bot, \Omega \otimes \bot)$	$(4, \Omega \otimes \bot), (12, \Omega \otimes \bot), (20, \Omega \otimes \bot), (35, \Omega \otimes \bot)$
$M \otimes \bot$	$(\bot, \Omega \otimes \bot)$	$(\bot, \Omega \otimes \bot)$	$(4, C \otimes mv), (12, F \otimes \bot), (20, V \otimes hv), (35, CM \otimes mv)$
$M - T \otimes \bot - 4$	$(\bot, \Omega \otimes \bot)$	$(\bot, F \otimes \bot)$	$(4, C \otimes mv), (12, F \otimes \bot), (20, F \otimes \bot), (35, F \otimes \bot)$
$M - T \otimes \bot - 20$	$(\bot, \Omega \otimes \bot)$	$(\bot, F \otimes \bot)$	$(4, F \otimes \bot), (12, F \otimes \bot), (20, V \otimes hv), (35, F \otimes \bot)$
$M - T \otimes \bot - 35$	$(\bot, \Omega \otimes \bot)$	$(\bot, F \otimes \bot)$	$(4, F \otimes \bot), (12, F \otimes \bot), (20, F \otimes \bot), (35, CM \otimes mv)$
$M - T \otimes \bot - 12$	$(\bot, \Omega \otimes \bot)$	$(\bot, \Omega \otimes \bot)$	$(4, C \otimes mv), (12, F \otimes \bot), (20, V \otimes hv), (35, CM \otimes mv)$
$M - T - A \otimes \bot - 4 - \bot$	$(\bot, OK \otimes \bot)$	$(\bot, \Omega \otimes \bot)$	$(4, \Omega \otimes \bot), (12, \Omega \otimes \bot), (20, \Omega \otimes \bot), (35, \Omega \otimes \bot)$
$M - T - A \otimes \bot - 20 - \bot$	$(\bot, OK \otimes \bot)$	$(\bot, \Omega \otimes \bot)$	$(4, \Omega \otimes \bot), (12, \Omega \otimes \bot), (20, \Omega \otimes \bot), (35, \Omega \otimes \bot)$
$M - T - A \otimes \bot - 35 - \bot$	$(\bot, OK \otimes \bot)$	$(\bot, \Omega \otimes \bot)$	$(4, \Omega \otimes \bot), (12, \Omega \otimes \bot), (20, \Omega \otimes \bot), (35, \Omega \otimes \bot)$
$M - T - T \otimes \bot - 4 - 4$	$(\bot, \Omega \otimes \bot)$	$(\bot, F \otimes \bot)$	$(4, C \otimes mv), (12, F \otimes \bot), (20, F \otimes \bot), (35, F \otimes \bot)$
$M - T - T \otimes \bot - 4 - 20$	$(\bot, \Omega \otimes \bot)$	$(\bot, \Omega \otimes \bot)$	$(4, C \otimes mv), (12, F \otimes \bot), (20, V \otimes hv), (35, CM \otimes mv)$
$M - T - T \otimes \bot - 20 - 4$	$(\bot, \Omega \otimes \bot)$	$(\bot, \Omega \otimes \bot)$	$(4, C \otimes mv), (12, F \otimes \bot), (20, V \otimes hv), (35, CM \otimes mv)$
$M - T - T \otimes \bot - 20 - 20$	$(\bot, \Omega \otimes \bot)$	$(\bot, F \otimes \bot)$	$(4, F \otimes \bot), (12, F \otimes \bot), (20, V \otimes hv), (35, F \otimes \bot)$
$M - T - T \otimes \bot - 35 - 4$	$(\bot, \Omega \otimes \bot)$	$(\bot, \Omega \otimes \bot)$	$(4, C \otimes mv), (12, F \otimes \bot), (20, V \otimes hv), (35, CM \otimes mv)$
$M - T - T \otimes \bot - 35 - 35$	$(\bot, \Omega \otimes \bot)$	$(\bot, F \otimes \bot)$	$(4, F \otimes \bot), (12, F \otimes \bot), (20, F \otimes \bot), (35, CM \otimes mv)$

Tab. 2. Une table *close* et *cohérente* pour apprendre un modèle PFSM du contrôleur de climatiseur

6 Test d'intégration

Après avoir présenté l'inférence de modèles pour les composants, nous décrivons ici comment exploiter ces algorithmes et ces modèles dans le développement d'un système par intégration de composants. On suppose que l'intégrateur logiciel se situe dans le cadre présenté en introduction, avec les hypothèses que nous y avons faites.

Dans une phase préparatoire de test "unitaire" des composants isolés, on construit pour chaque composant C un premier modèle PFSM $C^{(1)}$ selon l'algorithme décrit en 5. Ensuite, on assemble les composants d'un côté et leurs modèles de l'autre : les sorties d'un composant peuvent devenir les entrées d'un autre.

L'intégration se fait en deux étapes. Dans une première étape, on fournit au système (assemblé) les scénarios d'usage du système global (au besoin, si on ne dispose pas de tels scénarios, on peut en engendrer à partir de la définition d'interfaces). Les scénarios devront comporter normalement une instanciation des séquences d'entrées et de sortie avec des valeurs typiques des paramètres, ou des domaines de valeurs dans lesquelles on pourra choisir des valeurs pour les entrées. On fournit au système et à son modèle les séquences d'entrées paramétrées et on observe les sorties paramétrées. A ce stade, si les sorties ne sont pas celles

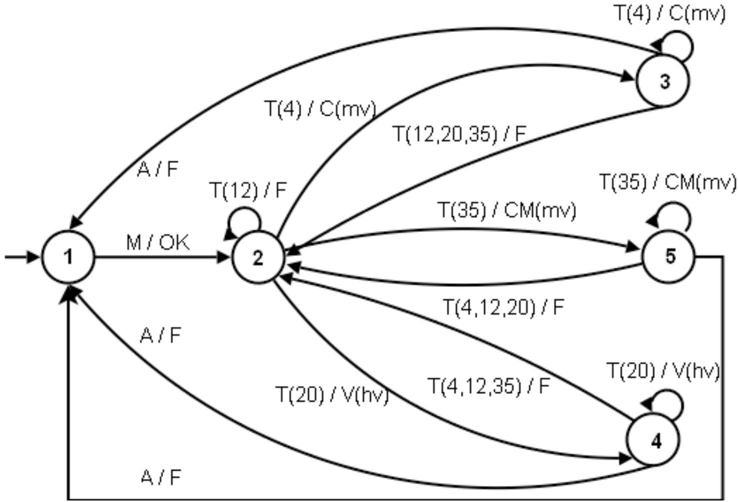


Fig. 4. Conjecture de PFSM

attendues (lorsqu'elles ont été spécifiées), une erreur est détectée, ou si elles diffèrent de celles produites par le modèle, on est en présence d'un contre-exemple qui sera réinjecté dans l'apprentissage pour enrichir les modèles des composants. Une première phase d'enrichissement incrémental des modèles des composants peut donc avoir lieu au cours de cette première étape.

Dans une deuxième étape, on procède au test d'interopérabilité, pour lequel on s'appuie sur les modèles, comme générateurs. Pour cela, on utilise des techniques de génération par exploration de modèles comme celles de [14], [3], [10] etc. Le choix des paramètres associés aux entrées externes du système sera fait en tenant compte des valeurs typiques fournies avec les scénarios d'usage, en essayant d'enrichir les cas déjà testés dans la première étape. Les sorties paramétrées (tant internes qu'externes) observées en testant le système sont confrontées à celles prévues par les modèles. Le test continue jusqu'à ce qu'une sortie soit incompatible avec le modèle (symbole de sortie ou paramètre différent de celui prévu par le modèle) ou qu'un critère de couverture défini pour l'arrêt du test soit atteint. Si une incompatibilité a été détectée, elle fournit un contre-exemple pour raffiner le modèle. L'expert procédant à l'intégration peut également être sollicité pour décider si le contre-exemple est un comportement admissible ou erroné du système intégré.

Les contre-exemples obtenus dans les deux étapes peuvent servir à raffiner les modèles. Ils sont alors injectés dans l'algorithme d'apprentissage comme séquences rajoutées à S pour les composants concernés. Les valeurs des paramètres compatibles avec les modèles sont également prises en compte à ce niveau pour enrichir les tables. On produit alors une nouvelle version du modèle du système avec les des modèles $C^{(i+1)}$ pour chaque composant C.

À la fin du processus d'intégration, on dispose donc d'un modèle paramétré pour chacun des composants qui est compatible avec tous les tests exécutés. Les interactions entre tous les composants auront été systématiquement testés selon le critère de sélection de test et de couverture que l'intégrateur se sera donné. Le processus de test aura également pu être interrompu en cas de découverte de fautes dans le système. Pour plus de détails sur l'algorithme d'intégration, voir [11]

7 Conclusion

Nous avons présenté une approche destinée à assister l'intégrateur de composants logiciels. Prenant acte de l'absence habituelle de modèles formels accompagnant les composants, nous proposons une méthode et des algorithmes permettant d'utiliser les techniques de génération de tests à partir de modèles, en s'appuyant sur des modèles eux-mêmes reconstruits et enrichis au cours du test. Comme les modèles sont eux-mêmes déduits des observations faites au cours des tests, ils ne peuvent servir d'oracles absolus. Mais ils permettent de dériver et de tester systématiquement les interactions entre les composants.

La méthode s'inscrit bien dans une assistance à base formelle au développement logiciel. L'ingénieur dispose d'outils automatiques pour construire les modèles, dériver et exécuter les tests. Il reste maître de l'architecture du système, de la définition des scénarios d'usage, et de l'analyse des fautes et des divergences entre le modèle et le système réel.

Comme les difficultés d'intégration entre composants provenant de sources différentes sont souvent liées à des discordances dans le traitement des valeurs échangées entre les composants, nous accordons un intérêt particulier à la construction de modèles paramétrés. Pour cela, nous avons développé de nouveaux algorithmes d'inférence de machines pour des modèles de ce type.

Nous poursuivons ce travail dans plusieurs directions. D'abord, bien qu'il soit plus expressif que les automates finis étudiés jusqu'ici en inférence de machines (à l'exception de [2]), notre modèle reste limité par l'absence de variables internes, ce qui ne permet de reconnaître que des machines ayant un nombre fini d'états. On pourrait envisager des modèles plus proches d'EFSM, éventuellement en supposant que l'intégrateur est capable de fournir des informations sur la structure de la machine. Le problème d'inférence de telles machines sera cependant difficile. Nous avons développé un outil, RALT, qui met en oeuvre les algorithmes d'inférence pour les automates accepteurs, les machines de Mealy et les automates paramétrés. Nous devrions l'appliquer prochainement à des cas d'étude issus des services de télécommunication développés par France Télécom. Au-delà du travail sur les modèles et l'approche de test incrémental, nous sommes également intéressés à quantifier, ou du moins à qualifier plus précisément, le niveau de confiance à accorder à l'assemblage à l'issue (et au cours) du processus d'intégration. Pour cela, nous travaillons sur la définition d'une notion d'approximation compatible avec notre approche. Enfin, la démarche de test est

paramétrée par les critères de couverture, et nous souhaitons développer la mise en relation formelle de ces critères avec l'approximation réalisée par les modèles.

Références

1. Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 2 :87–106, 1987.
2. Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines with parameters. In *FASE*, volume 3922 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2006.
3. C. Besse, A. Cavalli, M. Kim, and F. Zaidi. Two methods for interoperability tests generation : An application to the tcp/ip protocol. In *Proceedings of TestCom 2002*, Berlin, 2002.
4. Edith Elkind, Blaise Genest, Doron Peled, and Hongyang Qu. Grey-box checking. In *FORTE'06*, 2006.
5. Alex Groce, Doron Peled, and Mihalis Yannakakis. Adaptive model checking. In *Tools and Algorithms for Construction and Analysis of Systems*, 2002.
6. Andreas Hagerer, Hardi Hungar, Oliver Niese, and Bernhard Steffen. Model generation by moderated regular extrapolation. In *Fundamental Approaches to Software Engineering*, pages 80–95, 2002.
7. Hardi Hungar, Oliver Niese, and Bernhard Steffen. Domain-specific optimization in automata learning. In *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 315–327. Springer, 2003.
8. O. Koné and R. Castanet. Test Generation for Interworking Systems. *Computer Communications*, 23(7) :642–652, 2000.
9. Keqin Li, Roland Groz, and Muzammil Shahbaz. Integration testing of components guided by incremental state machine learning. In *TAIC PART*, pages 59–70. IEEE Computer Society, 2006.
10. Keqin Li, Roland Groz, and Muzammil Shahbaz. Integration testing of distributed components based on learning parameterized i/o models. In *FORTE*, volume 4229 of *Lecture Notes in Computer Science*, pages 436–450. Springer, 2006.
11. Erkki Makinen and Tarja Systa. MAS - an interactive synthesizer to support behavioral modelling in UML. In *ICSE '01 : Proceedings of the 23rd International Conference on Software Engineering*, pages 15–24, Washington, DC, USA, 2001. IEEE Computer Society.
12. Clémentine Nebut, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. Automatic test generation : A use case-driven approach. *IEEE Trans. Softw. Eng.*, 32(3) :140, 2006.
13. D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. In *Proceedings of FORTE'99*, Beijing, China, 1999.
14. A. Petrenko and N. Yevtushenko. Solving asynchronous equations. In *FORTE'98*, France, 1998.
15. Stephane S. Somé. Beyond scenarios : generating state models from use cases. In *Proceedings of SCESM*, 2002.
16. Neil Walkinshaw, Kirill Bogdanov, and Mike Holcombe. Identifying state transitions and their functions in source code. In *TAIC PART*, pages 49–58. IEEE Computer Society, 2006.

Déclinaison d’exigences de sécurité du système vers le logiciel, assistée par des modèles formels

Sophie Humbert[1], Jean-Marc Bosc[1], Charles Castel[2], Pierre Darfeuil[1], Yves Dutuit[3], Eric Focone[1] et Christel Seguin[2]

[1] TURBOMECA, 64511 Bordes Cedex
{jean-marc.bosc, pierre.darfeuil, eric.focone, sophie.humbert}@turbomeca.fr

[2] ONERA-CERT, 2 av. E. Belin, B.P. 4025, 31055 Toulouse Cedex
{castel, seguin}@cert.fr

[3] Université Bordeaux 1 / LAPS, 351 cours de la libération, 33405 TALENCE
yves.dutuit@iut.u-bordeaux1.fr

Résumé : Les processus traditionnels d'analyse de sécurité des systèmes embarqués préconisent d'éliciter les exigences de sécurité en tenant progressivement compte de l'architecture des systèmes puis des caractéristiques de leurs composants. Notre étude se propose d’améliorer la déclinaison des exigences de sécurité du niveau système global vers les sous-parties logicielles. Pour atteindre cet objectif, nous proposons d'étendre le processus traditionnel pour mieux cerner l'impact d'hypothétiques erreurs de conception logicielles et en dériver des exigences fonctionnelles. Cet article définit la nouvelle étape, présente comment elle s'intègre dans un processus classique et montre comment elle peut être supportée par l'utilisation de modèles formels AltaRica et Scade en vigueur dans le domaine. Les différents concepts sont illustrés sur la fonction de régulation d'un turbomoteur permettant de maintenir la vitesse de rotation du rotor d'un hélicoptère.

Mots-clés : modélisation formelle, AltaRica, SCADE, exigences de sécurité, déclinaison d’exigences, vérification par preuves formelles.

1 Introduction

La certification des avions ou hélicoptères nécessite de démontrer que les fonctions de haut niveau d’un aéronef satisfont certaines exigences de sécurité (au sens innocuité). La démonstration requiert de décomposer complètement et précisément ces exigences de haut niveau en exigences évaluables sur les constituants physiques les plus élémentaires de l’aéronef. Le travail présenté dans cet article porte sur les étapes de décomposition menant à des exigences de sécurité allouées aux logiciels applicatifs embarqués.
Le processus de décomposition actuel est guidé par des normes en vigueur dans le domaine aéronautique comme l’ « Aerospace Recommended Practice (ARP) 4754 » [1]. Trois niveaux de décomposition apparaissent : le niveau des fonctions de l’aéronef (le freinage, la propulsion, ..), le niveau des systèmes physiques (commande de vol, turbomoteur, …) permettant de réaliser les fonctions et enfin le niveau des équipements (électroniques, mécaniques, …). Les exigences portant sur les parties logicielles sont

élicitées tout d'abord au niveau système puis au niveau équipement. Dans cet article, nous nous focalisons sur les activités menées au niveau système.
Le niveau « système » comprend des étapes de conception variées, allant de la définition d'une architecture complexe incluant des redondances, aux spécifications détaillées des lois contrôlant les différents organes du système. Pour maîtriser ce large spectre, l'ARP 4754 recommande de réaliser une analyse préliminaire du système consolidée par une analyse de sécurité détaillée. La dérivation d'exigences d'un niveau au suivant suit une démarche assez systématique. On considère l'architecture préliminaire du système et les modes de défaillances pouvant impacter les grands composants identifiés. Si les modes de défaillances envisagés conduisent à violer une exigence de sécurité allouée au système complet, on dérive de nouvelles exigences sur les composants pour prévenir l'occurrence du problème. Les exigences dérivées seront essentiellement de deux types : « la probabilité d'occurrence du cas de défaillance imaginé doit être inférieur à seuil acceptable » ou bien « il faut implanter un mécanisme de sécurité pour s'accommoder de ce problème ».
La notation privilégiée pour décomposer un événement redouté en combinaisons booléennes de causes plus élémentaires est généralement l'arbre de défaillances ([2], [3]). L'approche est simple, effective dans de nombreux cas et permet de réaliser des analyses qualitatives (recherche de combinaison minimale de causes) et quantitatives (calcul de la probabilité d'occurrence de l'événement racine de l'arbre à partir des probabilités des défaillances élémentaires feuilles). Néanmoins, l'approche rencontre des limites lorsque l'on analyse des systèmes complexes ou que l'on rentre dans le détail de composants sophistiqués comme les calculateurs embarqués.
Les travaux sur les analyses de sécurité fondés sur des modèles formels de systèmes ([4], [5], [6]) tentent de dépasser cette limite. Ces travaux proposent de construire des modèles de propagation de panne, compositionnels et hiérarchiques, reflètant l'architecture des systèmes et de réaliser sur ces modèles des évaluations de sécurité classiques (extraction d'arbres de défaillances, recherche de causes élémentaires, quantification, … En particulier, le langage AltaRica [7] et les outils associés (voir par exemple [8], [9] ou [10]) ont été crées dans ce but. Dans cet article, nous proposons de montrer à travers un cas d'étude comment construire et analyser des modèles AltaRica s'insérant dans le processus de dérivation des exigences de sécurité allouées aux logiciels applicatifs.
Notre objectif final est de dériver des exigences de sécurité testables sur les spécifications détaillées des logiciels applicatifs les plus critiques. Pour la conception des logiciels de régulation des systèmes de propulsion, Turbomeca a choisi d'utiliser SCADE SUITE™ [11] basé sur le langage formel Lustre [12]. Ce choix se justifie par la possibilité d'utiliser un générateur de code qualifié pour la DO178B niveau A, qui permet d'une part d'éviter une campagne de tests unitaires coûteuse en temps et en ressources, et d'autre part une détection précoce d'erreurs à l'aide de simulations et de preuves formelles. Nous montrons comment les modèles AltaRica nous permettent de rentrer plus finement dans le détail des calculateurs et de mieux cerner les exigences allouées aux modules Scade d'intérêt. Néanmoins, la granularité des spécifications détaillées, destinées à générer le code, diffère de celle des modèles utilisés pour raisonner sur la propagation des défaillances, et ceci, indépendamment des différences introduites par

les langages AltaRica et Lustre. Nous terminons donc cet article en montrant le type de raisonnement qui permet de passer d'un niveau d'abstraction à l'autre et en tirons des conséquences pour la suite de nos travaux.
L'article est structuré de la manière suivante. Dans la section 2, nous introduisons notre cas d'étude : la fonction de régulation d'un turbomoteur permettant de maintenir la vitesse de rotation du rotor d'un hélicoptère. Nous présentons l'architecture globale du système telle qu'elle est appréhendée lors des analyses préliminaires. Puis nous montrons comment le logiciel applicatif est structuré de manière à faire ressortir les différents niveaux de décomposition à considérer dans l'analyse. Dans la section 3, nous présentons comment l'utilisation de modèles AltaRica peut être insérée dans le processus d'analyse de sécurité préliminaire et de dérivation d'exigences pour mieux cerner le rôle du logiciel applicatif. Dans la section 4, nous discutons le passage des modèles de sécurité préliminaire aux spécifications logicielles détaillées. Nous concluons en nous positionnant par rapport aux travaux existants et en discutant les pistes à développer.

2 Présentation du cas d'étude

Pour illustrer la démarche proposée, le cas d'étude choisi est le système pilotant le dosage du carburant dans un turbomoteur d'hélicoptère. Le rôle d'un turbomoteur est de maintenir la vitesse de rotation du rotor de l'hélicoptère quasi constante autour d'une valeur donnée en contrôlant la quantité de carburant injectée dans le moteur. Cette fonction de régulation est réalisée par du logiciel embarqué sur un calculateur. Celui-ci établit une consigne de déplacement du doseur de carburant (cf. figure 2) en fonction de paramètres variés et, en particulier, de mesures issues de différents capteurs moteur. Nous présentons par la suite l'architecture du système global puis nous détaillons l'architecture logicielle.
On notera que pour la clarté de l'exposé, même si ce cas d'étude est inspiré de la réalité, il n'en reprend pas toute la complexité.

2.1 L'architecture du système

La partie système intéressante est constituée du calculateur, d'un actionneur, d'un capteur et de l'organe (pointeau) limitant le débit du carburant. Les liaisons entre le calculateur d'une part, l'actionneur et le capteur de recopie de position d'autre part, sont réalisées par des faisceaux électriques. L'actionneur déplace une crémaillère, solidaire du pointeau, qui entraîne le capteur de recopie de position. Le changement de position du pointeau modifie la section de passage du carburant, ce qui permet d'augmenter ou réduire le débit du carburant injecté dans la chambre de combustion. L'analyse de sécurité préliminaire doit considérer l'ensemble de ces constituants.

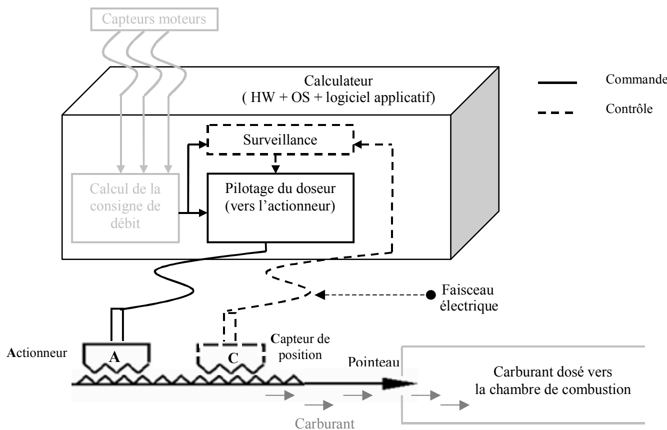


Figure 2 : Schéma de principe de fonctionnement du pilotage du doseur

NB : Les parties grises ne seront pas modélisées dans la suite, mais facilitent la compréhension du schéma fonctionnel. La partie logiciel automate (Operating System) n'est pas représentée graphiquement pour ne pas surcharger le schéma. Cependant, toutes les entrées et sorties du calculateur passent par la couche OS du calculateur (cf. partie 2.3).

2.2 L'architecture des logiciels applicatifs

Lorsque l'on entre dans le détail du calculateur qui pilote le doseur, on distingue
- les parties matérielles (processeurs, unités de gestions des entrées sorties, …)
- le **logiciel automate** (OSS : **O**perating **S**ystem **S**oftware). Assimilable à un « système d'exploitation », il permet d'interfacer le matériel avec le logiciel de régulation et gère l'ordonnancement des tâches du logiciel applicatif.
- le **logiciel applicatif** (CSS : **C**ontrol **S**ystem **S**oftware) qui réalise les fonctions de régulation et de surveillance du turbomoteur, dont le pilotage du doseur, notre cas d'étude.

Lorsque l'analyse de sécurité détaillée est réalisée, chacune de ces parties et leur intégration sont soigneusement étudiées. Dans la suite, nous détaillons uniquement l'architecture du logiciel applicatif

Les principales fonctions du logiciel applicatif, spécifié et produit par Turbomeca, sont :
- Le lancement (autotest, lecture EEPROM), le séquencement de l'application ;
- La gestion des acquisitions (capteurs moteur, commandes hélicoptère...) ;
- La régulation du moteur ;
- La protection du moteur (limites thermiques et mécaniques) ;
- Le pilotage des sorties (actionneurs, voyants) ;
- La communication avec les différents constituants du système (hélicoptère, autre moteur, maintenance…) ;

- Le contrôle santé moteur (fatigue cyclique, utilisation, contrôle performance moteur...)
- La gestion des pannes.

Ces fonctions sont décomposées et regroupées en constituants logiciels à l'aide des critères utilisés lors de la conception (fonctionnel, séquentiel, testabilité). Nous trouvons ainsi dans l'architecture statique du logiciel applicatif, les classes suivantes :

- Les **groupes** logiciels : un groupe logiciel est une partie du logiciel qui assure une fonction système de haut niveau (exemple : G_piloter_sorties) ;
- Les **composants** logiciels : un composant logiciel est une partie d'un groupe logiciel qui assure une fonction système modulaire dont la réalisation et le test peuvent être confiés à une équipe indépendante (exemple : C_piloter_surveiller_le_doseur) ;
- Les **modules** logiciels : un module logiciel est un constituant élémentaire du composant logiciel qui assure une bonne lisibilité du logiciel. Une bonne représentation est l'unité de compilation (exemple : controler_la_position_du_doseur) ;
- Les **fonctions** élémentaires **réutilisables** : certaines fonctions logicielles sont factorisées dans une librairie de base afin d'être réutilisées sur plusieurs projets logiciels. Ces fonctions représentent une « boîte à outils » multi-projet mathématique, logique, autres (exemple : filtre_de_panne_3etats, saturer, Bascule_RS…).

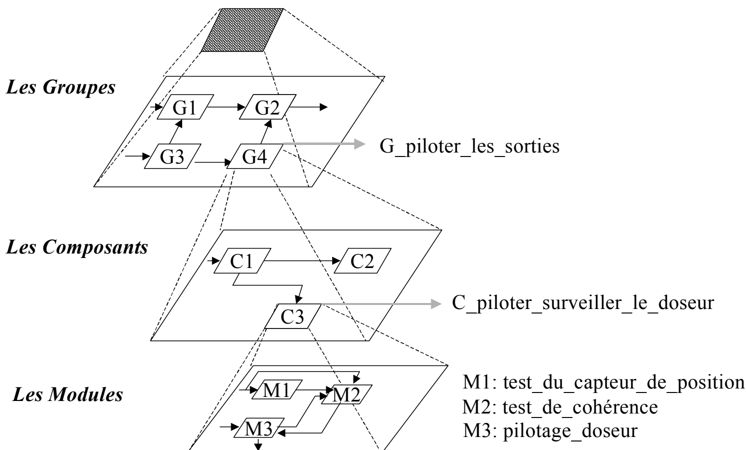


Figure 3 : Positionnement des fonctions étudiées dans l'architecture du logiciel applicatif

Pour les aspects dynamiques, un séquenceur global, définissant la tâche applicative appelée CSS, organise et séquence les groupes. Chaque groupe séquence l'ensemble de ses composants et éventuellement chaque composant séquence et organise ses modules si sa complexité le nécessite.

Pour le cas d'étude choisi, la partie applicative est le composant C_piloter_surveiller_le_doseur. Il contient les algorithmes de calcul du déplacement du doseur et deux modules de surveillance de l'actionneur doseur.
L'un, appelé « M2 : test_de_cohérence », réalise un test d'écart entre la demande de position de l'actionneur et la position lue par le capteur de position (c'est-à-dire l'écart entre la consigne de déplacement et le déplacement observé). Lorsque le test de cohérence détecte un écart trop important, la commande doseur est considérée comme perdue. On annonce alors au pilote une panne de la régulation automatique et l'on place le système dans un état sûr : on « gèle » le débit du carburant à la valeur qu'il avait juste avant la défaillance.
L'autre module, appelé « M1 : test_du_capteur_de_position » détecte les pannes propres au capteur de position. Cette détection se justifie pour être tolérant aux pannes de la surveillance, et éviter ainsi la perte de la régulation automatique (le débit continue à être piloté sans surveiller la commande de l'actionneur).

Le composant C_piloter_surveiller_le_doseur utilise deux services automate de l'OSS :

- lire_Cdp pour lire la mesure et la validité du capteur de position doseur ;
- commander_doseur pour piloter l'actionneur doseur.

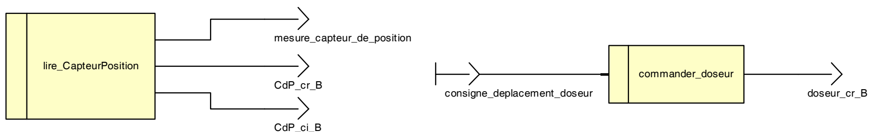


Figure 4 : les services automates de l'OSS

Dans notre cas d'étude, l'OSS fournit un compte-rendu de panne (CdP_cr_B) sur la mesure du capteur de position ainsi qu'une indication de « non utilisation » de la mesure du capteur de position (CdP_ci_B), lorsque celle-ci est momentanément considérée comme douteuse (par exemple lors de la mise sous tension du calculateur). Par ailleurs, l'OSS fournit un compte rendu de panne sur la sortie correspondant à la consigne de déplacement de l'actionneur (doseur_cr_B).

3 Analyse de sécurité de l'architecture à l'aide de modèles AltaRica

Cet exemple restreint montre bien la différence de niveau existant entre la granularité du calculateur vu comme un composant de l'architecture système globale et le détail de l'architecture de ce composant particulièrement complexe. Dans cette section, après un bref rappel du langage AltaRica, nous montrons comment les éléments sont intégrés dans l'analyse de sécurité et comment une modélisation AltaRica détaillée de l'architecture système permet de décomposer plus finement les exigences.

3.1. AltaRica

AltaRica est un langage formel conçu au LaBRI (Laboratoire Bordelais de Recherche en Informatique) pour faciliter la modélisation du comportement des systèmes potentiellement sujets à défaillances. C'est un langage compositionnel et hiérarchique basé sur la notion d'automate à contraintes.

Chaque automate (nœud) est interfacé par des "flux" (entrées/sorties) et peut contenir des variables d'état internes (mémoires, états de fonctionnement et de dysfonctionnement) qui évoluent lorsque surviennent des événements (par exemple défaillance). Le comportement d'un nœud est généralement défini par trois parties :

- l'initialisation des états internes ;
- les transitions indiquent les changements d'états internes du composant. Elles se présentent sous la forme de commandes gardées :

  condition de départ ├ événement → états internes d'arrivée ;
- les assertions se présentent sous la forme de contraintes booléennes sur les flux et les états. Elles permettent d'exprimer les relations entre entrées et sorties lorsque les flux sont orientés.

Un nœud peut être hiérarchisé : il peut intégrer et contrôler (en fonction de son propre état) un ensemble de composants ou de sous-systèmes. Enfin le système est construit à partir d'instances des nœuds que l'on interconnecte par leur interfaces.

L'emploi de ce langage se justifie pour plusieurs raisons. Tout d'abord, c'est un langage qui a été conçu spécialement pour faire des analyses de sécurité. D'une part, les automates à contraintes permettent de modéliser de manière concise et souple les logiques de changement de mode et les fonctions réalisées dans un mode de fonctionnement donné. Ceci est crucial pour les modèles de sûreté de fonctionnement. D'autre part, ces modèles peuvent être analysés à l'aide de multiples outils permettant de faire de la simulation, des analyses traditionnelles de sécurité (arbres de défaillances, Markov, Petri) et du model-checking. Enfin, il intéresse de plus en plus des entreprises comme Airbus, Dassault ou Turbomeca (voir [13], [14], ou [15] au sujet d'expériences réalisées dans le domaine aéronautique).

3.2. Insertion des modèles AltaRica dans la démarche d'analyse de sécurité préliminaire

Nous introduisons à présent la démarche d'analyse de sécurité préliminaire et montrons à travers le cas d'étude comment les modèles AltaRica peuvent être insérés et contribuer à une décomposition plus fine des exigences de sécurité. Nous n'entrerons pas dans le détail de la méthodologie de modélisation AltaRica, le lecteur intéressé pourra consulter [15] .

1 : Recensement des fonctions dans le périmètre de l'étude « système »

L'ensemble des fonctions matérielles et logicielles représentées figure 2, y compris les fonctions liées à l'OSS, à l'exception des fonctions grisées, fait partie du périmètre d'étude.

2 : Étude des défaillances potentielles pour chaque fonction envisagée (hors erreurs de conception logicielles)

Le comportement fonctionnel de chacune de ces fonctions est connu de par leur spécification. Leur comportement dysfonctionnel est extrait de l'analyse des modes de défaillance et de leur effets (AMDE voir [3]) menée classiquement pour chaque fonction. Il est communément reconnu que les parties purement logicielles ne s'usent pas et que par conséquent, leurs défaillances résultent soit d'erreur de conception, soit d'utilisations inappropriées. Les étapes de vérification ultérieures étant supposées éliminer avec une certaine confiance les erreurs d'implantation, on suppose généralement que les fonctions logicielles sont exemptes d'erreur de conception. En revanche, des hypothèses de défaillances peuvent être formulées sur les ressources utilisées par le logiciel applicatif comme par exemple une panne de CPU). A titre d'exemple, le tableau suivant donne la définition d'un mode de défaillance de la fonction réalisée par l'actionneur.

Fonction	Mode de défaillance	Effet local	Effet Système
Déplacer le doseur (réalisée par l'actionneur)	Action effectuée erronée	Déplacement du doseur erroné	Détection par le test de cohérence entre la mesure du capteur de position et la consigne de déplacement du doseur (fonction CSS) → Gel du doseur

Tableau 1 : Exemple d'une ligne d'AMDE matérielle

En résumé, l'analyse des modes de défaillances des fonctions du cas d'étude est fondée sur le fait que :

- Les défaillances matérielles prises en hypothèses au niveau de la chaîne de commande (cf. figure 2) ont pour effet la transmission erronée de la consigne de déplacement du doseur ;
- Au niveau de la chaîne de surveillance, les défaillances prises en hypothèses ont pour effet la transmission erronée de la mesure de position du doseur (défaillance du capteur de position ou du faisceau électrique).

3 : Création d'un modèle dysfonctionnel en langage AltaRica

L'objectif est d'étudier les conditions sous lesquelles les événements redoutés pourraient se produire. C'est ici que nous proposons de remplacer les modèles classiques comme les arbres de défaillances ou les diagrammes de fiabilité par un modèle AltaRica.

Avec l'approche AltaRica, chaque fonction est modélisée dans un nœud AltaRica, qui décrit son comportement fonctionnel et dysfonctionnel. A ce niveau d'analyse, le modèle AltaRica diffère sensiblement de la spécification détaillée du logiciel même si l'on suppose que la partie logicielle pure est exempte d'erreur. En effet, les modèles se focalisent sur la propagation des pannes, (est-ce que le composant transfère ou pas une valeur correct) et abstraient les détails des calculs. Prenons pour exemple la fonction test d'écart réalisée par le logiciel dont le rôle est de vérifier si deux variables réelles sont égales (à une tolérance près). Dans ce cas le test informe de la validité des données,

dans le cas contraire il informe d'une valeur erronée. En outre ce test est toujours effectué sauf si l'autorisation d'exécution ne lui est pas donnée.
En AltaRica, cette fonction test d'écart est abstraite de la manière suivante : en entrée, on a les deux variables à tester `position_calculee_prec` et `capteur_de_position`, de type énuméré à valeur dans le domaine `{correct, erroné}`, et la variable d'autorisation `autorisation_test` de type booléen. En sortie, on a la variable d'information sortie `ecart_courant_detecte_B` de type booléen (`true` en cas de détection, sinon `false`). La modélisation AltaRica est donnée ci après avec l'hypothèse d'une fonction sans défaillance (aucun état ni événement) :

```
node test_ecart
  flow
    position_calculee_prec:{correct, errone}:in;
    capteur_de_position:{correct, errone}:in;
    autorisation_test:bool:in;
    ecart_courant_detecte_B:bool:out;
  state // Lorsqu'il n'y a qu'un seul état, il n'est pas modélisé.
  event // Pas d'événement.
  trans // Pas de transition car il n'y a pas d'état modélisé.
  assert
  if autorisation_test=false then ecart_courant_detecte_B=false
  else if capteur_de_position=correct and position_calculee_prec=correct
  then ecart_courant_detecte_B=false
  else if capteur_de_position=errone or position_calculee_prec=errone
  then ecart_courant_detecte_B=true);
edon
```

Le modèle de la partie système intéressante est obtenu en connectant les nœuds selon les échanges fonctionnels (cf. figure 6).

4 : Analyse du modèle « système »

Pour notre cas d'étude, nous nous intéressons aux hypothétiques défaillances et erreurs de conception logicielles qui conduisent à l'événement redouté : **gel du doseur hors situation de recueil**, c'est-à-dire lorsque le doseur est gelé sans qu'il ne se soit produit de défaillances justifiant ce gel doseur.
Plusieurs méthodes d'analyse, soutenues par des outils tels que le logiciel Cecilia™ OCAS (Dassault Aviation) [8] sont utilisables : simulation interactive, génération automatiques d'arbres de défaillances et de séquences (suites de n événements aboutissant à l'événement redouté. L'analyse par génération de séquences de notre cas d'étude, montre que les seuls scénarios de panne qui conduisent au gel doseur sont une défaillance de l'actionneur ou une défaillance du faisceau électrique entre le calculateur et l'actionneur. Il s'agit de scénarios pour lesquels le gel doseur est un état système sûr justifié. Ainsi, nous pouvons valider l'architecture du point de vue de la sécurité.

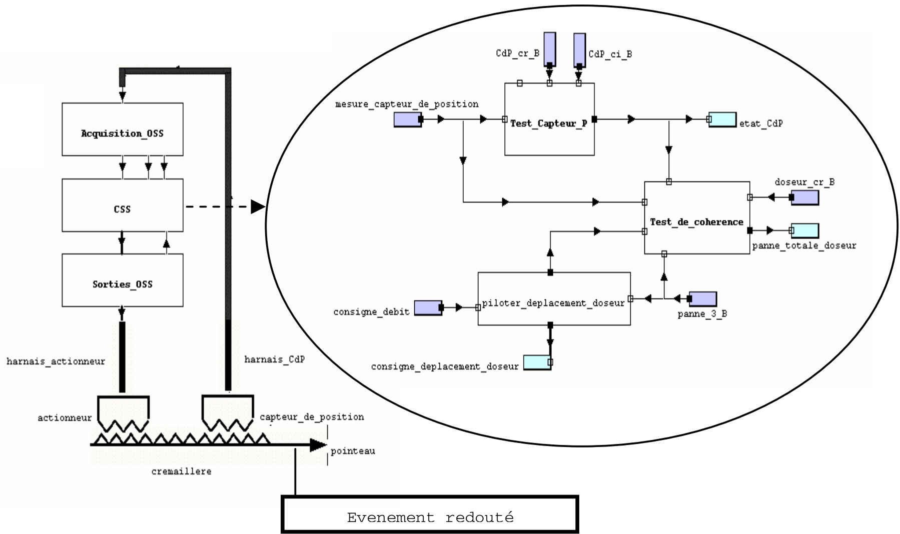


Figure 6 : Modèle AltaRica OCAS du cas d'étude

5 : Formulation des exigences fonctionnelles

L'analyse n'ayant pas nécessité de modification de l'architecture aucune exigence fonctionnelle n'est à rajouter à la spécification.

2+ : Identification d'hypothétiques erreurs de conception sur les fonctions logicielles

La modélisation précédente est complétée en tenant compte de comportements inattendus des fonctions logicielles (causés par exemple par d'hypothétiques erreurs de conception). Le tableau 2 montre cette extension pour la fonction test d'écart.

Le cas d'étude considéré comporte trois fonctions principales : le test du capteur de position, le test de cohérence (incluant le test d'écart), et la fonction de pilotage du doseur (calcul du déplacement). Pour les fonctions de détection, nous avons modélisé deux sortes d'erreurs de conception potentielles. La première est une erreur d'algorithme dont

Fonction	Mode de défaillance	Effet local	Effet Système
Test d'écart	Détection intempestive	Détection à tort d'une erreur de déplacement du doseur	Commande (intempestive) de gel → Gel du doseur
	Perte de la détection	Plus de détection possible d'une erreur de déplacement du doseur	Aucun (panne dormante) jusqu'à sollicitation.

Tableau 2 : Exemple d'une ligne de AMDE logicielle

la conséquence est la détection intempestive d'erreur (une valeur correcte sera considérée comme erronée); la seconde est aussi une erreur d'algorithme, mais qui a pour conséquence la non-détection de valeur erronée (cf. l'exemple tableau 2). Pour la fonction de pilotage du doseur, nous avons considéré qu'une erreur d'algorithme pouvait entraîner l'élaboration d'une consigne erronée.
Enfin, nous avons pris en compte d'hypothétiques erreurs de transmission des informations au niveau des acquisitions et des sorties de l'OSS.

3+ : Ajout dans le modèle des hypothétiques erreurs de conception logicielles
Les modèles des différents nœuds AltaRica (fonctions) sont modifiés en conséquence. Voici la modification du nœud AltaRica du test d'écart qui tient compte des deux modes de défaillances définis dans le tableau 2.

```
node test_ecart
  flow
    position_calculee_prec:{correct, errone}:in;
    capteur_de_position:{correct, errone}:in;
    autorisation_test:bool:in;
    ecart_courant_detecte_B:bool:out;
  state
    etat_test:{nominal, intempestif, perte};
  event
    EC_intempestif; // EC signifie Erreur de Conception
    EC_perte;
  trans
    etat_test=nominal |- EC_intempestif -> etat_test:=intempestif;
    etat_test=nominal |- EC_non_detection -> etat_test:=perte;
  assert
    if autorisation_test=false then ecart_courant_detecte_B=false
    else if (etat_test=nominal or etat_test=perte) and capteur_de_position=correct
    and position_calculee_prec=correct then ecart_courant_detecte_B=false
    else if (etat_test=nominal or etat_test=intempestif) and (capteur_de_position=
    errone or position_calculee_prec=errone) then ecart_courant_detecte_B=true
    //Voici les effets redoutés des hypothétiques erreurs de conception :
    else if etat_test=perte and (capteur_de_position=errone or
    position_calculee_prec=errone) then ecart_courant_detecte_B=false
    else if etat_test=intempestif and capteur_de_position=correct and
    position_calculee_prec=correct then ecart_courant_detecte_B=true ;
  init
    etat_test:=nominal;
edon
```

4+ : Analyse du modèle système enrichi
Un seul scénario impliquant un unique événement logiciel a été identifié :
`{'CSS.C_piloter_surveiller_doseur.Test_de_coherence.test_ecart.EC_intempestif'}`
Il s'agit d'une erreur de conception de la fonction logicielle : test d'écart du test de cohérence du composant C_piloter_surveiller_le_doseur du CSS, détectant à tort une panne.
Cet événement logiciel est critique puisque sa seule occurrence conduit à l'événement redouté.

5+ : Formulation des exigences de sécurité
La prise en compte de la criticité de l'événement logiciel précédent conduit à poser, sur le logiciel, l'exigence suivante : le test d'écart doit être exempt de fausse détection.

4 Déclinaison des exigences « système » vers le « logiciel »

L'étape suivante consiste à décliner les résultats de l'analyse de sécurité précédente en termes compréhensibles par le modèle SCADE du logiciel.

4.1. Modélisation « logiciel » : SCADE

Chez Turbomeca, le logiciel applicatif est modélisé en SCADE à l'aide de l'atelier SCADE SUITE™, d'édition, de validation formelle, simulation, de génération automatique de code et de documentation. Les modèles SCADE représentent graphiquement des fonctions définies en langage formel Lustre. Lustre est un langage synchrone, c'est-à-dire cadencé par le temps et non pas par des événements comme AltaRica. Il est compositionnel et hiérachique: les modèles se décomposent en parties élémentaires appelées « nœud ». Chaque nœud possède des entrées, des sorties, éventuellement des variables locales ainsi que des équations décrivant l'algorithme de la fonction.
Voici un exemple de planche SCADE, illustrant les concepts décrits section 2.2 :

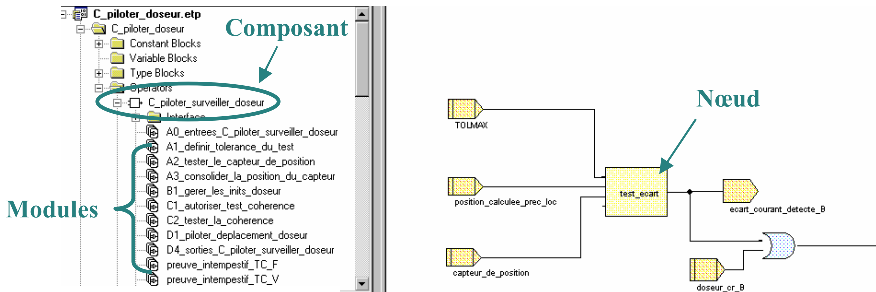


Figure 5 : Extrait de modèle SCADE

4.2 Correspondances entre les architectures des modèles AltaRica et SCADE

L'analyse de sécurité du niveau système a permis de cibler les événements logiciels jugés critiques pour le système, et les fonctions logicielles impliquées dans ces scénarios.
Les modèles AltaRica (niveau système) et SCADE (niveau logiciel applicatif) n'ayant pas le même niveau d'abstraction, il faut expliciter la correspondance entre ces deux modèles, afin de pouvoir cibler les parties logicielles SCADE concernées par les exigences de sécurité émises lors de l'analyse précédente.
Au niveau architecture des modèles, cette correspondance est assurée par l'utilisation d'une spécification de référence commune à l'analyse système et au développement du logiciel permettant de cibler les fonctions logicielles impliquées dans un scénario redou-

té. En effet, pour faciliter la réutilisation multi-projet, Turbomeca utilise une architecture de référence commune au système et au logiciel.

4.3 Déclinaison des exigences système vers le logiciel sur le cas d'étude

Une fois que le périmètre du modèle SCADE concerné a été identifié, les exigences doivent être exprimées en propriétés vérifiables sur ce modèle. La difficulté réside dans la traduction de termes d'un certain niveau abstrait (correct, erroné…) en des termes de niveau physique mettant en jeu des valeurs numériques. Examinons comment ce travail peut-être réalisé sur le cas d'étude.

L'analyse système a mis en avant l'événement logiciel critique suivant : **`CSS.C_piloter_surveiller_doseur.Test_de_coherence.test_ecart.EC_intempestif`**
La référence commune de spécification permet, par les noms utilisés, de localiser le périmètre SCADE concerné par l'événement : il s'agit de la fonction réalisant le test d'écart du test de cohérence du composant C_piloter_surveiller_le_doseur du CSS. De là, nous pouvons affiner le périmètre SCADE concerné : il s'agit du module SCADE nommé tester_la_cohérence. Par la coïncidence des noms des variables d'entrée et sortie des modèles AltaRica et SCADE, les nœuds SCADE réalisant l'algorithme de la fonction du test d'écart peuvent alors être identifiés.
L'événement redouté **`EC_intempestif`** doit être traduit en termes compris par les modèles SCADE. Pour cela on revient aux définitions utilisées pour la modélisation (cf tableau 2). La définition formelle se trouve dans les assertions faisant référence à l'événement intempestif du nœud AltaRica de la fonction test d'écart :

- La première partie de l'assertion explicite le comportement lorsqu'il se produit un intempestif :

```
if etat_test=intempestif and capteur_de_position=correct and
position_calculee_prec=correct then ecart_courant_detecte_B=true
```

- La seconde partie de l'assertion explicite le comportement dans le cas nominal :

```
if (etat_test=nominal or etat_test=perte) and capteur_de_position=correct
and position_calculee_prec=correct then ecart_courant_detecte_B=false
```

La première partie montre sous quelles conditions l'on constate un intempestif : c'est lorsque les deux entrées sont correctes. La seconde nous donne le comportement attendu lorsqu'il n'y a pas d'intempestif sous les mêmes conditions : c'est une absence de détection. Ainsi, nous pouvons en déduire une exigence formelle de « non intempestif » sur le logiciel applicatif :

```
capteur_de_position=correct and position_calculee_prec=correct
          imply ecart_courant_detecte_B=false
```

Cette exigence, compte tenu de la spécification de la fonction test d'écart peut être traduite par:

(1) $$|\, capteur_de_position - position_calculee_prec \,| < tolérance \Rightarrow ecart_courant_detecte_B = false$$

Remarque : nous proposons d’écrire l’exigence dans les spécifications du logiciel non seulement sous forme de propriété formelle, mais aussi textuellement afin d’en faciliter sa compréhension.

4.4 Analyse des exigences de sécurité sur le modèle SCADE

Pour évaluer la pertinence de la dérivation d'exigence, nous avons vérifié que les propriétés énoncées précédemment sont satisfaites par les modèles SCADE du logiciel à l'aide de l'outil de vérification exhaustive « design verifier »de SCADE SUITE™.
Cet outil évalue formellement une propriété, et fournit un des deux résultats suivants :
1 - la propriété est valide ;
2 - elle n’est pas valide, et le « design verifier » fournit un contre exemple (une combinaison d’instances (de valeurs) des variables qui mettent en défaut la propriété). Dans ce cas, il faut s’assurer que le scénario est plausible car la recherche peut nécessiter des abstractions pessimistes. Le cas échéant, une amélioration de la conception du logiciel doit être envisagée, afin d’éliminer la possibilité d’occurrence de ce contre exemple.

La propriété à tester est codée dans SCADE comme une équation (appelée preuve_intempestif dans la figure 7, et dont le code est représenté sur la partie droite de la figure). Pratiquement elle correspond à une équation supplémentaire (pour ne pas être intrusif dans le code) créée dans le composant contenant le périmètre de la fonction à valider. Les variables et paramètres de la propriété à vérifier sont celles du modèle logiciel écrit en SCADE. En particulier la valeur du seuil de tolérance est mémorisée dans la variable TOLMAX, variable utilisée à la fois dans l’écriture de la propriété et dans la fonction de test.
La vérification de l’exigence avec le « design verifier » affirme que cette propriété est valide. On conclut que l’événement logiciel critique ne peut en aucun cas se produire.

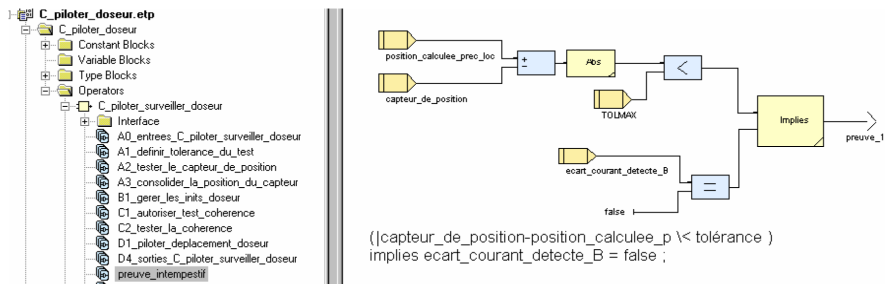


Figure 7 : propriété **(1)** écrite en SCADE

Comme le rapporte les paragraphes précédents, la preuve a été faite dans le composant SCADE C_piloter_surveiller_le_doseur avec les paramétrages du modèle logiciel.
Si nous nous étions intéressés à définir un domaine de valeur pour la validation de la propriété celle-ci devrait être considérée hors du composant. La valeur de la tolérance pourrait alors être laissée variable. Dans ce cas, l’exécution de la preuve fournit un

contre exemple avec une valeur de TOLMAX négative. On en conclut que si le paramétrage du code est erroné, alors l'événement redouté est susceptible de se produire. Une amélioration du code peut être aisément proposée dans ce cas là, il s'agit de toujours prendre la valeur absolue du paramètre, et ce pour éviter toute erreur de paramétrage.
Lorsqu'une modification est apportée à la conception du modèle SCADE, il faut réitérer l'exécution des preuves, afin de s'assurer de la non-régression (vérifier qu'il n'y a pas d'effets de bord relatifs à la modification).

5 Conclusion

La déclinaison des exigences de sécurité du système vers les sous-parties logicielles est aujourd'hui entravée par différents phénomènes. D'une part, il est difficile d'intégrer dans les analyses système un niveau de détail suffisant sur les parties logicielles. Cela provient de la complexité intrinsèque des parties logicielles, qui intègrent des fonctions variées mais aussi des techniques d'analyse traditionnelles peu compositionnelles, comme les arbres de défaillances. Pour lever ce premier obstacle nous avons proposé de réaliser les analyses de sécurité systèmes à l'aide de modèles AltaRica, bien adaptés à la prise en compte de système complexes. Une première contribution de cet article est de montrer à travers un exemple le niveau d'information et de modélisation AltaRica à introduire dans les phases d'analyses amont pour faciliter la déclinaison d'exigences vers le logiciel.
D'autre part, les processus actuels ne poussent pas à analyser l'impact des hypothétiques erreurs de conception logiciel sur l'architecture, le logiciel est supposé parfait en phase amont et l'on se doit de vérifier par la suite qu'il est bien conforme à ses spécifications. Notre expérience suggère que des modèles AltaRica définis au bon niveau de granularité permettent d'intégrer l'analyse des hypothétiques erreurs logicielles au même titre que les défaillances matérielles de manière peu coûteuse dès les phases amont. De plus l'exploitation de ces modèles a un effet bénéfique pour la suite des évaluations: la recherche des scénarios qui amènent à un événement redouté de niveau système a permis d'identifier dans notre exemple les cas de fonctionnement les plus critiques et de cibler les preuves de propriétés ou tests ultérieurs. Néanmoins, pour un système complexe, le nombre de scénarios amenant à un événement redouté peut être élevé. Il nous reste donc encore à proposer une démarche systématique de dépouillement des scénarios.
Un passage délicat reste la concrétisation des cas de fonctionnement abstraits en exigences SCADE. L'utilisation de modèles AltaRica distinguant les fonctions logicielles importantes pour l'analyse de sécurité permet de localiser assez naturellement les fonctions SCADE et les entrées/sorties sur lesquelles doivent porter les exigences dérivées. En revanche, la concrétisation des valeurs abstraites nécessite de se référer à la spécification des parties logicielles concernées. Nous souhaitons tracer plus formellement ces hypothèses de raffinement afin de générer plus automatiquement les exigences concrètes.
Pour éviter le problème d'interprétation, des travaux comme [6] proposent de réaliser les analyses de sécurité directement sur les modèles Scade. La démarche soulève au moins deux difficultés. D'une part la recherche de scénarios amenant à des événement redoutés explose et ne permet pas d'analyser l'architecture système dans sa globabilité. D'autre

part, le nombre et le détail des scénarios à analyser croîssent souvent avec le détail du modèle, rendant les résultats fournis difficilement exploitables. Une piste intéressante consisterait donc à mixer les deux types d'analyses : les parties à creuser peuvent être identifiées avec des modèles de la granularité des modèles Altarica puis l'investigation plus locale peut être directement réalisée sur les modèles SCADE détaillés.
Enfin, à notre connaissance, l'approche proposée n'avait pas été jusqu'alors explorée car jugée trop coûteuse. Notre travail montre que l'utilisation de techniques avancées de modélisation et d'analyse permet de dépasser cette limite. Nous allons poursuivre l'expérience à plus grande échelle pour pouvoir évaluer plus systématiquement les coûts et bénéfices de l'approche.

6 Bibliographie

[1] : ARP 4754, Certification considerations for highly-integrated or complex aircraft systems, SAE international, issued 1996-11.
[2] : ARP 4761, Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment, SAE international, issued 1996-12.
[3] : A. Villemeur, Sûreté de fonctionnement des systèmes industriels, Eyrolles, 1988.
[4] : www.isaac-fp6.org Projet européen « Improvement of Safety Activities on Aeronautical Complex systems » (ISAAC), référence FP6-2002-Aero-1-501848.
[5] O. Akerlund, P. Bieber, E. Boede, M. Bozzano, M. Bretschneider, C. Castel, A. Cavallo, M. Cifaldi, J. Gauthier, O. Lisagor, A. Lüdtke, S. Metge, C. Papadopoulos, T. Peikenkamp, L. Sagaspe, C. Seguin, H. Trivedi et L. Valacca, ISAAC, a framework for integrated safety analysis of functional, geometrical and human aspects, proceedings Embedded Real Time Software 2006, Toulouse (France), Janvier 2006
[6] : A. Joshi et M. Heimdahl, Model-based safety analysis of Simulink models using SCADE Design Verifier, proceeding Safecomp 2005, Fredrikstad (Norway), Septembre 2005, vol 3688 LNCS, Springer.
[7] : A. Arnold, A. Griffault, G. Point and A. Rauzy. The AltaRica formalism for describing concurrent systems, Fundamenta Informaticae, 40:109-124, 2000.
[8] : A. Rauzy, Mode automata and their compilation into fault trees. Reliability Engineering and System Safety, 78:1-12, 2002.
[9] : Aymeric Vincent, Conception et réalisation d'un vérificateur de modèles AltaRica, Thèse, LaBRI, Université Bordeaux I, Décembre 2003.
[10] : Manuel utilisateur OCAS V3.2, Dassault Aviation, 2005.
[11] : Manuel utilisateur SCADE, Esterel Technologies, 2006.
[12]: Halbwachs N., Caspi P., Raymond P., Pilaud D.: *The Synchronous Data Flow Programming LUSTRE*, Proceedings of the IEEE volume 79, N° 9, 1991.
[13] : C. Castel et C. Seguin, Modèles formels pour l'évaluation de la sûreté de fonctionnement des architectures logicielles d'avionique modulaire intégrée, AFADL, 2001.
[14] : P. Bieber, C. Castel, C. Kehren et C. Seguin, Analyse des exigences de sûreté d'un système électrique par model-cheking, Actes du congrès Lambda-Mu 14, 2004.
[15] : J-M. Bosc, C. Castel, P. Darfeuil, Y. Dutuit, S. Humbert et C. Seguin, Méthodologie de modélisation AltaRica pour la sûreté de fonctionnement d'un système de propulsion hélicoptère incluant une partie logicielle, actes du congrès Lambda Mu 15,

pulsion hélicoptère incluant une partie logicielle, actes du congrès Lambda Mu 15, 11/06.

On the Design and the Implementation of a Game-based Model for Open Systems: Current Status and Perspectives.*

Marco Faella[1] and Axel Legay[2]

[1] Dipartimento di Scienze Fisiche, Università di Napoli "Federico II", Italy
[2] Department of Computer Science, University of Liège, Belgium

Abstract. Sociable interfaces are a game-based model with rich communication primitives that facilitate the modeling of software and distributed systems. The first model of sociable interfaces has been introduced in [9], and then implemented in a tool called TICC [2]. While the primary goal of TICC was to perform the composition of two or more sociable interfaces, it has now evolved into a complete specification and verification tool. This paper presents the current status of the sociable interfaces model and of TICC. It discusses the improvements that have been brought to previous versions of the model and the tool, and investigates future directions of research.

1 Introduction

The trend in software and system engineering is towards component-based design: systems are designed by combining small components into bigger ones. Components offer thus the unit in which complex design problems can be decomposed, allowing the reduction of a single complex design problem into smaller design problems, more manageable in complexity, that can be solved in parallel. Components also provide a unit of reuse, defining the boundaries in which functionality can be packaged, documented and reused.

Components are designed to work as parts of larger systems: they make assumptions on their environment, and they expect that these assumptions will be met in the actual environment. In other words, a component is typically an open system which has some free inputs provided by other components and which in turn provides inputs to other components. It is thus obvious that the effective reuse of software requires adequate documentation of the component's behavior and the conditions under which it can be used, along with methods for checking that components are assembled in an appropriate way. Such documentation is commonly referred to as the *interface* of the component.

There have been many works on the design and implementation of good interfaces for components. Most of those works focus on capturing the *data dimension* of interfaces ("What are the value constraints on data communicated between components?") [22].

In a series of recent works [5, 14, 12], de Alfaro and Henzinger introduced *interface theories* that is as a formal notion of component interfaces that uses games to represent

* Axel Legay is supported by a F.R.I.A grant

the interaction between the behavior originating within a component and the behavior originating from the component's environment.

This game-based model is able to capture the *protocol* dimension of interfaces ("What are the temporal ordering constraints on communication events between components?") which makes it similar to a type system: indeed, it could be termed a "behavioral" type system for component interaction (see [17]).

In recent work [9], we introduced sociable interfaces — a concrete model of interface theories with rich communication primitives and design facilities. The theory of sociable interfaces was first presented in [9], and then implemented in a tool called TICC. The preliminary version of TICC was described in [2], but the tool (and so the model) has now been substantially improved[3].

The first goal of this paper is to present the current version of the sociable interface model and of TICC, as well as to discuss the perspectives for future work. The second goal is to make the link between the theory behind the tool and what the user really needs to know to use it.

Due to space limitations, we complete the practical part of the paper with a technical report [11].

2 Open Systems and Games, Where is the Link?

In this section we sketch the main features of the interface theories approach and we describe the link with game theory. The reader is referred to [8, 15] for more details.

In interface theories, an interface support the component-based design in the following ways.

Interface specification. An interface specifies how a component interacts with its environment. It describes the input assumptions that the component makes on the environment (methods that can be called) and the output guarantees it provides (methods calls, ...). Interfaces capture the I/O behavior of a component by an automaton whose syntax is similar to the I/O automata of [22]. Unlike traditional models, including I/O automata, that at every state must be receptive to every possible input event, in interface theories it is possible that inputs are illegal (cannot be accepted) at some states. One of the main advantages of making explicit assumptions about the environment is that it gives rise to an optimistic compatibility test when interfaces are composed, as explained below. Moreover, from a practical point of view, the ability to forbid inputs removes the need to specify "what happens" when taking an undesirable input. On the other hand, it is important to guarantee that the interface works in at least one environment, i.e. that it is well-formed. An interface is naturally modeled as a game between the players Input and Output. Input represents the environment: the moves of Input represent the inputs accepted from the environment. Output represents the component: the moves of Output represent the possible outputs generated by the component. Then, an interface is well-formed (i.e. can work in at least one environment) if the Input player has a winning strategy in the game, which means that

[3] The first version of the tool was limited to the composition of interfaces specified with a restrictive input language.

the environment can meet all input assumptions.

Interface Composition. Like most existing models, interfaces interact through the synchronization of common input and output events. The interpretation of inputs and outputs as assumptions and guarantees, respectively, implies that, when composing two interfaces *P* and *Q*, we have to ensure that *P*'s output guarantees satisfy *Q*'s input assumptions and vice versa. Concretely, consider the two interfaces *P* and *Q*, in one state of the composition. If *P* wants to emit an output that cannot be accepted by *Q* in that state (i.e. an output guarantee that violates an input assumption), then a *local incompatibility* occurs. While many approaches would be pessimistic and consider the two interfaces to be incompatible, the interface approach is optimistic, by assuming that the environment will steer away from locally incompatible states. Thus, two interfaces are *compatible* if there exists an environment to use the components together, and ensure that the assumptions of both are met. Interface composition thus consists in synthesizing the most liberal input strategy in the composite system that avoids all locally incompatible states. This can be done by classical game-theoretic algorithms [13].

Interface Refinement and Model Checking. Composition is certainly the most important operation that differentiates the theory of interfaces from other approaches. However it is also worth mentioning that, since the model distinguishes between Inputs and Outputs, the notion of refinement [21] naturally reduces to the one of alternating simulation [4] between the players of the components that have to be compared. Moreover, while this game-based model do not forbid model checking open systems with classical closed system logics such as branching time temporal logic CTL [7], it also offers the possibility of using logics that have been designed for open systems, such as alternating temporal logic [3].

As pointed out in [8], most of the ideas on which the approach of interface theories is based have been introduced earlier in the literature. The idea of specifying independent constraints for input and output behavior is present in trace theories [16]. All the algorithms used to solve the game are standard [4]. Refinement is based on alternating simulation [4]. Checking compatibility of composition can in certain cases be solved by controller synthesis [27]. Thus, the novelty in the game-based approach for open systems is not in any isolated algorithmic aspect, but rather in the recognition that refinement, composition, and synthesis can be homogeneously cast as game relations and problems. Note that the work done on game semantics for process algebra [25] and programming languages [1] also suggests that games provide an unifying framework for interaction between components. A comparison between those models and ours has been given in [17].

3 Before Sociable Interfaces

There have been many recent works on the design of concrete models of interface theories. They can be divided into two categories: (1) interface automata [12] that are asynchronous models that communicate via disjoint [4] input and output actions, and (2)

[4] This separation was mainly motivated to keep the distinction between the Input and the Output players.

interface modules [5] that are synchronous models that communicate via disjoint input and output variables.

While those models clearly show the applicability of the approach, they impose drastic restrictions on the way that components can communicate together. As an example, due to the distinction between the names of input and output actions (resp. variables), none of those models allow several components to communicate with a third one using the same action (resp. variable) name. Moreover, the fact that actions and variables cannot be combined introduces difficulties to the design of many interesting systems, such as those that manipulate global resources.

4 The Sociable Interface Model: Theory

This section gives a theoretical description of the sociable interface model. We assume a fixed set $\mathscr{V}$ of variables that are interpreted over a given domain $\mathscr{D}$. Given $V \subseteq \mathscr{V}$, a *state* over V is a mapping $s : V \rightarrow \mathscr{D}$ and $[\![V]\!]$ denotes the set of all states over V. For a set of variables $U \subseteq V$, and a state $s \in [\![V]\!]$, the projection of s on U is a state $t \in [\![U]\!]$ denoted as $s[U]$. For two disjoint sets of variables V_1 and V_2, and two states $s_1 \in [\![V_1]\!]$ and $s_2 \in [\![V_2]\!]$, the operation $(s_1 \circ s_2)$ composes the two states resulting in a new state $s = s_1 \circ s_2 \in [\![V_1 \cup V_2]\!]$, such that $s(x) = s_1(x)$ for all $x \in V_1$ and $s(x) = s_2(x)$ for all $x \in V_2$.

Given a set V of variables, we denote by $Preds(V)$ the set of first-order predicate formulas with free variables in V; for the moment, we assume that these predicates are written in some specified first-order language. We let $V' = \{x' \mid x \in V\}$ be the set of primed versions of variables in V. Intuitively, a variable $x' \in V'$ represents the *next value* of $x \in V$. Given a formula $\psi \in Preds(V)$ and a state $s \in [\![V]\!]$, we write $s \models \psi$ if the predicate formula ψ is true when its free variables are interpreted as specified by s. Given a formula $\rho \in Preds(V \cup V')$ and two states $s, s' \in [\![V]\!]$, we write $\langle s, s' \rangle \models \rho$ if the formula ρ holds when its free variables $x \in V$ are interpreted as $s(x)$, and its free variables $x' \in V'$ are interpreted as $s'(x)$. Given a set U of variables, we define the formula $Unchgd(U) = \bigwedge_{x \in U}(x' = x)$, which states that the variables in U do not change their value in a transition. Given a predicate $\psi \in Preds(V)$, we denote by ψ' the predicate obtained by substituting x by x' in ψ, for all $x \in V$.

With those definitions, we can define a sociable interface as follows:

Definition 1. A *sociable interface* is a tuple $M = (Act^G, Act^L, \mathscr{D}, V^G, V^L, V^H, W, \rho^{IL}, \rho^{IG}, \rho^O, \rho^L, \psi^I, \psi^O, I)$. Act^G is a set of set of *global actions*, and Act^L is a set of *local actions*. We set $Act = Act^G \cup Act^L$. V^G is a set of *global variables*, V^L is a set of *local variables*, and $V^H \subseteq V^G$ is a set of *history variables*. We require $V^L \cap V^G = \emptyset$. We set $V^{\text{all}} = V^L \cup V^G$ and $V = V^L \cup V^H$. All variables are interpreted over the domain $\mathscr{D}$. For each action $a \in Act$ we denote by $W(a)$ the set of variables that can be modified by a. For each $a \in Act^G$, the predicates $\rho^{IL}(a) \in Preds(V^{\text{all}} \cup (V^{\text{all}})')$, $\rho^{IG}(a) \in Preds(V^{\text{all}} \cup (V^G)')$, and $\rho^O(a) \in Preds(V^{\text{all}} \cup W(a)')$ are respectively the input local, the input global, and the output transition predicates for a. We require $\rho^{IL}(a)$ to be *deterministic* w.r.t. variables in V^L, that is, for all $a \in Act^G$, all $s \in [\![V^{\text{all}}]\!]$, and all $t \in [\![(V^G)']\!]$, there is a unique $u \in [\![(V^L)']\!]$ such that $s \circ t \circ u \models \rho^{IL}(a)$. We let $\rho^I(a) = \rho^{IL}(a) \wedge \rho^{IG}(a)$. For each $a \in Act^L$, the predicate $\rho^L(a) \in Preds(V^{\text{all}} \cup (V^L)')$

is the *local transition predicate* for a. Local transitions represent internal choices made by the component that cannot be viewed from the outside world. We define $\psi^O \in Preds(V^{\text{all}})$ and $\psi^I \in Preds(V^{\text{all}})$ to be respectively the output and input *invariant predicates* of M. Finally, we define $I \in Preds(V^{\text{all}})$ to be the predicate that characterizes the set of *initial states* of M.

This model differs from the one given in [9] by the addition of local actions and of an initial condition. The initial condition is the prelude announcement to a new feature of the tool: the manipulation of sets of states.

Sociable interfaces do not distinguish between input and output actions/variables. Rather, they associate a set of variables that can be modified by the action, as well as an output and an input transition relation that describe the ways in which the variables can be modified when the component, or its environment, output the action. In the model, one assumes that output and local transitions are the only responsible for the updates of the value of the variables. The global part of an input transition — which represents an input assumption — only makes assumptions on the value of the global variables. The local part of an input transition, instead, allows the interface to immediately react to the reception of an action from the environment, subject to two restrictions: First, the interface can only modify the value of its local variables, and second, it must do so in a *deterministic* fashion. These restrictions are necessary to ensure that each state change is driven by the component issuing the output action. The component which is receiving the action can express assumptions on how the action will update the global variables, but it cannot participate in the choice of the new values.

We denote $S = [\![V^{\text{all}}]\!]$ to be the set of states of M. As a shorthand, we let $\varphi^I = \{s \in S \mid s \models \psi^I\}$, $\varphi^O = \{s \in S \mid s \models \psi^O\}$. We define the restriction of transitions w.r.t. invariants, i.e. $\widehat{\rho}^I(a) = \rho^I(a) \wedge (\psi^I)'$, $\widehat{\rho}^O(a) = \rho^O(a) \wedge (\psi^O)' \wedge Unchgd(V^{\text{all}} \setminus W(a))$, and $\widehat{\rho}^L(a) = \rho^L(a) \wedge (\psi^O)' \wedge Unchgd(V^{\text{all}} \setminus V^L)$.

Note that variables whose next value is not specified in ρ^O (resp. ρ^L) are assumed to keep their value in $\widehat{\rho}^O$ (resp. $\widehat{\rho}^L$). The latter is reasonable since output (resp. local) transitions reflect the behaviors of the component. However, no assumption is made on ρ^I and $\widehat{\rho}^I$. The reason is that ρ^I represents the behaviors of the environment.

4.1 The Game under the Model

In this section, we assume a sociable interface $M = (Act^G, Act^L, \mathscr{D}, V^G, V^L, V^H, W, \rho^{IL}, \rho^{IG}, \rho^O, \rho^L, \psi^I, \psi^O, I)$, and we describe its semantics in term of a turn-based game between the two players Input and Output.

The game is played in an arena that is $S = [\![V^{\text{all}}]\!]$. At each round, from the current state, both players simultaneously choose a move that defines the next state of the game.

Definition 2 (Moves). The sets $\Gamma^I(M,s)$ and $\Gamma^O(M,s)$ of Input and Output moves at $s \in S$ are defined as follows:

$$\begin{aligned}
\Gamma^I(M,s) = & \{\Delta_0\} \times \{s' \in [\![V^{\text{all}}]\!] \mid s'[V] = s[V]\} \cup \\
& \{\langle a, s' \rangle \in Act^G \times [\![V^{\text{all}}]\!] \mid \langle s, s' \rangle \models \widehat{\rho}^I(a)\} \\
\Gamma^O(M,s) = & \{\Delta_0\} \cup \{\langle a, s' \rangle \in Act^G \times [\![V^{\text{all}}]\!] \mid \langle s, s' \rangle \models \widehat{\rho}^O(a)\} \cup \\
& \{\langle a, s' \rangle \in Act^L \times [\![V^{\text{all}}]\!] \mid \langle s, s' \rangle \models \widehat{\rho}^L(a)\}.
\end{aligned}$$

The $\{\Delta_0\}$ is an extra move to ensure that both players have a move to propose in each state. Note that, when Input plays the move Δ_0, it can also choose a new assignment to the history-free variables. This models the fact that history-free variables can be modified by environment actions that are not known to the interface.

At each game round, both players choose a move from the corresponding set of enabled moves. The outcome of their choice is defined as follows.

Definition 3 (Move Outcome). For all states $s \in S$ and moves $m^I \in \Gamma^I(M,s)$ and $m^O \in \Gamma^O(M,s)$, the *outcome* $\delta(M,s,m^I,m^O) \subseteq S$ of playing m^I and m^O at s can be defined as follows.

$$\delta(M,s,\langle\Delta_0,s'\rangle,\Delta_0) = \{s'\}, \qquad \delta(M,s,\langle\Delta_0,s'\rangle,\langle a,t'\rangle) = \{s',t'\},$$

$$\delta(M,s,\langle a,s'\rangle,\Delta_0) = \{s'\}, \qquad \delta(M,s,\langle a,s'\rangle,\langle b,t'\rangle) = \{s',t'\}.$$

The move outcomes show the priority that a "real" move has on the output Δ_0 move, but not on the input Δ_0 move. For $s \in S$, we define the set of *runs* starting from s as the set $Runs(M,s) \subseteq S^\omega$ of all infinite sequences $s_0s_1s_2\ldots$, such that $s_0 = s$, and for all $i{\geq}0$, $s_{i+1} \in \delta(M,s_i,m^I,m^O)$, for some $m^I \in \Gamma^I(M,s_i)$, $m^O \in \Gamma^O(M,s_i)$. We also set $Runs(M) = \bigcup_{s\in S} Runs(M,s)$. Given $i{\geq}0$, The finite prefix $\sigma_{0:i}$ of a run $\sigma = s_0s_1s_2\ldots s_i\ldots$ is the finite sequence $s_0s_1s_2\ldots s_i$ that is constituted of the $i+1$ first states of σ. A *strategy* for player $p \in \{I,O\}$ is a function π^p that for each run $\sigma \in Runs(M)$ associates a move $\pi^p(\sigma_{0:i}) \in \Gamma^p(M,s_i)$ to each finite prefix $\sigma_{0:i}$ of σ. We denote by Π^I_M and Π^O_M the set of input and output strategies for M, respectively. Let $\pi^I \in \Pi^I_M$ and $\pi^O \in \Pi^O_M$, the set $\hat{\delta}(M,s,\pi^I,\pi^O)$ of π^I and π^O from s consists of all runs $\sigma = s_0s_1s_2\ldots$ such that $s = s_0$, and for all $i{\geq}0$, $s_{i+1} \in \delta(M,s_i,\pi^I(\sigma_{0:i}),\pi^O(\sigma_{0:i}))$. Given a state $s \in S$ and a goal $\gamma \subseteq Runs(M,s)$, we say that s is *winning* for input (resp. output) with respect to γ, and we write $s \in Win^I(M,\gamma)$ (resp. $s \in Win^O(M,\gamma)$), iff there is $\pi^I \in \Pi^I_M$ (resp. $\pi^O \in \Pi^O_M$) such that for all $\pi^O \in \Pi^O_M$, $\hat{\delta}(M,s,\pi^I,\pi^O) \subseteq \gamma$ (resp. $\pi^I \in \Pi^I_M$, $\hat{\delta}(M,s,\pi^I,\pi^O) \subseteq \gamma$).

Definition 4 (Normal Form). *[9]* We say that M is in *normal form* iff $\varphi^I = Win^I(M,\Box\varphi^I)$, and $\varphi^O = Win^O(M,\Box\varphi^O)$, where $\Box X = \{s_0s_1s_2\ldots \in Runs(M) \mid \forall i{\geq}0\,.\, s_i \in X \subseteq S\}$.

The computation of $Win^I(M,\Box X)$ is referred to a safety game whose objective is $\Box X$. Definition 4 induces the trivial, but important lemma.

Lemma 1. *If M is in normal form, then it holds:*

$$\forall s \in \varphi^I \,.\, \forall (a,s') \in \Gamma^O(M,s) \,.\, s' \in \varphi^I,$$
$$\forall s \in \varphi^O \,.\, \forall (a,s') \in \Gamma^I(M,s) \,.\, s' \in \varphi^O.$$

Definition 5 (Well-formed Sociable Interface). *We say that M is* well-formed *iff (1) it is in normal form, and (2) $\varphi^I \cap \varphi^O \cap \{s \in S \mid s \models I\} \neq \emptyset$.*

Note that point (2) of the definition ensures that M is well-formed only if the states that are well-formed can be reached from its initial states.

5 The Sociable Interface Model: Practice

The sociable interface model of Section 4 has been implemented in a tool called TICC (Tool for Interface Compatibility and Composition) The documented code of TICC is freely available and can be downloaded from *http://dvlab.cse.ucsc.edu/dvlab/Ticc*. This website is a Wiki that also contains the documentation for the tool, and several additional examples. TICC allows users to specify sociable interfaces, called "modules", using a textual language based on guarded commands, perform operations on the modules, and verify properties of modules using functions that extend the capabilities of the OCaml [20] command-line. In the rest of the section, we describe the internal representation used by the tool. The next section outlines how the tool is used.

In the rest of this Section we consider a sociable interface $M = (Act^G, Act^L, \mathscr{D}, V^G, V^L, V^H, W, \rho^{IL}, \rho^{IG}, \rho^O, \rho^L, \psi^I, \psi^O, I)$. Internally, TICC relies on a symbolic representation of sociable interfaces based on MDDs [24, 26] that are used to represent the predicates ρ^{IL}, ρ^{IG}, ρ^O, ρ^L, ψ^I, ψ^O and I. MDDs are graph-like data structures that allow to represent and manipulate functions of the type $A \to \{\mathrm{T}, \mathrm{F}\}$ (i.e. predicates over A), for a finite set A of variables whose domain range over the Boolean and the bounded integers. It is well known that MDDs are a very compact representation on which Boolean operations and quantifier elimination can be performed efficiently. By abuse of notation, given a predicate $X \in Preds(V^{\mathrm{all}})$, we denote by X also the set of states $\{s \in S \mid s \models X\}$. However, in TICC, all the algorithms that will be described are implemented with MDDS that represent set of states.

We now consider the implementation of the first operation we need to perform on an interface, i.e. checking well-formedness. As mentioned in Section 4, this operation reduces to the computation of $Win^p(M, \Box\varphi)$ for $p \in \{I, O\}$. It is well known (see [4]) that the set of winning states can be characterized as a fixpoint of an operator involving the so-called *controllable predecessors operators* $Cpre^I(\cdot)$ and $Cpre^O(\cdot)$.

Definition 6 (Controllable Predecessor Operator). For all predicates $X \in Preds(V^{\mathrm{all}})$, we have:

$$Cpre^I(X) = \exists m^I \in \Gamma^I(M,s) \,.\, \forall m^O \in \Gamma^O(M,s) \,.\, \forall t \in \delta(M,s,m^I,m^O) \,.\, t \models X$$
$$Cpre^O(X) = \exists m^O \in \Gamma^O(M,s) \,.\, \forall m^I \in \Gamma^I(M,s) \,.\, \forall t \in \delta(M,s,m^I,m^O) \,.\, t \models X.$$

Intuitively, $Cpre^I(X) \in Preds(V^{\mathrm{all}})$ (resp.$Cpre^O(X) \in Preds(V^{\mathrm{all}})$) is a predicate that holds true for each state $s \in S$ from which the Input (resp. Output) player has a move that leads to X for each possible counter-move of the Output (resp. Input) player. We have the following definition.

For all $\varphi \in Preds(V^{\mathrm{all}})$, we have $Win^I(M, \Box\varphi) = \nu X.\ [\varphi \wedge Cpre^I(X)]$, and $Win^O(M, \Box\varphi) = \nu X.\ [\varphi \wedge Cpre^O(X)]$, where $\nu X \,.\, f(X)$ denotes the greatest fixpoint of the operator f. Since $Cpre^I(\cdot)$ (resp. $Cpre^O(\cdot)$) is monotonic, the fixpoints exist and can be computed by Picard iteration, i.e. $X_0 = \varphi, X_{i+1} = \varphi \wedge Cpre^I(X_i), \ldots$, and $X_n = X_{n+1} = Win^I(M, \Box\varphi)$. Due to the definition of the moves and move outcomes of our game, it is possible to apply reasoning that is identical to the one given in [9], and obtain the following result.

Lemma 2. *For all predicates* $\varphi \in Preds(V^{\mathrm{all}})$*, we have*

$$Win^I(M, \Box\varphi) = \nu X.\, [\varphi \wedge \forall Pre^O(X)],\ and\ Win^O(M, \Box\varphi) = \nu X.\, [\varphi \wedge \forall Pre^I(X)],$$

where

$$\forall Pre^O(X) = \bigwedge_{a \in Act^G} \forall (V^{\text{all}})'.\, (\widehat{\rho}^O(a) \Rightarrow X') \wedge \bigwedge_{a \in Act^L} \forall (V^{\text{all}})'.\, (\widehat{\rho}^L(a) \Rightarrow X')$$

$$\forall Pre^I(X) = \bigwedge (\exists (V^{\text{all}})'.\, X' \wedge Unchgd(V)) \wedge \bigwedge_{a \in Act^G} \exists (V^{\text{all}})'.\, (\widehat{\rho}^I(a) \wedge X').$$

Note that $\forall Pre^O(X) \in Preds(V^{\text{all}})$ and $\forall Pre^I(X) \in Preds(V^{\text{all}})$.
It is possible to obtain several algorithms to compute the fixpoints of $\forall Pre^O(X)$ and of $\forall Pre^I(X)$ simply by reorganizing the Picard iteration. Some of those algorithms have been implemented and documented in the tool.

5.1 An Introduction to the Use of TICC

This section is a summarized introduction to the use of TICC. More detailed examples that illustrate the full input language of the tool are given in [11].

We illustrate the modeling language of TICC by means of a very simple example: a fire detection system that is composed of a control unit and several smoke detectors. When a detector senses smoke (input *smoke*), it reports it by emitting an output *fire*. When the control unit receives the input *fire* from any of the detectors, it emits the output *call_fd*, corresponding to a call to the fire department. Additionally, an input *disable* disables both the control unit and the detectors, so that the smoke sensors can be tested without triggering an alarm.

Below, we provide the code for the control unit module (`ControlUnit`) and for one of the (several) fire detectors (`FireDetector1`):

```
module ControlUnit:
    var s: [0..3] // 0=waiting, 1=alarm raised, 2=fd called, 3=disabled
    initial : s = 0
    iinv : true
    ooinv : true
    input fire:     { local: s = 0 | s = 1 ==> s' := 1
                        else  s = 2         ==>         }
    input disable:  { local: true ==> s' := 3 }
    output call_fd: { s = 1 ==> s' = 2 }
endmodule

module FireDetector1:
    var s: [0..2] // 0=idle, 1=smoke detected, 2=inactive
    initial : s = 0
    iinv : true
    ooinv : true
    input smoke1:  { local: s = 0 | s = 1 ==> s' := 1
                     else   s = 2 ==>        } // do nothing if inactive
    output fire:   { s = 1        ==> s'  = 2 }
    input fire:    { } // other modules can detect fire too
    input disable: { local: true  ==> s' := 2 }
endmodule
```

The body of each module starts with the (possibly empty) list of its local variables (in the example, those variables are used to encode the locations of the modules). This list is followed by the declarations of the initial condition of the module and by its input and output invariants. Note that both the declaration of the initial condition and of the invariants can be omitted, in which case they are automatically set to true. The transitions are specified using guarded commands *guard* $\Rightarrow$ *command*, where *guard* and *command* are Boolean expressions over the local and global variables; as usual, primed variables refer to the values after a transition is taken. For instance, the output transition *fire* in module `FireDetector1` can be taken only when the local variable s has value 1, and it leads to a state where $s = 2$. When a transition starts with the keywords `input` or `output`, then the associated action is automatically considered to be global. The variables whose next value is not precised by the user are updated following the theory given in Section 4. We now go through the details of parsing a TICC program. We consider the fire detection systems and we suppose that it is given in a file whose name is `fire.si`. First the user has to invoke TICC using the command `ticc` from the shell. The result of this operation is an OCaml prompt from where one must type: "`open Ticc;;`". At this point the functions in the module of TICC become available at the top level (these functions are documented in the file *ticc/doc/api/Ticc.html* and in [11]). Next, one has to read a TICC program from a file, here `fire.si`. This is achieved by typing the command "`parse "fire.si";;`". The `parse` function reads in a .si file describing modules and possibly global variables, and places these definition into a global namespace. If the .si file does not follow the syntax of the input language, the function reports an appropriate error message. After parsing a TICC program, it is possible to construct the symbolic internal representation of each module. As an example, here is the command to construct the representation of `ControlUnit`: "`let controlunit = mk_sym "ControlUnit";;`". The command `mk_sym` constructs MDDs representations for all the predicates of the module. Moreover, it plays a safety game and restricts the invariants to ensure that the module is well-formed. Thus, there can be a difference between the module specified by the user and the one that will be used by the tool. Such differences can be detected by using the printout functions of TICC. As an example, the following command can be used to print out the content of module `ControlUnit`: "`print_symmod controlunit;;`". The printout functions are particularly useful for debugging in TICC. In the new release of the tool, we also added a random simulation function on symbolic modules. This function generates an HTML file with the result of the simulation (see Example 2). This is particularly useful in the early stages of model construction, to confirm that the model behaves as intended. Finally, let us note that one can also write *script files* for TICC. A script file is a file that groups a set of commands that can be executed in one step. One can invoke TICC to execute the script file with the following command from the shell prompt: "`ticc scriptfile`".

6 Composing Sociable Interfaces

In this section, we consider the composition of two sociable interfaces M_1 and M_2, where $M_i = (Act_i^G, Act_i^L, \mathscr{D}, V_i^G, V_i^L, V_i^H, W_i, \rho_i^{IL}, \rho_i^{IG}, \rho_i^O, \rho_i^L, \psi_i^I, \psi_i^O, I_i)$. In sociable interfaces, composition is done in four steps. The theory behind those steps has already

been given in [9] and the extension to our model is immediate. Here, we mainly focus on what the user really needs to know when performing the operation in TICC.

6.1 The Composability Condition

First, we need to check whether M_1 and M_2 are composable. This requires checking that if an action $a \in Act_1^G$ (respectively Act_2^G) of M_1 (resp. M_2) has an output transition that can modify a history variable of M_2 (resp. M_1), then M_2 (resp. M_1) has an input transition for action $a \in Act_2^G$ (resp. $a \in Act_1^G$). This check is the main motivation for distinguishing between history and history-free variables: an interface should only know all actions of other interfaces that modify its history variables. If we dropped the distinction, then an interface would have to know all actions of other interfaces that can change any of its variables, and this could greatly increase its number of transitions. Note that, in the new version of the tool, we implemented a pattern matching mechanism that can substantially reduce the number of transitions. As an example:

```
input a* : {implementation}
```

is equivalent to say that a module accepts all the input transitions whose first letter is "a". The utility of this mechanism is illustrated in the "house example" of [11].

6.2 The product

If the two interfaces are compatible, then we can define their product. The product of M_1 and M_2 is a sociable interface $M_{12} = (Act_{12}^G, Act_{12}^L, \mathscr{D}, V_{12}^G, V_{12}^L, V_{12}^H, W_{12}, \rho_{12}^{IL}, \rho_{12}^{IG}, \rho_{12}^O, \rho_{12}^L, \psi_{12}^I, \psi_{12}^O, I_{12})$. First, we have $Act_{12}^G = Act_1^G \cup Act_2^G$, $Act_{12}^L = Act_1^L \cup Act_2^L$, $V_{12}^G = V_1^G \cup V_2^G$, $V_{12}^L = V_1^L \cup V_2^L$, $V_{12}^H = V_1^H \cup V_2^H$, $\psi_{12}^I = \psi_1^I \wedge \psi_2^I$, $\psi_{12}^O = \psi_1^O \wedge \psi_2^O$, and $I_{12} = I_1 \wedge I_2$. The transitions of M_{12} are combinations of the transitions of M_1 and M_2. For each shared global action, the output transition of M_1 (resp. M_2) synchronizes (in our model, synchronization boils down to conjoining the corresponding predicates) with the *local part* of the input transition of M_2 (resp. M_1), and gives rise to output transitions in the product. The reason not to synchronize with the global part of the input is to ensure that only output transitions can modify the values of the global variables (inputs are only supposed to make assumptions on them). The input transitions of M_1 and M_2 corresponding to the same shared global actions are also synchronized, and lead to an input transition in the product. Finally, the interfaces interleave asynchronously on transitions labeled by non-shared global actions, and on local actions.

6.3 Locally Incompatible States

The product M_{12} can contain locally incompatible states in which one of the interfaces being composed wants to issue an output transition labeled by a shared global action, while the other interface does not have a corresponding (same action name) global input transition from that state which agrees with the output transition on the updates of global variables. We denote by *Good* the set of locally compatible states.

6.4 Synthesizing a Strategy

After computing M_{12} and *Good*, the next operations is to compute the set of states *Win* from which the Input player of M_{12} has a strategy to always stay in *Good*. In other words, we play a safety game whose arena is the set of states of the product, and whose objective is the set *Good*. The set *Win* is used to restrict the input invariant and the initial condition of M_{12} which is equivalent to restrict the environments in where it can be used. This is an optimistic approach, since two interfaces are considered to be compatible if they can work in at least one environment.

6.5 Implementation

The implementation of the composition operation is direct since it only requires Boolean combinations of predicates (for checking the compatibility condition, for computing M_{12}, and for computing the predicate representing the set *Good*) and the solution of a safety game (for computing *Win*) (see Section 5).

The composition of two compatible interfaces M_1 and M_2 is denoted by $M_1 \| M_2$. We have the following theorem.

Theorem 1. *For all sociable interfaces M_1, M_2, and M_3, either both $(M_1 \| M_2) \| M_3$ and $M_1 \| (M_2 \| M_3)$ are undefined, because some of the interfaces are not compatible, or $(M_1 \| M_2) \| M_3 = M_1 \| (M_2 \| M_3)$.*

Theorem 1 shows that sociable interfaces support incremental design, i.e. the components can be composed and checked in any order.

6.6 The Composition Operation in TICC

In TICC, the user performs composition with the function "`compose`" followed by the name of two symbolic modules. The result of this call is either a new symbolic module, or an error message if the two modules are not compatible.

Example 1. Consider the fire detection system of Section 5.1, and suppose also the existence of a faulty detector `Wrong_FireDetector2` which does not react on input `disable`. The code for this detector can be derived from the one of `FireDetector1` by not implementing input disabled and renaming input `smoke1` in `smoke2`. When `ControlUnit` and `FireDetector1` are composed, they synchronize on *fire* and *disable*. In the product, there are no locally incompatible states and the tool deduces that the two components can cooperate correctly in all environments. Note that the action *fire* survives in the composition *both* as an input and as an output, thus allowing `FireDetector1` $\|$ `ControlUnit` to be composed with other fire detectors. The composition of `ControlUnit` and `Faulty_FireDetector1` goes less smoothly. When the composition receives a *disable* action, the control unit shuts down ($s = 3$), while the faulty detector remains in operation. When the faulty detector senses smoke (input *smoke*2), it will emit *fire*: if the control unit has been disabled by the *disable* action, this causes an incompatibility. TICC diagnoses this incompatibility by synthesizing the following input restrictions:

- A restriction preventing the input *disable* if the faulty detector has detected smoke and is about to issue *fire*.
- A restriction preventing the input *smoke2* when `ControlUnit` is at $s = 3$ (disabled).

Since the actions *disable* and *smoke2* should be acceptable at any time, the new input restrictions for these actions are a strong indication that the composition `ControlUnit` $\|$ `Faulty_FireDetector1` does not work properly. However, we conclude that the two components are compatible since they can work together in at least one environment.

The incompatibility in Example 1 is exposed by the following series of OCaml commands:

```
# open Ticc;;
# parse "fire.si";;
# let controlunit = mk_sym "ControlUnit";;
# let fire1 = mk_sym "FireDetector1";;
# let wfire2 = mk_sym "Wrong_FireDetector2";;
# print_input_restriction (compose controlunit wfire2) "disable";;
# print_input_restriction (compose controlunit wfire2) "smoke2";;
```

The function `print_input_restriction` is used to print the restricted version of the input invariant.

Note that the fire detection system illustrates the "many-to-one" communication model of sociable interfaces: several fire detectors can communicate with the control unit using the action `fire`. This is done by allowing fire detectors to receive the action `fire` also as an input. In previous models that have to distinguish between input and output names, this would lead to an incompatibility, and the only solution would be to give an individual name to each fire detector. This has the double disadvantage of bounding the number of detectors that can communicate with the control unit and increasing its number of transitions (and so the internal representation used in the tool).

Due to space limitations, the reader is referred to [11] for many other illustrations of the application and the utility of the composition operation in TICC.

7 More on the Symbolic Representation

As already announced in Definition 1, sets of states can be manipulated through TICC. A set of states can be defined in TICC via a formula specifying constraints on the values of the variables. TICC can parse such a formula, and construct a symbolic representation (an MDD) that enables it to manipulate the set. TICC can combine such sets with the usual Boolean operators through several functions. The tool also contains an implementation of the classical CTL operators [7]. Those operators can be used to verify properties of components via model checking. TICC implements CTL operators with a slightly modified (and more efficient) version of the basic symbolic algorithms given in [7]. Those algorithms are based on the computation of the fixpoint of predecessor operators similar to those defined in Section 5. When model checking CTL, TICC views the module as a closed system whose transitions are the input and output moves of the game it induces.

```
1   open Ticc;;
2   parse "fire.si";;
3
4   let fire1 = mk_sym "FireDetector1";;
5   let controlunit = mk_sym "ControlUnit";;
6   let comp = compose fire1 controlunit;;
7
8   simulate comp "FireDetector1.s = 0 & ControlUnit.s = 0", 5, "r.html";;
9
10  let called_firemen = parse_stateset ("ControlUnit.s = 2");;
11  print_string "Can call the firemen:";;
12  print_stateset (ctl_e_f comp called_firemen);;
13  print_string "Always calls the firemen:";;
14  print_stateset (ctl_a_f comp called_firemen);;
```

Fig. 1. A script file illustrating CTL model-checking in TICC.

We describe now an example that illustrates the use of the symbolic representation and its application to CTL.

Example 2. Consider the fire detection system given in Section 5.1, and the script file in Figure 1. Line 10 builds the symbolic representation of a set ϕ consisting of the states where `ControlUnit.s = 2`. Line 12 prints the set of states that satisfy the CTL formula $\exists\Diamond\phi$, and line 14 prints the set of states that satisfy the CTL formula $\forall\Diamond\phi$. Note that Line 8 of the script illustrates the use of the function `simulate`, which was informally introduced at the end of Section 5.1.

The previous example is very simple and only shows a small fragment of the functionalities provided by the tool. In [11], we give a more complex example that also involves the use of global variables and of a closure function. The closure function (tt close in TICC) allows the user to close a module with respect to the occurrence of input transitions. This can be used to say that the environment is no longer able to provide a certain input.

8 Refinement

The notion of refinement is introduced to capture the relation between an abstract model of a component and a more detailed one, or between a model expressing a specification and a model describing an implementation. In an input-enabled setting, refinement is usually defined as trace containment or a simulation [22]; this ensures that all output behaviors of the implementation are allowed by the specification. Unfortunately, such definition does not stand in an non input-enabled setting, since it does not forbid the implementation to make stronger assumptions on the environment than the specification does. To overcome the problem, de Alfaro et al. suggest a contravariant definition [12] which replaces simulation by *alternating simulation*. More precisely, an interface Q refines an interface P if each input transition of P can be simulated in Q, and each output transition of Q can be simulated in P.

In the rest of this section, we propose a definition of refinement for sociable interfaces. This definition is an extension of the one given in [9]. We consider M_1 and M_2, two well-formed sociable interfaces, where $M_i = (Act_i^G, Act_i^L, \mathscr{D}, V_i^G, V_i^L, V_i^H, W_i, \rho_i^{IL}, \rho_i^{IG}, \rho_i^O, \rho_i^L, \psi_i^I, \psi_i^O, I_i)$, $V_i^{\mathrm{all}} = V_i^G \cup V_i^L$ and $S_i = [\![V_i^{\mathrm{all}}]\!]$. The sets Act_i^G, V_i^G, V_i^H, and W_i jointly define the *signature* of interface M_i.

Definition 7 (Signature). The signature $Sign(M_i)$ of an interface M_i is the tuple $(Act_i^G, V_i^G, V_i^H, W_i)$.

The following result shows that signature equality preserves composability.

Theorem 2. *Let N_1, N_2, and N_3 be three interfaces, such that (1) $Sign(N_1) = Sign(N_2)$, and (2) N_2 and N_3 are composable. For $i \in \{1,2,3\}$, let V_i^L be the set of local variables of N_i. If $V_1^L \cap V_3^L = \emptyset$, then N_1 and N_3 are composable.*

We now define a notion of alternating simulation for sociable interfaces. In addition to what has been said above, the definition must take into account the fact that the environment of an interface cannot see the local transitions. Given a state s of an interface, we define $\varepsilon{-}closure(s)$ to be the set of states that can be reached from s by applying zero or more local transitions.

Definition 8 (Refinement Relation). Assume that $Sign(M_1) = Sign(M_2)$. A relation $\preceq \subseteq S_1 \times S_2$ is a *refinement relation* iff $s \preceq t$ implies:

1. $s[V_1^G] = t[V_1^G]$;
2. for all $a \in Act_2^G$ and for all $t' \in S_2$ such that $\langle t,t' \rangle \models \widehat{\rho}_2^I(a)$ there exists $s' \in S_1$ such that $\langle s,s' \rangle \models \widehat{\rho}_1^I(a)$ and $s' \preceq t'$;
3. for all $a \in Act_1^G$ and for all $s' \in S_1$ such that $\langle s,s' \rangle \models \widehat{\rho}_1^O(a)$ there exists $t',t'' \in S_2$ such that $t' \in \varepsilon{-}closure(t)$ and $\langle t',t'' \rangle \models \widehat{\rho}_2^O(a)$ and $s' \preceq t''$.
4. for all $a \in Act_1^L$ and for all $s' \in S_1$ such that $\langle s,s' \rangle \models \widehat{\rho}_1^L(a)$ there is a state t' in $\varepsilon{-}closure(t)$ such that $s' \preceq t'$.

To gain some insight into Definition 8, consider that there is a refinement relation such that $s \preceq t$ if M_1 in state s can replace M_2 in state t in every context, without the environment noticing any difference. Thus, first of all s and t must agree on the values of the global variables known to M_1 (remember that $V_1^G = V_2^G$). Then, each input that can be accepted by M_2 from t must also be acceptable by M_1 in s. Conversely, each output that can be emitted by M_1 in s must also be emittable by M_2 in t, or in another state t' that is invisibly reachable from t. Finally, all local transitions of M_1 from s must correspond to zero or more local transitions of M_2 from t. We derive from these definitions a concept of refinement for sociable interfaces.

Definition 9 (Refinement). We say that M_1 *refines* M_2 iff *(i)* $Sign(M_1) = Sign(M_2)$, and *(ii)* there is a refinement relation $\preceq$ such that: for all $t \models \psi_2^I \wedge \psi_2^O \wedge I_2$ there is $s \models \psi_1^I \wedge \psi_1^O \wedge I_1$ such that $s \preceq t$.

Theorem 3. *Let N_1, N_2, and N_3 be three modules, such that (1) N_1 refines N_2, and (2) N_2 and N_3 are compatible. For $i \in \{1,2,3\}$, let V_i^L be the set of local variables of N_i. If $V_1^L \cap V_3^L = \emptyset$, then N_1 and N_3 are compatible.*

A preliminary refinement operation has been implemented in the new version of the tool in a function called "`refines`" that takes two symbolic modules as arguments. The result is either `true` if the first module refines the seconde one, and `false` otherwise. The implementation has already been discussed in [9] for the initial model — the extension to the model of this paper follows similar principles.

9 Conclusion and Perspectives

This paper considers sociable interfaces — a concrete game-based model with rich communication primitives to facilitate the modeling of software and distributed systems.

The sociable interfaces model has lead to a new tool named TICC. Several tools implementing game-based models have already been built before TICC. As an example, we mention the tool CHIC that implements the synchronous, variable-based interface theory of [5] which is able to handle pushdown games, a feature that TICC does not have (yet). Another example is the Ptolemy toolset [19] that implements the model of [12].

With respect to these other tools, TICC has the advantage of being able to use rich communication primitives to model components in a very compact and natural way.

The symbolic representation of TICC is also particularly attractive since it makes the tool very efficient and easily extensible. Both the tool and the sociable interfaces model are in constant evolution, and we are considering many improvements as well as several promising research directions.

One of our major concerns is the improvement of the existing functionalities. As an example, we plan to implement a new function that, given a symbolic set of states S and a CTL formula ϕ, will check if all states in S satisfy ϕ and will print a shortest counterexample trace if it is not the case. Such a functionality has been shown to be very useful for the design and the debugging of systems [23]. We are also considering the implementation of several functions that would give more feedback about a partial incompatibility between modules (see Example 1).

A new feature we are currently implementing consists in handling alternating-time temporal logic (ATL), a CTL-like logic designed for open systems, which has been proposed in [3]. The implementation is not straightforward since the algorithms of [3] first need to be framed into our model of game. As it is the case for the CTL logic, we also plan to consider counterexample traces.

Another promising research direction we are now investigating is a real-time extension of the Sociable Interface framework, along the lines of the *Timed Interfaces* of [14]. This is a large and complex endeavor, as the game-theoretic machinery of TICC will have to be replaced with one suited to real-time games [10]. An interesting application for the timed sociable interface model would be the scheduling of timed open systems. A timed version of the tool is desirable since it would constitute a platform for the implementation of various recent results on timed open systems (e.g., [18]).

As it is illustrated in [11], TICC is already able to deal with large systems. However, its efficiency is still limited by the existing algorithms that are used to manipulate the symbolic representation (see Section 5 and Section 7). In a future work, we plan to improve those algorithms by adapting recent works such as [6].

Acknowledgement

We thank James Worrel and Damien Ernst for helpful comments on various drafts of this paper.

References

1. S. Abramsky, D. R. Ghica, A. S. Murawski, and C.-H. Luke Ong. Applying game semantics to compositional software modeling and verification. In *Proc. 10th International Conference on Tools and techniques*, volume 2988 of *Lect. Notes in Comp. Sci.*, pages 421–435, Barcelona, Spain, 2004. Springer-Verlag.
2. B. Adler, L. de Alfaro, L. D. da Silva, M. Faella, A. Legay, V. Raman, and P. Roy. Ticc, a tool for interface compatibility and composition. In *Proceedings 18th International Conference on Computer Aided Verification (CAV)*, volume 4144 of *Lect. Notes in Comp. Sci.*, pages 59–62. Springer, 2006.
3. R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proc. 38th IEEE Symp. Found. of Comp. Sci.*, pages 100–109. IEEE Computer Society Press, 1997.
4. R. Alur, T.A. Henzinger, O. Kupferman, and M.Y. Vardi. Alternating refinement relations. In *CONCUR 98: Concurrency Theory. 9th Int. Conf.*, volume 1466 of *Lect. Notes in Comp. Sci.*, pages 163–178. Springer-Verlag, 1998.

5. A. Chakrabarti, L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. Synchronous and bidirectional component interfaces. In *CAV 02: Proc. of 14th Conf. on Computer Aided Verification*, volume 2404 of *Lect. Notes in Comp. Sci.*, pages 414–427. Springer-Verlag, 2002.
6. G. Ciardo and A. J. Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In *CHARME*, Lect. Notes in Comp. Sci., pages 146–161. Springer-Verlag, 2005.
7. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999.
8. L. de Alfaro. Game models for open systems. In *Proceedings of the International Symposium on Verification*, volume 2772 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2003.
9. L. de Alfaro, L. D. da Silva, M. Faella, A. Legay, P. Roy, and M. Sorea. Sociable interfaces. In *Proceedings of 5th International Workshop on Frontiers of Combining Systems*, volume 3717 of *Lect. Notes in Comp. Sci.*, pages 81–105. Springer, 2005.
10. L. de Alfaro, M. Faella, T.A. Henzinger, R. Majumdar, and M. Stoelinga. The element of surprise in timed games. In *CONCUR 03: Concurrency Theory. 14th Int. Conf.*, volume 2761 of *Lect. Notes in Comp. Sci.*, pages 144–158. Springer-Verlag, 2003.
11. L. de Alfaro, M. Faella, and A. Legay. An introduction to the tool ticc. Technical report, University of California Santa Cruz, 2006. Available at http://luca.soe.ucsc.edu/Publications.
12. L. de Alfaro and T.A. Henzinger. Interface-based design. In *Engineering Theories of Software Intensive Systems, proceedings of the Marktoberdorf Summer School*. Kluwer, 2004.
13. L. de Alfaro, T.A. Henzinger, and O. Kupferman. Concurrent reachability games. Technical Report UCB/ERL M98/33, University of California at Berkeley, 1998.
14. L. de Alfaro, T.A. Henzinger, and M. Stoelinga. Timed interfaces. In *Proceedings of the Second International Workshop on Embedded Software (EMSOFT 2002)*, Lect. Notes in Comp. Sci., pages 108–122. Springer-Verlag, 2002.
15. L. de Alfaro and M. Stoelinga. Interfaces: A game-theoretic framework to reason about open systems. In *FOCLASA 03: Proceedings of the 2nd International Workshop on Foundations of Coordination Languages and Software Architectures*, 2003.
16. D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. MIT Press, 1988.
17. M. Faella and A. Legay. Some models and tools for open systems. Technical report, University of California at Santa Cruz, 2005. Proceedings of FIT05.
18. T. Henzinger and V. Prabhu. Timed alternating-time temporal logic. In *FORMATS06*, Lect. Notes in Comp. Sci., Paris, France, 2006. Springer-Verlag. To appear.
19. E. A. Lee and Y. Xiong. A behavioral type system and its application in Ptolemy II. *Formal Aspect of Computing Journal*, 2003.
20. Xavier Leroy. Objective caml. http://caml.inria.fr/ocaml/index.en.html.
21. A. Lopes and J. Luiz Fiadeiro. Superposition: composition vs refinement of non-deterministic, action-based systems. *Formal Asp. Comput.*, 16(1):5–18, 2004.
22. N.A. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, 1996.
23. K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
24. R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and Their Applications. In *IEEE /ACM International Conference on CAD*, pages 188–191, Santa Clara, California, 1993. IEEE Computer Society Press.
25. S.Abramsky. Algorithmic game semantics and static analysis. In *SAS*, volume 3672, page 1. Lect. Notes in Comp. Sci., 2005.
26. A. Srinivasan, T. Kam, S. Malik, and R. Brayton. Algorithms for discrete function manipulation. In *Proceedings International Conference CAD (ICCAD-91)*, 1990.
27. W. Thomas. On the synthesis of strategies in infinite games. In *Proc. of 12th Annual Symp. on Theor. Asp. of Comp. Sci.*, volume 900 of *Lect. Notes in Comp. Sci.*, pages 1–13. Springer-Verlag, 1995.

Schémas de développement d'adaptateurs à l'aide de B

Arnaud Lanoix, Samuel Colin et Jeanine Souquières

LORIA – Nancy Université
Campus Scientifique, BP 239
F-54506 Vandœuvre lès Nancy cedex
{Arnaud.Lanoix,Samuel.Colin,Jeanine.Souquieres}@loria.fr

Résumé Dans une approche composants pour le développement de logiciels, les composants sont considérés comme des boîtes noires qui communiquent via leurs interfaces. L'interface fournie d'un composant peut être connectée à l'interface requise d'un autre composant si l'interface du premier composant implante les fonctionnalités requises par le second composant. Une description formelle de ces interfaces est nécessaire pour s'assurer de leur compatibilité. En général, les interfaces ne sont pas directement compatibles et un adaptateur doit être introduit. Nous proposons des schémas pour développer des adaptateurs et vérifier l'interopérabilité des composants.

Mots-clés : composant, adaptateur, vérification, construction sûre, raffinement

1 Introduction

L'approche conception de systèmes par assemblage de composants est une approche de développement intéressante et de plus en plus adoptée aujourd'hui [1]. Des composants logiciels "boîte noire" développés par ailleurs sont assemblés les uns avec les autres pour produire le système complet. Le processus d'assemblage sous-jacent est similaire aux méthodes de construction et de réutilisation développées dans d'autres disciplines comme le génie mécanique ou le génie électrique.

Les composants sont assemblés via leurs interfaces. Une interface *fournie* par un composant peut être connectée avec une interface *requise* d'un autre composant si la première offre toutes les fonctionnalités permettant d'implanter la seconde : les composants doivent être connectés de manière appropriée. Afin de garantir cette interopérabilité entre composants, nous considérons chaque connexion entre interfaces fournie et requise de l'architecture et montrons que les interfaces sont compatibles. Une description appropriée des interfaces est primordiale si l'on veut vérifier que l'assemblage est correct.

La spécification formelle des interfaces et la preuve de leur interopérabilité en utilisant la méthode formelle B a été étudiée dans [2, 3]. Grâce à B, nous prouvonsque le modèle de l'interface fournie est un *raffinement* correct de l'interface requise ; en d'autres termes, nous prouvons que l'interface fournie correspond à

une implantation correcte de l'interface requise et par conséquent, que les composants peuvent être connectés [2].

Dans la plupart des cas, des adaptateurs (ou médiateurs) entre composants, doivent être définis pour assurer l'interopérabilité. Un adaptateur est un programme qui réalise la correspondance entre une interface requise et une interface fournie, lorsque celles-ci ne sont pas directement compatibles. Une étude générale de la construction des adaptateurs et de leur preuve en termes du raffinement de l'interface requise incluant le modèle B de l'interface fournie est décrite dans [4, 5]. Cette étude a été étendue avec la prise en compte de modèles d'interfaces différents [6] et de propriétés de sécurité [7].

Dans cet article, nous systématisons notre approche et proposons différents schémas pour développer des adaptateurs et vérifier l'interopérabilité des composants ainsi connectés. Les points forts de notre approche sont :

- l'utilisation de notations simples et de haut niveau pour exprimer l'architecture du système et ses interfaces,
- des schémas d'adaptateurs utilisant les mécanismes classiques de composition et de raffinement,
- des guides pour développer incrémentalement ces adaptateurs,
- la preuve de l'interopérabilité des composants.

L'article est structuré de la manière suivante. La section 2 présente l'utilisation de la méthode B dans une approche composant. La section 3 présente la compatibilité directe entre deux interfaces et sa vérification à l'aide de B. La section 4 présente plusieurs cas d'adaptation de deux interfaces, ainsi que les schémas d'adaptateurs permettant d'exprimer et de vérifier l'interopérabilité. La section 5 s'intéresse au cas où le nombre de composants est supérieur à deux. Des travaux connexes sont discutés dans la section 6 et une conclusion avec des perspectives d'évolution termine ce papier. L'exemple utilisé tout au long de ce papier est celui du contrôle d'accès à un ensemble de bâtiments.

2 Description de l'approche

Dans l'approche composants que nous proposons [3], l'architecture du système est modélisée à l'aide de diagrammes UML 2.0 [8] annotés par des modèles B associés aux interfaces des différents composants. Les modèles B sont utilisés pour exprimer une spécification formelle des interfaces et ainsi vérifier systématiquement leur compatibilité.

2.1 La méthode B

La méthode B [9] est une méthode formelle basée sur la théorie des ensembles, permettant un développement incrémental grâce au raffinement. Un développement commence avec la définition d'une spécification abstraite qui est ensuite raffinée pas à pas jusqu'à l'obtention d'une implantation. Cette méthode a été appliquée avec succès dans le développement d'applications réelles complexes, comme le projet METEOR [10] ou le métro val [11]. Elle est supportée

par des outils robustes. Des obligations de preuves pour la consistance des invariants et la préservation du raffinement sont générées automatiquement par les outils [12, 13].

Dans notre approche, nous utilisons deux notions clés de la méthode B [9] :

- le raffinement qui permet un développement incrémental avec préservation de la correction à chaque étape du développement,
- les mécanismes de composition avec les clauses INCLUDES, PROMOTES et EXTENDS.

2.2 Architecture composants

Nous décrivons un système à base de composants à l'aide de plusieurs diagrammes UML :

- les diagrammes de structure composite expriment l'architecture globale du système en termes des composants et des interfaces à connecter ;
- les diagrammes de classes expriment les modèles de données et les signatures des méthodes des interfaces ;
- les PSMs, *Protocol State Machine*, expriment les protocoles d'utilisation pour certaines interfaces. Ces diagrammes ne seront pas utilisés dans ce papier ;
- les diagrammes de séquences permettent d'exprimer certaines interactions possibles entre composants connectés via leurs interfaces.

Les interfaces des composants sont ensuite spécifiées à l'aide de la méthode formelle B, augmentant le degré de confiance dans les systèmes développés : la correction des spécifications ainsi que la correction du processus de raffinement sont vérifiées à l'aide d'outils. Dans un processus de développement intégré, les modèles B peuvent être obtenus en appliquant des règles systématiques de transformation de UML vers B [14, 15].

2.3 Étude de cas : le contrôle d'accès

Nous illustrons notre propos à l'aide de l'étude de cas du contrôle d'accès à un ensemble de bâtiments [16]. L'objectif est de développer un système chargé de contrôler l'accès de certaines personnes aux différents bâtiments d'un lieu de travail. Le contrôle s'effectue sur la base de l'autorisation que chaque personne concernée possède. Cette autorisation doit lui permettre, sous le contrôle du système, d'entrer dans certains bâtiments et pas dans d'autres. Lorsqu'une personne se trouve à l'intérieur d'un bâtiment, sa sortie doit également être contrôlée par le système afin de savoir à tout instant qui se trouve dans un bâtiment donné.

Chaque personne autorisée dispose d'une carte d'accès avec un code. Des lecteurs de cartes sont installés à chaque entrée et sortie de bâtiment. À proximité de chaque lecteur se trouvent deux voyants, un rouge et un vert, chacun d'eux pouvant être allumé ou éteint. À chaque entrée et sortie de bâtiment se trouve un tourniquet normalement bloqué. Lorsqu'un tourniquet est débloqué par le

système, le passage éventuel d'une personne est détecté par un capteur. Chaque tourniquet n'est affecté qu'à une seule tâche, entrer ou sortir.

L'entrée et la sortie obéissent au protocole suivant :

– si la personne est autorisée à entrer dans le bâtiment concerné (elle est toujours autorisée à sortir), le voyant vert s'allume et le tourniquet se débloque. La spécification originelle fait état d'une contrainte de temps sur la durée de déblocage, contrainte que nous avons préféré abstraire en supposant qu'elle était gérée par le tourniquet lui-même. Dès que la personne franchit le tourniquet, le voyant vert s'éteint et le tourniquet se bloque immédiatement. Si la carte n'a pas été reprise au bout d'un certain laps de temps, elle est «avalée» par le lecteur ;
– si la personne n'est pas autorisée à entrer dans le bâtiment, le voyant rouge s'allume et le tourniquet reste bloqué. Ici encore, le retrait de la carte est soumis à une durée limite, au-delà de laquelle la carte est «avalée» par le lecteur.

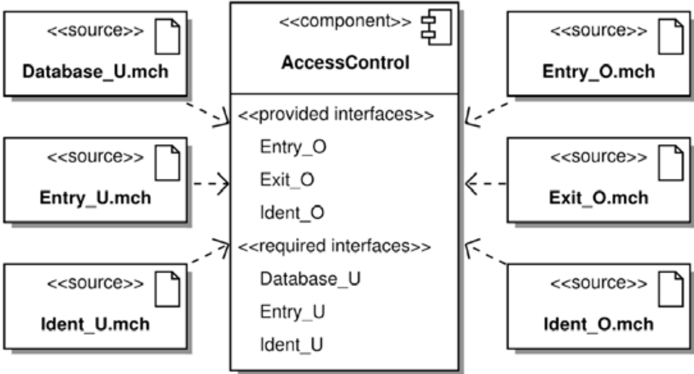


Fig. 1. Composant AccessControl

Dans une vue composant, le système de contôle d'accès peut être représenté par AccessControl, présenté figure 1. Des interfaces requises et fournies sont introduites pour répondre aux différents besoins exprimés dans le cahier des charges de ce système (nous indiquons les noms des méthodes de ces interfaces entre parenthèses) :

– les interfaces Ident_O et Ident_U proposent l'ensemble des fonctionnalités liées à l'identification par le contrôleur d'accès. Celui-ci commande le système d'identification par le biais de l'interface Ident_O (id_inserted, id_read, id_ejected, id_taken, id_retracted) et reçoit des informations en retour via Ident_U (read_id, accept_id, refuse_id) ;

– l'interface Database_U (has_permission) permet au contrôleur d'envoyer des requêtes à une base de données contenant les autorisations des usagers et des informations sur les personnes présentes dans les bâtiments ;
– l'interface Exit_O (has_left) permet d'informer le contrôleur lorsqu'un usager sort du bâtiment ;
– l'interface Entry_U (lock, unlock) permet au contrôleur d'accès de commander le blocage/déblocage de l'entrée ; l'interface Entry_O (entered) informe le contrôleur du passage d'un usager.

Des modèles B sont associés aux différentes interfaces. Ceux de Entry_U et Database_U sont présentés figures 8 et 10.

Pour répondre aux besoins exprimés par le composant AccessControl, nous disposons des composants suivants, présentés Figure 2.

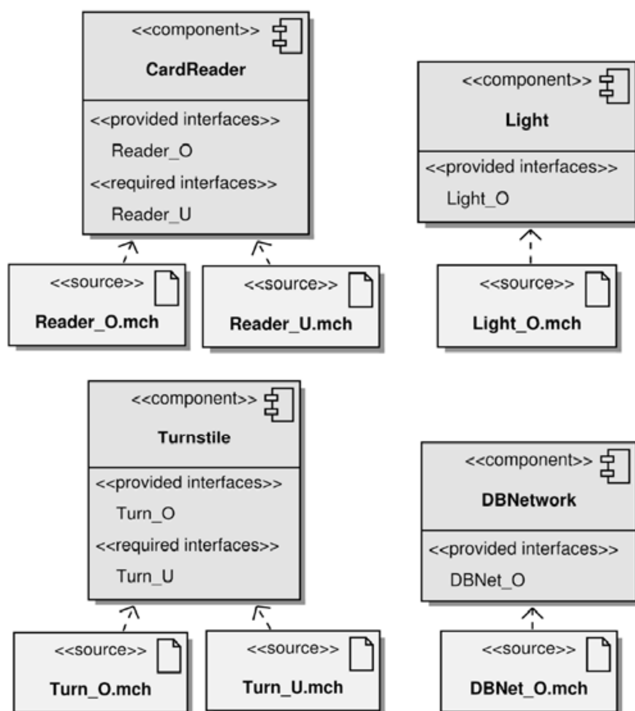


Fig. 2. Les composants1 CardReader, Light, Turnstile et DBNetwork

– Le composant CardReader fournit un pilote de périphérique à un lecteur de cartes. Ses deux interfaces Reader_O (read_card, eject_card, retract_card) et Reader_U (card_inserted, card_read, card_taken, card_retracted)correspondent à l'interfaçage entre le lecteur de cartes et son environnement.

– Le composant Light décrit le pilote de commande d'une lampe. L'interface Light_O (start,stop) permet d'allumer et d'éteindre la lampe.
– Le composant Turnstile fournit un pilote chargé de commander un tourniquet. L'interface Turn_O (block, unblock) fournit des méthodes pour commander l'ouverture et la fermeture du tourniquet (le modèle B associé est donné figure 8) ; Turn_U (pushed) propose une méthode pour informer du passage d'un usager.
– Le composant DBNetwork décrit un pilote permettant de connecter une base de données via l'interface DBNet_O (add_row, remove_row_uid, update_value, select_from_uid). Le modèle B associé est proposé figure 10.

L'architecture complète du système est décrite Figure 3 sous la forme d'un diagramme de structure composite d'UML. Elle utilise les composants précédemment décrits pour répondre aux besoins exprimés par AccessControl. Comme on peut le remarquer sur cette figure, il est nécessaire de développer des adaptateurs pour connecter ces différents composants. L'objet de ce papier est de proposer des schémas pour exprimer et vérifier des adaptateurs à l'aide de B. Entry, Database et Identification seront détaillés dans les sections suivantes.

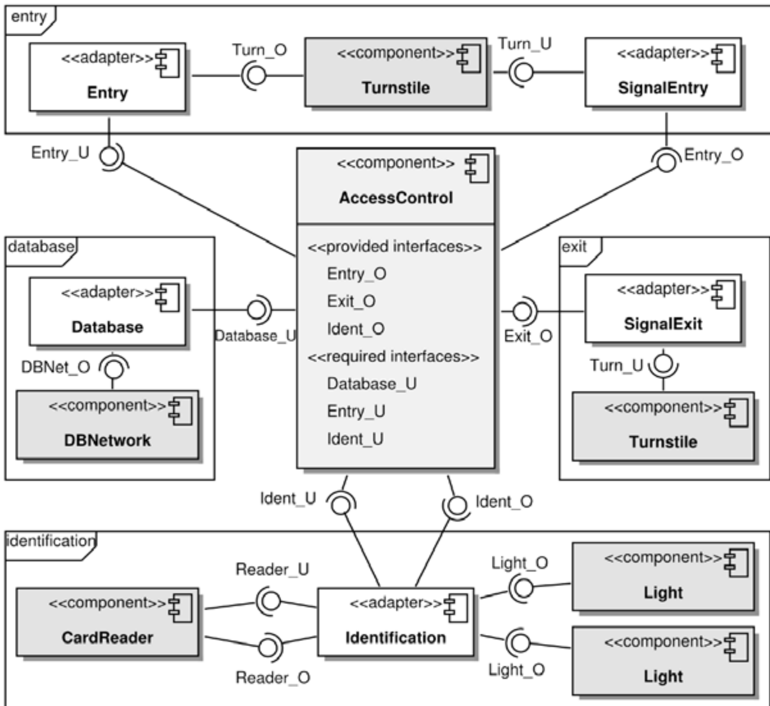


Fig. 3. Architecture globale du système de contrôle d'accès

3 Compatibilité entre deux interfaces

Pour vérifier que deux composants sont *interopérables*, c.à.d. qu'ils peuvent être connectés via leurs interfaces respectives, il faut s'assurer que ces interfaces sont *compatibles*.

Plus précisément, il s'agit de montrer que l'interface fournie implante bien les fonctionnalités nécessaires à l'interface requise [2]. Soient CompoU et CompoO deux composants représentés Figure 4, tels que :

– CompoU nécessite une interface IU, et
– CompoO implante une interface IO.

IO peut fournir plus de fonctionnalités que n'en nécessite IU. A l'aide des mécanismes de composition et de raffinement de B, nous pouvons vérifier la compatibilité entre IU et IO.

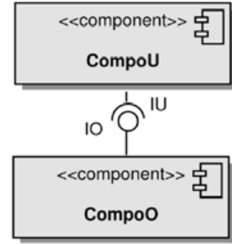


Fig. 4. Compatibilité entre IU et IO (UML)

Nous proposons d'utiliser le schéma de développement donné Figure 5 pour construire automatiquement un modèle B, appelé Connector, permettant de démontrer la compatibilité directe entre IU et IO. On prouve que Connector *raffine* le modèle B associé à IU en *incluant* le modèle B associé à IO (clause INCLUDES) et en *promouvant* les opérations OpeO de IO requises par IU (clause PROMOTES).

Le modèle B Connector, introduit pour vérifier formellement la compatibilité entre IU et IO, correspond à un *adaptateur* simple qui établit la connexion entre les composants CompoU et CompoO.

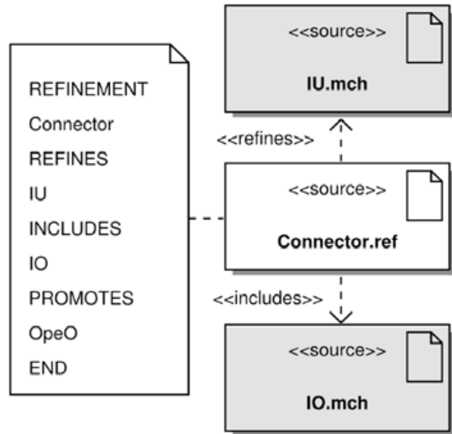


Fig. 5. Compatibilité entre IU et IO (B)

4 Adaptation entre deux interfaces

La plupart du temps, les interfaces entre deux composants ne sont pas *directement* compatibles et il est nécessaire de développer un adaptateur, c.à.d. un programme qui réalise les fonctionnalités nécessaires à l'interface requise en utilisant l'interface fournie [4].

Examinons le cas où les interfaces IU et IO des composants CompoU et CompoO ne sont pas directement compatibles. Développer un adaptateur consiste principalement à exprimer comment les attributs et les méthodes de l'interface requise IU sont implantés grâce à ceux de IO.

Plus précisément,

i) chaque attribut requis par IU doit être exprimé en utilisant les attributs de IO,

ii) chaque méthode nécessaire à IU doit être exprimée par une combinaison d'appels aux méthodes pertinentes de IO et

iii) les protocoles des interfaces IU et IO doivent être compatibles, c.à.d. que les ordres entre les appels de méthodes permis dans IU doivent aussi être permis dans IO.

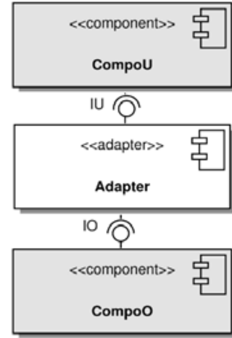


Fig. 6. Adaptateur entre IU et IO (UML)

L'adaptateur fournit l'interface requise IU tout en requérant l'interface fournie IO comme indiqué Figure 6. Tous les adaptateurs suivront ce schéma général, qu'il s'agisse d'appliquer un renommage ou de réaliser des correspondances plus complexes.

Nous proposons d'exprimer, à l'aide d'un raffinement B, l'adaptateur afin de prouver que l'adaptation est correctement exprimée. La figure 7 donne un squelette de l'adaptateur. Il reste bien sûr à compléter les clauses VARIABLES, INVARIANT et OPERATIONS pour respecter les règles *i)*, *ii)* et *iii)* énoncées ci-dessus. La preuve du raffinement assurera que le modèle B de l'adaptateur *raffine* le modèle B associé à IU tout en *incluant* correctement le modèle B associé à IO, c.à.d. que l'adaptation est correctement exprimée.

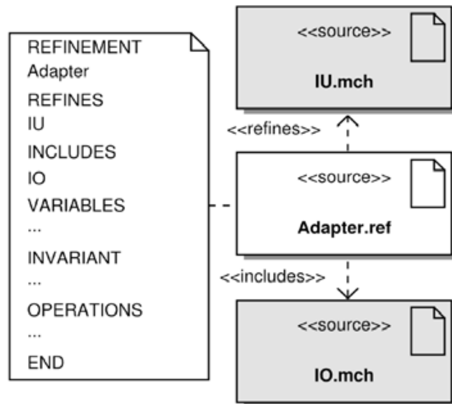
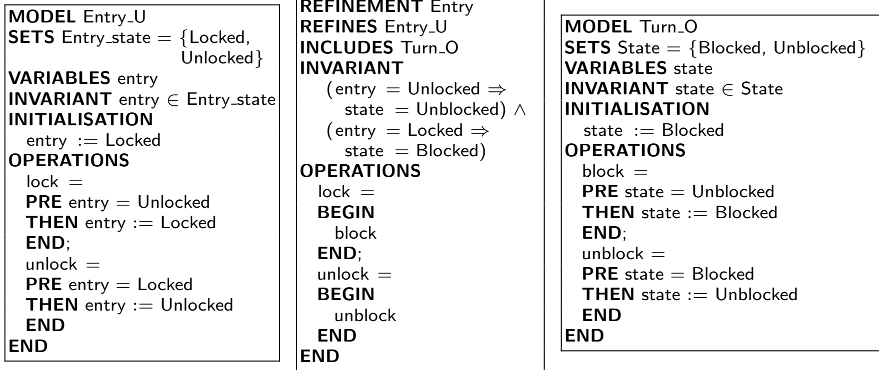


Fig. 7. Adaptateur entre IU et IO (B)

Exemple. Pour connecter AccessControl au composant Turnstile via les interfaces Entry_U et Turn_O, un adaptateur est nécessaire. Le schéma d'adaptation prédédent donne un squelette pour le modèle B de l'adaptateur Entry, comme indiqué Figure 8. Nous complétons ce modèle pour exprimer l'adaptation des éléments de Entry_U en utilisant ceux de Turn_O : ici, il s'agit d'un renommage.

Remarque. Il est à noter que le composant Turnstile est utilisé deux fois dans l'application, pour l'entrée et pour la sortie d'un bâtiment. D'autres adaptateurs sont donc nécessaires. Les adaptateurs SignalEntry et SignalExit sont similaires à un renommage près. Leur adaptation suit le même processus que celui de l'adaptateur Entry.



```
MODEL Entry_U
SETS Entry_state = {Locked,
                    Unlocked}
VARIABLES entry
INVARIANT entry ∈ Entry_state
INITIALISATION
  entry := Locked
OPERATIONS
  lock =
  PRE entry = Unlocked
  THEN entry := Locked
  END;
  unlock =
  PRE entry = Locked
  THEN entry := Unlocked
  END
END
```

```
REFINEMENT Entry
REFINES Entry_U
INCLUDES Turn_O
INVARIANT
   (entry = Unlocked ⇒
     state = Unblocked) ∧
   (entry = Locked ⇒
     state = Blocked)
OPERATIONS
  lock =
  BEGIN
    block
  END;
  unlock =
  BEGIN
    unblock
  END
END
```

```
MODEL Turn_O
SETS State = {Blocked, Unblocked}
VARIABLES state
INVARIANT state ∈ State
INITIALISATION
  state := Blocked
OPERATIONS
  block =
  PRE state = Unblocked
  THEN state := Blocked
  END;
  unblock =
  PRE state = Blocked
  THEN state := Unblocked
  END
END
```

Fig. 8. Adaptation simple du tourniquet en entrée

4.1 Modèles de données différents

Il n'est pas toujours facile d'exprimer chaque attribut requis en termes des attributs fournis, surtout si les modèles de données des interfaces IU et IO sont différents. Pour exprimer et vérifier cette correspondance, nous procédons étape par étape, en utilisant le mécanisme de raffinement de B. Un adaptateur peut être exprimé par une série de raffinements successifs commençant, au niveau le plus abstrait, par le modèle B de l'interface requise et se terminant avec l'inclusion du modèle B de l'interface fournie. Dans [6], nous proposons un processus d'adaptation des modèles B en trois étapes de raffinement. Le schéma de l'adaptateur correspondant est détaillé Figure 9.

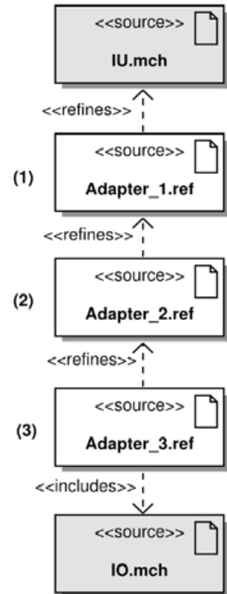


Fig. 9.

(1) Adaptation des variables

Il s'agit de préparer la correspondance entre les attributs de IU et ceux de IO :

- de nouvelles variables, qui ré-expriment les variables de IU sont introduites. Elles sont choisies afin de faciliter la mise en correspondance avec celles de IO ;
- le corps de chaque opération de IU est transformé pour prendre en compte ces nouvelles variables.

(2) Adaptation des types de données

Cette étape correspond au transtypage des données :

- les variables introduites à l'étape précédente sont toujours exprimées en termes des types de données de IU. Des fonctions de transtypage sont introduites afin de convertir les types de données de IU vers ceux de IO,

et réciproquement. De nouvelles variables sont également introduites par l'application des fonctions de transtypage sur les variables introduites à l'étape précédente ;
– le corps de chaque opération de IU est transformé pour tenir compte des modifications introduites sur les variables.

(3) Inclusion de l'interface fournie

Les deux étapes précédentes ont servi à préparer cette dernière étape qui consiste à inclure le modèle B de l'interface fournie :
– puisque les variables de IU ont été ré-exprimées et que les types de données ont été transformés, il est maintenant facile de mettre en correspondance les variables (modifiées) de IU avec celles de IO ;
– chaque opération de IU est exprimée en termes d'appels aux opérations de IO.

Le processus de développement précédent aide à construire l'adaptateur, mais aussi à réaliser la preuve de l'adaptation. La preuve complète est facilitée par la décomposition en plusieurs étapes. Il est plus facile de démontrer successivement chacune des étapes de l'adaptation plutôt que de démontrer l'ensemble des preuves en une seule étape. Il faut également souligner que les étapes **(1)**, **(2)** et **(3)** ne sont pas toujours toutes nécessaires et qu'il est quelquefois plus facile de subdiviser l'une des étapes en plusieurs raffinements, toujours pour faciliter la preuve.

Exemple. Afin de connecter les interfaces Database_U et DBNet_O, un adaptateur est nécessaire. Ces deux interfaces présentent des modèles de données différents. L'interface Database_U permet d'obtenir les portes (bâtiments) autorisées pour un utilisateur donné. L'interface DBNet_O permet de mémoriser dans une base de données des couples (Uid, Value) d'entiers naturels. Plusieurs étapes de raffinement, voir figure 10, sont nécessaires pour réaliser l'adaptation.

(1) Il n'y a pas d'étape d'adaptation des variables, puisque celles-ci peuvent être facilement mises en correspondance : les utilisateurs sont mis en correspondance avec le champ Uid de la base de données, et les portes avec le champ Value.

(2) L'adaptation des types de données est décomposée en deux étapes afin de faciliter la preuve :
– Dans Database_21, une fonction de transtypage user_cast est introduite afin de transformer le domaine de la relation permissions en le domaine les entiers naturels ; une nouvelle variable n_permissions est également introduite.
– Il s'agit maintenant de transformer le codomaine de n_permissions en le domaine des entiers naturels. Une fonction de transtypage, door_cast, ainsi qu'une nouvelle variable, nn_permissions, sont introduites dans Database_22.

(3) La dernière étape consiste à associer les champs Uid et Value de DBNet_O à nn_permissions, c'est-à-dire indiquer quelles sont les relations entre les

```
MODEL Database_U
SEES Types
VARIABLES permissions
INVARIANT
  permissions ∈ Users ↔ Doors
INITIALISATION
  permissions :∈ Users ↔ Doors
OPERATIONS
  result ⟵ has_permission(user, door) =
  PRE
    user ∈ Users ∧
    door ∈ Doors
  THEN
    result := bool(user↦door ∈ permissions)
  END
END
```

```
REFINEMENT Database_21
REFINES Database_U
SEES Types
CONSTANTS user_cast
PROPERTIES user_cast ∈ Users ⤖ ℕ1
VARIABLES n_permissions
INVARIANT
  n_permissions ∈ ℕ1 ↔ Doors ∧
  n_permissions = (user_cast⁻¹; permissions)
INITIALISATION
  n_permissions :∈ ℕ1 ↔ Doors
OPERATIONS
  result ⟵ has_permission(user, door) =
  BEGIN
    result := bool(
        user_cast(user)↦door ∈ n_permissions )
  END
END
```

```
REFINEMENT Database_22
REFINES Database_21
SEES Types
CONSTANTS door_cast
PROPERTIES door_cast ∈ Doors ⤖ ℕ1
VARIABLES nn_permissions
INVARIANT
  nn_permissions ∈ ℕ1 ↔ ℕ1 ∧
  nn_permissions = (n_permissions; door_cast)
INITIALISATION
  nn_permissions :∈ ℕ1 ↔ ℕ1
OPERATIONS
  result ⟵ has_permission(user, door) =
  BEGIN
    result := bool(
      user_cast(user)↦door_cast(door) ∈
      nn_permissions )
  END
END
```

```
REFINEMENT Database_3
REFINES Database_22
SEES Types
INCLUDES DBNet_O
VARIABLES latest_query
INVARIANT
  latest_query ⊆ ℕ ∧
  nn_permissions = (table(Uid)⁻¹; table(Value))
INITIALISATION latest_query := ∅
OPERATIONS
  result ⟵ has_permission(user, door) =
  BEGIN
    latest_query ⟵
        select_from_uid ( (user_cast(user)) );
    result :=bool(door_cast(door) ∈ latest_query)
  END
END
```

```
MODEL DBNet_O
SETS
  Indices = {Uid, Value}
VARIABLES
  table
INVARIANT
  table ∈ Indices → (ℕ1 ⇸ ℕ)
∧ dom(table(Uid)) = dom(table(Value))
INITIALISATION
  table :|( table ∈ Indices → (ℕ1 ⇸ ℕ1) ∧
    dom(table(Uid)) = dom(table(Value)) )
OPERATIONS
  values ⟵ select_from_uid(uid) =
  PRE uid ∈ ℕ
  THEN
    IF uid ∈ ran(table(Uid))
    THEN
      values := table(Value)[(table(Uid))⁻¹[{uid}]]
    ELSE
      values := ∅
    END
  END
END
```

Fig. 10. Adaptation des modèles de données

structures de données. C'est également lors de cette étape que le corps des méthodes se réduit à l'appel des méthodes correspondantes du composant fourni (ici, has_permission appelle select_from_uid).

4.2 Protocoles d'appel complexes

Une autre difficulté pour le développement d'un adaptateur correct consiste à établir un protocole d'appel aux méthodes de l'interface fournie pour réaliser les méthodes de l'interface requise. Ce protocole peut être exprimé à l'aide de diagrammes de séquences UML 2.0. Un (ou plusieurs) diagramme de séquences sert à exprimer les appels aux méthodes de IU (fournies par l'adaptateur), puis les appels résultants de l'adaptateur vers les méthodes de IO. La preuve de raffinement du modèle B de l'adaptateur permet de s'assurer que les appels aux méthodes de IO sont valides (preuves d'inclusion) et le raffinement en lui-même assure que l'adaptation est correcte.

5 Généralisation de l'adaptation

La démarche proposée dans la section 4 est généralisée à la connection simultanée de plus de deux interfaces. L'adaptateur réalise les interfaces requises des différents composants à connecter tout en utilisant les interfaces fournies des composants [5].

L'adaptateur raffine les modèles B des différentes interfaces requises. Ceci s'exprime en B en introduisant un modèle abstrait intermédiaire qui *étend* les modèles des interfaces requises. Ce modèle est ensuite raffiné par le modèle B de l'adaptateur, Adapter2.ref, comme illustré Figure 11 (dans le cas où deux interfaces sont à réaliser). Pour réaliser les différentes interfaces requises, l'adaptateur inclut les modèles B associés aux interfaces fournies des différents composants.

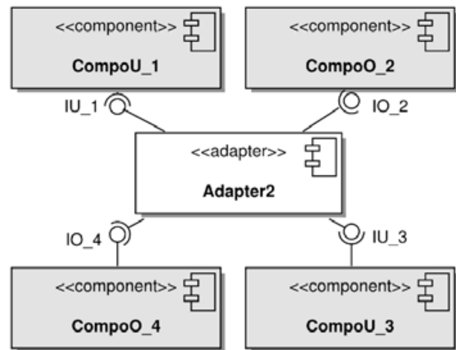


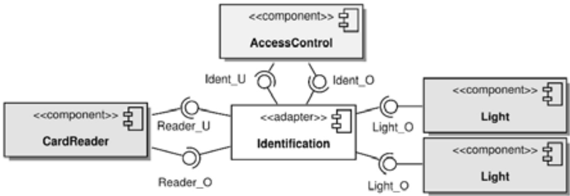
Fig. 11. Adaptateur entre interfaces

B propose un mécanisme de renommage associé à la clause INCLUDES, permettant d'utiliser dans un adaptateur, plusieurs instances d'un même composant via des interfaces fournies identiques.

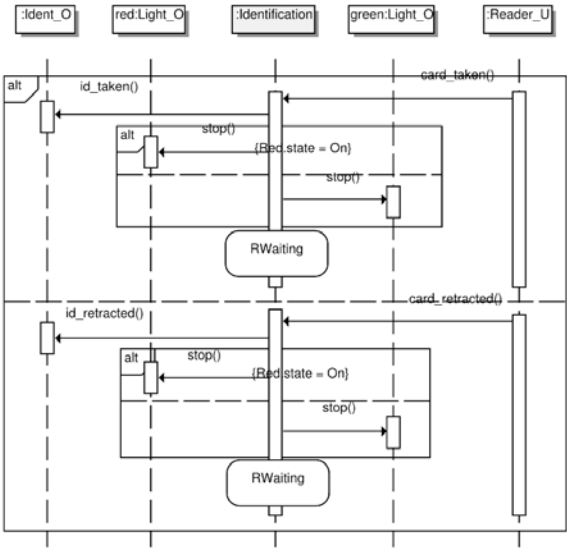
Plusieurs étapes de raffinement peuvent être nécessaires pour faciliter le développement et la preuve de l'adaptateur. Comme plusieurs composants sont en jeu, le protocole d'appels que doit réaliser l'adaptateur peut être complexe

et il est souvent plus profitable de commencer par l'exprimer à l'aide d'un diagramme de séquences, mettant en jeu les différentes interfaces des composants à connecter.

Exemple. Pour répondre aux besoins exprimés par Ident_U et Ident_O à propos de l'identification d'un usager vis-à-vis de AccessControl, nous proposons d'utiliser un composant CardReader via ses interfaces Reader_U et Reader_O et deux instances du composant Light via son interface Light_O. Un adaptateur Identification est nécessaire pour assembler ces différentes interfaces en jeu, comme indiqué figure 12(a). Celui-ci doit expliciter comment implanter les interfaces requises Ident_U et Reader_U en utilisant les interfaces fournies Ident_O, Reader_O et Light_O.



(a) Composants UML



(b) Diagramme de séquences (extrait)

```
REFINEMENT Identification
REFINES Identification_abs
INCLUDES
  Red.Light_O, Green.Light_O,
  Reader_O,
  Ident_O
VARIABLES reader_state
INVARIANT
  (Green.state = On ⇒
    Red.state = Off) ∧
  (Red.state = On ⇒
    Green.state = Off) ∧
  ( signal = No_signal ⇒
      (Green.state = Off ∧
       Red.state = Off)
  )
INITIALISATION
  reader_state := RWaiting
OPERATIONS
  card_taken =
  BEGIN
    id_taken ;
    SELECT Red.state = On
      THEN Red.stop
    WHEN Green.state = On
      THEN Green.stop
    END;
    reader_state := RWaiting
  END;
  card_retracted =
  BEGIN
    id_retracted ;
    SELECT Red.state = On
      THEN Red.stop
    WHEN Green.state = On
      THEN Green.stop
    END;
    reader_state := RWaiting
  END
END
```

(c) Identification

Fig. 12. Adaptateur d'identification

Le diagramme de séquences proposé figure 12(b) permet dans un premier temps d'expliciter le protocole d'appel de l'adaptateur. A chaque méthode des interfaces requises Ident_U et Reader_U, on associe la réaction de l'adaptateur, c.à.d. les appels aux méthodes nécessaires des interfaces fournies. Par exemple, la méthode accept_id() de Ident_U correspond à une notification d'autorisation d'accès d'un usager par le système de contrôle d'accès. L'adaptateur doit réagir en termes des interfaces fournies, en demandant l'allumage d'une lampe verte (Green.start()) afin de confirmer visuellement à l'usager son autorisation d'accès, en éjectant la carte (eject_card()) et en notifiant au contrôleur d'accès l'éjection de la carte (id_ejected()).

Un premier modèle B abstrait nécessaire, Identification_abs, étend les interfaces Reader_U et Ident_U. Le modèle B de l'adaptateur proprement dit est donné figure 12(c). Celui-ci doit raffiner le modèle Identification_abs afin de vérifier que l'adaptateur réalise les différentes interfaces requises, tout en incluant les modèles B des différentes interfaces fournies :

- L'invariant de Identification établit un lien entre les variables à fournir reader_state et signal et les variables fournies par les interfaces Ident_O, Reader_O et Light_O.
- Chacune des méthodes de Reader_U et de Ident_U est reformulée en termes d'appels aux méthodes correspondantes des modèles inclus. La méthode accept_id, par exemple, est réexprimée par la séquence d'appels Green :start ; eject_card ; id_ejected.

Remarque. Les états des interfaces mises en œuvre restent disjoints, c.à.d. qu'il n'est pas possible de lier ces états, que ce soit au niveau des préconditions des méthodes ou des modifications effectuées, en utilisant les nouveaux états introduits dans le raffinement. Ce phénomène est induit par l'utilisation de composants indépendants. L'adaptateur doit créer des liens qui n'existent pas : ceux-ci imposent un style de programmation défensif, traduit par l'utilisation dans notre exemple de gardes (clause SELECT) plutôt que de préconditions (style offensif).

Modèles B\OPs	évidentes	OPs	interactives
Types	1	0	0
Turn_O	5	0	0
Turn_U	3	0	0
Entry_O	3	0	0
Entry_U	5	0	0
Exit_O	3	0	0
Entry	**9**	**2**	**0**
Signal_Entry	**3**	**0**	**0**
Signal_Exit	**3**	**0**	**0**
DBNet_O	12	10	4
Database_U	3	0	0
Database_21	**6**	**2**	**2**
Database_22	**6**	**2**	**2**
Database_3	**5**	**8**	**2**
Light_O	5	0	0
Reader_O	7	0	0
Reader_U	9	0	0
Ident_O	11	0	0
Ident_U	7	0	0
Identification_abs	**9**	**0**	**0**
Identification	**62**	**6**	**2**
TOTAL	**177**	**30**	**12**

Tab. 1. Obligations de preuves

Les différents adaptateurs présentés ainsi que les interfaces nécessaires ont tous été validés avec B4free. Cela nous permet d'assurer que les adaptations sont correctes et que les différents composants mis en jeu dans l'exemple du système de contrôle d'accès pourront interagir correctement. Le détail des obligations de preuves (OPs) est donné dans le tableau 1.

6 Etat de l'art

Les travaux de recherche relatifs à l'adaptation de composants sont nombreux et la nécessité de disposer de mécanismes d'assemblage performants pour les réaliser a été reconnue dès les années 1990 [17, 18, 19, 20].

Une des premières approches concernant la réutilisation de modules avec adaptation de leurs interfaces est celle proposée par Purtilo et Atlee [21] : ils proposent un langage dédié, Nimble, où l'adaptation entre interfaces requises et fournies est effectuée par le développeur. Notre approche est assez voisine avec l'utilisation de UML et B comme langages, reposant sur des standards et des outils de vérification.

Des approches pragmatiques ont porté sur l'analyse des problèmes sous-jacents à l'adaptation de composants existants.Une définition formelle de l'interopérabilité et de l'adaptation de composants a été introduite dans [22]. Dans ce cadre, la spécification du comportement d'un composant est décrite à l'aide de machines à états finis pour lesquelles il existe des techniques et des outils efficaces permettant la vérification de la compatibilité des protocoles.

Zaremski et Wing [23] proposent une approche intéressante pour comparer deux composants logiciels, permettant de décider si un composant peut être remplacé par un autre. Ils utilisent les spécifications algébriques pour modéliser le comportement des composants et le prouveur Larch pour prouver la correspondance entre composants.

Reussner et Schmidt considèrent une certaine classe de problèmes dans le contexte des systèmes concurrents [24, 25]. L'incompatibilité des protocoles est résolue par le génération d'adaptateurs en utilisant les interfaces décrites en termes de machines à états finis.

Les travaux présentés dans [26] proposent un processus de génération d'adaptateurs. De nombreux travaux actuels sont dédiés à l'adaptation dynamique [27], qui va plus loin que notre approche : l'adaptation des composants s'effectue lors de l'exécution en recherchant le composant adapté [28, 29]. Ces méthodes se basent sur l'hypothèse de l'existence de relations d'héritages (avec une possible transitivié) entre une interface fournie et une classe qu'on sait pouvoir utiliser. Elles sont donc fortement basées sur la notion de (sous-)typage dans un contexte de programmation objet, et donc sont moins flexibles en termes d'expressivité que notre approche, bien qu'elles apportent l'adaptation dans un contexte dynamique.

Le papier [30] présente un cadre pour modéliser des architectures composants en utilisant des techniques formelles (réseaux de Petri et CSP) : les connexions entre interfaces requises et fournies sont représentées par des transformations de graphes utilisant des notions de composition, d'extension et de raffinement. Notre approche est similaire avec l'utilisation de B pour exprimer les transformations comme des raffinements entre interfaces requises et fournies.

Braccalia & al [31] spécifient un adaptateur comme un ensemble de correspondances entre les méthodes et les paramètres des composants requis et fournis. Un adaptateur est formalisé par un ensemble de propriétés exprimées à l'aide du π-calcul.

La génération automatique d'adaptateurs est limitée à une certaine classe de problèmes car la vérification de l'interopérabilité repose sur la décidabilité de l'inclusion des composants. Dans notre approche, nous proposons des schémas pour construire et vérifier les adaptateurs, en fonction de différents cas de figures de l'architecture, sans aller jusqu'à leur génération automatique.

7 Conclusion

L'approche composants est un paradigme bien connu et utilisé dans le développement de logiciels, aussi bien dans le milieu académique que dans le milieu industriel. Dans cette approche, les composants sont considérés comme des boîtes noires décrites en termes de leur comportement visible et de leurs interfaces, qu'elles soient requises ou fournies.

Des adaptateurs doivent être définis pour construire un système à l'aide de composants. Un adaptateur est un programme qui définit comment les interfaces requises sont réalisées en termes des interfaces fournies : il exprime la correspondance entre variables, types et opérations. Nous proposons une approche formelle pour développer ces adaptateurs avec des schémas pour les construire et les vérifier, en fonction des différents cas de figures de l'architecture. Nous ne proposons pas de les générer automatiquement.

Grâce à l'utilisation de la méthode B, de ses mécanismes d'assemblage et de raffinement pour modéliser les interfaces et les adaptateurs, nous obtenons la preuve de l'interopérabilité entre les différents composants. Le prouveur B garantit que l'adaptateur est une implantation correcte des fonctionnalités attendues en termes des composants existants. La vérification de l'interopérabilité entre les composants connectés est effectuée aux niveaux signature, sémantique et protocole.

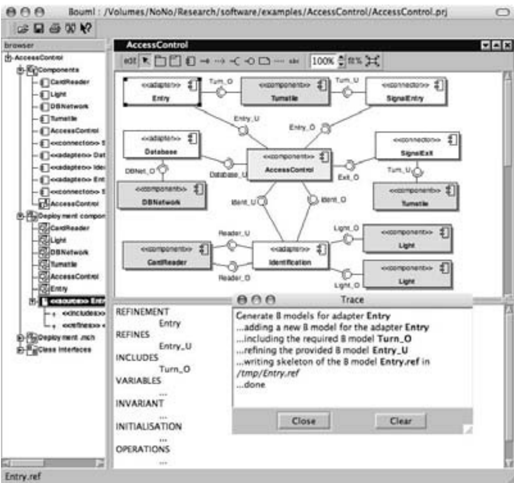


Fig. 13. BOUML pour générer un modèle B

L'implantation d'un plugin pour BOUML[1] fondé sur les schémas de développement présentés dans cet article est en cours : la figure 13 montre la génération du squelette du modèle B correspondant à l'adaptateur Entry.

L'extension de l'approche avec la prise en compte de propriétés de sécurité dans une architecture composants existante, sans modification de ses fonction-

[1] http ://bouml.free.fr

nalités de base [7] est en cours d'étude. Ce travail doit également être complété par un outil d'aide à la détection des incompatibilités.

Références

[1] Szyperski, C. : Component Software. ACM Press, Addison-Wesley (1999)

[2] Chouali, S., Heisel, M., Souquières, J. : Proving Component Interoperability with B Refinement. Electronic Notes in Theoretical Computer Science (ENTCS) **160** (2006) 157–172

[3] Hatebur, D., Heisel, M., Souquières, J. : A Method for Component-Based Software and System Development. In : Proceedings of the 32nd Euromicro Conference on Software Engineering And Advanced Applications, IEEE Computer Society (2006) 72–80

[4] Mouakher, I., Lanoix, A., Souquières, J. : Component Adaptation : Specification and Verification. In : Proc. of the 11th Int. Workshop on Component Oriented Programming (WCOP'06), satellite workshop of ECOOP 2006. (2006) 23–30

[5] Lanoix, A., Souquières, J. : A Trustworthy Assembly of Components using the B Refinement. e-Informatica Software Engineering Journal (ISEJ) **2**(1) (2008) 19 pages, published in advances of print.

[6] Colin, S., Lanoix, A., Souquières, J. : trustworthy interface compliancy : data model adaptation. In : Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA), Satellite workshop of ETAPS, Electronic Notes in Theoretical Computer Science (ENTCS) (March 2007) 13 pages. to be published.

[7] Lanoix, A., Hatebur, D., Heisel, M., Souquières, J. : Enhancing Dependability of Component-based Systems. In : Reliable Software Technologies Ada-Europe 2007. LNCS, Springer Verlag (June 2007) 14 pages. to be published.

[8] Object Management Group (OMG) : UML Superstructure Specification. (2005) version 2.0.

[9] Abrial, J.R. : The B Book. Cambridge University Press (1996)

[10] Behm, P., Benoit, P., Meynadier, J. : METEOR : A Successful Application of B in a Large Project. In : Integrated Formal Methods, IFM99. Volume 1708 of LNCS., Springer Verlag (1999) 369–387

[11] Badeau, F., Amelot, A. : Using b as a high level programming language in an industrial project : Roissy val. In : ZB 2005 : Formal Specification and Development in Z and B, 4th International Conference of B and Z Users. Volume 3455 of LNCS., Springer-Verlag (2005) 334–354

[12] Steria – Technologies de l'information : Obligations de preuve : Manuel de référence, version 3.0. (1998)

[13] Clearsy : B4free (2004) `http://www.b4free.com`.

[14] Meyer, E., Souquières, J. : A systematic approach to transform OMT diagrams to a B specification. In : Proceedings of the Formal Method Conference. Number 1708 in LNCS, Springer-Verlag (1999) 875–895

[15] Ledang, H., Souquières, J. : Modeling class operations in B : application to UML behavioral diagrams. In : ASE'2001 : 16th IEEE International Conference on Automated Software Engineering, IEEE Computer Society (2001) 289–296

[16] AFADL'2000 : Etude de cas : système de contrôle d'accès. In : Journées AFADL, Approches formelles dans l'assistance au développement de logiciels. (2000) actes LSR/IMAG.

[17] Brown, A.W., Wallnan, K.C. : Engineering of component-based systems. In : Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '96), IEEE Computer Society (1996) 414

[18] Heineman, G., Ohlenbusch, H. : An evaluation of component adaptation techniques. Technical Report WPI-CS-TR-98-20, Department of Computer Science, Worcester Polytechnic Institute (February 1999)

[19] Heisel, M., Santen, T., Souquières, J. : Toward a formal model of software components. In : Proc. 4th International Conference on Formal Engineering Methods - ICFEM'02. Number 2495 in LNCS, Springer-Verlag (2002) 57–68

[20] Canal, C., Murillo, J.M., Poizat, P. : Software adaptation. L'Objet **12**(1) (2006) 9–31

[21] Purtilo, J., Atlee, J. : Module reuse by interface adaptation. Software - Practice and Experience **21**(6) (1991) 539–556

[22] Yellin, D.D.M., Strom, R.E. : Protocol specifications and component adaptors. ACM Transactions on Programming Languages and Systems **19**(2) (1997) 292–333

[23] Zaremski, A.M., Wing, J.M. : Specification matching of software components. ACM Transaction on Software Engeniering Methodolology **6**(4) (1997) 333–369

[24] Schmidt, H.W., Reussner, R.H. : Generating adapters fo concurrent component protocol synchronisation. In Crnkovic, I., Larsson, S., Stafford, J., eds. : Proceeding of the 5th IFIP International conference on Formal Methods for Open Object-based Distributed Systems. (2002) 213–229

[25] Reussner, R.H., Schmidt, H.W., Poernomo, I.H. : Reasoning on software architectures with contractually specified components. In Cechich, A., Piattini, M., Vallecillo, A., eds. : Component-Based Software Quality : Methods and Techniques. (2003) 287–325

[26] Poizat, P., Salaün, G., Tivoli, M. : An Adaptation-based Approach to Incrementally Build Component Systems. In : FACS'06, Electronic Notes in Theoretical Computer Science (2006) to appear.

[27] WCAT2006 : Coordination and Adaptation Techniques : Bridging the Gap Between Design and Implementation. In Becker, S., Canal, C., Diakov, N., Murillo, J.M., Poizat, P., Tivoli, M., eds. : Proceedings of the Third International Workshop on Coordination and Adaptation Techniques for Software Entities. (2006)

[28] Mätzel, K.U., Schnorf, P. : Dynamic component adaptation. Technical report, Ubilab laboratory, Union Bank of Switzerland, Zürich, Switzerland (June 1997)

[29] Kniesel, G. : Type-safe delegation for run-time component adaptation. Lecture Notes in Computer Science **1628** (1999) 351–366

[30] Ehrig, H., Padberg, J., Braatz, B., Klein, M., Orejas, F., Perez, S., Pino, E. : A generic framework for connector architectures based on components and transformation. In : FESCA'04, satellite of ETAPS'04. Volume 108 of ENTCS. (2004) 53–67

[31] Bracciali, A., Brogi, A., Canal, C. : A formal approach to component adaptation. In : Journal of Systems and Software. Volume 74., Elsevier Science Inc. (2005) 45–54

Intégration de Propriétés Temporelles dans des Applications à Base de Composants

Sébastien Saudrais[†], Olivier Barais[†], Laurence Duchien[††] et Noël Plouzeau[†]

[†]IRISA France, Projet Triskell [⋆] – [††]INRIA France, Projet ADAM
{ssaudrai, barais, plouzeau}@irisa.fr, duchien@lifl.fr

Résumé Prendre en compte le temps dans le développement d'applications à base de composants reste une tâche difficile pour le concepteur car le temps n'est pas quantifiable comme les autres ressources et ne peut donc être structuré de manière isolée. Face à ce défi, cet article présente une technique pour intégrer des propriétés temporelles dans des composants logiciels traditionnels. L'objectif de nos travaux est donc d'aider l'architecte à introduire le temps dans les composants sans interférer avec les fonctionnalités déjà définies. Pour cela, nous définissons des motifs correspondant à des propriétés de temps, comme par exemple le temps de réponse ou la période. Nous intégrons ensuite ces motifs aux composants afin d'obtenir des composants avec des informations de temps. Nous appliquons le principe de la séparation des préoccupations pour permettre la spécification du temps de manière indépendante de la spécification fonctionnelle traditionnelle. Ceci facilite l'intégration des outils de vérification temporelle dans le développement de logiciels en permettant l'emploi des méthodes formelles pour le temps lors de l'assemblage de composants, ceci sans modifier le processus de conception existant.

1 Introduction

Beaucoup d'applications modernes incluent le temps comme une caractéristique importante de leur comportement, comme par exemple les applications de communication (*chat* vidéo, édition de texte concurrente, etc). Fondées généralement sur des approches à base de composants, construites comme un assemblage de modules logiciels réutilisables, ces approches prennent difficilement en compte les propriétés liées au temps lors de la phase de construction de l'application. Or, les composants logiciels impliquent des opérations et des coordinations complexes, et sont écrits principalement dans des langages tels que Java, C++ ou C#. Dès lors, travailler avec des propriétés de temps n'est pas aisé pour le développeur de composants. Des langages de modélisation tel que UML incluent des notions de temps informelles [1] et de fait rarement utilisées lors de l'assemblage des composants. De plus, même si certains langages de programmation et leur cadre logiciel associé incluent aussi des caractéristiques temporelles [10], tous ces modèles de temps ne sont pas suffisamment formels pour les utiliser lors de la conception des composants ou lors de la phase d'assemblage. Par conséquent, bien

[⋆] Ce travail est financé par ARTIST2, le réseau d'excellence de la conception de systèmes embarqués.

[1] par exemple en utilisant des profils [26]

que le temps joue un rôle important dans ces applications, il est souvent délaissé au niveau spécification du processus de conception et peu connecté au reste du processus de développement de l'application.

D'un point de vue plus théorique, de nombreux formalismes existent pour décrire les comportements temporels et effectuer des vérifications sur ceux-ci : on compte parmi ces formalismes les réseaux de Petri temporels [23,27] et les automates temporisés[4]. Le formalisme des automates temporisés peut être utilisé pour décrire des applications [5] et peut être vérifié par rapport à des propriétés temporelles. Le formalisme des réseaux de Petri temporels peut servir à modéliser des sytèmes avec des informations temporelles [9]. Dans ce article, nous voulons réconcilier les mécanismes formels de temps avec les architectures construites à base de composants. Pour aboutir à cela, nous fusionnons un méta-modèle classique de composants représentant la définition des concepts manipulés par un architecte avec des extensions pour le comportement et les spécifications formelles de temps. Ces extensions ont été choisies pour promouvoir une bonne séparation des préoccupations selon deux axes : spécification/comportement et synchronisation/temps. Les activités de spécification se fondent sur une logique temporelle avec des extensions qualitatives de temps tandis que le comportement sera représenté par des automates temporisés. Afin d'introduire du temps dans les composants classiques, au niveau spécification et comportemental, nous avons défini un procédé pour aider l'architecte à prendre en compte le temps lors de la conception de l'application.

La suite de cet article est organisée comme suit : la section 2 présente une vue du méta-modèle choisi pour décrire nos composants avec un intérêt spécifique pour les parties sur la spécification et le comportement. La section 3 détaille les formalismes de temps utilisés et montre comment les ajouter dans les composants. Enfin, la section 4 présente les travaux connexes et la section 5 conclut et présente les travaux futurs.

2 Analyse et conception

Notre méta-modèle à composants est fondé sur un sous-ensemble du modèle à composants UML 2.0. Ce sous-ensemble comprend les concepts suivants : composants, ports, interfaces et connecteurs. La notation résultante, définie à l'aide d'un méta-modèle, ressemble à celle utilisée par d'autres approches comme par exemple celles du document [22].

Le modèle de conception est organisé en deux parties principales : la partie structurelle décrivant l'architecture de l'application et la partie comportementale décrivant les interactions du composant avec l'environnement.

2.1 Eléments structurels du meta-modèle à composants

Notre méta-modèle à composants est dérivé des concepts du diagramme d'architecture d'UML 2.0 pour la partie structurelle. Cependant, pour limiter la complexité du langage manipulé par l'architecte, nous avons supprimé certains concepts et enlevé tous les points de variations sémantiques d'UML 2.0.

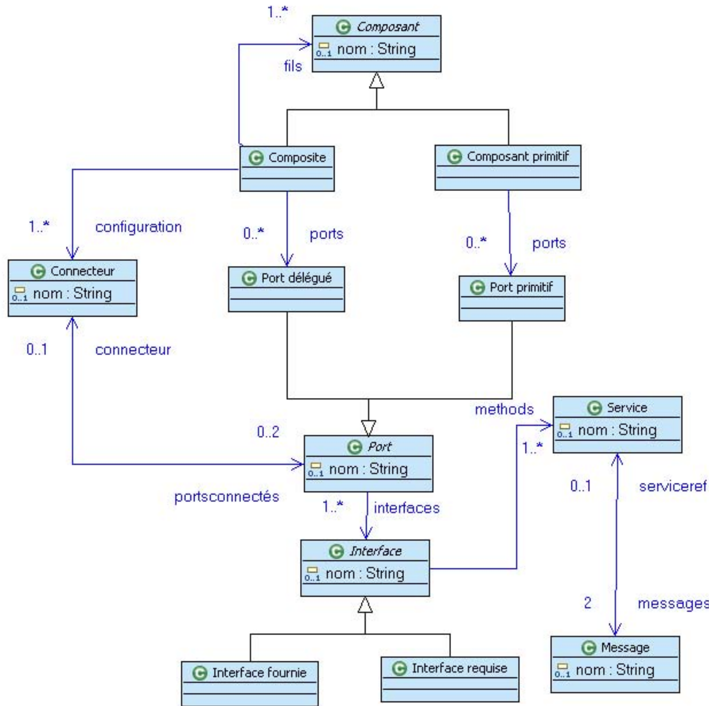


FIG. 1. Partie structurelle du méta-modèle de composant

Dans notre modèle à composants, un composant fournit des *services* et peut requérir des services d'autres composants. Les services sont accessibles par des ports uniquement. Un *port* est un point de connexion sur un composant définissant deux ensembles d'*interfaces* : *fournies* et *requises*.

Notre modèle de composant distingue deux types de composants : *primitif* et *composite*. Les composants *primitifs* contiennent du code exécutable et sont les briques de base de construction dans les assemblages de composants logiciels. Les services fournis et requis par un composant primitif sont accessibles grâce à des *ports primitifs* qui sont les seuls points d'entrée d'un composant primitif.

Les *composites* sont utilisés comme mécanisme pour gérer un groupe de composants comme un tout, en cachant certaines fonctionnalités des composants de ce groupe. Notre modèle de composant n'impose pas de limite dans le nombre de niveaux de composition. Il existe deux façons de définir l'architecture d'une application : utiliser les *connecteurs* entre les ports de composants ou utiliser un composite pour encapsuler un groupe de composants, appelées aussi respectivement composition horizontale ou composition verticale. Un connecteur associe le port d'un composant avec un port situé sur un autre composant. Deux ports peuvent être liés ensemble seulement si les interfaces

requises par l'un sont fournies par l'autre et vice-versa. Les services fournis et requis par un composant fils d'un composant composite sont accessibles grâce à des *ports délégués* qui sont aussi les seuls points d'entrée d'un composant composite. Un port délégué d'un composite est connecté à un et un seul port de composant fils. La partie structurelle est présentée sur la figure 1.

Contrat de composant En UML 2.0, il n'existe pas de sémantique claire pour définir les conditions de compatibilité de deux ports. Il semble raisonnable de décider que dans une architecture, la liaison entre deux ports est correcte si toutes les services fournis par le premier type de port sont requis par l'autre et inversement. Mais la vérification de type, souvent utilisée dans la programmation à objets, ou la correspondance de signature [32] a montré ses limites [24]. La compatibilité entre deux prototypes de services ne peut pas garantir leur bonne utilisation. C'est pourquoi nous enrichissons les propriétés visibles du composant. L'augmentation des capacités d'expression d'un modèle type boîte noire permet de simplifier l'intégration des composants. L'enrichissement des spécifications est souvent utilisé dans les techniques de conception par contrat de développement logiciel assurant des architectures logicielles de haut niveau. Elles garantissent que tous les composants d'un système respectent leurs attentes. Dans notre approche, comme présenté dans [8], nous identifions quatre niveaux de contrats. Le premier niveau, contrats basiques ou syntaxiques, est simplement requis pour que le système fonctionne. Il contient des informations fournies par les prototypes des opérations. Le second niveau, contrats comportementaux, augmente le niveau de confiance dans un contexte séquentiel. C'est un ensemble de contraintes sur les opérations, appelé contrats d'assertion. Ce niveau ajoute des pré et post conditions sur une opération pour garantir un niveau d'utilisation correct. Le troisième niveau traite des contrats de synchronisation, il augmente la confiance dans un contexte distribué ou concurrent. Il spécifie le comportement externe du composant. Le quatrième niveau concerne les contrats de qualité de service (QdS). Il exprime la qualité du service et est la plupart du temps négociable. L'architecte doit être capable de concevoir chaque niveau indépendamment et doit être capable de garantir durant l'étape de conception que l'architecture respecte les contrats des composants.

2.2 Comportement du composant

Avec les définitions d'interface et de service, le composant déclare des éléments structurels sur les services fournis et requis. La spécification de comportement définit l'interaction du composant avec son environnement. Le comportement est décrit par une algèbre de processus pour laquelle nous utilisons le modèle des automates à entrées/sorties [20] pour vérifier le système.

L'algèbre de processus. Pour spécifier le comportement du composant, nous utilisons une algèbre de processus simple, inspirée de FSP [21]. L'algèbre de processus est fondée sur une expression décrivant un ensemble de traces (séquences d'évènements). Appliqué à un composant, un évènement est une abstraction d'un appel de service ou d'une réponse à un appel. Par exemple, un appel de $m1$ sur l'interface $i1$ du port $p1$ est noté par $p1.i1.m1$ et une réponse à ce service $p1.i1.m1\$$. Chaque évènement est

émis par un composant et reçu par un autre. L'appel de $m1$ par l'interface $i1$ du port $p1$ est vu comme l'émission de $!p1.i1.m1$ par le composant $C1$ (noté par un évènement $!C1.p1.i1.m1$). La réception de $p3.i2.m1$ est vue par $C2$ comme $?p3.i2.m1$ (noté par $?C2.p3.i2.m1$ du point de vue de $C2$).

Les opérateurs utilisés dans les protocoles de comportement sont : $\rightarrow$ pour la séquence, $|$ le choix 'alterné' et $*$ la répétition finie. Cette algèbre sert à représenter le comportement des composants primitifs.

Le modèle d'automate à entrée/sortie. En plus de l'algèbre de processus, nous utilisons le formalisme des automates à entrée/sortie pour effectuer la vérification. Chaque comportement défini avec l'algèbre de processus est transformé en automate à E/S.

Definition 1. *(Automate à entrée/sortie) Un automate à entrée/sortie est un n-uplet* (S, L, T, s_0) *avec :*
- S *est un ensemble fini non vide d'états ;*
- L *est un ensemble fini non vide de labels.* $L = I \bigcup O$ *où* I *est un ensemble d'entrées et* O *les sorties et* $I \bigcap O = \emptyset$ *;*
- $T \subseteq S(L \bigcup \{\tau\})S$ *est un ensemble fini de transition où* τ *est une action interne non observable, ;*
- $s_0 \in S$ *est l'état initial.*

Composition d'automates à entrée/sortie La composition de composants dans notre modèle est fondée sur la synchronisation d'une sortie d'un composant avec l'entrée du composant qui lui est connectée.

La composition des automates à entrée/sortie est associative et commutative. Quand l'architecte compose plusieurs composants, l'ordre de composition n'est donc pas important.

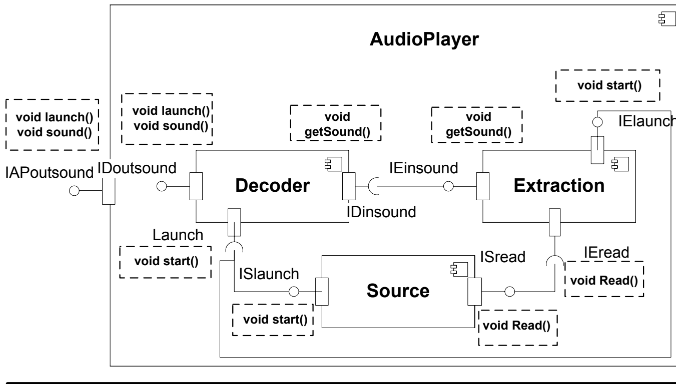
Pour rester cohérent avec le sous-ensemble UML2 sélectionné, cette algèbre de processus peut être vue comme une représentation textuelle d'un sous-ensemble de diagrammes de séquences où les rôles, identifiés dans le diagramme, sont les ports du composant.

2.3 Exemple

La figure 2 illustre notre modèle avec un exemple de composant `AudioPlayer`. Ce composant fournit une interface `IAPoutsound` qui contient deux services `launch` et `sound`. Ce composant est un composite de trois composants `Decoder`, `Extraction` et `Source`. La partie haute de la figure montre la représentation structurelle de ce composant dans notre modèle. La partie basse montre l'automate $A1$ décrivant le comportement du composant `Decoder`.

3 Ajout des propriétés temporelles dans les composants

Quand toutes les propriétés fonctionnelles sont définies pour le composant, l'architecte peut lui ajouter des propriétés extra-fonctionnelles. Ces propriétés sont souvent



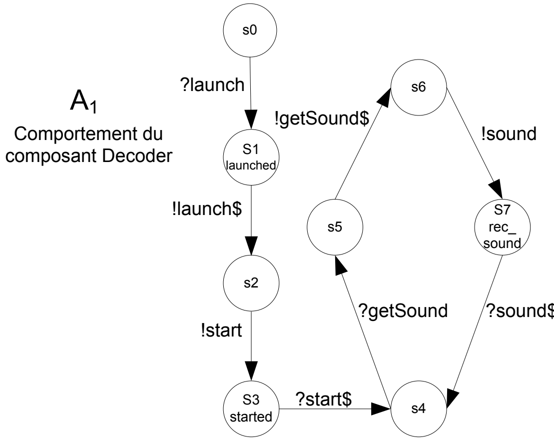


FIG. 2. Exemple : Un composant AudioPlayer

orthogonales aux propriétés fonctionnelles (intersection non vide entre les propriétés) et difficile à ajouter. Les propriétés de temps constituent une catégorie importante des propriétés extra-fonctionnelles. Nous définissons une méthode pour ajouter les propriétés temporelles dans les composants après avoir défini les propriétés fonctionnelles. Nous modifions certaines parties du composant sans pour autant en changer ses fonctionnalités. Les informations temporelles peuvent être ajoutées dans les composants à deux endroits : sur le comportement du composant lui-même et dans les contrats attachés aux interfaces requises. Ces deux emplacements sont les plus adaptés à l'ajout des informations de temps, car ils représentent ce que le composant fournit et requiert et sont utilisés lors de la composition des composants. Pour ajouter du temps dans le comportement du composant, nous utilisons la théorie des automates temporisés [4] et nous définissons un ensemble de motifs pour aider l'architecte à ajouter les informations. Pour les propriétés temporelles dans les contrats, nous utilisons une logique temporelle temporisée (*Timed Computation Tree Logic* [3]). Nous définissons aussi des motifs représentant les structures les plus fréquentes pour aider à l'écriture des contrats temporels.

3.1 Ajout du temps dans le comportement des composants

Pour décrire le comportement temporel, nous avons choisi le formalisme des automates temporisés (AT). Ces AT vont remplacer le comportement originel du composant dans sa description.

Automates temporisés Un automate temporisé est un automate étendu avec des horloges qui sont un ensemble de variables augmentant uniformément avec le temps. Formellement, un automate temporisé est défini comme suit.

Definition 2. *(Automate temporisé) Un automate temporisé est un n-uplet* $A =< S, X, L, T, \iota, P >$ *où :*
- S *est un ensemble fini de localités ;*
- X *est un ensemble fini d'horloges. A chaque horloge, nous assignons une estimation* $v \in V$, $v(x) \in R^+$ *pour tout* $x \in X$ *;*
- L *est un ensemble fini de labels ;*
- T *est un ensemble fini de transitions. Chaque transition* t *est un tuple* $< s, l, \psi, , s' >$ *où* $s, s' \in S$, $l \in L$, $\psi \in \Psi_X$ *est la condition de progression,* Ψ_X *est l'ensemble des prédicats sur* X *définis par* $x \sim c$ *où* $x \in X$ *et* $\sim \in \{<, \leq, =, \geq, >\}$ *et* $c \in N.$ *;*
- ι *est l'invariant de* A. $\iota \in \Phi_X$ *où* Φ_X *est l'ensemble des fonctions* $\phi : S \rightarrow \Psi_X$ *liant chaque localité* s *à un prédicat* ψ *;*
- P *associe un ensemble de propositions atomiques à un état.*

Un état d'un automate temporisé est un couple *localité et estimation d'horloges* satisfaisant l'invariant de la localité. Deux types de transition sont possibles entre les états : discrètes sans incrément de temps et avec incrément de temps.

Motifs temporels Pour faciliter l'ajout de temps dans le comportement, nous définissons un ensemble de motifs temporels basés sur ceux définis partiellement dans [14] : temps de réponse, délai, temps d'exécution, période d'appel de service, durée, etc. Nous expliquons ici deux motifs principaux : temps de réponse et temps d'exécution.

Temps de réponse Le motif *temps de réponse* (RT) décrit comment exprimer un temps de réponse avec un automate temporisé. Le temps de réponse est le temps entre l'appel de service et son acquittement. Par exemple, pour exprimer un temps de réponse sur le service $getSound$, une horloge doit être initialisée lors de l'appel à $getSound$ et doit être comparée à une valeur lors de la réception de son acquittement. Ce motif requiert trois paramètres : l'appel de service $service$, l'opérateur de comparaison $\sim \in \{<, \leq, =, \geq, >\}$ et la valeur c. L'automate RT de la Figure 3 représente le motif générique du temps de réponse. Le motif est composé de trois localités, deux transitions et une horloge. Cette dernière est initialisée sur la première transition avec l'appel de service et vérifiée sur la seconde avec l'acquittement de l'appel. La deuxième transition a la propriété $call_service_begin$ et la troisième $call_service_end$. Ces propriétés seront utilisées lors la vérification des contrats. Quand le motif est ajouté au comportement du composant, les deux transitions ne sont pas obligatoirement consécutives, d'autres

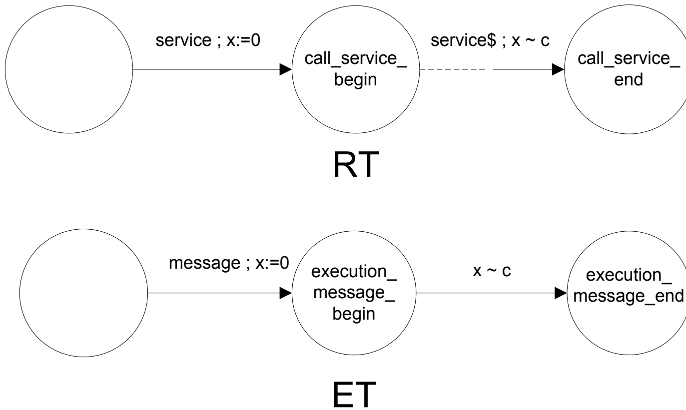


FIG. 3. Motifs temps de réponse et temps d'exécution

transitions peuvent être présentes entre elles. Ceci est représenté sur la partie de la figure RT par la ligne pointillée entre la seconde location et la seconde transition.

Temps d'exécution Le motif *temps d'exécution* (ET) est utilisé pour représenter une exécution avec un automate temporisé. Le temps d'exécution est le temps pris pour effectuer un traitement. Par exemple, après la réception de la réponse à un appel de service, le composant peut avoir besoin d'un temps pour traiter cette réponse. Le motif a trois paramètres : le message $message$, l'opérateur de comparaison $\sim$ et la valeur c. L'automate ET de la Figure 3 représente le motif générique de temps d'exécution. Le motif comprend trois localités, deux transitions et une horloge. L'horloge est initialisée sur la première transition avec le message et comparée sur la seconde sans aucun label. La seconde localité a la propriété $execution_message_begin$ et la troisième $execution_message_end$. Contrairement au motif temps de réponse, les deux transitions de ce motif doivent être consécutives. En effet, le composant ayant reçu des données ne peut rien faire d'autre. C'est pourquoi la seconde transition et la troisième localité n'existent pas dans le comportement d'origine et seront créées lorsque le motif sera appliqué. Cette façon d'ajouter ce motif n'est pas la seule. Par exemple, le composant peut être capable de recevoir des informations ou d'en envoyer pendant le traitement. Dans ce cas, la garde de la seconde transition est ajoutée à toutes les transitions sortantes de la seconde localité de l'automate d'origine.

Comportement temporel Après avoir défini un ensemble de motifs de temps, nous allons maintenant expliquer comment les ajouter au comportement du composant. L'architecte choisit les différents motifs qu'il souhaite ajouter au composant. Une fois les motifs sélectionnés, ils seront automatiquement intégrés au comportement du composant en transformant l'automate originel en automate temporisé. Les motifs seront combinés successivement avec cet automate pour obtenir le comportement temporisé final. Nous illustrons le processus d'ajout en déroulant pas à pas l'ajout de deux motifs au comportement du composant $decoder$.

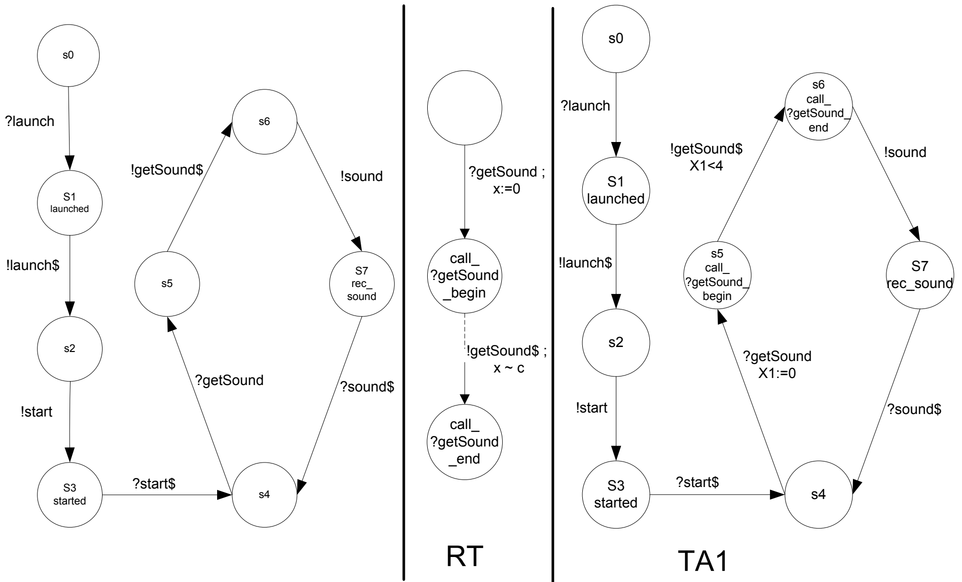


FIG. 4. Ajout du motif temps de réponse

Premièrement, comme illustré Figure 4, le motif temps de réponse est sélectionné avec les paramètres suivants : $getSound$ pour l'appel de service, l'opérateur $<$ et la valeur 4. Une horloge $x1$ est d'abord ajoutée à l'automate. Celle-ci est initialisée sur la transition $?getSound$ partant de la localité $s5$. La propriété $call_getSound_begin$ est ajoutée à la localité $s6$ cible de cette transition. La transition portant l'acquittement de l'appel de service est ensuite sélectionnée. La garde $x1 < 4$ est ajoutée à cette transition et la propriété $call_getSound_begin$ est ajoutée à la localité cible.

Nous appliquons ensuite le motif temps d'exécution avec les paramètres $!getSound\$$, l'opérateur $<$ et la valeur 2. Une deuxième horloge $x2$ est ajoutée et initialisée sur la transition portant $!getSound\$$. Une nouvelle localité $s6_exec$ est ajoutée. Les transitions sortantes de $s6$ deviennent les transitions sortantes de $s6_exec$. Une nouvelle transition entre $s6$ et $s6_exec$ est ajoutée avec la garde $x2 < 2$. Les propriétés $execution_!getSound\$_begin$ et $execution_!getSound\$_end$ sont ajoutées respectivement aux localités $s6$ et $s6_exec$. Le nouvel automate du comportement est montré par l'automate $TA2$ de la figure 5. Ce nouveau comportement du composant ne change pas le comportement d'origine : on peut obtenir $A1$ à partir de $TA2$ en enlevant les horloges puis les transitions sans étiquette.

3.2 Ajout de temps dans les contrats

Le second emplacement où le temps est ajouté est dans les contrats des composants. Les trois premiers niveaux de contrats sont utilisés pour assembler les composants en respectant les propriétés syntaxiques, comportementales et de synchronisation [6]. Nous

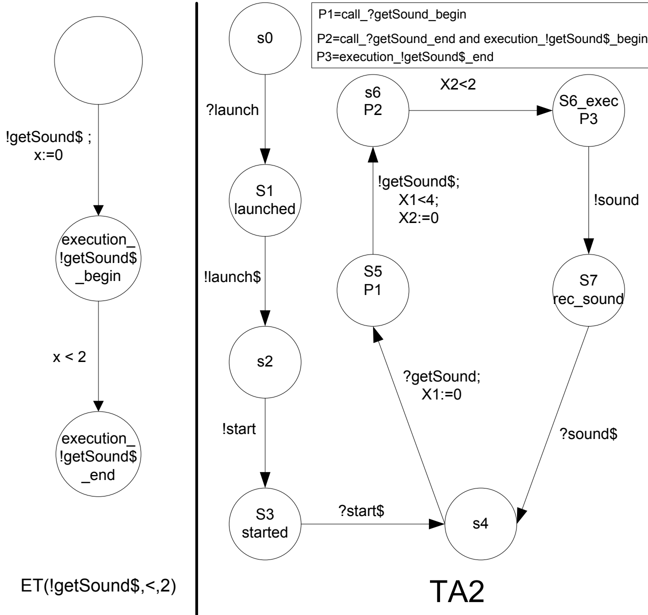


FIG. 5. Ajout du motif temps d'exécution

ajoutons un quatrième niveau de contrats pour la qualité de service de temps. Pour exprimer ces contrats temporels, nous utilisons une logique temporelle temporisée, TCTL. Ces nouveaux contrats seront utilisés lors de la phase de composition des automates temporisés pour valider la compatibilité entre les composants.

TCTL TCTL est une extension de la logique temporelle CTL [13] avec des opérateurs quantitatifs temporels. En CTL, la formule $\exists\Diamond p$ exprime que le prédicat p peut devenir vrai le long de certains chemins d'exécution, mais sans information sur le moment où il devient vrai. L'extension TCTL peut répondre à cette deuxième question : on peut enrichir la formule précédente afin de préciser le moment où p devient vrai. Par exemple la formule $\exists\Diamond_{<5}p$ est vraie le long des chemins d'exécution où p est vraie avant 5 unités de temps.

Soient P un ensemble de propriétés et N l'ensemble des entiers naturels.

Definition 3. *(Syntaxe) Les formules ψ en TCTL sont définies par :*
$\psi := p|false|\psi_1 \to \psi_2|\exists\psi_1 U_{\sim c}\psi_2|\forall\psi_1 U_{\sim c}\psi_2$
où $p \in P$, $c \in N$, et $\sim \in \{<, \leq, =, \geq, >\}$.

Des abrévations sont définies par :
- $\exists\Diamond_{\sim c}\psi$ pour $\exists true U_{\sim c}\psi$ (possibilité),
- $\forall\Diamond_{\sim c}\psi$ pour $\forall true U_{\sim c}\psi$ (des localités le long de toutes les exécutions),
- $\exists\Box_{\sim c}\psi$ pour $\neg\forall\Diamond_{\sim c}\neg\psi$(toutes les localités le long d'une exécution),

– $\forall\Box_{\sim c}\psi$ pour $\neg\exists\Diamond_{\sim c}\neg\psi$ (toutes les localités le long de toutes les exécutions).

Nous interdisons l'utilisation de plus d'une horloge dans la même expression pour éviter le problème de l'analyse en avant [11].

Contrats temporels A l'instar des contrats des trois premiers niveaux, un contrat de qualité de service est attaché aux interfaces requises. Afin de faciliter la définition des contrats temporels, nous définissons un ensemble de motifs s'inspirant de [18]. Ces motifs de contrats sont des squelettes qui devront être complétés par l'architecte et produiront des formules TCTL. Celui-ci pourra aussi écrire les contrats directement en TCTL. Pour créer un nouveau contrat temporel, l'architecte sélectionne le motif désiré et fournit les valeurs des paramètres. Des exemples de motifs sont :

- temps de réponse de c de l'appel de service foo : $call_foo_begin \rightarrow \forall\Box(\forall\Diamond_{\sim c}call_foo_end)$
- période de c de la propriété p : $\forall\Box(\forall\Diamond_{\sim c}p)$
- temps de c entre deux propriétés $p1$ et $p2$: $p1 \Rightarrow \forall\Box(\forall\Diamond_{\sim c}p2)$

Certains contrats sont automatiquement créés lorsque des motifs de temps sont ajoutés par l'architecte. Par exemple, lors du choix du motif de temps de réponse sur un appel de service externe, ($?getSound$ par exemple), un contrat est implicitement créé avec ce motif. En effet ce type de propriétés temporelles mettant en jeu des acteurs extérieurs au composant peut se répercuter sur le comportement et sur ce que le composant requiert. La formule TCTL est donc automatiquement créée avec les paramètres du motif.

3.3 Vérifier les propriétés de temps lors de la composition de composants

Pour vérifier les propriétés de temps lors du processus de composition, nous utilisons le *model-checker* Kronos [12] qui permet de vérifier si un automate temporisé satisfait une formule TCTL ou non. Le comportement de chaque composant primitif est décrit par un automate temporisé et les contrats temporels sont exprimés à l'aide de la logique TCTL. Lorsque l'automate temporisé ne satisfait pas la formule, Kronos identifie les localités où la formule n'est pas vérifiée. Par exemple, si les contrats requis par l'environnement sur le composant $decoder$ sont :

- recevoir $sound$ périodiquement toutes les 7 unités de temps : $\forall\Box(\forall\Diamond_{<7}rec_sound)$
- recevoir $sound$ au plus 5 unités de temps après $launch$:
  $launched \Rightarrow \forall\Box(\forall\Diamond_{<5}rec_sound)$

Kronos répond $vrai$ quand la formule est vérifiée sur l'automate temporisé. Sinon, Kronos répond par la négative et fournit les localités précédant celle où rec_sound est vraie.

L'utilisation des motifs et des contrats TCTL présentés dans cette section permet d'intégrer les informations liées au temps dans une modélisation à base de composants. Du fait de l'orthogonalité entre les contrats TCTL par rapport aux autres contrats liés au comportement et grâce aux motifs, l'étape d'intégration est bien isolée des autres tâches de développement de l'application.

4 Travaux connexes

Le premier objectif de notre article est d'intégrer du temps en tant que qualité de service dans un modèle à composants. De nombreux résultats de recherche ont montré l'utilité de langages dédiés pour décrire les architectures logicielles appelés aussi langage de description d'architecture (ADL). Grâce à la précision de la sémantique de ces langages, des suites d'outils ont été développées pour analyser la cohérence des architectures logicielles et pour les prototyper. SOFA [16], par exemple, étend le langage de description d'interface de l'OMG (IDL *Interface Description Language*) afin de spécifier l'architecture des applications à composants. SOFA fournit aussi une algèbre de processus pour spécifier le comportement externe du composant. Mais en utilisant SOFA, l'architecte ne peut pas décrire la QdS fournie et requise des composants.

Le standard AADL [28] est l'un des premiers ADLs fournissant un mécanisme pour spécifier les quatre niveaux de contrat. Mais AADL est une abstraction de bas niveau et est fortement connecté avec l'implantation. De plus, AADL n'est pas encore connecté avec des outils utilisant les informations de QdS pour analyser la cohérence de l'architecture.

Dans le monde UML, de nombreux profils existent pour la conception de systèmes temps-réel : *Scheduling, Performance and Time* (SPT-UML) [26] de l'OMG, MARTE [1]. Ces profils définissent des concepts et une syntaxe pour décrir des systèmes temps-réel mais la sémantique de ces profils reste imprécise. CQML [33] est un langage lexical pour décrire les spécifications de QdS. Il peut être intégré avec UML et utilisé à différents niveaux d'abstraction. Cependant CQML est intégré à peu d'outils et ne peut donc pas être intégré efficacement dans le processus de développement. Le projet OMEGA [2] fournit des méthodes formelles pour vérifier la cohérence de modèles UML 2. L'approche OMEGA propose un profil UML dédié au développement des systèmes temps réel embarqués critiques. La modélisation est basée sur le langage UML étendu à l'aide du mécanisme de stéreotype. Ce profil définit un raffinement formel du profil SPT-UML de l'OMG. Ce profil associé à la boîte à outils IFx permet de simuler et de valider formellement des propriétés dynamiques de modèles UML. Pour le moment, ce profil permet uniquement de spécifier l'architecture et ne fournit pas d'outils pour générer du code pour une partie de l'implantation.

UPPAAL [19] est un environnement d'outils intégrés pour modéliser, valider et vérifier des systèmes temps-réel modélisés par des réseaux d'automates temporisés. UPPAAL permet d'évaluer des formules CTL sur les automates mais pas TCTL et ne peut donc pas évaluer le quatrième niveau de contrats. Bip [7] est un cadre logiciel pour modéliser des composants hétérogènes. Les composants Bip peuvent être étendus avec des horloges mais le modèle de temps est discret et simulé.

Les travaux de [31] proposent l'ajout d'interfaces temporisées pour les composants. Les interfaces temporisées définissent des propriétés temporelles que l'on souhaite intégrer aux composants. Contrairement à notre approche, dans leurs travaux les automates des interfaces ne sont pas directement ajoutés au comportement du composant. La cohérence entre les automates temporisés définis au niveau des interfaces des composants et les automates traduisant le comportement des composants doit être vérifiée pour garantir que la propriété de temps désirée est compatible avec le composant.

Notre second objectif est d'avoir une bonne séparation des préoccupations afin de pouvoir ajouter la qualité de service de temps sans interférer sur les fonctionnalités des composants. Dans [17], Klein et al. proposent un opérateur de composition (tissage) fondé sur la sémantique des scénarios. Ce travail utilise les *Message Sequence Charts* (MSC) comme langage de scénarios, mais les MSC et les automates à E/S utilisés pour spécifier le comportement sont des langages identiques du point de vue de l'opérateur de tissage. Cependant, l'opérateur de tissage peut aider l'architecte à intégrer la spécification de l'aspect comportemental mais il n'est pas compatible avec les automates temporisés et l'intégration de la QdS dans la spécification du composant. Notre approche peut être vue comme un premier travail pour la composition de temps. Nous ne fournissons pas actuellement de langage de point de coupe définissant la façon dont nous intégrons nos motifs.

5 Conclusion et perspectives

La séparation des préoccupations permet une conception plus facile, augmente la testabilité et la maintenabilité de l'application. La séparation des préoccupations est souvent utilisée pour placer dans différentes unités des préoccupations techniques comme la sécurité, la persistance ou la traçabilité. Cet article considère le temps comme une préoccupation et propose des méthodes pour aider l'architecte à intégrer des informations de qualité de service temporelle pendant la spécification et la conception d'application à composants. L'approche met en avant des motifs de conception pour le comportement et la définition de contrats. L'introduction de ces motifs dans les composants se fait automatiquement, l'architecte n'ayant qu'à paramétrer les motifs choisis.

Le travail présenté dans cet article fait partie d'une approche globale qui a pour objectif la réduction de l'écart entre le modèle de spécification et l'implémentation [30]. L'approche propose un processus unifié pour la conception et l'implémentation des composants. Elle peut aider les architectes dans la gestion du temps dans leur application, en fournissant un ensemble d'outils vérifiant la cohérence des composants tout au long du développement de l'application. La partie spécification abordée dans cet article permet d'obtenir une description formelle de l'application. L'ajout d'un motif est vu comme une opération du méta-modèle des automates temporisés implanté en Kermeta [25]. La partie fonctionnelle de cette description est ensuite fournie à un programmeur tandis que la partie qualité de service temporelle est transmise à un logiciel moniteur afin de vérifier l'implémentation à l'exécution [29]. Ce moniteur est construit automatiquement par une transformation de modèles écrites en Kermeta. A partir d'automates temporisés nous construisons automatiquement une application Giotto [15] (cadre logiciel offrant une abstraction de temps).

Nous travaillons actuellement à la mise en œuvre de ces motifs comme un aspect au niveau modèle (au sens de la conception par aspects). Le but est de décrire un nouvel élément au niveau modèle pour réutiliser des modèles de qualité de service. De plus, nous travaillons à la définition d'un langage de coupe pour simplifier l'intégration d'un même modèle de qualité de service dans plusieurs composants de l'architecture. Cela permettra de construire une couche de qualité de service pouvant être composée avec d'autres aspects de l'architecture.

Références

1. MARTE UML profile RFP. voted at OMG. http ://www.omg.org/cgi-bin/doc ?realtime/2005-02-06.
2. Webpage of the OMEGA IST project. http ://www-omega.imag.fr/.
3. R. Alur, C. Courcoubetis, and D.L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1) :2–34, 1993.
4. R. Alur and D.L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2) :183–235, 1994.
5. T. Amnell, E. Fersman, P. Pettersson, W. Yi, and H. Sun. Code synthesis for timed automata. *Nordic J. of Computing*, 9(4) :269–300, 2002.
6. O. Barais. *Construire et Maîtriser l'Evolution d'une Architecture à base de Composants*. PhD thesis, LIFL, INRIA Futurs, Lille,France, 2005.
7. A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in bip. In *SEFM '06 : Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.
8. A. Beugnard, J-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7) :38–45, 1999.
9. A. Bobbio. System modelling with petri nets. In *Colombo and A. Saiz de Bustamante, editors, Systems Reliability Assessment*, 1990.
10. G. Bollella and J. Gosling. The real-time specification for java. *Computer*, 33(6) :47–54, 2000.
11. P. Bouyer. Untameable timed automata ! In *STACS '03 : Proceedings of the 20th Annual Symposium on Theoretical Aspects of Computer Science*, pages 620–631, London, UK, 2003. Springer-Verlag.
12. M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos : A model-checking tool for real-time systems. In *Proc. 1998 Computer-Aided Verification, CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, Vancouver, Canada, June 1998. Springer-Verlag.
13. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2) :244–263, 1986.
14. S. Graf and I. Ober. A real-time profile for UML and how to adapt it to SDL. In *SDL Forum 2003, July 1-4, Stuttgart*, volume 2708 of *LNCS*, July 2003.
15. T.A. Henzinger, C.M. Kirsch, and B. Horowitz. Giotto : A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1) :84–99, January 2003.
16. T. Kalibera and P. Tuma. Distributed component system based on architecture description : The sofa experience. In *On the Move to Meaningful Internet Systems - DOA, CoopIS and ODBASE*, pages 981–994, London, UK, October 2002. Springer-Verlag. *ISBN : 3-540-00106-9*.
17. J. Klein, L. Hélouet, and J.M. Jézéquel. Semantic-based weaving of scenarios. In *proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06)*, Bonn, Germany, March 2006. ACM.
18. S. Konrad and B. H.C.Cheng. Real-time specification patterns. In *ICSE27*, pages 372–381, May 2005.

19. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2) :134–152, Oct 1997.

20. N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3) :219–246, 1989.

21. J. Magee. Behavioral analysis of software architectures using ltsa. In *Proceedings of the 21st international conference on Software engineering*, pages 634–637. IEEE Computer Society Press, 1999. *ISBN : 1-58113-074-0*.

22. N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. In *IEEE Transactions on Software Engineering*, volume 26, page 23, January 2000.

23. P. M. Merlin. *A study of the recoverability of computing systems.* PhD thesis, Department of Information and Computer Science, University of California, Irvine, CA, 1974.

24. B. Meyer. Applying "design by contract". *Computer*, 25(10) :40–51, 1992.

25. P-A. Muller, F. Fleurey, and J-M. Jézéquel. Weaving executability into object-oriented meta-languages. In L. C. Briand and Clay Williams, editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2005.

26. OMG. *UML Profile for Schedulability, Performance, and Time Specification*, January 2005. Version 1.1.

27. C. Ramchandani. *Analysis of asynchronous concurrent systems by timed petri nets*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1974.

28. As-2 Embedded Computing Systems Committee SAE. Architecture Analysis & Design Language (AADL). SAE Standards n^o AS5506, November 2004.

29. S. Saudrais, O. Barais, and L. Duchien. Using model-driven engineering to generate qos monitors from a formal specification. *edoc-workshop*, page 45, 2006.

30. S Saudrais, O. Barais, and N. Plouzeau. Composants avec propriétés temporelles. In M. C. Oussalah, F. Oquendo, D. Tamzalit, and T. Khammaci, editors, *CAL*, pages 143–149. Hermes Science, 2006.

31. B. Schätz. Interface descriptions for embedded components. In *3rd Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER'05)*, Paderborn, 2005.

32. A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4) :333–369, 1997.

33. J. Øyvind Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, Department for Informatics, University of Oslo, June 2001.

Vers la génération automatique de tests à partir d'arbres de tâches

Laya Madani and Ioannis Parissis

Laboratoire Informatique de Grenoble
BP 53 - 38041 Grenoble Cedex 9 - France
{laya.madani, ioannis.parissis}@imag.fr

Résumé L'arbre de tâches décrit l'interaction entre l'application interactive et l'utilisateur. Il comporte ainsi des informations sur le comportement de ce dernier. Ces informations peuvent être utilisées pour la génération de tests. En s'appuyant sur nos travaux précédents sur le test d'applications interactives au moyen de l'approche synchrone, on présente ici une méthode pour la génération automatique des données de test à partir d'arbres de tâches. L'arbre de tâche est automatiquement analysé et le comportement de l'utilisateur est extrait sous la forme d'une machine d'états finis à entrées-sorties. Cette dernière est construite conformément à une sémantique formelle définie à cet effet pour tous les opérateurs de l'arbre de tâche. Ce modèle est ensuite utilisé pour générer des tests simulant le comportement de l'utilisateur.

1 Introduction

Les applications interactives sont aujourd'hui présentes dans plusieurs domaines critiques comme le contrôle de vol ou le contrôle de processus industriels critiques. Elles assurent également l'accès à des services commerciaux divers comme les téléphones mobiles, les systèmes de réservation ou les services de télécommunication. Leur correction devient ainsi un enjeu très important et leur développement requiert une validation rigoureuse. Plusieurs méthodes automatiques basées sur des spécifications formelles ont été proposées pour la modélisation et la vérification de systèmes interactifs. Dans la plupart de ces méthodes, comme FSM (Formal System Modelling) [7], LIM (Lotos Interactor Model) [8] ou ICO (Interactive Cooperative Object) qui est basée sur les réseaux de Petri [14,13], l'application interactive est spécifiée formellement comme un modèle abstrait. Les propriétés sont vérifiées sur ce modèle par des techniques traditionnelles de vérification formelle comme le model-checking. L'utilisation de la méthode B a également été suggérée [2] pour assurer que les propriétés de l'application interactive sont préservées pendant le processus du raffinement. Dans tous les cas, la vérification requiert une démarche lourde de spécification que la plupart de concepteurs des applications interactives ne peuvent pas effectuer.

Nous avons récemment proposé [11] d'utiliser l'approche synchrone pour le test des systèmes interactifs. Contrairement aux méthodes ci-dessus, cette

technique requiert une spécification partielle du comportement de l'utilisateur de l'application. Cette spécification, fournie sous la forme d'un ensemble d'expressions Lustre [5], peut être renforcée par des probabilités d'occurrence ou des spécifications de scénarios. Ensuite, plusieurs stratégies de génération peuvent être appliquées pour engendrer automatiquement des tests. La génération des entrées est effectuée "à la volée" (les entrées sont calculées en fonction des entrées et des sorties précédentes). Cependant, ce modèle synchrone du comportement de l'utilisateur n'est pas facile à construire pour les concepteurs des applications interactives, non familiers de ces langages. Pour cette raison, nous avons étudié des méthodes de génération de données de test basées sur des modèles qui sont plus habituels dans le processus de développement des applications interactives, les *arbres de tâches*. Ces derniers décrivent les interactions entre l'application et l'utilisateur.

La contribution principale de ce papier est la définition formelle de la transformation de l'arbre de tâche en un modèle comportemental de l'utilisateur qui peut être utilisé pour la génération de données de test. Ce modèle est une machine d'états finis à entrées-sorties, qui peut être parcourue selon plusieurs stratégies pour simuler les entrées de l'utilisateur afin de tester l'application. Le paragraphe 2 présente le formalisme de description d'arbres de tâches que nous utilisons, CTT. Dans le paragraphe 3 on définit formellement la transformation de l'arbre de tâche vers une machine à E/S et on montre comment cette machine peut être utilisée pour générer les séquences de test.

2 Modèles de tâches

2.1 Définitions et notations

Les modèles de tâches sont souvent utilisés dans la conception d'applications interactives. Dans ces modèles, les tâches sont représentées d'une manière hiérarchique [1] : une tâche est composée de sous-tâches liées par des opérateurs temporels. Cela signifie que le modèle de tâches fournit des informations sur les sous-tâches qui doivent être exécutées afin d'accomplir une autre tâche plus complexe, ainsi que des informations sur le comportement de système interactif. Nous utilisons la notation bien connue CTT (*ConcurTaskTrees*) [12]. CTT distingue quatre types de tâches. Une **tâche usager** est une activité cognitive sans interaction avec le système (la réflexion pour résoudre un problème, lecture d'un message...). Une **tâche application** est une action du système, comme l'affichage du résultat d'une requête. Une **tâche interactive** correspond à une action de l'usager avec un feedback immédiat du système, comme l'édition d'un document. Enfin, une **tâche abstraite** est composée d'autres tâches liées par des opérateurs temporels :

Activation	**T1 >> T2**	**T2 activée par l'exécution de la tâche T1**
Activation - passage d'information	**T1 []>> T2**	**Idem mais T1 fournit des informations à T2**
Choix	**T1 [] T2**	**Une seule des tâches T1 et T2 est exécutée**
Entrelacement (Concurrence indépendante)	**T1 ||| T2**	**Les actions de T1 et T2 sont effectuées dans un ordre quelconque**
Concurrence avec échange d'information	**T1 |[]| T2**	**Idem mais synchronisation pour échange d'information.**
Désactivation	**T1 [> T2**	**T1 désactivée quand la première action de T2 est effectuée.**
Suspendre-reprendre	**T1 |> T2**	**T2 interrompt T1 ; quand T2 est terminée, T1 reprend dans l'état ou elle a été interrompue**
Itération	**T***	**T exécutée de manière itérative jusqu'à ce qu'une autre tâche la désactive.**
Itération finie	**T(n)**	**T exécutée n fois.**
Tâche optionnelle	**[T]**	**L'exécution de T est optionnelle.**

2.2 Un exemple : Memo

L'application interactive Memo [4] permet d'annoter des localisations physiques avec des notes ("post-it") digitales. Lorsqu'une note a été placée à un endroit physique, elle peut être ensuite lue/portée/supprimée par d'autres utilisateurs. L'utilisateur de Memo est équipé d'un GPS et d'un magnétomètre pour permettre le calcul de la localisation et de l'orientation de ce dernier. Il porte également un casque dont la demi-transparence permet la fusion entre les données électroniques (les notes digitales) et l'environnement physique. Trois tâches principales sont définies dans Memo : (1) le changement d'orientation et de localisation de l'utilisateur mobile ; le système affiche sur le casque les notes visibles selon la position et l'orientation courantes de l'utilisateur mobile (2) la manipulation d'une note (récupérer, placer et supprimer une note) et (3) la sortie du système. L'utilisateur mobile peut ainsi récupérer une note qu'il portera pendant son déplacement. Il ne peut porter qu'une seule note à la fois. Enfin, il peut supprimer une note portée ou visible dans son casque. La figure 1 présente l'arbre des tâches de l'application "Memo" dans la notation CTT. Cet arbre montre que l'utilisateur peut utiliser l'application itérativement (l'opérateur d'itération *) et peut être interrompu (l'opérateur de désactivation [>) par la tâche "exit". La tâche "use memo system" est un choix entre les trois tâches suivantes : découvrir le terrain ("explore the ground"), la manipulation d'une note visible ("handle a displayed note") ou la manipulation d'une note portée

(”handle a carried note”). Si le système affiche une note (la tâche ”memoDisplayed”), l’utilisateur peut (opérateur d’activation >>) récupérer ou supprimer cette note. Si l’utilisateur est en train de porter une note (”memoCarried”), l’utilisateur peut placer ou supprimer cette note.

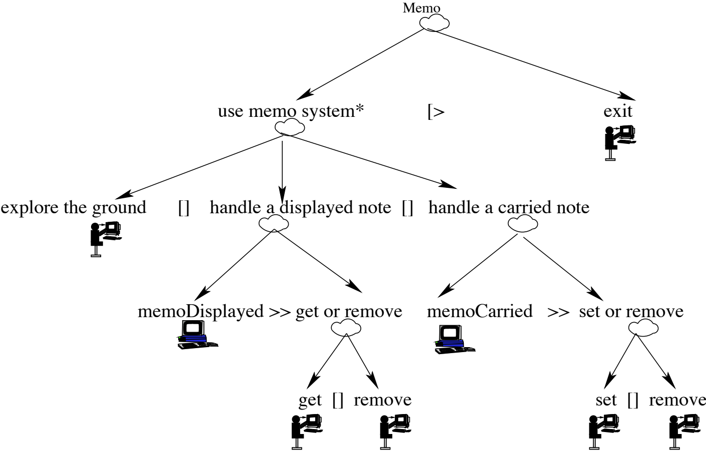


Fig. 1. Arbre de tâches de Memo

3 Extraction du modèle de l’utilisateur

3.1 Motivation et hypothèses

Lors de récents travaux de recherche [11], nous avons montré qu’il est possible de tester une application interactive au moyen de Lutess [6], un environnement de test de programmes synchrones. Ces programmes ont un comportement cyclique : à chaque top d’une horloge globale, ils lisent leurs entrées et calculent instantanément leurs sorties. Lutess requiert la spécification de l’environnement du programme sous test sous la forme de propriétés invariantes écrites en langage Lustre [9]. A partir de cette spécification non-déterministe, Lutess construit un générateur de données de test : à chaque top, le générateur produit aléatoirement un vecteur valide d’entrées et l’envoie au logiciel, qui réagit en émettant un vecteur de sortie. Le générateur produit un nouveau vecteur d’entrée et le cycle se répète tandis qu’un oracle automatique observe les entrées et les sorties du programme.

D’un point de vue formel, la spécification de l’environnement est une machine d’états finis $M_{env} = (V_o, V_i, V_{sve}, env, t_e)$:

- i et o étant les vecteurs des variables d'entrée et de sortie du logiciel, V_i et V_o sont leurs ensembles de valeurs associés ;
- V_{sve} est l'ensemble des états de l'environnement (ensemble de valeurs des variables d'état sve) ;
- la fonction booléenne $env : V_{sve} \times V_i \longrightarrow Bool$ définit à tout moment t l'ensemble des valeurs du vecteur d'entrée i conforme à la spécification de l'environnement (i.e. ensemble de valeurs rendant vraie la fonction env dans l'état où se trouve l'environnement à l'instant t) ;
- la fonction de transition $t_e : V_{sve} \times V_i \times V_o \longrightarrow V_{sve}$ calcule l'état de l'environnement après chaque échange d'entrées et sorties avec le logiciel.

L'algorithme de génération de test choisit un vecteur valide d'entrée conforme à la spécification de l'environnement (une valeur des variables d'entrée rendant vraie la fonction de l'environnement env). Pour chaque état, tous les vecteurs d'entrée valides possèdent la même probabilité d'être sélectionnés si le mode de génération choisi est la *simulation aléatoire de l'environnement*. Lutess fournit des stratégies de génération guidées par des *profils opérationnels*, les *schémas comportementaux* [6] et le test guidé par des *propriétés de sûreté* [15] où les entrées sélectionnées sont potentiellement plus aptes à détecter des violations des propriétés. L'utilisation de Lutess pour le test des systèmes interactifs est fondée sur l'hypothèse suivante : une application interactive peut être assimilée, sous certaines conditions, à un programme synchrone [3]. En effet, l'hypothèse de synchronisme est vérifiée si le logiciel est capable de prendre en compte toute évolution de son environnement externe. Ainsi, une application interactive peut être vue comme un programme synchrone si toutes les actions de l'utilisateur et autre stimuli externes sont pris en compte lors de son exécution.

Pour utiliser Lutess dans le cadre du test d'applications interactives, il faut fournir la spécification de l'environnement dans le langage Lustre enrichi utilisé par Lutess. Ce formalisme nous semble être d'un niveau inhabituellement bas pour les concepteurs de ce type d'applications. Afin de leur permettre d'utiliser l'environnement de Lutess sans écrire une telle spécification, nous proposons d'extraire un modèle de comportement de l'utilisateur à partir de l'arbre de tâches écrit en notation CTT. Ce modèle est une machine d'états finis à entrées-sorties, capable de supporter la génération automatique de données de test. Pour cela, il est nécessaire de définir précisément la sémantique des opérateurs CTT, adaptée à la génération de test.

3.2 Définitions préliminaires

Comme il a été mentionné dans la section 2, une tâche en notation CTT peut être une tâche usager, une tâche abstraite, une tâche application ou une tâche interactive.

Les tâches usager ne sont pas intéressantes du point de vue de la génération de tests, car elles ne correspondent à aucune interaction avec le système (ni action ni réaction).

Nous assimilons une tâche application o à une machine d'états élémentaire à deux états, dont la seule transition consiste à passer de l'état initial à l'état final en émettant une sortie. Formellement, elle est modélisé par la machine $M_o = (Q_o, qi_o, qf_o, I_o, O_o, trans_o)$ où :

- $Q_o = \{qi_o, qf_o\}$ est un ensemble d'états ; qi_O , qf_O sont respectivement l'état initial et l'état final de M_O
- $I_o = \{\mu\}$ est un ensemble d'entrée ; μ est une entrée vide (l'utilisateur ne procède à aucune action).
- $O_o = \{o\}$ est un ensemble de sortie.
- $trans_o = \{qi_o \xrightarrow{\mu/o} qf_o\}$ est un ensemble de transitions. La notation $qi_o \xrightarrow{\mu/o} qf_o$ signifie qu'il y a une transition de l'état qi_o vers l'état qf_o avec l'entrée vide μ est la sortie o.

La principale hypothèse de notre approche concerne les tâches interactives. Nous supposons que ces tâches sont modélisées par des machines d'états finis à E/S et que cette modélisation doit être fournie en même temps que l'arbre des tâches. Cette hypothèse est également discutée dans la conclusion de l'article.

Formellement, pour une tâche interactive T, on définit la machine d'états finis à E/S $M_T = (Q_T, qi_T, qf_T, I_T, O_T, trans_T)$ où :

- Q_T est un ensemble d'états.
- qi_T est l'état initial. C'est un état source.
- qf_T est l'état final. C'est un état puits.
- I_T est l'ensemble des entrées de l'application pour la tâche T.
- O_T est l'ensemble des sorties de l'application pour la tâche T.
- $trans_T \subseteq Q_T \times (I_T \cup \{\mu\}) \times O_T \times Q_T$ est l'ensemble de transitions de la tâche T. Si $(q_T, a, b, p_T) \in trans_T$, on écrit $q_T \xrightarrow{a/b} p_T$. Quelquefois on omet l'entrée et la sortie de la transition : $q_T \xrightarrow{c} p_T$ signifie $c = e/s$.

Finalement, une tâche abstraite est composée d'autres sous-tâches liées par des opérateurs de CTT. Une machine à E/S peut être aussi associée à une tâche abstraite. Elle résulte de la combinaison des machines à E/S de ses sous-tâches comme expliqué dans le paragraphe suivant.

3.3 Transformation d'une tâche abstraite en une machine à E/S

Opérateur d'activation ($A >> B$ ou $A[\,] >> B$) Soit T=A> >B ou T=A[]> >B. Nous considérons que les informations passant de la tâche A à la tâche B, lorsque l'opérateur $[\,] >>$ est utilisé, n'ont pas d'intérêt du point de vue de la génération de données de test, car cette communication est interne à l'application. Puisque B commence dès que A termine, l'état final de A, qf_A, sera l'état initial de B , qi_B, ($qf_A = qi_B$). La figure 2 illustre la définition formelle suivante :

$$M_T(Q_T, qi_T, qf_T, I_T, O_T, trans_T)$$
$$= M_A(Q_A, qi_A, qf_A, I_A, O_A, trans_A) >> M_B(Q_B, qi_B, qf_B, I_B, O_B, trans_B)$$
$$= M_A(Q_A, qi_A, qf_A, I_A, O_A, trans_A)[\,] >> M_B(Q_B, qi_B, qf_B, I_B, O_B, trans_B)$$

$Q_T = Q_A \cup Q_B$ avec $qf_A = qi_B$ et $Q_A \cap Q_B = \{qf_A\} = \{qi_B\}$
$qi_T = qi_A$, $qf_T = qf_B$, $I_T = I_A \cup I_B$, $O_T = O_A \cup O_B$, $trans_T = trans_A \cup trans_B$

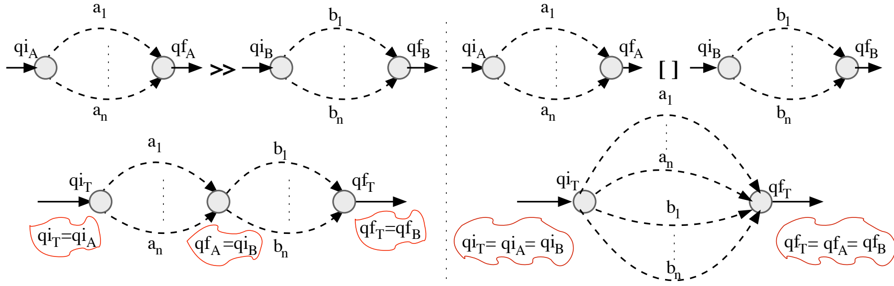


Fig. 2. Sémantique des opérateurs activation " $>>$ " et choix "$[\,]$"

Opérateur de Choix ($A[\,]B$) Soit $T = A[\,]B$. T est réalisée en choisissant une tâche entre A et B. Ainsi, la tâche T commence et se termine en même temps que la tâche choisie. En conséquence, l'état initial de T, qi_T, sera confondu avec l'état initial de A , qi_A et de B , qi_B ($qi_T = qi_A = qi_B$). L'état final de T, qf_T, sera identique à l'état final de A, qf_A, et de B, qf_B ($qf_T = qf_A = qf_B$). La figure 2 illustre cette définition formelle :

$$M_T(Q_T, qi_T, qf_T, I_T, O_T, trans_T)$$
$$= M_A(Q_A, qi_A, qf_A, I_A, O_A, trans_A)[\,]M_B(Q_B, qi_B, qf_B, I_B, O_B, trans_B)$$

$Q_T = Q_A \cup Q_B$ avec $qi_T = qi_A = qi_B$ et $qf_T = qf_A = qf_B$ et $Q_A \cap Q_B = \{qi_A, qf_A\} = \{qi_B, qf_B\}$

$I_T = I_A \cup I_B$, $O_T = O_A \cup O_B$, $trans_T = trans_A \cup trans_B$

Opérateur d'entrelacement (Concurrence indépendante) (A ||| B) Soit $T = A \; ||| \; B$. La figure 3 montre un exemple d'utilisation de la définition formelle de la machine résultante ci-dessous :

$$M_T(Q_T, qi_T, qf_T, I_T, O_T, trans_T) \\ = M_A(Q_A, qi_A, qf_A, I_A, O_A, trans_A) ||| M_B(Q_B, qi_B, qf_B, I_B, O_B, trans_B)$$

$Q_T = Q_A \times Q_B$, $qi_T = (qi_A, qi_B)$, $qf_T = (qf_A, qf_B)$, $I_T = I_A \cup I_B$, $O_T = O_A \cup O_B$

La relation $trans_T$ est définie comme suit :

$$(q_A, q_B) \xrightarrow{c}_T (p_A, p_B) \; ssi \begin{cases} soit \; (q_A \xrightarrow{c}_A p_A) \; et \; q_B = p_B \\ soit \; (q_B \xrightarrow{c}_B p_B) \; et \; q_A = p_A \end{cases}$$

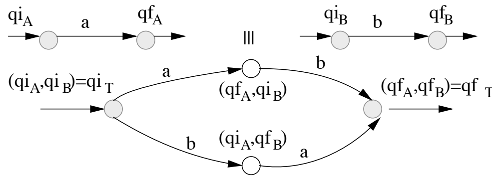


Fig. 3. Un exemple de l'opérateur d'entrelacement"|||"

Opérateur de désactivation ($A[> B$) Soit $T = A[> B$. Dans la machine à E/S de T, M_T, à partir de chaque état de M_A(sauf l'état final qf_A), il y a une transition étiquetée par la première action de la tâche B vers l'état correspondant de la machine M_B. Puisque T se termine lorsque soit A s'est terminée sans interruption, soit B s'est terminée après avoir interrompu A , l'état final de T, qf_T, coïncide avec l'état final de A ($qf_T = qf_A$) et également l'état final de B ($qf_T = qf_B$). La figure 4 illustre la définition formelle :

$$M_T(Q_T, qi_T, qf_T, I_T, O_T, trans_T) \\ = M_A(Q_A, qi_A, qf_A, I_A, O_A, trans_A)[> M_B(Q_B, qi_B, qf_B, I_B, O_B, trans_B)$$

$Q_T = Q_A \cup Q_B$ avec $qi_T = qi_A = qi_B$ et $qf_T = qf_A = qf_B$ et $Q_A \cap Q_B = \{qi_T, qf_T\}$

$I_T = I_A \cup I_B$, $O_T = O_A \cup O_B$

$trans_T = trans_A \cup trans_B \cup trans_{AB}$ où la relation $trans_{AB} \subseteq (Q_A \setminus \{qf_A\}) \times (I_B \cup \{\mu\}) \times O_B \times Q_B$ est définie ci-dessous :

$q_A \xrightarrow{b}_{AB} q_B \; ssi \; qi_B \xrightarrow{b}_B q_B$

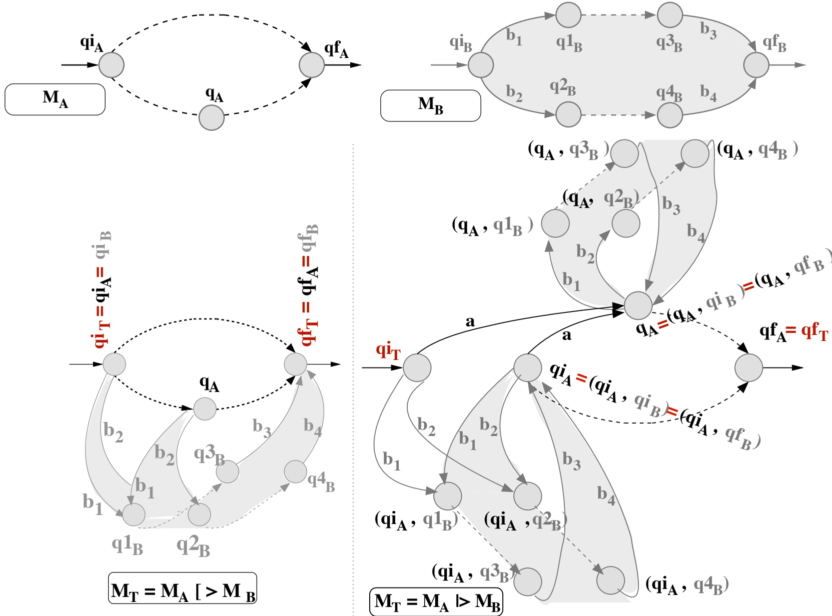


Fig. 4. Sémantique de désactivation" [> "et suspendre-reprendre " |> "

Opérateur suspendre-reprendre ($A \mid > B$) Soit $T = A \mid > B$. Dans la machine M_T, à partir de chaque état $q_A$de M_A (sauf l'état final qf_A), il y a une transition étiquetée par la première action de la tâche B vers l'état correspondant de la machine M_B avec mémorisation de l'état de départ ($q_A = (q_A, qi_B)$). Lorsque toutes les actions de B se sont exécutées, l'exécution de A reprend à partir de l'état mémorisé ($(q_A, qf_B) = q_A$). L'état initial d'une tâche étant par définition un état source, nous ajoutons un nouvel état qi_T dont les transitions sortantes sont les mêmes que celles de qi_A. La figure 4 illustre la composition entre les machines à E/S de deux tâches A et B définie formellement comme suit :

$M_T(Q_T, qi_T, qf_T, I_T, O_T, trans_T)$
$= M_A(Q_A, qi_A, qf_A, I_A, O_A, trans_A) \mid > M_B(Q_B, qi_B, qf_B, I_B, O_B, trans_B)$

Notation $Q_A^{-fin} = Q_A \setminus \{qf_A\}$
$Q_T = Q_A \cup (Q_A^{-fin} \times Q_B) \cup \{qi_T\}$ avec $qf_T = qf_A$
et $\forall q_A \in Q_A^{-fin} : q_A = (q_A, qi_B) = (q_A, qf_B)$
$I_T = I_A \cup I_B$, $O_T = O_A \cup O_B$, $trans_T = trans_A \cup trans_{AB} \cup trans_{\dot{T}}$
Où la relation $trans_{AB} \subseteq (Q_A^{-fin} \times Q_B) \times (I_B \cup \{\mu\}) \times O_B \times (Q_A^{-fin} \times Q_B)$ est définie ci-dessous:
$(q_A, q_B) \xrightarrow{b}_{AB} (p_A, p_B) \quad ssi \quad q_B \xrightarrow{b}_B p_B \; et \; q_A = p_A$
Et la relation $trans_{\dot{T}} \subseteq \{qi_T\} \times (I_T \cup \{\mu\}) \times (Q_A \cup (\{qi_A\} \times Q_B))$ est définie ci-dessous :
$qi_T \xrightarrow{a}_{\dot{T}} q_A \; ssi \; qi_A \xrightarrow{a}_A q_A$
$qi_T \xrightarrow{b}_{\dot{T}} (qi_A, q_B) \; ssi \; qi_B \xrightarrow{b}_B q_B$

Opérateur d'itération (A^*) Soit $T = A^*$. Dans la machine à E/S correspondant à la tâche répétitive T, l'état final de A, qf_A est le même que son état initial, qi_A ($qf_A = qi_A$). Puisque par définition l'état initial est source, nous ajoutons un nouvel état qi_T. Les mêmes actions, qui peuvent s'exécuter à partir de qi_A, peuvent également s'exécuter à partir de qi_T. La figure 5 donne une représentation graphique de la définition formelle suivante :
$M_T(Q_T, qi_T, qf_T, I_T, O_T, trans_T) = M_A(Q_A, qi_A, qf_A, I_A, O_B, trans_A)^*$
$Q_T = Q_A \cup \{qi_T, qf_T\}$ avec $qf_A = qi_A$
$I_T = I_A$, $O_T = O_A, trans_T = trans_{\dot{A}} \cup trans_A$
Où la relation $trans_{\dot{A}} \subseteq \{qi_T\} \times (I_A \cup \{\mu\}) \times O_A \times Q_A$ est définie ci-dessous :
$qi_T \xrightarrow{a}_{\dot{A}} q_A \; ssi \; qi_A \xrightarrow{a}_A q_A$
L'état final de la tâche itérative qf_T n'est pas atteignable parce que, par définition, l'itération est infinie (elle continue jusqu'à sa désactivation par une autre tâche).

Opérateur d'itération finie ($A(n)$) On considère deux tâches A et T et un entier *n*, tels que :$T = A(n); n > 1$. Un compteur (i) est utilisé dans la définition de cette machine. Pendant la répétition de la tâche A (quand $1 \leq i \leq n-1$), l'état final de A dans la *ième* itération sera le même que l'état initial dans la *(i+1)ème* itération ($(qf_A, i) = (qi_A, i+1)$). La figure 5 illustre la définition formelle suivante :
$M_T(Q_T, qi_T, qf_T, I, O, trans_T) = M_A(Q_A, qi_A, qf_A, I, O, trans_A)(n)$

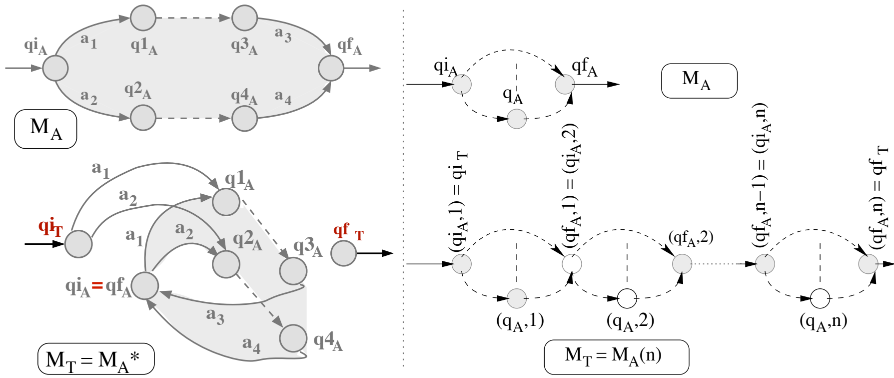


Fig. 5. Sémantique des opérateurs itération "*" et itération finie

$Q_T = \{(q_A, i)\ ;\ \forall q_A \in Q_A \& \forall i\ ;\ 1 \leq i \leq n\}$ avec $(qf_A, i) = (qi_A, i + 1)\,,\ \forall i \in [1..n-1]$

$qi_T = (qi_A, 1)\ ,\ qf_T = (qf_A, n)$

Où la relation $trans_T$ est définie ci-dessous :

$(q_A, i)\ \xrightarrow{a}_T\ (s_A, j)\ ssi\ q_A\ \xrightarrow{a}_A\ s_A\ et\ i = j$

La tâche optionnelle ($[A]$) Les tâches optionnelles doivent être utilisées avec les opérateurs d'activation ($[A] >> B = (A >> B)[]B$) ou de concurrence ($[A]|||B = (A|||B)[]B$) [10,16]. La machine de la tâche optionnelle est construite à partir des machines des autres tâches selon l'opérateur de la composition et en utilisant la sémantique définie ci-dessus.

Exemple Reprenons l'exemple de la figure 1. La tâche abstraite *"Memo"* est définie à l'aide de cinq tâches interactives : *"get", "set", "remove", "explore the ground"*, *"exit"* et deux tâches application : *"memoDisplayed", "memoCarried"*. Ces tâches sont modélisées par les machines à E/S illustrées par la figure 6. Lorsque l'utilisateur envoie la commande "*get*" sur une note affichée au système Memo, l'application réagit en envoyant le message "*memo is taken*" et la note disparaît du terrain. Quand l'utilisateur supprime une note (portée ou affichée), le système envoie le message *"memo is removed*". Enfin, lorsque l'utilisateur porte une note *("memoCarried*") et envoie la commande "*set*", le système envoie le message "*memo is set*" et la note est reposée sur le terrain.

Les deux tâches abstraites "get or remove" et "set or remove" sont modélisées par les deux machines à E/S illustrées par la figure 7. Chaque machine résulte de la combinaison des machines à E/S de ses sous-tâches ($M_{\text{get or remove}}$=

M_{get}[] M_{remove}, $M_{set\ or\ remove}$= M_{set}[] M_{remove}) en appliquant la sémantique de l'opérateur "choix".

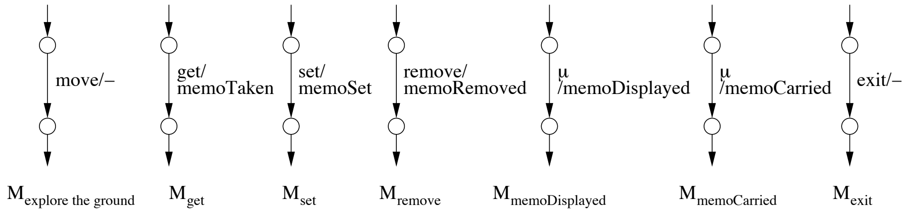


Fig. 6. La machine à E/S pour les tâches:
"explore the ground", "get", "set", "remove","memoDisplayed" , "memoCarried" et 'exit"

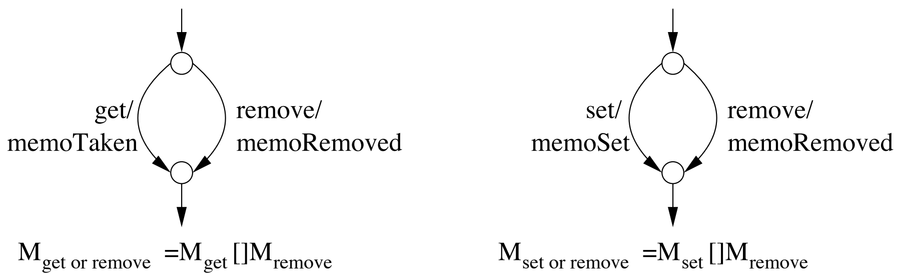
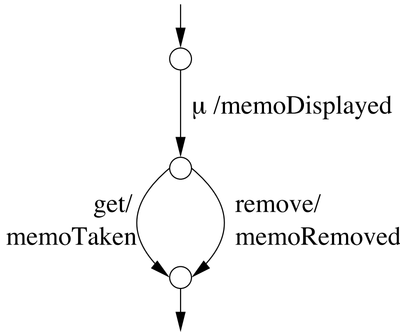


Fig. 7. La machine à E/S pour les tâches: *"get or remove", "set or remove"*

Ainsi, en appliquant la sémantique des opérateurs définie précédemment, on obtient les machines à E/S de toutes les tâches de l'exemple de la figure 1. Les figures 8, 9 montrent les machines des tâches : *"handle a displayed note"*, *handle a carried note"* et *"use Memo system"*. La machine de la tâche itérative *"use Memo system*"* (figure 10) est obtenue à partir de la machine $M_{use\ Memo\ system}$en considérant l'état final q_4 équivalent à l'état initial q_1 (toutes les transitions destinées à l'état final q_4 dans la machine $M_{use\ Memo\ system}$ sont destinées à l'état q_1 dans la machine $M_{use\ Memo\ system*}$) et en ajoutant un nouvel état source q_0, du lequel peuvent s'exécuter les mêmes actions, qui peuvent s'exécuter à partir de q_1.

La machine à E/S de la tâche *"Memo"* est illustrée par la figure 11. Cette machine est obtenue à partir de $M_{use\ Memo\ system*}$ (figure 10) en ajoutant pour chaque état une transition étiquetée par l'action *"exit"* vers l'état final q_5.



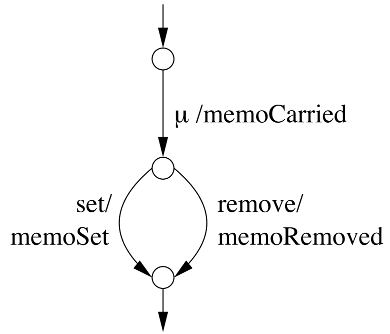


Fig. 8. La machine à E/S pour les tâches:*"handle a displayed note", "handle a carried note"*

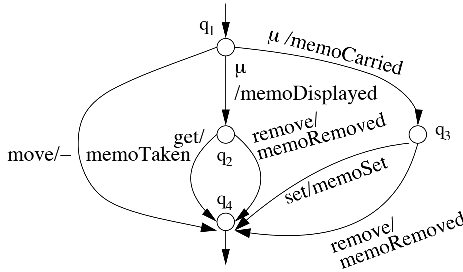


Fig. 9. La machine à E/S pour la tâche: *"use Memo System"*

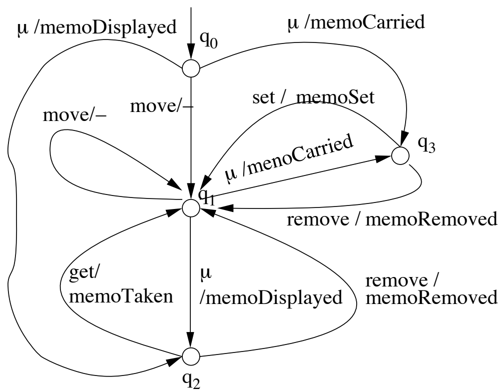


Fig. 10. La machine à E/S pour la tâche: *"use Memo System*"*

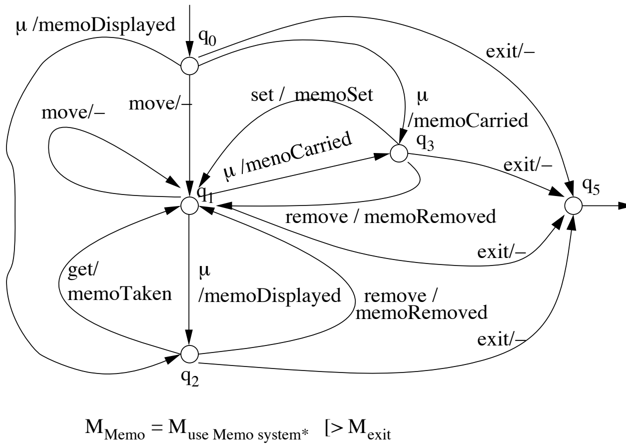


Fig. 11. La machine à E/S pour la tâche: *"Memo"*

3.4 Simulation du comportement de l'utilisateur

Algorithm 1

1. ***var***
2. $preq, q \in Q_T$, $pFollowingq \in 2^{Q_T}$, $i \in (I_T \cup \{\mu\})$, $oset \in 2^{O_T}$
3. ***begin***
4. $q \leftarrow qi_T$
5. ***do forever***
6. $oset \leftarrow \emptyset$
7. $i \leftarrow random(beh_T(q))$
8. ***if*** $(i \neq \mu)$ ***then*** $write(i)$
9. $wait(C)$
10. $read(oset)$
11. $preq \leftarrow q$
12. $pFollowingq \leftarrow pTrans_T(preq, i, oset)$
13. ***if*** $(pFollowingq \neq \emptyset)$ ***then*** $q \leftarrow random(pFollowingq)$
14. ***else*** $q \leftarrow preq$
15. ***end do***
16. ***end***

La simulation du comportement de l'utilisateur, nécessaire pour engendrer les séquences de tests, peut se faire sur la machine à E/S associée à l'arbre de tâches. Cette machine est semblable à celle utilisée par Lutess à l'exception de l'absence d'entrée ou de sortie sur certaines transitions. Nous proposons dans ce paragraphe une adaptation de l'algorithme de génération l'aide de deux fonctions beh_T et $pTrans_T$ définies ci-dessous :

Définition : $beh_T : Q_T \longrightarrow 2^{I_T \cup \mu}$ où
$beh_T(q) = \{i \in (I_T \cup \{\mu\}) \,|\, \exists o, p,\; q \xrightarrow{i/o}_T p\}$ est l'ensemble de toutes les entrées valides de l'application à l'état q.

Définition : $pTrans_T : Q_T \times (I_T \cup \{\mu\}) \times 2^{O_T} \longrightarrow 2^{Q_T}$ où
$pTrans_T(q, i, os) = \{p |\; q \xrightarrow{i/o}_T p, o \in os\}$ est l'ensemble des transitions issues de l'état q ayant i pour entrée et dont la sortie est dans os.

La génération de données de test est illustrée par l'algorithme 1. La fonction *random*, appliquée à un ensemble fini, retourne un élément aléatoire de cet ensemble. Suivant cet algorithme, à chaque état une entrée aléatoire est choisie ($i \leftarrow random(beh_T(q))$) parmi les entrées sur les transitions issues de cet état. L'entrée choisie est envoyée à l'application interactive. Puis, les sorties de l'application sont lues $(read(oset))$. L'ensemble des états successeurs possibles est calculé en fonction de l'entrée envoyée et les sorties lues $(pFollowingq \leftarrow pTrans_T(preq, i, oset))$. De cet ensemble, un état aléatoire est choisi, et ainsi de suite.

No du pas	Entrée	**Sorties**	**Transitions possibles**	**Transition choisie**
1	$move$	$-$	$q_0 \xrightarrow{move/-} q_1$	$q_0 \xrightarrow{move/-} q_1$
2	μ	$-$	-	-
3	$move$	$memoDisplayed$	$q_1 \xrightarrow{move/-} q_1$	$q_1 \xrightarrow{move/-} q_1$
4	μ	$memoDisplayed$	$q_1 \xrightarrow{\mu/memoDisplayed} q_2$	$q_1 \xrightarrow{\mu/memoDisplayed} q_2$
5	get	$memoTaken,$ $memoCarried$	$q_2 \xrightarrow{get/memoTaken} q_1$	$q_2 \xrightarrow{get/memoTaken} q_1$
6	$move$	$memoCarried,$ $memoDisplayed$	$q_1 \xrightarrow{move/-} q_1$	$q_1 \xrightarrow{move/-} q_1$
7	μ	$memoCarried,$ $memoDisplayed$	$q_1 \xrightarrow{\mu/memoCarried} q_3,$ $q_1 \xrightarrow{\mu/memoDisplayd} q_2$	$q_1 \xrightarrow{\mu/memoCarried} q_3$
8	set	$memoSet,$ $memoDisplayed$	$q_3 \xrightarrow{set/memoSet} q_1$	$q_3 \xrightarrow{set/memoSet} q_1$
9	μ	$memoDisplayd$	$q_1 \xrightarrow{\mu/memoDisplayed} q_2$	$q_1 \xrightarrow{\mu/memoDisplayed} q_2$
10	$remove$	$memoRemoved$	$q_2 \xrightarrow{remove/memoRemoved} q_1$	$q_2 \xrightarrow{remove/memoRemoved} q_1$

Table 1. Un extrait de trace suite à la simulation de l'utilisateur de Memo

Un résultat possible de l'application de cet algorithme sur la machine $\mathrm{M}_{\text{use Memo system*}}$ avec le système Memo est présenté dans la table 1. L'état de la machine est initialisé à q_0 (4. $q \leftarrow q_0$).

Dans le pas no 1 du test, l'entrée est aléatoirement choisie parmi les entrées possibles à partir de q_0(7. $i \leftarrow random(beh_T(q_0)))$: $beh_T(q_0) = \{move, \mu\}$. L'entrée $move$ est choisie et envoyée, puis le générateur attend pour que l'application réagisse (9. $wait(c)$). L'application n'envoie pas de sortie (10. $read(oset)$) :

on obtient $oset = \emptyset$. Ensuite, l'ensemble des transitions possibles est calculé (12. $pFollowingq \leftarrow pTrans_T(q_0, move, \emptyset)$) : $pFollowingq = \{q_0 \overset{move/-}{\longrightarrow} q_1\}$. La seule transition de cet ensemble est choisie et on arrive dans l'état q_1(étapes 13, 14 de l'algorithme).

Dans le cas du pas no 2, l'entrée vide μ est choisie : dans ce cas aucune entrée n'est envoyée à l'application (étape 8). L'ensemble de sorties lues est vide, et donc l'ensemble des transitions possibles de la machine $M_{\text{use Memo system}*}$ est également vide (étape 12 de l'algo). En conséquence, on reste dans le même état (étapes 13, 14 de l'algorithme).

Dans le pas no 3, l'entrée $move$ est envoyée. L'application réagit en envoyant $memoDisplayed$. L'ensemble des transitions possibles est calculé (étape 12 de l'algorithme) : $pFollowingq = \{q_1 \overset{move/-}{\longrightarrow} q_1\}$. La seule transition de cet ensemble est choisie pour arriver dans l'état $q1$ (étapes 13, 14 de l'algo).

Le pas no 4 se déroule de manière similaire et mène à l'état q_2.

Dans le cas du pas no 5, $beh_T(q_2) = \{get, remove\}$, l'entrée est choisie aléatoirement dans cet ensemble :$i = get$ (l'étape 7 de l'algorithme). L'application envoie deux sorties $memoTaken$, $memoCarried$. Dans la machine $M_{\text{use Memo system}*}$, il y a une seule transition possible dans ce cas ($pFolloingq = \{q_2 \overset{get/memoTaken}{\longrightarrow} q_1\}$). Cette transition est choisie pour arriver dans l'état q_1.

4 Conclusion et perspectives

Nous avons présenté une méthode de génération de données de test à partir de l'arbre de tâche décrivant le fonctionnement d'une application interactive. Nous avons montré que l'arbre de tâches peut être transformé en une machine à E/S modélisant le comportement de l'utilisateur conformément à une sémantique formelle que nous avons définie. Cette machine peut être simulée d'une manière analogue à celle mise en oeuvre au sein de l'outil Lutess. Nous soulignons certaines hypothèses faites dans ce travail qui méritent une étude plus approfondie et de ce fait en constituent des perspectives immédiates :

- Les tâches interactives sont supposées être modélisées sous la forme de machines d'E/S. Cela suppose un effort de modélisation supplémentaire de la part du concepteur de l'arbre des tâches ; mais cet effort est, à notre sens, le strict minimum requis pour une approche formelle.
- Nous supposons que les arbres de tâches sont "bien formés". Cela signifie qu'ils contiennent uniquement des tâches interactives et des tâches application qui activent d'autres tâches interactives. Une définition plus formelle de cette notion est en étude.

D'autres perspectives de ce travail sont l'enrichissement des arbres de tâches avec des annotations exploitables par des outils de test. Par exemple, l'association de probabilités d'occurrence des tâches peut permettre la mise en oeuvre des techniques de génération basées sur des profils opérationnels.

Références

1. Dittmar A. More precise descriptions of temporal relations within task models. In *DS-VIS*, pages 151–168, 2000.
2. Y. Aït Ameur, M. Baron, and P. Girard. Formal validation of hci user tasks. In *Software Engineering Research and Practice*, pages 732–738, 2003.
3. A. Benveniste, B. Caillaud, and P. Le Guernic. From synchrony to asynchrony. In *CONCUR*, pages 162–177, 1999.
4. J. Bouchet and L. Nigay. Icare : a component-based approach for the design and development of multimodal interfaces. In *CHI Extended Abstracts*, pages 1325–1328, 2004.
5. P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre : A declarative language for programming synchronous systems. In *POPL*, pages 178–188, 1987.
6. L. du Bousquet, F. Ouabdesselam, I. Parissis, J.-L. Richier, and N. Zuanon. Lutess : a testing environment for synchronous software. In *Tool Support for System Specification, Development and Verification*, pages 48–61. Advances in Computer Science. Springer Verlag, June 1998.
7. D. J. Duke and M. D. Harrison. Abstract interaction objects. *Comput. Graph. Forum*, 12(3) :25–36, 1993.
8. G. P. Faconti, N. Zani, and F. Paternò. The input model of standard graphics systems revisited by formal specification. *Comput. Graph. Forum*, 11(3) :237–251, 1992.
9. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9) :1305–1320, 1991.
10. Andreas Lecerof and Fabio Paternò. Automatic support for usability evaluation. *IEEE Trans. Software Eng.*, 24(10) :863–888, 1998.
11. L. Madani, C. Oriat, I. Parissis, J. Bouchet, and L. Nigay. Synchronous testing of multimodal systems : An operational profile-based approach. In *ISSRE*, pages 325–334, 2005.
12. G. Mori, F. Paternò, and C. Santoro. Ctte : Support for developing and analyzing task models for interactive system design. *IEEE Trans. Software Eng.*, 28(8) :797–813, 2002.
13. D. Navarre, Ph. A. Palanque, R. Bastide, A.Schyn, M. Winckler, L. Porcher Nedel, and C. M. Dal Sasso Freitas. A formal description of multimodal interaction techniques for immersive virtual reality applications. In *INTERACT*, pages 170–183, 2005.
14. Ph. A. Palanque and R. Bastide. Verification of an interactive software by analysis of its formal specification. In *INTERACT*, pages 191–196, 1995.
15. I. Parissis and J. Vassy. Strategies for automated specification-based testing of synchronous software. In *16th International Conference on Automated Software Engineering*, San Diego, USA, November 2001. IEEE.
16. Conte E. Mancini C. Mori G. Paternò F., Ballardin G. Specification of notation of task modelling methodology. Technical report, CNUCE-C.N.R., Pisa, Italy, January 2000.

Test de logiciels synchrones : apports de la programmation par contraintes

Besnik Seljimi, Ioannis Parissis

Laboratoire d'Informatique de Grenoble
BP53 38041 Grenoble Cedex 9
{Besnik.Seljimi, Ioannis.Parissis}@imag.fr

Résumé Nous présentons une extension de l'environnement Lutess basée sur la programmation par contraintes, visant à générer des données de test pour les logiciels synchrones avec des entrées-sorties numériques. La Programmation Logique avec Contraintes (PLC) offre un environnement qui permet d'envisager cette extension de manière simple et efficace sans avoir à développer une solution ad hoc. De plus, il est intéressant d'étudier les moyens d'exploiter cette nouvelle représentation du problème de la génération de test et les avantages que la PLC peut y apporter, en particulier pour l'adaptation à ce nouveau contexte des nombreuses stratégies de test que Lutess propose. Plusieurs méthodes de génération ont été étudiées dans ce cadre : la génération aléatoire, le guidage par les probabilités et par les propriétés de sûreté. Un autre objectif de notre travail est d'étendre Lutess de façon à prendre en compte des hypothèses ou des connaissances sur le fonctionnement du programme et de rendre le guidage plus efficace.

1 Introduction

Les travaux sur le test des logiciels synchrones [6] ayant abouti au développement de l'environnement Lutess [1] avaient pour principale motivation la volonté d'apporter un moyen de vérification complémentaire à la preuve formelle par « model-checking » [3]. Cette dernière se limitant essentiellement à la vérification de propriétés définies sur des signaux booléens du programme, cette même limitation s'est retrouvée dans les outils de test développés. La possibilité de prendre en compte des spécifications incluant des relations entre valeurs et expressions numériques a toujours été considérée comme une extension indispensable, les logiciels synchrones ne se limitant pas à des entrées sorties booléennes. Cette extension est abordée ici en s'appuyant sur la Programmation Logique avec Contraintes (PLC) [4]. Ce choix est motivé par la richesse des environnements de résolution de contraintes qui permet d'envisager cette extension de manière simple et efficace sans avoir à développer une solution ad hoc.

L'utilisation de la PLC à des fins de génération de tests a souvent été suggérée en particulier dans le cas de systèmes réactifs [8]. Notons en particulier Gatel [5], un outil de génération de tests pour programmes Lustre. Gatel traduit

le programme sous test et ses spécifications en une représentation en contraintes équivalente. La résolution de ces dernières vise la satisfaction d'un objectif défini par l'utilisateur. Gatel a été utilisé pour tester des programmes réalistes. Ses principales différences avec Lutess résident dans le traitement des spécifications et du programme (Gatel les combine en une seule représentation tandis que Lutess les traite comme des composants distincts) et l'absence dans Gatel de stratégie de génération élaborée (les objectifs de test sont en effet définis par l'utilisateur).

L'approche proposée pour l'extension de Lutess diffère également de celle adoptée dans Lurette [9] qui adopte un environnement spécifique de résolution de contraintes basé sur les graphes de décision binaire et des polyèdres. Par ailleurs, on ne retrouve pas dans ce dernier les stratégies de génération automatique de Lutess mais plutôt des langages de spécification de scénarios de test. Dans le cadre de cette extension nous devons traiter deux problèmes distincts. D'une part, l'utilisation de contraintes de nature numérique pose en des termes nouveaux le problème de la sélection de données d'entrée pertinentes. En particulier, la sélection de données de manière équitable, assurée dans le cadre de la version booléenne de Lutess par une étude exhaustive de l'arbre décrivant l'ensemble des solutions, ne peut être assurée de la même manière. De plus, « l'équité » est un concept qui, dans le cadre numérique, appelle à une autre définition. D'autre part, la traduction en contraintes des spécifications Lustre enrichies utilisées par les différentes stratégies de Lutess nécessite une définition précise de la transformation à effectuer et la démonstration du principe de génération de données dans ce nouveau modèle de calcul.

Le paragraphe 2 est une présentation détaillée du contexte de ce travail : le test des programmes synchrones. En particulier, les modèles formels manipulés par Lutess y sont rappelés, leur compréhension étant nécessaire à l'explication des extensions proposées. Ces dernières sont exposées dans les paragraphes 3 et 4.

2 Contexte : test de programmes synchrones

2.1 Programmation synchrone & langage Lustre

Un programme est dit *synchrone*, s'il vérifie l'hypothèse de synchronisme : sa réaction aux entrées est théoriquement instantanée. En pratique, on considère qu'un logiciel a un comportement synchrone s'il réagit à son environnement avant toute évolution de ce dernier. Ainsi, si à l'instant t le logiciel reçoit les entrées i_t depuis son environnement externe, il émet les sorties o_t à ce même instant (fig. 1). Aucun changement de i_t ne peut survenir avant l'instant suivant $t+1$ (où une nouvelle entrée i_{t+1} sera disponible). Cette propriété permet

de s'abstraire des problèmes de temporalité, très importants quand il s'agit de réaliser des *systèmes réactifs*. Les logiciels synchrones sont déterministes : leur exécution peut être vue comme une séquence d'entrées/sorties où à chaque instant les sorties sont complètement déterminées par les entrées à l'instant t, ainsi que de l'état (abstraction des valeurs passées des entrées).

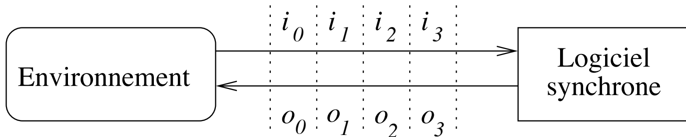


FIG. 1. Programme synchrone

Lustre est un langage synchrone flot de données où chaque variable définit un flot. Les expressions Lustre sont construites à l'aide de variables, de constantes, des opérateurs arithmétiques et logiques habituels, ainsi que de deux opérateurs spécifiques : l'opérateur *précédent* (noté *pre*) introduit un décalage du flot d'une unité de temps, et l'opérateur *suivi par* (noté −>) permet l'initialisation des flots. Ainsi, si *E* et *F* sont deux expressions Lustre, l'expression *E−>pre(F)* définit le flot : $e_0, f_0, f_1, f_2, ...$ Les programmes Lustre sont structurés en *noeuds* (fig. 2). Un *noeud* comporte un ensemble d'équations qui définissent chacun des paramètres de sortie en fonction des paramètres d'entrée à l'aide de variables locales. Une fois défini, chaque noeud peut être ensuite utilisé dans d'autres noeuds comme tout autre opérateur.

```
node Programme(<entrées>) returns (<sorties>)
var
   <variables locales>
let
   <sorties>=f(<entrées>,<variables locales>);
tel
```

FIG. 2. Structure syntaxique d'un programme Lustre

Considérons un exemple de programme réactif contrôlant un climatiseur dont la fonction est de commander les différents dispositifs du système de climatisation (émission d'air chaud ou froid en fonction de la température). Le programme prend en entrée un signal *Bouton*, actif si l'utilisateur a appuyé sur le bouton marche/arrêt, et 3 signaux (*Tinf*, *Tok* et *Tsup*) indiquant respectivement

que la température est inférieure, égale ou supérieure à celle fixée par l'utilisateur. Le vecteur de sortie *o* est constitué des signaux *En_marche* indiquant si le climatiseur est en marche, *Froid* et *Chaud* indiquant si le climatiseur fonctionne en mode chauffage ou refroidissement et *Inactif* si aucun des deux modes n'est actif (quand la température visée est atteinte). Dans la figure 3 on donne une réalisation possible de ce contrôleur dans le langage Lustre.

```
node Clim(Bouton, Tinf, Tok, Tsup : bool)
   returns (En_marche, Froid, Inactif, Chaud : bool)
let
   En_marche = Bouton -> pre En_marche and not(Bouton)
      or not(pre En_marche) and Bouton;
   Froid = En_marche and Tsup;
   Inactif = En_marche and Tok;
   Chaud = En_marche and Tinf;
tel
```

FIG. 3. Contrôleur du climatiseur en Lustre

Le tableau 1 donne la trace d'une exécution illustrant le fonctionnement du climatiseur. On peut remarquer que pour les mêmes entrées, le logiciel peut produire des sorties différentes (instants t_1 et t_7). Ceci est dû à la mémoire interne du logiciel. En effet, le climatiseur reste en marche entre deux appuis successifs sur le bouton, mémorisant ainsi le fait qu'on a déjà appuyé sur le bouton dans le passé. Notons que les signaux *Froid*, *Inactif* et *Chaud* ne peuvent pas être vrais si le climatiseur n'est pas en marche.

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
Bouton	0	**1**	0	0	0	0	0	**1**	0
Tinf	0	**0**	1	1	0	0	0	**0**	1
Tok	0	**1**	0	0	1	0	0	**1**	0
Tsup	1	**0**	0	0	0	1	1	**0**	0
En_marche	0	**1**	1	1	1	1	1	**0**	0
Froid	0	**0**	0	0	0	1	1	**0**	0
Inactif	0	**1**	0	0	1	0	0	**0**	0
Chaud	0	**0**	1	1	0	0	0	**0**	0

TAB. 1. Chronogramme représentant une exécution possible du climatiseur

L'opérateur *pre* constitue une mémoire de la valeur de l'expression associée et l'ensemble des expressions du type *pre E* définissent l'état du programme Lustre. Cet état représente en effet une abstraction des valeurs passées des va-

riables d'entrée et de sortie du logiciel. Les sorties o_t d'un logiciel à un instant t sont alors complètement définies en fonction de l'état du programme et des entrées i_t à ce même instant.

2.2 L'environnement Lutess

Principes Lutess [1] est un outil de test fonctionnel des logiciels réactifs synchrones (fig. 4). Il intègre la génération des jeux de tests et leur exécution, afin de produire automatiquement et dynamiquement des données en respectant les contraintes d'environnement du programme. Le programme sous test est donné sous sa forme exécutable et est supposé avoir un comportement synchrone. L'oracle examine la séquence d'entrées-sorties du logiciel et donne un verdict sur leur concordance avec les propriétés spécifiées par le testeur. La figure 5 est un exemple de spécification d'un oracle en Lustre. Il vérifie, sur l'exemple du climatiseur déjà vu, si le climatiseur émet de l'air chaud quand il fait froid.

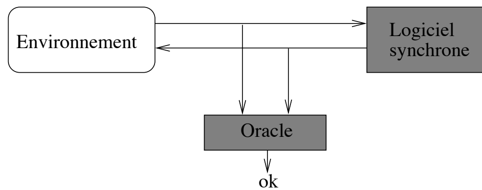


FIG. 4. Principe de fonctionnement de Lutess

```
node Oracle(Bouton,Tinf,Tok,Tsup,
   En_marche,Froid,Inactif,Chaud : bool) returns (ok : bool)
let
   ok = implies(En_marche and Tinf, Chaud);
tel
```

FIG. 5. Réalisation de l'oracle en Lustre

La génération de tests nécessite la fourniture d'une spécification (*noeud de test*), excluant au moyen de propriétés invariantes les comportements irréalistes de l'environnement (opérateur *environment*) qui ne doivent pas être pris en compte pendant le test. Le noeud de test est repéré par le mot clé *testnode*, et correspond à une extension de la grammaire de Lustre (fig. 6). Il prend en entrée (resp. sortie) les sorties (resp. entrées) du programme sous test. L'exemple

de la figure 7 définit une spécification de l'environnement du climatiseur stipulant que la température ne peut pas passer de *Tinf* (froid) à *Tsup* (chaud) sans passer par la température fixée par l'utilisateur (*Tok*).

```
testnode Environnement(<sorties du logiciel>)
   returns (<entrées du logiciel>)
var
   <variables locales>
let
   environment(C1,...,Cn);
   <définition des variables locales>
tel
```

FIG. 6. Structure syntaxique d'un noeud de test

```
testnode Env_clim(En_marche, Froid, Inactif, Chaud : bool)
   returns (Bouton, Tinf, Tok, Tsup : bool)
let
   environment( true -> (not(Tsup) or not(pre Tinf)) );
tel
```

FIG. 7. Exemple de spécification de l'environnement du climatiseur en Lustre

Simulation de l'environnement Un noeud de test est transformé en un simulateur d'environnement qui est une une machine d'états finis M_{env} = $(S_{env}$, $s_{init_{env}}$, V_i, V_o, env, $trans_{env})$ où :

- S_{env} est l'ensemble des états de l'environnement (ensemble de valeurs des variables d'état sve) ;
- $s_{init_{env}}$ est l'état initial de l'environnement ;
- i et o étant les vecteurs des paramètres d'entrée et de sortie du logiciel sous test, V_i et V_o sont leurs ensembles de valeurs associés ;
- La fonction $env : S_{env} \times V_i \rightarrow \{false, true\}$ définit à tout moment t l'ensemble des valeurs du vecteur d'entrée i conformes à la spécification de l'environnement (i.e. ensemble de valeurs rendant vraie la fonction env dans l'état où se trouve l'environnement à l'instant t) ;
- sel_{env} est une méthode qui dans l'état $s \in S_{env}$ sélectionne un vecteur d'entrées $i \in V_i$ qui respecte l'environnement : $env(s, i) = true$;
- La fonction de transition $trans_{env} : S_{env} \times V_i \times V_o \rightarrow S_{env}$ calcule l'état de l'environnement après chaque échange d'entrées-sorties avec le logiciel synchrone.

Ainsi, au noeud de la figure 7 correspond la machine d'états finis M_{env} :

- Paramètres d'entrée : $i = \{Bouton, Tinf, Tok, Tsup\}$
- Paramètres de sortie : $o = \{En_marche, Froid, Inactif, Chaud\}$
- État : $s_{env} = \{sv_0, sv_1\}$, état initial $s_{init_{env}} = \{(sv_0, sv_1)|sv_0 = true\}$
- Fonction de transition $trans_{env}$: $sv_0 = false, sv_1 = Tinf$
- Fonction de l'environnement env : $sv_0 \vee not(sv_0) \wedge (not(Tsup) \vee not(sv_1))$

Le coeur du générateur de tests consiste en une animation des spécifications de l'environnement (c.à.d un parcours de la machine M_{env}). La sélection d'une donnée de test consiste à chaque instant t à (1) déterminer l'ensemble de valuations possibles V_i du vecteur d'entrée i_t qui dans l'état courant sve_t respecte les contraintes d'environnement : $env(sve_t, i_t) = true$, (2) choisir un élément i_t dans l'ensemble V_i et (3) après avoir envoyé les entrées i_t au logiciel et récupéré les sorties o_t, calculer l'état suivant de l'environnement (réalisation de la fonction $trans_{env}$). env étant une fonction booléenne, elle peut être représentée efficacement par un Graphe de Décision Binaire (*BDD - Binary Decision Diagram*). C'est le choix de représentation qui a été fait dans la première version de Lutess et ses évolutions ultérieures, qui ne traitent que les programmes à entrées et sorties booléennes. Cette représentation facilite le choix des valeurs du vecteur d'entrée i. En effet, à chaque pas de génération, les variables d'état sont complètement connues et un simple parcours permet d'accéder au sous-graphe représentant les valeurs que les entrées peuvent prendre dans l'état courant. De plus, en étiquetant chaque arc du graphe avec le nombre des vecteurs d'entrée valides qui lui sont associés, on garantit l'équiprobabilité de tirage pour chacun des vecteurs d'entrée.

Guidage par des probabilités Le guidage par des probabilités conditionnelles permet de rendre l'opération de test proche d'un profil d'usage [2]. A titre d'illustration, considérons de nouveau l'exemple du climatiseur : pour tester effectivement la propriété $(En_marche \wedge T < Tuser) \Rightarrow Chaud$, il peut être intéressant de favoriser une situation où $En_marche = true$ en considérant que la probabilité d'appuyer sur *Bouton* est faible (resp. élevée) si $En_marche = true$ (resp. $En_marche = false$). Dans Lutess, l'utilisateur peut affecter à une variable d'entrée une probabilité d'occurrence qui ne sera effective que si une condition, également spécifiée par l'utilisateur, se trouve vérifiée.

Génération guidée par les propriétés de sûreté Le principe du test guidé par les propriétés de sûreté [7] est de choisir un vecteur d'entrées tel que la valeur de vérité de la propriété de sûreté dépende des sorties calculées par le programme. Par exemple, si i et une entrée et o une sortie d'un programme

devant satisfaire la propriété $i \Rightarrow o$, l'entrée $i = vrai$ doit être générée (sinon, la propriété est vraie quelle que soit la valeur de o). Le guidage par les propriétés de sûreté favorise la génération de l'entrée $i = vrai$, à condition que les contraintes d'environnement le permettent. De même, pour la propriété $true \rightarrow \mathbf{pre}\ i \Rightarrow o$ il faut engendrer l'entrée $i = vrai$ à l'instant courant pour que la propriété soit violable à l'instant suivant. D'une manière générale, ce type de guidage consiste à engendrer à un instant t des entrées qui peuvent mener le logiciel dans une situation où la propriété peut être violée à un instant $t + k$, k étant le nombre maximum d'instants à considérer.

Les propriétés de sûreté qui vont guider la sélection des vecteurs d'entrées sont décrites dans le noeud de test de manière similaire aux contraintes d'environnement et identifiées par l'opérateur *safety*. En termes de sémantique, il est nécessaire d'étendre le simulateur d'environnement défini plus haut de telle sorte que les propriétés de sûreté y soient intégrées. Nous parlerons alors de *simulateur guidé par les propriétés de sûreté*. Les propriétés de sûreté peuvent être assimilées à un programme synchrone et de ce fait être représentées par un automate d'états finis $M_{prop} = (S_{prop}, s_{init_{prop}}, V_i, V_o, prop, trans_{prop})$ où la fonction $prop : S_{prop} \times V_i \times V_o \rightarrow \{false, true\}$ définit à tout moment t la valeur de vérité de la propriété. Étant donnée la machine associée à l'environnement $M_{env} = (S_{env}, s_{init_{env}}, V_i, V_o, env, trans_{env})$, le simulateur d'environnement guidé par les propriétés de sûreté peut être formellement défini par la machine $M_{Ps} = (S_{Ps}, s_{init_{Ps}}, V_i, V_o, pert, trans_{Ps})$ où :

- $S_{Ps} = S_{env} \times S_{prop}$ est l'ensemble des états ;
- $s_{init_{Ps}} = (s_{init_{env}}, s_{init_{prop}})$ est l'état initial ;
- V_i et V_o sont les ensembles de valeurs de vecteurs d'entrées i et de sorties o ;
- $trans_{Ps} : S_{Ps} \times V_i \times V_o \rightarrow S_{Ps}$ est la nouvelle fonction de transition. Elle est définie par :
  $trans_{Ps}((s_{env}, s_{prop}), i, o) = (trans_{env}(s_{env}, i, o), trans_{prop}(s_{prop}, i, o))$.
- La fonction $pert : S_{Ps} \times V_i \rightarrow \{false, true\}$ définit les vecteurs d'entrées dits *pertinents* : $pert(s_{Ps}, i) = \exists o \in V_o,\ (s_{Ps} = (s_{env}, s_{prop})) \wedge env(s_{env}, i) \wedge (\neg prop(s_{prop}, i, o) \vee (\exists i' \in V_i, pert(trans_{Ps}(s_{Ps}, i, o), i')))$;

La différence principale avec le simulateur d'environnement réside dans le calcul de l'ensemble des vecteurs *pertinents*. Ce sont les vecteurs qui respectent l'environnement et sont susceptibles de violer la propriété de sûreté sur un chemin de longueur $1..k$. Dans [10], trois différentes stratégies de calcul sont définies. La *stratégie d'union* considère tous les chemins de longueur $1..k$ qui mènent le logiciel à une situation susceptible de violer la propriété de sûreté (autrement dit *état suspect*). La *stratégie d'intersection* ne considère que les vecteurs d'entrée qui font partie de tous les chemins de longueur $1..k$ qui mènent

à un état suspect. La *stratégie paresseuse* choisit le vecteur qui mène le plus rapidement à un état suspect.

3 Extension de Lutess au moyen de la PLC

Considérons une nouvelle version du climatiseur ayant comme entrées la valeur de la température ambiante *Tamb* et la valeur de la température voulue par l'utilisateur *Tutil* (fig. 8), toutes deux de type entier. Le programme va alors calculer la température de l'air sortant de la soufflerie *Tsort*.

```
node Clim(Bouton : bool; Tamb, Tutil : int)
returns (En_marche : bool; Tsort : int);
```

FIG. 8. Interface du climatiseur numérique en Lustre

Pour décrire l'environnement de ce programme on a besoin d'utiliser des équations comportant des variables entières. Par exemple, on peut spécifier que la température ambiante ne peut varier que d'un degré entre deux instants successifs, ou alors que les températures sont comprises dans l'intervalle [-10,40]. La prise en compte par les techniques de génération de Lutess d'une telle spécification nécessite une représentation et un mode de calcul différents. Ce cadre est offert par la programmation par contraintes.

3.1 Principes de programmation par contraintes

Formellement, un problème de satisfaction de contraintes peut être représenté par un triplet (X, D, C) où :

- $X = \{X_1, X_2, ..., X_n\}$ est un ensemble de variables
- $D = \{D_1, D_2, ..., D_n\}$ est un ensemble de domaines non vides. Chaque domaine D_i représente l'ensemble de valeurs que peut prendre la variable X_i.
- $C = \{C_1, C_2, ..., C_m\}$ est un ensemble de contraintes. Chaque contrainte C_i définit une relation sur un sous-ensemble de variables qui spécifie les combinaisons de valeurs permises pour ce sous-ensemble.

L'instanciation d'une partie ou de l'ensemble des variables $\{X_i = v_i, X_j = v_j, ...\}$ définit un état du problème. Une instanciation qui satisfait l'ensemble des contraintes est dite consistante. Lorsque l'ensemble des variables est instanciée sans violer les contraintes du système, cette instanciation représente une solution du système de contraintes. Lors de la résolution d'un système de contraintes, des algorithmes de propagation sont mis en oeuvre. Le but de ces algorithmes est de

déduire les valeurs qui ne peuvent pas constituer une solution (cette opération est appelée *filtrage*). A titre d'illustration, considérons deux entiers $x, y \in (1..3)$ et la contrainte $x < y$. Dans ce cas $y = 1$ ne peut pas faire partie d'une solution et la valeur 1 est enlevée du domaine de y. Le filtrage permet dans ce cas de réduire les domaines des variables $x \in (1..2)$ et $y \in (2..3)$. Le domaine obtenu après le filtrage est appelée *domaine réduit*. Cependant le filtrage ne permet pas toujours de déterminer complètement les solutions du système. L'*énumération* consiste à faire des choix arbitraires puis recommencer le filtrage. Si un choix mène à une contradiction, il est remplacé par un autre et ainsi de suite.

3.2 Simulation aléatoire avec des expressions numériques

Pour générer des données de test qui respectent la spécification de l'environnement, à chaque instant t, il faut déterminer l'ensemble sve_t des valeurs possibles pour les variables d'entrée i_t du programme, telles que $env(sve_t, i_t) = true$ et ensuite choisir un élément dans cet ensemble. On se propose de modéliser cette fonction comme un problème de satisfaction de contraintes. Il s'agit de spécifier un système de contraintes (X, D, C) qui associe aux variables $X = sve_t \cup i_t$ un ensemble de contraintes C, telle que toutes les valeurs de i_t qui satisfont l'ensemble des contraintes représentent un des vecteurs d'entrée permis par l'environnement.

La figure 9 présente une nouvelle spécification de l'environnement du climatiseur qui impose que les températures *Tamb* et *Tutil* soient dans des intervales raisonnables et que la température ne change pas de plus d'un degré entre deux instants successifs.

```
testnode Env_clim(En_marche : bool; Tsort : int)
   returns(Bouton : bool; Tamb, Tutil : int);
var dT : int;
let
   dT = 0 -> Tamb - pre Tamb;
   environment( dT >= -1 and dT <= 1
      and Tamb > -20 and Tamb < 60
      and Tutil > 10 and Tutil < 40 );
tel
```

FIG. 9. Description de l'environnement du climatiseur

La difference de entre la température ambiante *Tamb* et la température à l'instant précédent *pre Tamb* introduit une variable d'état mémorisant la valeur de *Tamb* entre deux instants. L'état de la machine d'états finis correspondant à

cet exemple est constitué de deux variables : sv_0 pour distinguer l'état initial et sv_1 pour mémoriser la valeur de *Tamb*.

- Variables d'état : $sve = \{sv_0, sv_1\}$
- État initial : $sv_0 = true$
- Fonction de transition : $sv'_0 = false \wedge sv'_1 = Tamb$
- Contraintes d'environnement :
  - $(sv_0 \wedge dT = 0) \vee (\neg sv_0 \wedge dT = Tamb - sv_1)$
  - $dT \geq -1 \wedge dT \leq 1 \wedge T_{amb} \geq -20 \wedge T_{amb} \leq 60 \wedge T_{util} \geq 10 \wedge T_{util} \leq 60$

Techniquement, la traduction en un système de contraintes consiste à générer automatiquement 3 prédicats[1]. Le prédicat *environnement/2* exprime les contraintes d'environnement sur les variables d'entrée en fonction de l'état courant.

```
environnement(Etat, Entrees) :-
    Etat = [SV0, SV1],
    Entrees = [Bouton, Tamb, Tutil],
    bool(Bouton), int(Tamb), int(Tutil),

    and(SV0, DT #= 0) or and(neg(SV0), DT #= Tamb - SV1),
    DT #>= -1, DT #=< 1, Tamb #>= -20, Tamb #=< 60, Tutil
#>= 10, Tutil #=< 40.
```

La fonction de transition est modélisée comme un prédicat *transition/4* qui exprime les contraintes liant le nouvel état à l'ancien en fonction des variables d'entrée et de sortie. Dans cet exemple il y a deux contraintes définissant les nouvelles valeurs des deux variables d'état.

```
transition(Etat, Entrees, Sorties, EtatSuiv) :-
    Etat = [SV0, SV1],
    Entrees = [Tamb, Tutil, Bouton],
    Sortie = [Froid,Inactif,Chaud,En_marche],
    EtatSuiv = [SV0p, SV1p],
    SV0p #= 0,
    SV1p #= Tamb.
```

Le prédicat *etat_initial/1* définit l'état initial de l'environnement représenté par la contrainte $sv_0 = 1$.

```
etat_initial(Etat) :- Etat = [1, 0].
```

A l'aide de ces trois prédicats, on peut produire des données d'entrée conformes aux contraintes d'environnement. Le prédicat *chemin/2* donne une suite de valeurs d'entrée correspondant à un parcours sur un chemin dans la machine d'états finis de l'environnement. Une séquence de test (prédicat *test/1*) est un chemin partant de l'état initial de l'environnement :

[1] La syntaxe utilisée est celle de l'environnement ECLiPSe Prolog.

```
chemin(Longueur, Etat) :- Longueur > 0,
    environnement(Etat, Entrees),
    selection(Entrees),
    envoyer(Entrees),
    récuperer(Sorties),
    transition(Etat, Entrees, Sorties, EtatSuiv),
    chemin(Longueur-1, EtatSuiv).
test(Longueur) :- etat_initial(Etat), chemin(Longueur,
Etat).
```

Les deux prédicats *envoyer(Entrees)* et *recuperer(Sorties)* assurent la communication avec le programme sous test (exécutable). Le prédicat *selection(Entrees)* permet de choisir une entrée parmi celles qui sont permises par l'environnement, c.à.d. une solution du système de contraintes du prédicat *environnement*.

La simulation aléatoire consiste à affecter aléatoirement aux variables des valeurs dans leur domaine réduit, jusqu'à obtenir une affectation complète qui vérifie l'ensemble des contraintes. Notons que, contrairement à la version booléenne de Lutess, l'énumération aléatoire ne garantit pas l'équiprobabilité entre les différentes solutions possibles (l'ensemble des vecteurs d'entrée conformes à l'environnement), même si aucune des solutions n'est exclue. Cependant, on peut s'interroger sur la pertinence d'un tirage équiprobable quand les domaines des variables sont de taille très différente. Considérons, par exemple, la contrainte $b \in (0..1) \wedge x \in (0..2^{32}-1) \wedge (b \Rightarrow x = 0)$. Dans une sélection équiprobable entre toutes les solutions, la probabilité de tirer une solution telle que $b = 1, x = 0$ est quasi nulle ($\frac{1}{2^{32}+1}$), alors que cette solution a manifestement un intérêt particulier. Par ailleurs, une sélection aléatoire dans le domaine de chaque variable peut donner une distribution différente selon l'ordre des variables selon lequel cette sélection est effectuée. Ainsi, si on choisit d'abord la variable x, on obtient une probabilité de $\frac{1}{2^{33}}$ d'avoir la solution $b = 1, x = 0$ alors que si on choisit d'abord la variable b, la probabilité d'avoir cette même solution est de $\frac{1}{2}$. Instancier d'abord les variables ayant le plus petit domaine réduit, ce qui, en particulier revient à considérer les booléens avant les variables entières, pourrait permettre d'avoir une distribution plus judicieuse du point de vue du test.

Illustration A titre d'illustration de la simulation aléatoire de l'environnement, le tableau 2 montre des données de test générées par l'outil Lutess en utilisant la description d'environnement de la fig. 9. On peut remarquer que la témperature *Tamb* change au plus d'un degré entre deux instants succéssifs.

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}
Bouton	0	1	1	1	1	0	1	1	1	1	0	1	0	0
Tamb	**-12**	**-13**	**-14**	**-14**	**-14**	**-14**	**-15**	**-15**	**-14**	**-15**	**-15**	**-16**	**-16**	**-15**
Tutil	16	17	30	24	25	33	21	24	34	38	11	21	28	29
En_marche	0	1	0	1	0	0	1	0	1	0	0	1	1	1
Tsort	25	27	44	36	38	48	33	37	50	55	19	33	42	43

TAB. 2. Chronogramme de valeurs générées pour l'environnement numérique

3.3 Génération guidée par les probabilités conditionnelles

L'utilisation de la PLC permet d'étendre le guidage par les probabilités aux contraintes numériques en permettant l'affectation de probabilités non seulement aux variables d'entrée mais à toute expression définie dans le noeud de test. Ainsi, l'utilisateur peut spécifier des probabilités conditionnelles sur des expressions, en utilisant la notation *prob(C,E,P)* où :

- *C* est une condition portant sur les valeurs passées des paramètres d'entrée/sortie.
- *E* est une expression Lustre de type booléen.
- *P* est une constante réelle dans l'intervalle $(0.0..1.0)$.

L'expression *prob(C,E,P)* signifie : si *C=true* alors la probabilité que *E=true* est *P*. Dans l'exemple du climatiseur, pour favoriser la situation *En_marche=true*, on peut écrire :

```
testnode Env_clim(En_marche :bool; Tout : int)
   returns(Bouton : bool; Tamb, Tutil : int;);
var dT : int;
let
   dT = 0 -> Tamb - pre T;
   environment( dT >= -1 and dT <= 1
      and Tamb > -20 and Tamb < 60
      and Tutil > 10 and Tutil < 40 );
   prob(pre En_marche, Bouton, 0.1);
   prob(not(pre En_marche), Bouton, 0.9);
tel
```

FIG. 10. Description de l'environnement du climatiseur avec des probabilités

Pour prendre en compte les probabilités, on traduit l'ensemble des conditions et des expressions en contraintes. Si on a une liste de probabilités prob(C_1, E_1, P_1), . . . , prob(C_n, E_n, P_n), on tire aléatoirement n valeurs réelles $x_1, \ldots, x_n$ dans l'intervalle $(0.0..1.0)$. Pour chaque i on pose la contrainte $C_i \Rightarrow (x_i \leq P_i \Leftrightarrow E_i)$. Il est possible que l'ensemble des contraintes ainsi exprimées (y compris

les contraintes d'environnement) ne puissent pas être satisfaites. Dans ce cas, aucune entrée n'est possible et on considère qu'il y a une incohérence dans la spécification des probabilités. Face à ce problème, nous avons deux choix :

– Considérer que l'utilisateur exprime ces probabilités comme un moyen simple de guider la génération, sans se soucier de leur distribution effective et de leur cohérence. Dans ce cas on peut envisager la possibilité de continuer la génération en essayant de satisfaire un maximum de contraintes à chaque pas, mais pas forcément toutes.
– Considérer, au contraire, que la génération doit s'arrêter et que l'utilisateur doit rectifier la spécification de probabilités.

Les probabilités exprimées dans l'exemple de la figure 10 sont automatiquement traduites comme suit :

```
probabilites([prob(SV2, Bouton, 0.1),
    prob(neg(SV2), Bouton, 0.9)]).
```

où SV2 est une variable d'état introduite à cause de l'expression *pre En_marche*. Le principe décrit plus haut est alors implémenté par l'algorithme suivant :

```
probabilites([]).
probabilites( [ prob(C,E,P) | S ] ) :-
    random(X, 0.0, 1.0), B #= X$=<P, proba(C,E,B),
    probabilites(S).
proba(C,E,B) :- or( neg(C) , B #= E ).
```

Le prédicat *proba/3* pose la contrainte $C \Rightarrow (B \Leftrightarrow E)$ où B est la valeur de vérité de la comparaison entre la valeur aléatoire X est la probabilité P. Si la condition est vérifiée alors $B \Leftrightarrow E$, ce qui signifie qu'on pose E si $X \leq P$ et $\neg E$ si $X > P$. Cette implémentation impose la vérification de l'ensemble des probabilités exprimées et en cas d'incohérence la génération est arrêtée. En revanche, si on ne s'intéresse pas à la cohérence, on peut continuer le test même si on ne peut pas satisfaire l'ensemble des probabilités. Pour ce faire, un moyen simple est de créer un point de choix dans l'arbre de recherche avec deux alternatives : $C \Rightarrow (B \Leftrightarrow E)$ et $C \Rightarrow (\neg B \Leftrightarrow E)$. Notons qu'avec cette solution, une association de probabilité est plus importante que toutes celles qui suivent dans le noeud de test.

Illustration Le tableau 3 ci dessous, montre sur une séquence générée pour la spécification de la figure 10. On peut remarquer la valeur générée de *Bouton* est telle à favoriser de longues séquences avec *En_marche*=*true*.

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}
Bouton	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**0**
Tamb	17	16	15	14	13	13	13	14	15	16	17	18	19	18
Tutil	16	13	34	11	25	14	38	14	27	18	22	29	34	36
En_marche	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**1**
Tsort	16	12	40	10	29	14	46	14	31	18	23	32	39	42

TAB. 3. Chronogramme de valeurs générées en utilisant des probabilités

3.4 Génération guidée par les propriétés de sûreté

Nous partons des mêmes principes que pour la simulation aléatoire pour réaliser le générateur guidé par les propriétés de sûreté (à partir de la machine d'états finis définie dans 2.2). La différence majeure est dans la définition de la fonction *pert* qui définit l'ensemble des vecteurs pertinents :

$$pert(s_{Ps}, i) = \exists o \in V_o \wedge (s_{Ps} = (s_{env}, s_{prop})) \wedge env(s_{env}, i) \wedge$$
$$(\neg prop(s_{prop}, i, o) \vee \exists i' \in V_i \wedge pert(trans_{Ps}(s_{Ps}, i, o), i'))$$

En fait, cette définition est récursive et elle suppose l'existence de sorties o telles que, dans le futur, la propriété puisse être violée. Pour réaliser cette fonction, on pose les contraintes sur la propriété récursivement sur un nombre k de pas prédéfini, en associant pour chaque pas une valeur de vérité de la propriété. Ceci est fait à l'aide du prédicat *propriete/4*, automatiquement généré à partir de la spécification de la propriété de sûreté. A titre d'illustration, la propriété de sûreté $false \rightarrow pre\ i \Rightarrow o$ peut être représentée par :

```
propriete(Etat,Entrees,Sorties,ValVerite) :-
    Etat=[SV0,SV1], Entrees=[I], Sorties=[O],
    ValVerite #= or(not(SV0), SV1 and implies(I,O)).
```

Les valeurs de vérité associées aux contraintes de la propriété de sûreté dans les k pas futurs, nous permettent de réaliser les différentes stratégies (cf. 2.2) pour le calcul des vecteurs d'entrée :

- La *stratégie d'union* peut être réalisée simplement on posant la contrainte qu'au moins une des valeurs de vérité est fausse (la propriété peut être violée dans au moins un des chemins de longueur inférieure à k).
- A l'opposé, la *stratégie d'intersection* est obtenue en imposant que toutes ces valeurs soient fausses (la propriété peut être violée dans tous les chemins de longueur inférieure à k).
- Enfin, la *stratégie paresseuse* consiste à chercher le chemin de plus petite longueur qui peut violer la propriété. Ceci est réalisé en associant aux valeurs de vérité la valeur $faux$, dans l'ordre, à partir du chemin de longueur 1 jusqu'aux chemins de longueur k.

Illustration Le tableau 4 montre une séquence de test générée en ajoutant à la spécification de l'environnement la propriété de sûreté :

```
implies(En_marche and Tamb<Tutil, Tsort>Tutil)
```

On peut remarquer que les valeurs générées respectent la condition *Tamb* < *Tutil*, afin de maximiser les chances de violer cette propriété.

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}
Bouton	1	0	1	1	1	0	1	0	1	0	0	0	1	1
Tamb	**8**	**7**	**6**	**5**	**4**	**4**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**9**
Tutil	**16**	**13**	**34**	**11**	**25**	**14**	**38**	**14**	**27**	**18**	**22**	**29**	**36**	**36**
En_marche	0	0	1	0	1	1	0	0	1	1	1	1	0	1
Tsort	18	15	43	13	32	17	49	17	34	21	26	35	44	45

TAB. 4. Chronogramme de valeurs générées en utilisant le guidage par des propriétés de sûreté

4 Prise en compte d'hypothèses sur le programme

Le guidage par les propriétés de sûreté tel qu'il est défini plus haut présente un inconvénient important : le programme sous test étant considéré comme une boîte noire, plusieurs des vecteurs considérés comme pertinents peuvent, en réalité, ne pas l'être. En effet, la définition de la pertinence s'appuie sur une séquence d'entrées et de sorties qui doivent être produites avant d'atteindre un état suspect. Or, ne disposant pas de description ou de spécification du programme sous test (boîte noire) nous sommes obligés de considérer que ce dernier peut produire beaucoup plus de valeurs de sortie qu'il n'en est réellement capable. L'extension que nous proposons ici vise à introduire dans le processus de génération de tests des hypothèses sur le programme sous test. Ces hypothèses peuvent être issues d'une analyse du programme, dans le cas où le code source ou une spécification sont disponibles, être issues de l'analyse des résultats de tests préalablement effectués (qui ont montré la validité de certaines propriétés) ou bien être le simple fruit de l'expérience du concepteur des tests. Quelle que soit leur origine, ces hypothèses peuvent fournir une spécification partielle du programme sous test et leur exploitation peut améliorer la pertinence des vecteurs de test engendrés. En effet, au moment de la sélection de ces vecteurs, le générateur peut tenir compte de ces hypothèses et exclure ceux qui, bien qu'initialement considérés comme pouvant mener à un état suspect, donnent lieu à une réaction du programme rendant une violation des propriétés impossible.

Formellement, de façon similaire aux machines définies précédemment, on définit une hypothèse par un automate d'états finis M_{hyp} =(S_{hyp}, $s_{init_{hyp}}$, V_i, V_o, hyp, $trans_{hyp}$). Si on considère le simulateur d'environnement guidé par les

propriétés de sûreté M_{Ps} =(S_{Ps}, $s_{init_{Ps}}$, V_i, V_o, $pert$, $trans_{Ps}$) alors, un simulateur d'environnement guidé par les propriétés de sûreté et prenant en compte des hypothèses sur le programme, est défini par l'automate M =(S, s_{init}, V_i, V_o, val, $trans$) où :

- $S = S_{Ps} \times S_{hyp}$ est l'ensemble des états ;
- $s_{init} = (s_{init_{Ps}}, s_{init_{hyp}})$ est l'état initial ;
- V_i et V_o sont les ensembles de valeurs de vecteurs d'entrées i et de sorties o ;
- $trans : S \times V_i \times V_o \rightarrow S$ est la nouvelle fonction de transition. Elle est définie par :
  $trans((s_{Ps}, s_{hyp}), i, o) = (trans_{Ps}(s_{Ps}, i, o), trans_{hyp}(s_{hyp}, i, o))$
- La fonction $val : S_{Ps} \times V_i \rightarrow \{false, true\}$ définit les vecteurs d'entrées valides. Ce sont les vecteurs *pertinents* qui respectent les hypothèses sur le programme
  $val(s, i) = (s = (s_{Ps}, s_{hyp})) \wedge pert(s_{Ps}, i) \wedge hyp(s_{hyp}, i)$;

Illustration Dans l'exemple du climatiseur, on peut considèrer l'hypothèse que l'appui sur le *Bouton* change la valeur de *En_marche* entre deux instants[2]. Dans Lutess, nous pouvons la spécifier à l'aide du mot-clé *hypothesis*. Le tableau 5, montre une séquence générée pour la spécification suivante :

```
hypothesis( true -> Bouton = En_marche<>pre(En_marche) );
safety( implies(En_marche and Tamb<Tutil, Tsort>Tutil) );
```

On peut remarquer que le *Bouton* est appuyé lorsque le climatiseur n'est pas en marche afin de produire une séquence avec *En_marche=true*. La violation de la propriété de sûreté ne depend alors que de la valeur de *Tsort* augmentant ainsi les chances de violer la propriété de sûreté.

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}
Bouton	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Tamb	13	14	13	12	11	12	11	10	10	10	10	10	10	9
Tutil	36	24	33	28	26	32	25	17	22	29	29	23	18	29
En_marche	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
Tsort	43	27	39	33	31	38	29	19	26	35	35	27	20	35

TAB. 5. Illustration du guidage par des propriétés de sûreté avec prise en compte d'hypothèses

[2] L'hypothèse peut être une abstraction du programme sous forme de propriétés vérifiées par celui-ci.

5 Conclusion

La réalisation des différentes techniques de génération de test de Lutess au moyen de la PLC permet l'extension simple et efficace de l'outil à la prise en compte des données numériques. De plus, la puissance de la PLC permet de réaliser simplement des extensions supplémentaires, comme l'association de probabilités d'occurrence à des expressions, plutôt qu'à des simples variables ou bien la prise en compte d'hypothèses sur le programme. Toutefois, la définition formelle des techniques proposées doit être renforcée pour être adaptée à ce nouveau contexte. Ainsi, l'association de probabilités à des expressions peut introduire des incohérences dont la détection n'est pas actuellement réalisée.

La réalisation d'une version de Lutess intégrant la PLC est en cours de finalisation. Si des tests sur des programmes de taille réduite ont permis de valider les techniques énoncées, la validation de l'outil sur des études de cas plus conséquentes (en particulier avec un nombre important de contraintes à gérer) est une des perspectives immédiates de ce travail.

Références

1. L. du Bousquet, F. Ouabdesselam, I. Parissis, J.-L. Richier, and N. Zuanon. Lutess : a testing environment for synchronous software. In *Tool Support for System Specification, Development and Verification*, pages 48–61. Advances in Computer Science. Springer Verlag, June 1998.
2. L. du Bousquet, F. Ouabdesselam, and J.-L. Richier. Expressing and implementing operational profiles for reactive software validation. In *9th International Symposium on Software Reliability Engineering*, Paderborn, Germany, november 1998.
3. N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language lustre. *IEEE Trans. Software Eng.*, 18(9) :785–793, 1992.
4. Lassez J.-L. Jaffar J. Constraint logic programming. In *14th. ACM Symposium on Principles of Programming Languages (POPL'87)*, pages 111–119, 1987.
5. Bruno Marre and Agnès Arnould. Test sequences generation from lustre descriptions : Gatel. In *ASE*, pages 229–, 2000.
6. Ioannis Parissis and Farid Ouabdesselam. Specification-based testing of synchronous software. In *SIGSOFT FSE*, pages 127–134, 1996.
7. Ioannis Parissis and Jérôme Vassy. Thoroughness of specification-based testing of synchronous programs. In *ISSRE*, pages 191–202, 2003.
8. H. Pretschner A., Lötzbeyer. Model based-testing with constraint logic programming : First results and challenges. In *2nd ICSE Workshop on Automated Program Analysis, Testing and Verification*, pages 1–9, 2001.
9. Pascal Raymond, Xavier Nicollin, Nicolas Halbwachs, and Daniel Weber. Automatic testing of reactive systems. In *IEEE Real-Time Systems Symposium*, pages 200–209, 1998.
10. Jérôme Vassy. *Génération automatique de cas de test guidée par les propriétés de sûreté*. PhD thesis, Université Joseph Fourier, Grenoble, France, october 2004.

Test fonctionnel de conformité vis-à-vis d'une politique de contrôle d'accès*

Frédéric Dadeau, Amal Haddad, and Thierry Moutet

Laboratoire d'Informatique de Grenoble (LIG)
UMR 5217
681, rue de la Passerelle
BP. 72
38402 Saint-Martin d'Hères cedex
FRANCE

{Frederic.Dadeau,Amal.Haddad,Thierry.Moutet}@imag.fr

Résumé Les travaux présentés dans cet article s'articulent autour de la validation du contrôle d'accès défini par des politiques de sécurité. Nous nous intéressons à la validation par génération de tests à partir d'un modèle fonctionnel écrit en B. Nous utilisons l'outil Meca, qui prend en entrée un modèle fonctionnel et une description d'une politique de sécurité sous la forme de machines abstraites B, et qui génère un noyau de sécurité pour le modèle fonctionnel. Ce noyau de sécurité est en charge d'intercepter tous les accès des sujets aux objets, et restreint les comportements à ceux satisfaisant les exigences de sécurité.

Nous proposons une approche définissant des objectifs de tests de conformité vis-à-vis d'une politique de contrôle d'accès. L'objectif est de s'assurer que les appels à des opérations dans un contexte licite (resp. illicite) sont bien autorisés (resp. sont bien bloqués) sur l'implantation sous test. Nous présentons une étude de cas complète, sur une application de type porte-monnaie électronique, modélisée en B et implantée en Java, et agrémentée d'une politique de sécurité discrétionnaire.

Mots-clés : Contrôle d'accès, test de conformité, Meca, politique de sécurité, noyau de sécurité

1 Introduction

Les systèmes actuels sont de plus en plus demandeurs en terme de sécurité. Par sécurité, on entend généralement cinq propriétés essentielles : l'*authentification*, qui permet de s'assurer de l'identité d'une entité donnée ou de l'origine d'une communication ou d'un fichier, la *confidentialité* qui empêche la divulgation d'une information tenue secrète, l'*intégrité*, qui vise à garantir qu'une information ne sera pas modifiée sans en avoir eu l'autorisation, la *disponibilité*, qui garantit l'accès aux informations pour les utilisateurs autorisés, et la *non-répudiation* qui empêche le reniement d'action ou de messages anciens.

* Ce travail a été en partie financé par le projet RNTL POSE (ANR-05-RNTL-01001)

Dans l'objectif d'évaluer la sécurité, les *Critères Communs* (CC) ont récemment été introduits. Il s'agit d'une norme (ISO 15408) qui valide la sécurité d'un système du point de vue logiciel et matériel. Les critères communs définissent une cible de sécurité (TOE – Target Of Evaluation) qui doit présenter un certain nombre d'exigences se répartissant en deux catégories : les exigences fonctionnelles de sécurité et les exigences d'assurance de sécurité. Les premières évaluent les mécanismes déployés pour garantir la sécurité, tandis que les secondes visent à garantir la bonne mise en œuvre de ces mécanismes et leur adéquation à la cible de sécurité. Nous nous intéressons plus particulièrement au cas du contrôle d'accès à des informations sensibles, qui est une sous-classe de la classe "Protection des données de l'utilisateur" (FDP – Functional Data Protection), appartenant à la catégorie des exigences fonctionnelles de sécurité. Le contrôle d'accès vise à s'assurer que des *sujets* ont ou non le droit d'accéder à des *objets*. Les droits ou les interdictions sont définis à l'aide de règles dépendantes d'attributs de sécurité.

Un système de contrôle d'accès doit intercepter toutes les tentatives d'accès non autorisés. La définition d'un tel système se base sur les trois éléments suivants. (*i*) Les politiques de sécurité, qui décrivent des règles de haut niveau de contrôle d'accès et décident lesquels sont autorisés ou interdits. (*ii*) Les modèles de sécurité, qui sont des formalisations des politiques de sécurité et de leur fonctionnement. Ils sont utilisés pour prouver des propriétés de sécurité dans les systèmes. Enfin, (*iii*) les mécanismes de sécurité définissent des fonctions de bas niveau (locigielles et matérielles) permettant d'implanter les contrôles imposés par la politique de sécurité.

On distingue différents types de politiques de contrôles d'accès. Les politiques discrétionnaires (Discretionary Access Control – DAC) [9] accordent au possesseur d'une information (généralement son créateur) tous les droits d'accès, ainsi que la possibilité de passer ces droits à d'autres utilisateurs à leur discrétion. Les politiques obligatoires (Mandatory Access Control – MAC) [3,4] décrètent des règles incontournables qui régissent les droits des sujets et des objets. Enfin, les politiques basées sur les rôles (Role-Based Access Control – RBAC) [6] permettent d'accorder des droits d'accès à des sujets en fonction des rôles qu'ils jouent dans une organisation.

Un travail préliminaire [7] a présenté l'outil Meca, qui considère un modèle fonctionnel d'un système, exprimé en B, et une politique de sécurité décrite par une machine abstraite B, et qui génère un noyau de sécurité interceptant les accès illicites des sujets aux objets. Les politiques de contrôle d'accès acceptées par Meca peuvent être de n'importe lequel des trois types précédents (DAC, MAC ou RBAC). Nous souhaitons coupler ce mécanisme avec un moyen de générer automatiquement des tests de conformité, de manière à s'assurer que la sécurité liée au système est correctement implantée. Notre objectif est donc de produire des tests guidés par ces principes de sécurité. Pour ce faire, nous considérons un modèle de sécurité dissocié du modèle fonctionnel du système. Nous nous appuyons sur les résultats produits par l'analyse de modèle dans Meca pour produire ces tests. Nous illustrons notre approche sur une étude de

cas complète, partant d'une spécification d'un porte-monnaie électronique, pour lequel on souhaite sécuriser l'accès, jusqu'à son implantation en Java.

Cet article s'organise de la manière suivante. Nous présentons en partie 2 l'approche que nous avons mise en place. Après un rapide survol des principales fonctionnalités de la méthode B en partie 3, nous introduisons l'utilisation d'un modèle dédié à la sécurité en partie 4. Nous expliquons la contribution principale de ces travaux, la génération de tests basée sur ce modèle de sécurité, en partie 5. Une expérimentation sur une étude de cas est donnée dans la partie 6. Puis, nous discutons de la technique employée et des extensions ultérieures en partie 7. Enfin, nous présentons les conclusions et les perspectives de ces travaux en partie 8.

2 Présentation de l'approche

L'objectif des travaux présentés dans cet article est de définir un mécanisme de génération de jeux de tests dédiés à exercer le contrôle d'accès d'une application. Basiquement, le contrôle d'accès se caractérise par un ensemble de sujets qui sont autorisés, ou non, à manipuler des objets. Ainsi, les interactions entre sujets et objets sont décrites par l'intermédiaire de permissions ou d'interdictions.

Les tests que nous nous proposons de générer s'appuient sur ce mécanisme, en souhaitant explorer le plus grand nombre de possibilités concernant les conditions d'utilisation des objets. Nous souhaitons effectuer des tests de conformité du contrôle d'accès, ce qui sous-entend vérifier que l'implantation permet bien l'accès dans des conditions licites d'utilisation et refuse bien l'accès dans des conditions illicites. Nous adoptons une approche de test fonctionnel boîte-noire, basée sur un modèle [2]. Dans cette configuration, un modèle est utilisé pour produire les cas de tests et donner l'oracle. L'implantation est, quant à elle, développée à part, sans lien avec le modèle, si ce n'est l'existence de points de connexion à travers des API définies dans le cahier des charges.

Pour ce faire, nous disposons de l'outil Meca, qui permet, à partir d'un ensemble de règles données dans un modèle de sécurité et d'un modèle fonctionnel, de produire un noyau de sécurité qui contrôle les appels des différentes opérations pour n'autoriser que les exécutions des opérations qui sont licites, et détecter (et signaler) les appels aux opérations qui sont illicites. Le modèle fonctionnel sécurisé produit par Meca fournit l'oracle, qui consiste à vérifier la conformité entre la prédiction du modèle et la réalité de l'implantation : dans des conditions d'activation supposées similaires, si une implantation permet un accès, ce dernier est-il autorisé sur le modèle ? On notera qu'en terme de sécurité, l'implantation peut dans certains cas refuser des accès pourtant autorisés par le modèle de sécurité. En effet, il est possible que certains aspects purement fonctionnels aient été omis dans le modèle de sécurité et qu'une implantation soit plus restrictive dans ses cas d'utilisation (par exemple, des limitations introduites sur les domaines des variables).

Notre approche se résume par la Fig. 1. A partir d'une spécification informelle des besoins, nous considérons deux modèles, un modèle fonctionnel et un modèle

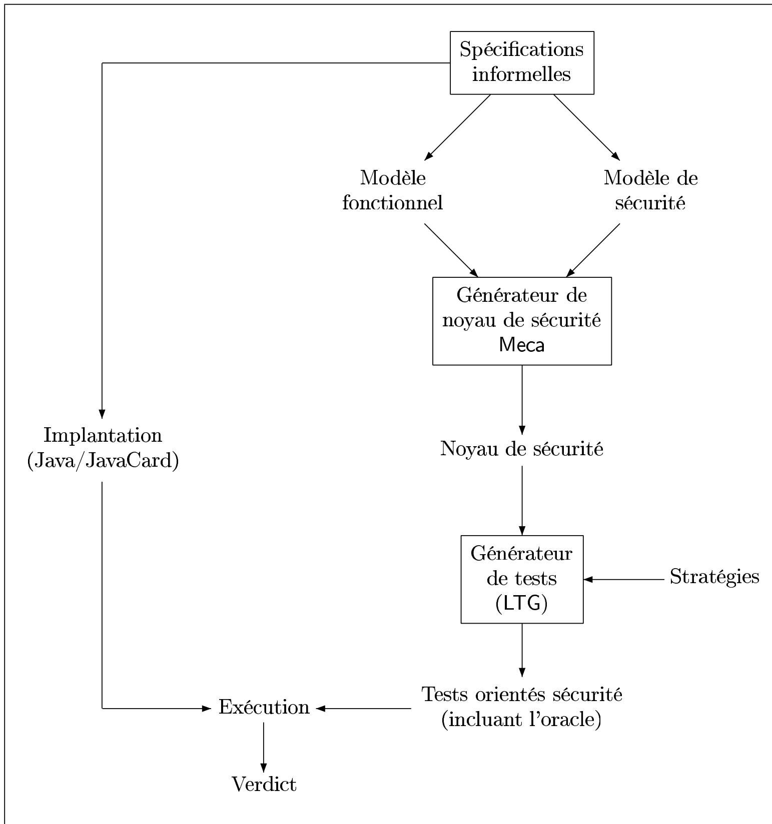


Fig. 1. Principe de l'approche proposée

de sécurité, tous deux écrits en B. Le modèle fonctionnel est plus ou moins détaillé, mais il doit introduire les éléments sur lesquels portent la politique de sécurité (attributs de sécurité, opération à sécuriser, etc.). Dans notre cas, les objets contrôlés sont les opérations du modèle B fonctionnel. En parallèle, nous concevons un autre modèle B, décrivant les règles d'accès des sujets aux objets contenus dans le modèle fonctionnel.

Ces deux modèles sont fournis en entrée de l'outil Meca qui produit un noyau de sécurité en charge de rajouter des vérifications, protégeant les accès des sujets aux objets en fonction de la politique d'accès choisie. Ceci nous produit une sur-couche du modèle fonctionnel, que nous désignons par "noyau de sécurité". Nous utilisons ensuite *LEIRIOS Test Generator* (LTG) [13,8], un générateur automatique de tests fonctionnels à partir d'un modèle B basé sur des critères de couverture du modèle. Nous verrons en partie 5 en quoi le modèle fonctionnel

sécurisé généré par Meca est adapté à la stratégie de génération de tests proposée par LTG. Ces tests sont ensuite concrétisés et exécutés sur une implantation Java. Les prédictions issues de l'exécution du modèle nous permettent d'établir le verdict du test.

Notons que l'implantation Java sur laquelle sont joués les tests a été developpée indépendamment du modèle fonctionnel et du modèle de sécurité. Elle représente ainsi la vision que le développeur a du cahier des charges. Le processus de test consiste à s'assurer qu'aucun élément inhérent à la sécurité de l'application n'a été oublié lors du développement. Pour évaluer notre approche, nous exerçons les tests générés sur des mutants, variantes de notre implantation. Ces variations sont guidées, dans le sens où elles affectent la vérification des conditions de sécurité dans l'implantation, simulant des erreurs classiques.

3 La méthode B

La *méthode B* [1] est une méthode formelle permettant le développement de logiciels corrects par construction. Cette caractéristique repose essentiellement sur des mécanismes permettant de vérifier par la preuve, à l'aide d'outils dédiés, qu'un système vérifie certaines propriétés exprimées avec des notations mathématiques. Elle permet en particulier d'écrire une spécification formelle sous la forme de *machines abstraites*. Celles-ci comportent des *données* (qui peuvent être exprimées par des types simples comme des entiers ou des booléens mais également par des ensembles ou des relations), des *propriétés invariantes* sur ces données et des *services* permettant d'initialiser et de faire évoluer ces données par l'intermédiaire de substitutions. Ces derniers sont représentés par des opérations.

A partir d'une machine abstraite, il est possible de continuer son développement par un travail de reformulation de la spécification et d'enrichissement du modèle initial. Ce processus de raffinemement est un mécanisme intégré de la méthode B. Son objectif est de garantir qu'à chaque étape du raffinement, on produit un modèle conforme au modèle réalisé à l'étape précédente. Cette validité est assurée aux moyens de vérifications statiques et d'obligation de preuves. Le dernier niveau de raffinement correspond à l'implantation. A cette étape, toutes les données ou substitutions manipulées doivent avoir un équivalent informatique. Il est ainsi possible de faire une traduction du modèle formel dans un langage exécutable comme Ada ou C. Cette traduction peut même être automatisée au moyen d'outils spécialisés comme l'*AtelierB*.

Dans le cadre de notre expérimentation, nous avons utilisé le langage B pour modéliser une application bancaire de type porte-monnaie électronique. La variable *balance* correspond au montant disponible sur la carte. Un *code pin utilisateur* permet de sécuriser la lecture, le crédit et le débit sur la carte. Un *code pin banque* permet de réinitialiser le *code pin utilisateur* en cas de blocage de celui-ci. Enfin, une variable *mode* représente le cycle de vie de la carte. Cette variable peut avoir trois états. En mode personnalisation (PERSO), les seuls traitements

```
MACHINE
    epurse
SETS
    MODE={PERSO,USE,INVALID}
DEFINITIONS
    BPC == 0..9999
VARIABLES
    bpc,bptry,isBankAuth,       /* code pin banque, nombre d'essai */
    mode,isOpenSess,...         /* mode et ouverture de session */
INVARIANT
    bpc ∈ BPC ∧ bptry ∈ 0..3 ∧ mode ∈ MODE ∧ ...
OPERATIONS
    setBpc(pin) =
       PRE
           pin ∈ BPC ∧          /* typage du pin en paramètre */
           isOpenSess = btrue ∧ /* la session doit être ouverte */
           mode = PERSO          /* bpc est modifiable en mode PERSO */
       THEN
           bpc := pin ||         /* mise à jour code pin banque */
           bptry := 3            /* compteur d'essai initialisé à 3 */
       END
    ...
END
```

Fig. 2. Schéma du modèle fonctionnel de l'application pour l'opération setBpc

autorisés correspondent à l'installation des codes pin. Une fois la configuration des codes pin effectuée, la carte passe en mode utilisation (`USE`). Celui-ci permet d'effectuer n'importe quel traitement relatif au solde à condition de s'authentifier au préalable au moyen du bon code pin. Enfin, le mode devient invalide (`INVALID`) à l'issu d'un blocage du code pin utilisateur. Dans ce mode, il est seulement possible de réinitialiser le code pin utilisateur après s'être authentifié avec le code pin banque.

La Figure 2 présente le modèle fonctionnel de l'application qui a été utilisée pour notre expérimentation. Seule l'opération `setBpc` est présentée. Cette opération permet de fixer le code pin de la banque, ce qui ne peut avoir lieu que dans le cas où une session a été ouverte et si la carte est en phase de personnalisation. Les propriétés de sécurité ne sont pas toutes exprimées de manière explicite dans cette spécification fonctionnelle même si celles-ci doivent être prises en compte au niveau de l'implantation. Un modèle dédié à la sécurité complète donc les aspects fonctionnels. Ceci est présenté dans la partie suivante.

4 Modèle dédié à la sécurité

Nous commençons par présenter le format du modèle de sécurité. Nous nous intéressons ensuite au noyau de sécurité produit par Meca.

4.1 Format du modèle de sécurité

Comme nous l'avons représenté en Fig. 1, les entrées de Meca sont d'une part un modèle fonctionnel et d'autre part un modèle de politique de sécurité. Le modèle fonctionnel contient une description des entités sensibles du système ainsi que leur comportement fonctionnel.

Le modèle de politique de sécurité contient d'abord la description des deux entités qui interviennent dans le contrôle d'accès : les *sujets* et les *objets*. Les objets représentent les entités passives du modèle fonctionnel, dont les accès doivent être sécurisés. Les sujets sont les entités actives pour lesquelles on souhaite contrôler l'accès aux objets. Le modèle contient ensuite la liste des permissions qui sont données dans le contrôle d'accès. Pour Meca, il n'est possible de décrire que des permissions exprimant des autorisations. Implicitement tout ce qui n'apparaît pas dans les permissions est considéré comme interdit.

Nous avons également proposé un format de contrôle d'accès des utilisateurs (UAC - User Access Control) en fonction de conditions portant sur des attributs de sécurité. Il permet de contrôler l'exécution des opérations. La Figure 3 décrit le format d'entrée de Meca du type UAC. On remarquera que cette machine ne présente pas d'opérations et que les permissions sont exprimées à travers une constante.

Dans le cadre du modèle UAC, les permissions accordées aux sujets dépendent entre autres de l'état interne du système. Ainsi, certaines entités du système restreignent les permissions accordées dans le cadre d'une politique de sécurité. De telles entités sont appelées des *attributs de sécurité*. Par exemple, le terminal administratif est autorisé à enregistrer le code de la banque seulement lorsque la carte se trouve dans le mode personnalisation. Ceci se traduit dans le modèle UAC par la règle suivante :

$$\texttt{(mode = PERSO)} \Rightarrow \texttt{(terminalAdministratif} \mapsto \texttt{setBpc)} \in \texttt{permission}$$

```
MACHINE
    uac_Policy
SETS
    SUBJECT={s1,s2,...};           /* sujets du contrôle d'accès */
    OBJECT={op1,op2,...}           /* opérations du modèle fonctionnel */
CONSTANTS
    permission, ...
PROPERTIES
    permission ∈ SUBJECT ↔ OBJECT ∧
    condition ⇒ (s1 ↦ op1) ∈ permission ∧
    ...
```

Fig. 3. Schéma général du format du modèle de sécurité

4.2 Format du noyau de sécurité

Meca génère un noyau de sécurité qui correspond au modèle fonctionnel de l'application, auquel ont été ajoutées les conditions de sécurité qui servent à faire appliquer le contrôle d'accès. Ces conditions sont synthétisées à partir des permissions définies dans le modèle de politique de sécurité. En configurant certaines options de Meca, il est possible de paramétrer la forme du noyau de sécurité généré. Ainsi, Meca propose de générer un noyau qui est soit de forme offensive, soit de forme défensive.

Dans la forme défensive, chaque opération contient une précondition, plus deux branchements conditionnels protégeant l'accès à l'opération. Dans la précondition, on retrouve uniquement les informations de typage des paramètres d'appel, extraits du modèle fonctionnel. Les deux décisions doivent être satisfaites pour accéder à l'opération protégée. La première décision est la condition de sécurité. Elle correspond à une reformulation des règles du modèle de sécurité et pour l'opération considérée. La seconde décision est la condition fonctionnelle, qui correspond à la précondition de l'opération dans le modèle fonctionnel. La Figure 4 représente le format général du noyau de sécurité défensif pour le format UAC. On notera qu'une opération du noyau de sécurité est produite pour chaque opération du modèle fonctionnel qui est contrôlée.

```
MACHINE
    uac_def_SecurityKernel
OPERATIONS
    rs, rf, o1,...,oN ⟵ execute_op(su,i1,...,iN) ≙
        PRE
            su ∈ SUBJECT ∧ ...          /* typage des paramètres */
        THEN
            IF Condition_Secu           /* condition du mod. de sécurité */
            THEN      rs := OK ||
                IF Condition_Func       /* précondition du mod. fonct. */
                THEN  rf := OK ||
                    o1,...,oN := op(i1,...,iN) /* opération du mod. fonct. */
                ELSE  rf := KO || skip
                END
            ELSE      rs := KO ||
                IF    Condition_Func    /* précondition du mod. fonct. */
                THEN  rf := OK || skip
                ELSE  rf := KO || skip
                END
            END
        END
...
```

Fig. 4. Schéma général du format du noyau de sécurité défensif pour le format UAC

Dans le cadre de notre expérimentation, nous utilisons la forme défensive du noyau de sécurité pour faire de la génération de cas de test. Un autre intérêt de cette forme est de pouvoir faire de la surveillance à l'exécution en utilisant un moniteur basé sur le noyau de sécurité. Son rôle est de gérer tout appel aux opérations de l'application en interceptant tout accès illicite. Comme nous l'avons présenté au début de cette partie, Meca peut aussi générer une forme offensive du noyau de sécurité. Dans ce cas, les deux conditions telles qu'elles apparaissent dans la forme défensive sont rassemblées au sein de la précondition. Cette forme est plutôt utilisée pour faire de la preuve et n'est pas l'objet de cet article.

Sachant que notre objectif se place dans le cadre du contrôle d'accès, nous avons choisi de séparer les deux conditions afin d'obtenir un contrôle plus fin. A cet effet, nous avons utilisé deux paramètres de retour, `rs` et `rf`, qui nous permettent d'observer, pour chaque appel, la satisfaction respective de la condition de sécurité et de la condition fonctionnelle.

Pour que le comportement fonctionnel de l'opération soit assuré (ce qui se traduit par l'appel de l'opération issue du modèle fonctionnel), il est nécessaire que la condition de sécurité et la condition fonctionnelle soient satisfaites (`rs=OK` et `rf=OK`). Lorsque la condition de sécurité n'est pas satisfaite (`rs=KO`), l'opération ne doit jamais être exécutée.

Un exemple d'opération du noyau de sécurité est donné en Fig. 5. On retrouve la *condition fonctionnelle* qui régit son exécution, issue du modèle fonctionnel : $\texttt{isOpenSess} = \texttt{TRUE} \wedge \texttt{pin} \in \texttt{BPC} \wedge \texttt{mode} = \texttt{PERSO}$, ainsi que la *condition de sécurité* générée par Meca, en fonction du modèle de sécurité : $\texttt{mode} = \texttt{PERSO} \wedge \texttt{su} = \texttt{terminalAdministratif}$.

5 Test de conformité à une politique de sécurité

Nous nous intéressons dans cette partie à la description du test vis-à-vis d'une politique de contrôle d'accès. Nous commençons par définir les tests que nous souhaitons obtenir. Puis, nous présentons comment s'établit le verdict de conformité par rapport au contrôle d'accès. Pour finir, nous présentons l'outil qui réalisera le calcul des cas de tests.

5.1 Stratégies de test du contrôle d'accès

C'est le noyau de sécurité qui est exploité pour produire les tests de conformité. En effet, celui-ci traduit l'aspect opérationnel de la politique de sécurité. Une opération du noyau de sécurité matérialise une tâche dans le système ; ici, il s'agit d'exécuter une opération du modèle fonctionnel par le noyau de sécurité.

Notre but est de nous assurer que lorsque la condition de sécurité est satisfaite, l'opération agit comme prévu, en fonction de sa précondition. Plus précisément, si cette dernière est satisfaite, le système évolue, dans le cas contraire, il reste dans le même état et aucune valeur de retour de l'opération du modèle fonctionnel ne peut être élaborée. Ainsi, seule l'activation licite d'une opération

```
rs,rf ← execute_setBpc(pin,su) ≙
    PRE
        su ∈ SUBJECT ∧ pin ∈ BPC
    THEN
        IF mode = PERSO ∧ su = terminalAdministratif
        THEN
            rs := OK ||
            IF isOpenSess = btrue ∧ pin ∈ BPC ∧ mode = PERSO
            THEN rf := OK || setBpc(pin)
            ELSE rf := KO
            END
        ELSE
            rs := KO ||
            IF isOpenSess = btrue ∧ pin ∈ BPC ∧ mode = PERSO
            THEN rf := OK
            ELSE rf := KO
            END
        END
    END
```

FIG. 5. L'opération setBpc du noyau de sécurité

est susceptible de modifier l'état interne du système et/ou d'accéder aux données du système. Pour ce faire, il est nécessaire de procéder à une phase d'observation qui nous permettra de décider du verdict du test.

Du point de vue de la stratégie de génération de test, nous cherchons à exploiter le comportement de chaque opération avec la satisfaction, ou la non-satisfaction de sa condition de sécurité. Ainsi, nous nous intéressons à la couverture des conditions de sécurité et des conditions fonctionnelles. Plus précisément, nous souhaitons que nos tests exhibent chacune des combinaisons suivantes :

- Satisfaction des conditions de sécurité et fonctionnelle (rs = OK et rf = OK)
- Satisfaction de la condition de sécurité et non-satisfaction de la condition fonctionnelle (rs = OK et rf = KO)
- Non-satisfaction de la condition de sécurité et satisfaction de la condition fonctionnelle (rs = KO et rf = OK)
- Non-satisfaction de la condition de sécurité et non-satisfaction de la condition fonctionnelle (rs = KO et rf = KO)

L'implantation doit tenir compte de la sécurité. Pour s'assurer qu'elle respecte bien la politique de sécurité, nous proposons de jouer les cas de test générés à partir du noyau de sécurité sur l'implantation. Lors de l'exécution des cas de test sur l'implémentation, la spécification nous permet d'établir les verdicts des tests pour décider si le test réussit ou échoue. L'établissement du verdict est discuté à présent.

5.2 Établissement du verdict

L'établissement du verdict se base sur les sorties produites par l'implantation par rapport aux sorties calculées par le modèle. Le modèle B à partir duquel sont calculés les tests encapsule les appels et retourne systématiquement une valeur représentant la satisfaction de la condition de sécurité (`rs`) et une valeur représentant la satisfaction de la condition fonctionnelle (`rf`) (cf. partie 4.2).

Nous supposons que l'implantation sous test est capable de faire la distinction entre une terminaison normale et une terminaison erronée, dûe sans distinction à une condition de sécurité incorrecte ou une condition fonctionnelle incorrecte. On notera que cette hypothèse est réaliste vis-à-vis d'une implantation de type JavaCard. En effet, un tel programme retourne un code nommé Status Word dont la valeur, définie dans le cahier des charges, indique les éventuelles erreurs (typiquement, 9000 signifie une terminaison normale, toute autre valeur signifie une terminaison anormale parce qu'une erreur d'un certain type est détectée). Nous faisons l'hypothèse qu'une terminaison anormale de l'application n'entraîne aucune évolution du système, et que celui-ci n'est susceptible d'évoluer que lorsque l'opération a terminé normalement.

Nous devons établir le verdict à partir des résultats produits par l'implantation et des résultats attendus donnés par le modèle de sécurité, et nous renseignant sur la conformité de l'implémentation. Ces verdicts sont établis suivant le Tableau 1.

Les tests qui nous intéressent sont ceux qui détectent une non-conformité liée spécifiquement à la condition de sécurité, c'est-à-dire lorsque l'implantation permet l'exécution normale du programme alors qu'une erreur était attendue (numéros (2), (3) et (4)), ou lorsqu'une erreur est détectée dans le programme alors que le modèle n'en attendait pas (numéro (5)). Typiquement, le cas (3) est intéressant car l'implantation a autorisé l'exécution de l'opération alors que le modèle prévoyait une erreur due au contrôle d'accès, ce qui signifie que l'implantation a, par exemple, oublié de prendre en compte un cas de refus de l'accès à l'application. Tout ceci correspond à une notion de conformité au sens strict (col. 3 du tableau).

Résultat donné par le programme	Modèle fonctionnel (`rs`,`rf`)	Verdict du test (conf. stricte)	Verdict du test (conf. du refus)	#
Terminaison normale	(OK,OK)	*Succès*	*Succès*	(1)
	(OK,KO)	*Échec*	*Échec*	(2)
	(KO,OK)	*Échec*	*Échec*	(3)
	(KO,KO)	*Échec*	*Échec*	(4)
Terminaison anormale (erreur de sécurité ou erreur fonctionnelle)	(OK,OK)	*Échec*	*Succès*	(5)
	(OK,KO)	*Succès*	*Succès*	(6)
	(KO,OK)	*Succès*	*Succès*	(7)
	(KO,KO)	*Succès*	*Succès*	(8)

Tab. 1. Etablissement du verdict en fonction du retour du programme

Néanmoins, la spécification peut possiblement accepter plus de comportements que l'implantation. Par exemple, le fait que le solde sur le porte-monnaie soit limité peut entraîner un comportement restreint une fois le programme embarqué sur une carte. De ce fait, nous ne cherchons pas à établir une conformité stricte entre l'implantation et le modèle de sécurité, mais nous cherchons plutôt à nous assurer que les comportements autorisés par l'application sont effectivement des comportements licites. De ce fait, nous distinguons dans le tableau la conformité “stricte” (tous les comportements autorisés par l'application sont autorisés par le modèle, et tous les comportements refusés par l'implantation sont refusés par le modèle) et la conformité relative au “refus” (col. 4 du tableau) : tous les comportements refusés par le modèle sont refusés par l'application, et l'application peut refuser des comportements autorisés dans le modèle.

Nous nous intéressons à présent à l'outil LTG que nous utilisons pour générer les suites de tests.

5.3 L'outil LEIRIOS Test Generator

LEIRIOS Test Generator (LTG) [8,13] est un outil de génération automatique de tests fonctionnels. Il peut prendre en paramètres différents formats dont des spécifications écrites en B. Les tests produits par LTG se décomposent comme illustré en Fig. 6.

Les cibles de tests sont d'abord calculées par rapport au modèle du système à tester. Une cible de test est l'activation d'un comportement extrait d'une opération du modèle B. Un comportement se définit basiquement comme un chemin d'exécution possible dans l'opération. Le moteur d'animation de LTG, basé sur l'animation symbolique à contraintes, est ensuite utilisé pour produire le *préambule*. Il s'agit d'une trace d'exécution menant, depuis l'état initial, à un état cible. Le *corps* du test en lui-même consiste en l'activation du comportement sélectionné, à partir du contexte définissant la cible de test. Une phase d'*observation* intervient ensuite. Celle-ci permet l'appel à des opérations spécifiques du modèle afin d'obtenir la valeur de variables du modèle. Cette phase permettra ensuite, lors de l'exécution du test, de comparer les résultats rendus par l'implantation par rapport aux résultats attendus, calculés sur le modèle. Enfin, le *postambule* est une séquence d'appels d'opérations permettant de remettre le système dans

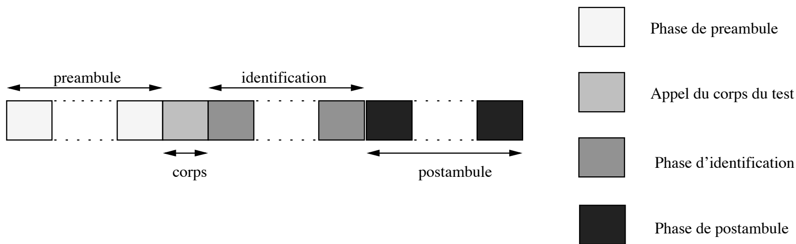


Fig. 6. Composition d'un cas de test produit par LTG

l'état initial. Il permet d'enchaîner les tests, sans avoir à passer par une remise à zéro manuelle du système, qui n'est pas toujours possible.

Calcul des cas de test Le processus de génération de tests de LTG s'appuie sur la couverture du modèle, au niveau structurel, au niveau des décisions et au niveau des données.

Couverture structurelle des opérations. Les tests produits par LTG ciblent la couverture des opérations du modèle fonctionnel. Pour ce faire, LTG considère les différents chemins du graphe de flot de contrôle de l'opération, qu'il nomme comportements.

Exemple 1 (Découpage en comportements) Pour cet exemple, nous considérons l'opération `execute_setBpc(su,pin)` extraite du noyau de sécurité produit par Meca, et présentée dans la Fig. 5. Cette opération se décompose en quatre comportements, correspondant aux imbrications de IF-THEN-ELSE, nommés cpt_1, cpt_2, cpt_3 et cpt_4, comme suit :

cpt_1	$\texttt{su} \in \texttt{SUBJECT} \wedge \texttt{pin} \in \texttt{BPC} \wedge \texttt{mode} = \texttt{PERSO} \wedge \texttt{su} = \texttt{terminalAdministratif}$ $\wedge\ \texttt{isOpenSess} = \texttt{btrue} \Longrightarrow \texttt{rs} := \texttt{OK} \wedge \texttt{rf} := \texttt{OK}\ \Vert \texttt{setBpc(pin)}$
cpt_2	$\texttt{su} \in \texttt{SUBJECT} \wedge \texttt{pin} \in \texttt{BPC} \wedge \texttt{mode} = \texttt{PERSO} \wedge \texttt{su} = \texttt{terminalAdministratif}$ $\wedge\ \texttt{isOpenSess} \neq \texttt{btrue} \Longrightarrow \texttt{rs} := \texttt{OK} \parallel \texttt{rf} := \texttt{KO}$
cpt_3	$\texttt{su} \in \texttt{SUBJECT} \wedge \texttt{pin} \in \texttt{BPC} \wedge (\texttt{mode} \neq \texttt{PERSO} \vee \texttt{su} \neq \texttt{terminalAdministratif})$ $\wedge\ \texttt{isOpenSess} = \texttt{btrue} \Longrightarrow \texttt{rs} := \texttt{KO} \parallel \texttt{rf} := \texttt{OK}$
cpt_4	$\texttt{su} \in \texttt{SUBJECT} \wedge \texttt{pin} \in \texttt{BPC} \wedge (\texttt{mode} \neq \texttt{PERSO} \vee \texttt{su} \neq \texttt{terminalAdministratif})$ $\wedge\ \texttt{isOpenSess} \neq \texttt{btrue} \Longrightarrow \texttt{rs} := \texttt{KO} \parallel \texttt{rf} := \texttt{KO}$

Une cible de test est représentée par la condition d'activation du comportement considéré. Cette condition d'activation est calculée comme étant la conjonction des prédicats composant un comportement. Par exemple, la condition d'activation du comportement cpt_3 de l'exemple précédent est : $\texttt{su} \in \texttt{SUBJECT} \wedge \texttt{pin} \in \texttt{BPC} \wedge (\texttt{mode} \neq \texttt{PERSO} \vee \texttt{su} \neq \texttt{terminalAdministratif}) \wedge \texttt{isOpenSess} = \texttt{btrue}$.

Couverture des décisions des opérations. En complément, LTG propose d'affiner la couverture du modèle en sélectionnant un critère de couverture des décisions relatives aux comportements [10]. Pour ce faire, il applique différentes réécritures portant sur les disjonctions du prédicat définissant la cible de test. Chacune de ces réécritures permet de satisfaire un critère spécifique de couverture des décisions. Le Tableau 2 décrit ces réécritures, ainsi que les critères associés [11].

La réécriture 1 laisse la disjonction telle qu'elle, ne produisant ainsi qu'une seule cible de test, saitsfaisant indifféremment P_1 ou P_2. La réécriture 2 produit deux cibles de test, visant la satisfaction de chacun des littéraux composant la disjonction. La réécriture 3 produit également deux cibles de test visant l'activation de chacun des littéraux composant la disjonction de manière exclusive

Nř	Réécriture de $P_1 \vee P_2$	Critère de couverture des décisions
1	$P_1 \vee P_2$	Decision Coverage (DC)
2	P_1, P_2	Condition/Decision Coverage (C/DC)
3	$P_1 \wedge \neg P_2$, $\neg P_1 \wedge P_2$	Modified Condition/Decision Coverage (MC/DC)
4	$P_1 \wedge P_2$, $P_1 \wedge \neg P_2$, $\neg P_1 \wedge P_2$	Multiple Condition Coverage (MCC)

Tab. 2. Critères de couverture des décisons en fonction des réécritures

par rapport aux autres littéraux. Enfin, la réécriture 4 permet de produire trois cibles de test, de manière à exhiber toutes les possibilités de satisfaire une disjonction.

Couverture des données. LTG propose également une gestion assez fine des valeurs des variables et des paramètres d'entrée d'une opération sous test. Ainsi, toutes les données qui ne sont pas précisées dans la cible du test peuvent se voir appliquer une sélection consistant à choisir par exemple : une seule valeur, des valeurs aux bornes des domaines (pour les domaines ordonnés), ou encore une énumération de toutes les valeurs possibles pour les domaines finis.

Lien avec le modèle fonctionnel sécurisé LTG est un outil qui se base sur l'analyse d'un modèle pour produire des tests boîte-noire pertinents. Nous nous sommes fixés des besoins de test liés au modèle du noyau de sécurité, décrits dans la partie précédente. La stratégie de génération de tests de LTG satisfait les critères que nous nous sommes fixés.

- La couverture structurelle des opérations nous permet de considérer tous les cas d'exécution d'une opération, par rapport à ses conditions d'accès et fonctionnelles.
- La couverture des disjonctions permet de jouer finement sur les différentes configurations du système provoquant l'autorisation ou le refus de l'accès. De plus, un mécanisme de résolution de contraintes élimine les cibles inconsistantes.
- La couverture des données permet l'énumération des paramètres d'entrée, augmentant ainsi le nombre de cas de tests. Dans notre cas, en énumérant toutes les valeurs possibles des sujets (paramètres `su`) nous sommes en mesure de couvrir de manière exhaustive tous les cas d'accès (ou de refus) pour chacun des sujets sur les différents objets.

6 Expérimentation

Nous nous intéressons à présent à la mise en œuvre de ces techniques dans le cadre d'une expérimentation[1]

[1] Les spécifications, les modèles et l'application utilisés pour cette expérimentation sont disponibles à l'adresse : `http://ww-lsr.imag.fr/Les.Personnes/Amal.Haddad/afadl07/`

6.1 Génération de tests abstraits

LTG impose que le modèle fonctionnel fourni en entrée ne contienne pas d'appels d'opérations dans les substitutions, et que les données aient des domaines finis. Nous modifions donc le noyau de sécurité pour aplatir les appels d'opérations et nous bornons la variable représentant le solde du porte monnaie (initialement déclarée comme INT).

Nous donnons le noyau de sécurité modifié en entrée de LTG, qui génrère les cibles relatives aux branches d'exécutions de ce modèle. Nous choisissons le critère de couverture des décisions MCC de manière à avoir le plus de combinaisons possibles. De plus, pour chaque paramètre d'opération, nous choisissons d'énumérer toutes les valeurs possibles. Il en va de même pour les variables d'état qui sont impliquées dans les conditions de sécurité et les conditions fonctionnelles des opérations ciblées. Les variables numériques (comme les codes pin) sont instanciées aux limites, en laissant LTG sélectionner une valeur extremum du domaine des variables. Nous énumérons toutes les valeurs des variables ayant un domaine fini et de faible cardinalité (par exemple, le mode).

Intuitivement, ces choix nous permettent de tester les conditions d'accès pour tous les sujets possibles (les sujets étant en paramètre des opérations testées). De plus, l'énumération des valeurs des variables d'état nous permet de couvrir un grand nombre de configurations à partir desquelles réaliser les tests.

Nous ne choisissons pas d'opération d'observation ; nous laissons au pilote de test le soin d'observer la terminaison d'une opération. Pour simplifier, nous choisissons de créer de nouveaux objets pour chaque cas de test, ce qui nous dispense de générer des postambules.

Le nombre de tests est donné par le tableau 3, pour chacune des opérations du modèle fonctionnel. Au total, notre suite de test se compose de 203 cas de test. On remarque que les nombres de cibles générées pour `setHpc` et `getBalance` sont supérieures aux nombres de cas de tests. Ceci est dû aux combinaisons de littéraux, issus de la réécriture choisie, qui produisent des cibles de tests inatteignables.

Opération	**Nombre de cibles**	**Nombre de cas de test**
`setBcp`	18	18
`beginSession`	6	6
`endSession`	6	6
`setHpc`	36	24
`authBank`	18	18
`checkPin`	24	32
`getBalance`	24	15
`debit`	42	46
`credit`	29	45
Total	**231**	**203**

Tab. 3. Résultat de la génération de tests pour l'exemple

6.2 Concrétisation des tests

Les tests abstraits produits par LTG sont ensuite concrétisés pour être exécutés sur l'implantation sous test. Celle-ci se présente sous la forme d'une classe Java qui décrit le porte-monnaie et dispose des mêmes points d'entrée que le modèle fonctionnel. Les méthodes sont susceptibles de déclencher des exceptions relatives au non-respect des conditions du cahier des charges.

Un pilote de test spécifique, écrit en Java, est en charge de l'exécution des cas de test et de l'observation de la terminaison des méthodes. Soit la méthode termine normalement et aucune exception n'est déclenchée, soit la méthode déclenche une exception et celle-ci est rattrapée par le pilote de test. Le verdict est ensuite établi par une fonction qui considère cette terminaison, ainsi que les valeurs attendues pour `rs` et `rf` issues du modèle, et qui calcule le verdict, *Succès* ou *Échec*, selon le tableau 1 (colonne “refus”).

Nous avons joué notre suite de tests sur notre application, et nous n'avons détecté aucune erreur. Nous avons donc cherché à évaluer la pertinence de notre suite de tests. Pour ce faire, nous nous sommes tournés vers une analyse mutationnelle.

6.3 Évaluation des tests produits

L'analyse mutationnelle consiste à introduire volontairement des erreurs dans un programme correct. Une variante erronée du programme est appelée *mutant*. Ces mutants servent à mesurer l'efficacité d'une suite de test, au sens où cette dernière doit être capable de détecter que les mutants sont incorrects. Quand la suite de test identifie une erreur dans un mutant, on dit que le mutant est tué.

Nous avons généré 30 mutants de notre application (considérée comme référence). Les mutations que nous avons introduites concernent spécifiquement l'implantation des vérifications des conditions d'accès. Les mutations que nous avons introduites concernent l'évaluation des différentes conditions d'accès régissant l'exécution des méthodes de l'application. Elles simulent des erreurs classiques de programmation. Ainsi, nous avons effectué les modifications suivantes :

- remplacement de `&&` par `||`, et inversement ;
- remplacement d'un prédicat d'une conjonction ou d'une disjonction par `true` ou `false` ;
- négation des prédicats dans les `if` ;
- suppression d'une vérification.

Ces mutations guidées représentent des erreurs de programmation les plus classiques. Notre suite de test a réussi à tuer les 30 mutants, nous confortant dans l'efficacité de la méthode de génération de tests que nous avons proposée.

7 Discussion

Pour s'assurer de la sécurité d'une application, il est nécessaire de vérifier que si une opération n'a pas été appelée dans des conditions d'accès licites, alors le

système doit rester dans le même état, ou alors ne pas effectuer de modification qui compromettrait ultérieurement la sécurité du système. C'est l'hypothèse que laquelle nous sommes partis.

Dans notre expérimentation, si l'opération de l'application termine anormalement (en déclenchant une exception), nous avons supposé qu'aucune variable n'était modifiée. Néanmoins, ce n'est pas nécessairement le cas. D'une part, le modèle peut admettre des évolutions des attributs de sécurité lorsque les accès sont refusés, tels que des compteurs de ratification. D'autre part, notre notion d'observation des résulats obtenus étant assez grossière, nous ne sommes pas en mesure de nous assurer que si aucun attribut n'est supposé évoluer dans le modèle, c'est effectivement le cas dans l'implantation.

Pour résoudre le premier point, nous sommes actuellement en train de travailler à l'évolution du modèle de sécurité, de manière à ce qu'il prenne en compte la dynamique des attributs de sécurité. Ceci aura nécéssairement un impact sur les formats, à la fois du modèle de sécurité et du noyau de sécurité.

Pour nous assurer qu'en dehors des conditions d'accès licites le système n'évolue pas, comme identifié dans le second point, il est nécessaire d'observer les valeurs des variables d'état. Ceci peut être réalisé de manière directe, si le système sous test fournit des primitives permettant de récupérer des valeurs de variables (potentiellement sensibles) du système. Cette hypothèse est réaliste ; en effet, les fabricants de carte sont parfois amenés à ajouter de telles primitives dans le cadre de la phase de validation, avant de les retirer pour la version finale du logiciel. Néanmoins, nous aimerions ne pas avoir à demander ce genre de contraintes à notre système ; nous envisageons donc une autre solution.

Notre seul point de jonction entre le modèle et l'application étant les opérations, nous proposons de nous assurer que le système n'a pas évolué en considérant une phase d'observation particulière. Celle-ci consiste à tenter d'effectuer, à la suite du corps du test, des opérations qui devraient être refusées du point de vue du contrôle d'accès. Si ces opérations peuvent effectivement être activées sur l'implantation alors, même si le corps du test (l'opération que l'on teste à l'origine) est conforme aux prédictions, il rend possible l'exécution d'actions illicites, ce qui est une erreur. Dans ce cas, le test serait un échec. Concrètement, nous chercherions à activer, suite au corps du test, toutes les opérations qui sont susceptibles de déclencher une erreur d'accès (sur le modèle `rs=KO` et `rf=OK`). De ce fait, nous sommes capables d'étendre notre pouvoir de décision du verdict du test, sans pour autant demander une fonctionnalité supplémentaire à l'application.

Nous travaillons actuellement à la mise en place de telles stratégies de détection, employant une approche combinatoire et une technique de filtrage des cas de tests vis-à-vis du résultat obtenu.

8 Conclusion et Perspectives

Nous avons présenté dans cet article une approche visant à valider de manière automatique une implantation vis-à-vis d'une politique de sécurité décrivant le

contrôle d'accès. Nous nous sommes focalisés sur une politique décrivant des permissions. Nous avons défini une notion de conformité d'une application vis-à-vis du contrôle d'accès. Notre approche considère un noyau de sécurité, sous la forme d'un modèle B, dont la couverture permet de réaliser, avec l'outil LTG, des tests pertinents par rapport aux propriétés d'accès. Pour valider notre démarche, nous avons proposé une expérimentation sur un exemple réaliste d'un porte-monnaie électronique, auquel est associée une politique de sécurité.

Cette approche est proposée dans le cadre du projet RNTL POSE[2] qui vise à valider la sécurité des systèmes embarqués de type carte à puce.

Peu de travaux similaires se sont intéressés à l'utilisation des modèles pour la génération du test du contrôle d'accès. L'approche classique est l'audit. Néanmoins, des approches ont été menées dans le cadre du test de contrôle d'accès dans des réseaux, une politique de sécurité décrivant ainsi un ensemble de permissions pour accéder à des services par des machines. L'approche présentée dans [12] utilise une formalisation par automates de Mealy pour décrire la politique de sécurité du réseau. Les cas de tests sont ensuite générés classiquement à partir de ces machines à états finis. Dans le cadre du projet POTESTAT [5], les auteurs formalisent le contrôle d'accès par un ensemble de formules temporelles, destinées à tester le contrôle d'accès d'un réseau. Le test de contrôle d'accès se prête bien au domaine des réseaux car il s'agit d'un système moins complexe à modéliser que le fonctionnement d'une application de type carte à puce. Ainsi, un modèle de sécurité suffit pour décrire une cible de test, le test en lui-même se résumant à envoyer des paquets sur le réseau, à partir d'une configuration précise (adresse IP, sous-réseau, etc.) et d'observer si les paquets sont acceptés ou rejetés. De ce fait, les attributs du sécurité du réseau sont complètement indépendants et ne sont pas sujet à des évolutions. Il n'est donc pas nécessaire de modéliser le réseau pour générer des tests. Notre approche se distingue des précédentes du fait que nous utilisons, en plus du modèle de sécurité, un modèle fonctionnel décrivant le système. Celui-ci nous permet d'être plus proche de l'implantation et de nous confronter à n'importe quel système, du moment que son comportement fonctionnel peut être modélisé.

Dans cet article, nous nous sommes intéressés à la notion de contrôle d'accès qui est une sous-classe de la classe FDP des CC. Nous pensons que ces travaux peuvent s'étendre à d'autres spécifications fonctionnelles de sécurité (SFR), telles que la gestion des droits d'accès (classe Administration de la sécurité – FMT) des CC, ou la protection des données (sous-classe FDP_RIP de la classe FDP).

D'autre part, nous souhaiterions formaliser la notion de couverture du contrôle d'accès. Nous avons, dans ce papier, considéré cette notion de manière intuitive et elle nous a guidé dans le choix de critères de couverture du modèle par l'outil LTG. L'objectif de cette étape est de pouvoir ensuite mesurer cette couverture et ainsi pouvoir évaluer différentes suites de test en fonction de celle-ci. En conséquence, nous pensons utiliser cette mesure de couverture pour filtrer des cas de tests, ou travailler sur la réduction de suites de tests, qui pourraient par exemple être produites par une approche combinatoire. Cette volonté est inscrite dans

[2] http ://www.rntl-pose.info

la sous-classe ATE_COV, de la classe Assurance des Tests (ATE) des Critères Communs (CC), qui décrit les exigences relatives à la complétude de la couverture des tests vis-à-vis des descriptions des fonctions de sécurité.

Références

1. J.-R. Abrial. *The B-book : assigning programs to meanings.* Cambridge University Press, 1996.
2. B. Beizer. *Black-Box Testing : Techniques for Functional Testing of Software and Systems.* John Wiley & Sons, New York, USA, 1995.
3. D. Elliot Bell and Leonard J. LaPadula. Secure computer systems : A mathematical model, volume ii. *Journal of Computer Security*, 4(2/3) :229–263, 1996.
4. K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachusetts, April 1977.
5. Vianney Darmaillacq, Jean-Claude Fernandez, Roland Groz, Laurent Mounier, and Jean-Luc Richier. Test generation for network security rules. In M. Ümit Uyar, Ali Y. Duale, and Mariusz A. Fecko, editors, *TestCom*, volume 3964 of *Lecture Notes in Computer Science*, pages 341–356. Springer, 2006.
6. Ferraiolo D.F. and Kuhn Richard. Role-Based Access Control. In *Proceedings of the 15th NIST-NSA National Computer Security Conference*, pages 554–563, Baltimore, MD, USA, October 1992. Nat'l Inst. Standards and Technology.
7. Amal Haddad. Meca : A tool for access control models. In *B'2007, the 7th Int. B Conference*, volume 4355 of *LNCS*, pages 281–284, Besançon, France, January 2007. Springer.
8. E. Jaffuel and B. Legeard. LEIRIOS Test Generator : Automated Test Generation from B Models. In *B'2007, the 7th Int. B Conference*, volume 4355 of *LNCS*, pages 277–281, Besançon, France, January 2007. Springer.
9. Butler W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1) :18–24, 1974.
10. B. Legeard, F. Peureux, and M. Utting. Controlling test case explosion in test generation from B formal models. *Software Testing, Verification and Reliability, STVR*, 14(2) :81–103, 2004.
11. A.J. Offutt, Y. Xiong, and S. Liu. Criteria for generating specification-based tests. In *Proceedings of the* 5th *IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'99)*, pages 119–131, Las Vegas, USA, October 1999. IEEE Computer Society Press.
12. Diana Senn, David A. Basin, and Germano Caronni. Firewall conformance testing. In Ferhat Khendek and Rachida Dssouli, editors, *TestCom*, volume 3502 of *Lecture Notes in Computer Science*, pages 226–241. Springer, 2005.
13. M. Utting and B. Legeard. *Practical Model-Based Testing : A Tools Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

Développement formel de circuits électroniques par la méthode B

Yann Zimmermann*

LORIA - équipe MOSEL - B.P. 239 - 54506 Vandœuvre-Lès-Nancy Cedex
yann.zimmermann@loria.fr

Résumé Cet article expose une méthode de modélisation de circuits électroniques synchrones basée sur la méthode B événementielle. Dans ce formalisme adapté à la modélisation de systèmes, un circuit est modélisé par un ensemble d'événements. Certains principes de modélisation ont été dégagés pour l'application de la méthode à la modélisation de circuits synchrones. Un ensemble d'outils permet de valider par la preuve le modèle et ensuite d'obtenir le code simulable et synthétisable VHDL ou SystemC du circuit. Les différentes phases de modélisation sont illustrées à l'aide d'un exemple de protocole d'accès à un bus géré par un arbitre.

1 Introduction

Dans le développement de systèmes corrects, on applique des méthodes formelles pour obtenir la correction du système par preuve. L'effort additionnel nécessaire pour la modélisation du système et la preuve des propriétés dans le développement est habituellement payant dans les environnements où la sécurité est critique, ou bien quand le coût d'un échec est très grand. Les systèmes électroniques devenant de plus en plus complexes, le besoin en méthodes formelles se fait de plus en plus sentir.

La méthode B [3] est l'une des méthodes formelles qui possède suffisamment d'outils supports pour être utilisée en pratique aujourd'hui. Son domaine d'application est le développement de logiciels corrects par construction. Elle est utilisée, par exemple, dans [4] pour la vérification de circuits par traduction du code VHDL en B. Cette démarche ne permet pas la modélisation à haut niveau d'abstraction. Notre approche est inverse : partir d'un modèle B de haut niveau et produire du VHDL.

Plus récemment, la méthode B a aussi été utilisée dans le domaine du développement de systèmes [5,6], donnant lieu au B événementiel [7,8]. Certains efforts ont été faits depuis pour l'appliquer également au développement de matériel [9,10], mais sans aller jusqu'à l'implantation. Notre but est de dégager les principes de modélisation qui permettent de modéliser des circuits électroniques synchrones et de les traduire vers des langages de description de circuits.

D'autres méthodes (et formalismes) utilisées dans la modélisation de matériel incluent le "model-checking" (par exemple basé sur les State Charts [11]), les

---

* Ce travail a été accompli dans le cadre d'une thèse [1] en convention CIFRE entre la société KeesDA [2] et le laboratoire LORIA.

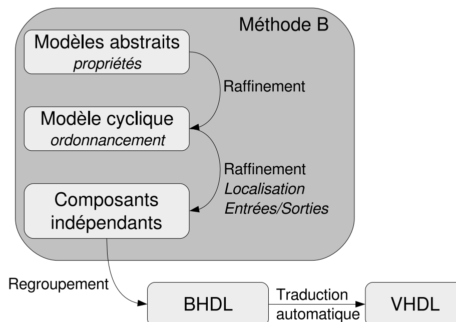
Actions Systems [12,13], la vérification (par exemple ACL2 [14], Coq [15]), ou la transformation (par exemple Alpha [16]). En comparaison de ces approches, le raffinement proposé par la méthode B a l'avantage de mieux faire face aux systèmes complexes car il permet de diviser la preuve de correction en parties plus facilement gérables.

Les principes de modélisation de circuits électroniques ont été dégagés à partir d'une étude de cas réaliste [17] choisie par Volvo dans le cadre du projet PUSSEE [18]. Cette étude consistait à modéliser un contrôleur d'accès à un bus série géré par un protocole distribué. Ce contrôleur a été traduit vers VHDL, simulé et synthétisé.

Notre méthodologie de développement est basée sur le raffinement. La démarche consiste à commencer par des modèles très abstraits, ce qui permet d'exprimer et de prouver des propriétés systèmes plus facilement et sans se soucier de l'implantation. Un invariant prouvé sur un modèle abstrait est préservé lors d'un raffinement. Le système est ensuite raffiné pas à pas en tenant compte des spécificités d'une implantation par circuit. La preuve accompagne l'ensemble du processus de développement.

Dans la suite de cet article, nous présentons d'abord les principes de la méthode B événementielle puis nous montrons les principes de développement de circuits synchrones en nous appuyant sur l'exemple d'un bus géré par un arbitre. Nous présentons d'abord une modélisation abstraite du système, puis nous montrons comment modéliser le cycle d'horloge, et enfin nous illustrons sur l'exemple comment séparer les composants du système. Ensuite nous expliquons comment obtenir un modèle BHDL®[1], à partir duquel il existe des traducteurs automatiques vers VHDL et SystemC. La démarche de modélisation est résumée sur la figure 1.

Fig. 1. Démarche de modélisation



[1] BHDL est une marque déposée par KeesDA

2 Modélisation par événements

En B événementiel, le système est observé de l'extérieur. L'observateur regarde le système et voit une succession d'événements. Chacun des événements modifie l'état du système. La modélisation événementielle consiste à spécifier quand l'observateur peut voir (observer) chaque événement, et quelles modifications sont apportées à l'état. Le système évolue de façon spontanée, on dit que les événements *se déclenchent*.

Suivant que l'observateur se trouve près ou loin du système, il distinguera plus ou moins de détails, le système sera modélisé à des niveaux d'abstraction différents.

Le modèle d'un système en B événementiel spécifie l'état du système et sa dynamique. La dynamique est modélisée en donnant une collection d'événements. Suivant l'état dans lequel se trouve le système, certains événements peuvent se déclencher, d'autres non.

2.1 État du système

L'état du système est caractérisé par une liste de variables. Quelle que soit l'évolution du système, l'état doit toujours satisfaire un *invariant* spécifié par le modélisateur. Cet invariant exprime la vue statique du système, c'est-à-dire ce qui ne change pas. En particulier, on y trouvera les types des variables.

Par ailleurs, une *initialisation* doit être donnée pour toutes les variables. L'initialisation fixe les valeurs portées par les variables quand aucun événement ne s'est encore produit.

2.2 Événements

Un événement est modélisé par une substitution généralisée [3]. En pratique, on se limitera aux trois formes d'événements ci-dessous, où $v, w, ...$ sont les variables d'état du système, P est un prédicat (la garde de l'événement) et S une substitution.

```
WHEN
  P(v, w, ...)
THEN
  S(v, w, ...)
END
```

```
ANY x, y, ... WHERE
  P(x, y, ..., v, w, ...)
THEN
  S(x, y, ..., v, w, ...)
END
```

```
BEGIN
  S(v, w, ...)
END
```

Un événement se divise en deux parties : sa garde et sa substitution. La garde est un prédicat sur l'état du système qui spécifie quand l'événement peut se déclencher. Lorsque deux événements ont leurs gardes vraies, un seul, dont le choix est non déterministe, se déclenche. La substitution spécifie de quelle manière l'événement modifie l'état du système. La troisième forme ci-dessus est le cas particulier où il n'y a pas de garde (garde toujours évaluée à *vrai*).

La première forme signifie que l'événement *peut* se déclencher lorsque P est évalué à *vrai*. En cas de déclenchement, la substitution S est appliquée.

Dans la deuxième forme, $x, y...$ sont des variables locales différentes des variables d'état. Ce sont des variables locales utilisées pour le calcul. Dans ce cas, le prédicat P ne constitue plus seulement une condition de déclenchement de l'événement. Il impose aussi des contraintes sur les variables locales qui ne sont pas nécessairement déterministes. Sous cette forme, l'événement peut se déclencher s'il est possible de donner aux variables locales des valeurs telles que P soit évalué à *vrai*. La garde d'un tel événement est $\exists(x, y, ...) \cdot P(x, y, ..., v, w...)$.

La troisième forme ci-dessus est le cas particulier où il n'y a pas de garde (garde toujours évaluée à vrai).

2.3 Obligations de preuve

Pour être correct, un modèle doit satisfaire un certain nombre de contraintes. Dans le cadre d'une méthode formelle comme B, ces contraintes sont vérifiées par preuve. La méthode B possède des outils permettant de générer automatiquement des obligations de preuve. Un assistant de preuve peut décharger une bonne partie des preuves automatiquement et permet une interaction avec l'utilisateur pour les preuves restantes.

Pour un système B, les obligations de preuve générées ont pour but d'assurer que :

- l'initialisation du système établit l'invariant,
- tout déclenchement d'un événement préserve l'invariant,
- les raffinements d'événements sont corrects : la garde concrète doit impliquer la garde abstraite et la substitution concrète est un raffinement de la garde abstraite.
- un raffinement n'introduit pas de possibilité de blocage dans le système,
- les nouveaux événements introduits dans un raffinement rendent toujours le contrôle aux anciens événements,
- tout nouvel événement doit être un raffinement de la substitution *skip* (substitution qui ne modifie pas l'état).

À chaque pas de raffinement, un *invariant de collage* est donné : il s'agit d'un invariant qui exprime la relation entre les nouvelles variables introduites par le raffinement et les variables du modèle plus abstrait.

3 Modélisation de circuits

Modéliser un système nécessite de décider quels sont les événements qui sont observés de l'extérieur. Le système peut être modélisé à différents niveaux d'abstraction. On peut modéliser une vue très abstraite en oubliant la nature physique du système. On peut aussi modéliser le système de façon très détaillée.

Il existe déjà des langages de description de circuits (VHDL, SystemC ...) à partir desquels des outils permettent la *synthèse* de la description très bas niveau. Notre objectif est d'obtenir un modèle du système qui puisse être traduit automatiquement vers des langages comme VHDL puis synthétisé.

Le niveau de détails que nous visons est d'observer les valeurs qui sont portées par les fils et les registres du circuit. C'est-à-dire que l'état sera constitué de variables représentant des fils et des registres. Les événements exprimeront de quelle manière les valeurs des variables sont modifiées par les composants. Ainsi, les événements représenteront le "code" (au sens VHDL) des composants.

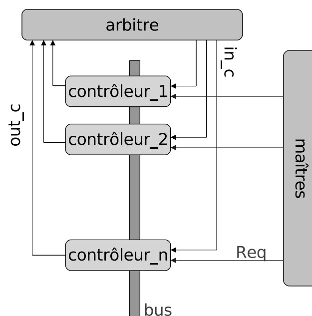
Les différentes phases de modélisation seront illustrées sur l'exemple d'un protocole d'accès à un bus géré par un arbitre : on commence par faire une modélisation abstraite du système orientée par les propriétés qu'on souhaite garantir. Ensuite le système est raffiné de façon à introduire le cycle d'horloge. Le système est de nouveau raffiné de façon à séparer les différents composants du système et enfin les entrées sont modélisées de façon explicite.

3.1 Présentation de l'exemple

Le système (cf. figure 2) consiste en un ensemble de composants souhaitant envoyer des messages sur un bus. Pour éviter les conflits, le bus est géré par un arbitre et les composants passent par des contrôleurs pour accéder au bus. Notre but est de modéliser les contrôleurs, le reste (dont l'arbitre) faisant partie de l'environnement des contrôleurs.

Les contrôleurs reçoivent des requêtes (*Req*) de la part de maîtres lorsque ceux-ci souhaitent envoyer un message. Ils envoient des demandes d'utilisation du bus à l'arbitre via les sorties *out_c* et reçoivent les réponses via les entrées *in_c*.

Fig. 2. Exemple d'un bus avec arbitre



Le bus est une partie analogique du système, tous les contrôleurs envoient en permanence une valeur sur le bus. On considère que le bus est dominé par 0. C'est-à-dire que si au moins un contrôleur envoie 0 sur le bus, la valeur du bus est 0. Pour que le bus soit dans l'état 1 (état récessif) il faut que tous les contrôleurs envoient 1 sur le bus. Lorsqu'un composant n'a rien à envoyer sur le bus, il envoie la valeur 1.

3.2 États des composants

Le B événementiel ne permet pas de séparer les modèles des différents composants du système. Au contraire, à des niveaux abstraits, on manipule un état global : les états des différents composants ne sont pas séparés. Au niveau de l'implantation il faut que chaque composant ait son propre état, c'est-à-dire un ensemble de variables qu'il est le seul à pouvoir modifier. De la même manière chaque composant doit avoir son propre ensemble d'événements, chargé de manipuler ses propres variables.

Dans notre exemple, nous avons affaire à plusieurs composants identiques : les contrôleurs. Le code ne sera pas dupliqué autant de fois qu'il y a de contrôleurs, au contraire le nombre de contrôleurs peut rester non déterminé (mais en nombre fini). Pour cela nous déclarons un ensemble abstrait *CTRL* qui représente l'ensemble des contrôleurs. Pour modéliser l'état des contrôleurs nous utilisons des variables de type fonction : par exemple pour modéliser les sorties du contrôleur vers le bus on utilisera la variable *cbus* de type $CTRL \rightarrow \{0, 1\}$; ainsi, pour un contrôleur c, $cbus(c)$ est la valeur envoyée sur le bus par c.

3.3 Modélisation abstraite

On modélise d'abord le système en utilisant les propriétés que l'on souhaite exprimer. La propriété que l'on souhaite garantir dans notre exemple est que le bus est utilisé par au plus un contrôleur à la fois. Pour cela, on utilise une variable *bus* modélisant l'ensemble des contrôleurs utilisant le bus. La propriété s'exprime par un invariant disant que la variable *bus* est soit vide soit un singleton, ce qui peut s'exprimer par le premier invariant ci-dessous : si l'ensemble *bus* contient deux éléments ils sont nécessairement égaux. Par ailleurs on souhaite également garantir que seul le contrôleur ayant le contrôle du bus peut envoyer des 0 sur le bus ; un composant n'ayant pas le contrôle du bus doit envoyer 1 sur le bus. La propriété s'exprime par le deuxième invariant ci-dessous : si un composant envoie 0 sur le bus, il est nécessairement dans l'ensemble *bus*.

$$\begin{array}{l}\text{INVARIANT}\\ \quad \forall(c1, c2) \cdot (c1 \in bus \wedge c2 \in bus \Rightarrow c1 = c2) \wedge \\ \quad \forall c \cdot (c \in CTRL \wedge cbus(c) = 0 \Rightarrow c \in bus)\end{array}$$

La variable *bus* est initialisée à l'ensemble vide pour indiquer qu'aucun contrôleur n'utilise le bus initialement. Les sorties sur le bus sont initialisées à 1 puisqu'aucun contrôleur n'utilise le bus.

$$\begin{array}{l}\text{INITIALISATION}\\ \quad bus := \varnothing \parallel cbus := CTRL \times \{1\}\end{array}$$

Le modèle lui-même est constitué de quatre événements. L'événement *take* permet, lorsque le bus est libre ($bus = \varnothing$), à un contrôleur de prendre le bus. Dans ce modèle, le choix du contrôleur est non déterministe. L'événement *send* permet au contrôleur ayant le contrôle du bus ($c \in bus$) d'envoyer des données

sur le bus, la donnée envoyée est choisie de manière non déterministe ($cbus(c) :\in \{0,1\}$[2] signifie que $cbus(c)$ prend une des deux valeurs 0 ou 1). L'événement *free* permet au contrôleur utilisant le bus de le libérer ($bus := \varnothing$). Enfin, l'événement *nothing* modélise le comportement des contrôleurs n'ayant pas le bus : ils envoient toujours la valeur 1 sur le bus.

take $=$
ANY c WHERE
$c \in CTRL \wedge$
$bus = \varnothing$
THEN
$bus := \{c\} \parallel$
$cbus(c) := 1$
END

send $=$
ANY c WHERE
$c \in bus$
THEN
$cbus(c) :\in \{0,1\}$
END

free $=$
ANY c WHERE
$c \in bus$
THEN
$bus := \varnothing \parallel$
$cbus(c) := 1$
END

nothing $=$
ANY c WHERE
$c \in CTRL - bus$
THEN
$cbus(c) := 1$
END

Dans notre approche de la modélisation de circuits, nous nous imposons que dans tous les modèles (y compris les modèles abstraits) les comportements de tous les composants dans toutes les situations soient modélisés. Cela nous semble important car dans un système électronique, un composant ne peut pas se bloquer, un composant électronique fait toujours quelque chose. Si on n'avait pas l'événement *nothing* dans notre exemple, le comportement des contrôleurs n'ayant pas le bus ne serait pas modélisé lorsque $bus \neq \varnothing$. Avec l'événement *nothing*, notre modèle stipule que les contrôleurs qui n'ont pas le bus envoient 1 sur le bus, ce qui est une part non négligeable de la spécification.

3.4 Modélisation du cycle d'horloge

Cette section montre comment on peut raffiner l'exemple précédent pour se diriger vers une implantation des contrôleurs.

Notre objectif est de modéliser des circuits électroniques synchrones. Ce type de circuit est gouverné par une entrée particulière : l'horloge, qui est cyclique. Il s'en suit que le circuit a lui-même un comportement cyclique. En un cycle, le circuit prend de nouvelles entrées et calcule sont état interne ainsi que les sorties (la fréquence de l'horloge doit être réglée pour cela).

Par ailleurs il peut exister des communications à l'intérieur d'un cycle. Pour modéliser cela, nous divisons le cycle en périodes successives, tous les composants se situant dans la même période à un moment donné. On interdit toute communication entre composants à l'intérieur d'une même période, les communications se font d'une période à une autre. Une période est un moment de concurrence entre les composants.

Pour modéliser ce cycle divisé en périodes successives, nous introduisons un ordonnancement entre les événements. Nous utilisons pour cela des variables d'ordonnancement qui sont des ensembles de composants. À la période numéro i correspond une variable E_i. Lorsqu'un composant se trouve dans l'ensemble

[2] Formellement la notation $cbus(c) :\in \{0,1\}$ n'est pas correcte en B. Il faudrait écrire ANY x WHERE $x \in \{0,1\}$ THEN $cbus(c) := x$ END

E_i cela signifie que ce composant doit accomplir la période i, qu'il ne pourra accomplir que lorsque tous les composants auront accompli la période précédente ; lorsqu'un composant accomplit la période i, il passe de l'ensemble E_i à l'ensemble E_{i+1} (ou E_0 s'il s'agissait de la dernière période). Un événement de la période numéro i prend la forme nommée *event_i* sur la figure 3.

On s'autorise cependant une simplification dans la modélisation de l'environnement : on permet aux événements modélisant l'environnement de faire basculer tous les composants de E_e à E_{e+1} d'un seul coup, e étant le numéro de la période correspondant à l'environnement, ceci est décrit par l'événement nommé *env* sur la figure 3.

Fig. 3. Modélisation de l'ordonnancement

event_i =	**env =**
ANY c WHERE	WHEN
$c \in E_i \wedge$	$E_e = CTRL \wedge$
$E_{i-1} = \varnothing \wedge$	$E_{e-1} = \varnothing \wedge$
...	...
THEN	THEN
... $\|$	... $\|$
$E_i := E_i - \{c\} \|$	$E_e := \varnothing \|$
$E_{i+1} := E_{i+1} \cup \{c\}$	$E_{e+1} := CTRL$
END	END

Dans notre exemple, nous raffinons notre modèle abstrait en introduisant un cycle divisé en quatre périodes. Une première période pour modéliser la mise à jour des requêtes des maîtres aux contrôleurs. La deuxième période est le traitement des requêtes pour fournir les demandes auprès de l'arbitre. La troisième période est celle où l'arbitre (que nous considérons faire partie de l'environnement) traite les demandes et fournit l'accès à au plus un des composants. Enfin, la quatrième période est celle où les contrôleurs récupèrent les réponses de l'arbitre et envoient des données sur le bus. Le modèle contient donc deux périodes pour l'environnement (la première et la troisième) et deux périodes pour les contrôleurs.

Ce raffinement introduit sept nouvelles variables : la variable *Req* qui l'ensemble des composants faisant une requête d'accès au bus, *out_c* qui modélise les sorties des contrôleurs vers l'arbitre, in_c qui modélise les sorties de l'arbitre vers les contrôleurs, et les variables d'ordonnancement $E0$, E1, $E2$ et $E3$.

Première période La première période est modélisée par un seul événement pour l'environnement (il traite tous les composants d'un seul coup). Cet événement modélise la mise à jour des requêtes des maîtres aux contrôleurs. Les requêtes sont représentées par l'ensemble *Req* contenant l'ensemble des composants faisant une requête. La substitution $Req : (Req \subseteq CTRL)$ signifie que *Req* devient un sous-ensemble de $CTRL$.

```
env =
  WHEN
    E0 = CTRL ∧ E3 = ∅
  THEN
    Req : (Req ⊆ CTRL) ||
    E0 := ∅ ||
    E1 := CTRL
  END
```

Deuxième période La deuxième période est constituée de trois événements. Si le bus est utilisé ($bus \neq \varnothing$) c'est l'événement *req_no* qui se déclenche pour tous les contrôleurs, faisant ainsi en sorte qu'aucune demande n'est envoyée à l'arbitre. Si le bus n'est pas utilisé, l'événement *req_ok* permet aux contrôleurs faisant partie des requêtes ($c \in Req$) d'envoyer une demande à l'arbitre ($out_c(c) := 1$). L'événement *req_ko* stipule que les contrôleurs ne faisant pas partie des requêtes ($c \in CTRL - Req$) ne font pas de demande auprès de l'arbitre ($out_c(c) := 0$).

```
req_ok =
  ANY c WHERE
    c ∈ E1 ∧ E0 = ∅ ∧
    c ∈ Req ∧
    bus = ∅
  THEN
    out_c(c) := 1 ||
    E1 := E1 − {c} ||
    E2 := E2 ∪ {c}
  END

req_ko =
  ANY c WHERE
    c ∈ E1 ∧ E0 = ∅ ∧
    c ∈ CTRL − Req ∧
    bus = ∅
  THEN
    out_c(c) := 0 ||
    E1 := E1 − {c} ||
    E2 := E2 ∪ {c}
  END

req_no =
  ANY c WHERE
    c ∈ E1 ∧ E0 = ∅ ∧
    bus ≠ ∅
  THEN
    out_c(c) := 0 ||
    E1 := E1 − {c} ||
    E2 := E2 ∪ {c}
  END
```

Troisième période La troisième période modélise l'arbitrage en deux événements. Si l'arbitre a reçu au moins une demande (il existe un c tel que $out_c(c) = 1$) alors l'événement *arb_ok* donne le bus à l'un de ces contrôleurs ($in_c(c)$ prend la valeur 1, tous les autres in_c prennent la valeur 0). Notons que le choix du contrôleur auquel on donne l'accès est non déterministe. Si l'arbitre n'a reçu aucune demande ($out_c \rhd \{1\} = \varnothing$) l'événement *arb_ko* ne donne le bus à personne (tous les in_c sont mis à 0).

```
arb_ok =
  ANY c WHERE
    E2 = CTRL ∧ E1 = ∅ ∧
    c ∈ CTRL ∧
    out_c(c) = 1
  THEN
    in_c := CTRL × {0} <+ {c ↦ 1} ||
    E2 := ∅ ||
    E3 := CTRL
  END

arb_ko =
  WHEN
    E2 = CTRL ∧ E1 = ∅ ∧
    out_c ▷ {1} = ∅
  THEN
    in_c := CTRL × {0} ||
    E2 := ∅ ||
    E3 := CTRL
  END
```

Quatrième période Nous retrouvons dans la quatrième période les quatre événements qui étaient déjà présents dans l'abstraction. Outre l'ajout de l'ordonnancement à tous les événements, les deux événements *take* et *nothing* sont raffinés. L'événement *nothing* est raffiné simplement en rendant sa garde plus forte en ajoutant la condition $in_c(c) = 0$, c'est-à-dire qu'il ne pourra se déclencher que pour les contrôleurs auxquels l'arbitre n'a pas attribué le bus. La garde de l'événement *take* est raffinée en ajoutant la condition $in_c(c) = 1$ et en enlevant la condition $bus = \varnothing$, l'obligation de preuve générée est donc la suivante :

$$c \in E3 \wedge E2 = \varnothing \wedge in_c(c) = 1 \Rightarrow bus = \varnothing$$

dont la preuve nécessite de prouver les trois invariants ci-dessous, on voit que l'obligation de preuve est une conséquence directe du premier invariant. Le prédicat $singleton(X)$ signifie que l'ensemble X est un singleton.

$$
\begin{array}{l}
\text{INVARIANT} \\
\quad \forall c \cdot (c \in E3 \wedge in_c(c) = 1 \Rightarrow bus = \varnothing) \wedge \\
\quad \forall c \cdot (c \in E2 \wedge in_c(c) = 1 \Rightarrow bus = \varnothing) \wedge \\
\quad (E3 \neq \varnothing \Rightarrow (in_c \rhd \{1\} \neq \varnothing \Rightarrow singleton(in_c \rhd \{1\})))
\end{array}
$$

$$
\begin{array}{l}
\textbf{take} = \\
\quad \text{ANY } c \text{ WHERE} \\
\quad\quad c \in E3 \wedge E2 = \varnothing \wedge \\
\quad\quad \boldsymbol{in_c(c) = 1} \\
\quad \text{THEN} \\
\quad\quad bus := \{c\} \parallel \\
\quad\quad cbus(c) := 1 \\
\quad\quad E3 := E3 - \{c\} \parallel \\
\quad\quad E0 := E0 \cup \{c\} \\
\quad \text{END}
\end{array}
\qquad
\begin{array}{l}
\textbf{send} = \\
\quad \text{ANY } c \text{ WHERE} \\
\quad\quad c \in E3 \wedge E2 = \varnothing \wedge \\
\quad\quad c \in bus \\
\quad \text{THEN} \\
\quad\quad cbus(c) :\in \{0, 1\} \parallel \\
\quad\quad E3 := E3 - \{c\} \parallel \\
\quad\quad E0 := E0 \cup \{c\} \\
\quad \text{END}
\end{array}
$$

$$
\begin{array}{l}
\textbf{free} = \\
\quad \text{ANY } c \text{ WHERE} \\
\quad\quad c \in E3 \wedge E2 = \varnothing \wedge \\
\quad\quad c \in bus \\
\quad \text{THEN} \\
\quad\quad bus := \varnothing \parallel \\
\quad\quad cbus(c) := 1 \\
\quad\quad E3 := E3 - \{c\} \parallel \\
\quad\quad E0 := E0 \cup \{c\} \\
\quad \text{END}
\end{array}
\qquad
\begin{array}{l}
\textbf{nothing} = \\
\quad \text{ANY } c \text{ WHERE} \\
\quad\quad c \in E3 \wedge E2 = \varnothing \wedge \\
\quad\quad c \in CTRL - bus \wedge \\
\quad\quad \boldsymbol{in_c(c) = 0} \\
\quad \text{THEN} \\
\quad\quad cbus(c) := 1 \\
\quad\quad E3 := E3 - \{c\} \parallel \\
\quad\quad E0 := E0 \cup \{c\} \\
\quad \text{END}
\end{array}
$$

Initialisation Nous ajoutons à l'initialisation les variables Req, out_c, in_c et les variables d'ordonnancement. On initialise $E0$ à $CTRL$ pour indiquer qu'on commence par la période $E0$, les autres variables d'ordonnancement étant initialisées à l'ensemble vide.

$$
\begin{array}{l}
\text{INITIALISATION} \\
\quad bus := \varnothing \parallel cbus :\in CTRL \rightarrow \{1\} \parallel Req : (Req \subseteq CTRL) \parallel \\
\quad out_c :\in CTRL \rightarrow \{0, 1\} \parallel in_c :\in CTRL \rightarrow \{0, 1\} \parallel \\
\quad E0, E1, E2, E3 := CTRL, \varnothing, \varnothing, \varnothing
\end{array}
$$

3.5 Séparation des composants

Dans les modèles abstraits, certaines variables peuvent modéliser un état global du système et pas l'état d'un seul composant. Il faut alors raffiner le modèle pour remplacer cette variable par une ou plusieurs variables, chacune appartenant à un seul composant. On appelle aussi cette étape *localisation* car il s'agit de faire du raffinement de données pour obtenir un modèle où les données sont dans des états locaux aux composants.

Dans notre exemple, il y a deux variables à remplacer. D'une part la variable *bus* qui représente l'ensemble des contrôleurs utilisant le bus. Nous la remplaçons par deux variables : la variable *bus_used* indiquant si le bus est utilisé ou non, cette variable est mise à jour par l'environnement en début de cycle, et la variable *bbus* qui est telle que $bbus(c) = 1$ si et seulement si $c \in bus$. D'autre part la variable *Req* représentant l'ensemble des requêtes est remplacée par la variable *bReq* qui est telle que $bReq(c) = 1$ si et seulement si $c \in Req$. Les invariants de collage sont donnés ci-dessous.

$$
\begin{array}{l}
\text{INVARIANT} \\
\quad (E1 \neq \varnothing \Rightarrow (bus_used = bool(bus \neq \varnothing))) \wedge \\
\quad (E3 \neq \varnothing \Rightarrow bus = dom(bbus \rhd \{1\}) \wedge \\
\quad Req = dom(bReq \rhd \{1\})
\end{array}
$$

Dans les événements ci-dessous, les modifications par rapport au modèle abstrait sont écrites en **gras**.

La première période est raffinée de façon à prendre en compte le remplacement de *Req* par *bReq* ainsi que la mise à jour de la nouvelle variable *bus_used*.

$$
\begin{array}{l}
\textbf{env} = \\
\quad \text{WHEN} \\
\qquad E0 = CTRL \;\wedge E3 = \varnothing \\
\quad \text{THEN} \\
\qquad \boldsymbol{bReq :\in CTRL \rightarrow \{0,1\}} \parallel \\
\qquad \boldsymbol{bus_used := bool(dom(bbus \rhd \{1\}) \neq \varnothing)} \parallel \\
\qquad E0 := \varnothing \parallel \\
\qquad E1 := CTRL \\
\quad \text{END}
\end{array}
$$

On raffine la deuxième période en opérant le remplacement des conditions du type $c \in Req$ par $bReq(c) = 1$ ainsi qu'en remplaçant les conditions du type $bus = \varnothing$ par $bus_used = FALSE$.

$$
\begin{array}{l}
\textbf{req_ok} = \\
\quad \text{ANY } c \text{ WHERE} \\
\qquad c \in E1 \wedge E0 = \varnothing \wedge \\
\qquad \boldsymbol{bReq(c) = 1} \wedge \\
\qquad \boldsymbol{bus_used} = FALSE \\
\quad \text{THEN} \\
\qquad out_c(c) := 1 \parallel \\
\qquad E1 := E1 - \{c\} \parallel \\
\qquad E2 := E2 \cup \{c\} \\
\quad \text{END}
\end{array}
$$

$$
\begin{array}{l}
\textbf{req_ko} = \\
\quad \text{ANY } c \text{ WHERE} \\
\qquad c \in E1 \wedge E0 = \varnothing \wedge \\
\qquad \boldsymbol{bReq(c) = 0} \wedge \\
\qquad \boldsymbol{bus_used} = FALSE \\
\quad \text{THEN} \\
\qquad out_c(c) := 0 \parallel \\
\qquad E1 := E1 - \{c\} \parallel \\
\qquad E2 := E2 \cup \{c\} \\
\quad \text{END}
\end{array}
$$

$$
\begin{array}{l}
\textbf{req_no} = \\
\quad \text{ANY } c \text{ WHERE} \\
\qquad c \in E1 \wedge E0 = \varnothing \wedge \\
\qquad \boldsymbol{bus_used} = TRUE \\
\quad \text{THEN} \\
\qquad out_c(c) := 0 \parallel \\
\qquad E1 := E1 - \{c\} \parallel \\
\qquad E2 := E2 \cup \{c\} \\
\quad \text{END}
\end{array}
$$

Enfin, il faut raffiner les événements de la quatrième période (cf. figure 4) de façon à remplacer les conditions de la forme $c \in bus$ par $bbus(c) = 1$ et en écrivant la substitution $bbus(c) := 1$ à la place de $bus := \{c\}$ dans l'événement *take* et $bbus(c) := 0$ à la place de $bus := \varnothing$ dans l'événement *free*. Remarquons que dans l'événement *free* la substitution $bbus(c) := 0$ correspond plus directement à la substitution $bus := bus - \{c\}$, cependant comme on sait que *bus* est exactement le singleton $\{c\}$ (car on sait que $c \in bus$ par la condition $bbus(c) = 1$ et que *bus* est au plus un singleton), cela est bien équivalent à $bus := \varnothing$.

Fig. 4. Raffinement de la quatrième période

take = ANY c WHERE $c \in E3 \wedge E2 = \varnothing \wedge$ $in_c(c) = 1$ THEN $\boldsymbol{bbus(c) := 1} \parallel$ $cbus(c) := 1$ $E3 := E3 - \{c\} \parallel$ $E0 := E0 \cup \{c\}$ END	**send =** ANY c WHERE $c \in E3 \wedge E2 = \varnothing \wedge$ $\boldsymbol{bbus(c) = 1}$ THEN $cbus(c) :\in \{0, 1\} \parallel$ $E3 := E3 - \{c\} \parallel$ $E0 := E0 \cup \{c\}$ END
free = ANY c WHERE $c \in E3 \wedge E2 = \varnothing \wedge$ $\boldsymbol{bbus(c) = 1}$ THEN $\boldsymbol{bbus(c) := 0} \parallel$ $cbus(c) := 1$ $E3 := E3 - \{c\} \parallel$ $E0 := E0 \cup \{c\}$ END	**nothing =** ANY c WHERE $c \in E3 \wedge E2 = \varnothing \wedge$ $\boldsymbol{bbus(c) = 0} \wedge$ $in_c(c) = 0$ THEN $cbus(c) := 1$ $E3 := E3 - \{c\} \parallel$ $E0 := E0 \cup \{c\}$ END

3.6 Explicitation des entrées

En observant le modèle obtenu dans la section précédente on peut se rendre compte qu'il existe encore deux cas de non déterminisme. D'une part les deux événements *send* et *free* ont la même garde, d'autre part on a l'affectation non déterministe $cbus :\in \{0, 1\}$ dans l'événement *send*.

Ce non déterminisme est typiquement un moyen de modéliser implicitement des entrées. Pour les modéliser explicitement on ajoute deux variables : la variable *msg_c* qui indique si le message est terminé ou pas et la variable *binp* qui est la donnée à envoyer sur le bus.

$$
\begin{array}{l}
\textbf{env} = \\
\quad \text{WHEN} \\
\qquad E0 = CTRL \ \wedge E3 = \varnothing \\
\quad \text{THEN} \\
\qquad bReq :\in CTRL \rightarrow \{0,1\} \parallel \\
\qquad bus_used := bool(dom(bbus \rhd \{1\}) \neq \varnothing) \parallel \\
\qquad \boldsymbol{binp :\in} CTRL \boldsymbol{\rightarrow \{0,1\}} \parallel \\
\qquad \boldsymbol{msg_c :\in} CTRL \boldsymbol{\rightarrow \{0,1\}} \parallel \\
\qquad E0 := \varnothing \parallel \\
\qquad E1 := CTRL \\
\quad \text{END}
\end{array}
$$

La condition $msg_c(c) = 0$ signifie qu'on souhaite continuer à envoyer des données, elle est mise en garde de l'événement *send*, son opposée est mise en garde de l'événement *free*. Par ailleurs, la sortie sur le bus *cbus* est maintenant modifiée de manière déterministe en lui assignant la valeur de l'entrée *binp*.

$$
\begin{array}{l}
\textbf{send} = \\
\quad \text{ANY } c \text{ WHERE} \\
\qquad c \in E3 \wedge E2 = \varnothing \ \wedge \\
\qquad bbus(c) = 1 \ \wedge \\
\qquad \boldsymbol{msg_c(c) = 0} \\
\quad \text{THEN} \\
\qquad \boldsymbol{cbus(c) := binp(c)} \parallel \\
\qquad E3 := E3 - \{c\} \parallel \\
\qquad E0 := E0 \cup \{c\} \\
\quad \text{END}
\end{array}
\qquad
\begin{array}{l}
\textbf{free} = \\
\quad \text{ANY } c \text{ WHERE} \\
\qquad c \in E3 \wedge E2 = \varnothing \ \wedge \\
\qquad bbus(c) = 1 \ \wedge \\
\qquad \boldsymbol{msg_c(c) = 1} \\
\quad \text{THEN} \\
\qquad bbus(c) := 0 \parallel \\
\qquad cbus(c) := 1 \\
\qquad E3 := E3 - \{c\} \parallel \\
\qquad E0 := E0 \cup \{c\} \\
\quad \text{END}
\end{array}
$$

Le seul non déterminisme qui reste dans le modèle B se trouve dans les événements modélisant l'environnement : non déterminisme dans le choix des entrées dans *env* et dans le choix du contrôleur qui obtient le bus dans *arb_ok*.

4 Bilan sur les preuves

Le développement de cet exemple a été réalisé en six modèles : le modèle initial plus cinq raffinements. Par rapport aux modèles présentés dans ce papier, il y a un raffinement de plus dans le tableau ci-dessous ; ceci est dû au fait que nous avons utilisé deux raffinements pour d'une part introduire la variable *bus_used* et d'autre part supprimer la variable *bus*. Le tableau ci-dessous résume le nombre d'obligations de preuves pour chaque modèle. L'avant dernière colonne indique pour chaque modèle le nombre de preuves déchargées automatiquement, la colonne précédente correspond au nombre de preuves faites de manière interactive. Pour réaliser les preuves, nous avons utilisé l'outil B4free et son interface Click'n'Prove. La dernière colonne indique le nombre moyens de "clics" par preuve interactive, elle donne une indication de la difficulté des preuves

Raffinement	Obligations de preuve	interactives	automatiques	Taille moyenne
0	16	4	12	3,75
1	105	9	96	3,66
2	13	2	11	3,5
3	37	15	22	8,8
4	8	3	5	15
5	10	3	7	4,33
Total	189	19%	81%	

5 Traduction

Les outils de traduction vers VHDL et SystemC développés par KeesDA [2] fonctionnent à partir d'un modèle appelé BHDL qui est un modèle B déterministe et constitué d'un seul événement. Pour obtenir ce modèle BHDL, nous regroupons les événements en un seul. Ceci se fait en deux étapes : d'abord le regroupement des événements de chaque période de façon à obtenir un seul événement par période, et ensuite l'implantation de l'ordonnancement.

Le regroupement des événements d'une même période se fait selon la règle ci-dessous introduisant le IF [19]. Cette règle ne modifie pas la sémantique du système.

$$\frac{\begin{array}{l}\text{WHEN } P \wedge Q \text{ THEN } S \text{ END}\\ \text{WHEN } P \wedge \neg Q \text{ THEN } T \text{ END}\end{array}}{\begin{array}{l}\text{WHEN } P \text{ THEN}\\ \quad \text{IF } Q \text{ THEN } S \text{ ELSE } T \text{ END}\\ \text{END}\end{array}} \qquad \frac{\begin{array}{l}\text{ANY } c \text{ WHERE } P \wedge Q \text{ THEN } S \text{ END}\\ \text{ANY } c \text{ WHERE } P \wedge \neg Q \text{ THEN } T \text{ END}\end{array}}{\begin{array}{l}\text{ANY } c \text{ WHERE } P \text{ THEN}\\ \quad \text{IF } Q \text{ THEN } S \text{ ELSE } T \text{ END}\\ \text{END}\end{array}}$$

Pour pouvoir appliquer cette règle sur la troisième période de notre exemple, on peut reformuler l'événement *arb_ok* en WHEN $E2 = CTRL \wedge E1 = \varnothing \wedge out_c \rhd \{1\} \neq \varnothing$ THEN $in_c : (\exists c \cdot (c \in dom(out_c \rhd \{1\}) \wedge in_c = CTRL \times \{0\} \Leftarrow \{c \mapsto 1\}))\|E2 := \varnothing\|E3 := CTRL$ END

Une fois le regroupement des événements de chaque période effectué, nous avons quatre événements dont les gardes ne contiennent que des variables d'ordonnancement. Si ce n'est pas le cas c'est que le modèle ne respecte pas la règle de modélisation selon laquelle doivent être modélisés explicitement les comportements de tous les composants dans toutes les situations.

La deuxième étape consiste à supprimer les variables d'ordonnancement L'ordonnancement exprimé ici est une séquentialité dans le déclenchement des événements, ce qui s'implante en BHDL en utilisant la composition séquentielle.

Afin de donner une règle générale pour ce regroupement séquentiel, nous opérons la transformation suivante sur le modèle : nous considérons que l'ensemble $CTRL$ est un singleton $\{c\}$, par conséquent nous transformons les gardes ANY c WHERE $E_k = \varnothing \ \wedge \ c \in E_{k+1}$ en WHEN $E_k = \varnothing \ \wedge \ E_{k+1} = CTRL$ et les substitutions $E_k := E_k - \{c\}$ (resp. $E_k := E_k \cup \{c\}$) en $E_k := \varnothing$ (resp. $E_k := CTRL$). Cette transformation trouve sa justification dans le fait que nous construisons le modèle BHDL d'*un* contrôleur et non pas du système entier. De plus, comme nous avions plusieurs composants identiques, nous avions utilisé des

variables de type fonction pour représenter l'état des composants. Maintenant nous souhaitons traduire le code d'un seul composant. Nous changeons donc les variables du type fonction $CTRL \rightarrow T$ vers des variables simples du type T. Pour cela, il suffit simplement de remplacer tous les termes du type $v(c)$ par v dans les événements.

Pour effectuer le regroupement on utilise la règle générale donnée ci-dessous. Les indices $k-1$, $m+1$ et $p+1$ étant entendus modulo le nombre de périodes.

$$\frac{\begin{array}{l} evA = \textsc{when}\ E_{k-1} = \varnothing\ \wedge\ E_k = CTRL \\ \qquad\qquad \textsc{then}\ S \parallel E_m := \varnothing \parallel E_{m+1} := CTRL\ \textsc{end} \\ evB = \textsc{when}\ E_m = \varnothing\ \wedge\ E_{m+1} = CTRL \\ \qquad\qquad \textsc{then}\ T \parallel E_p := \varnothing \parallel E_{p+1} := CTRL\ \textsc{end} \\ S^\star = skip \text{ si } evA \text{ fait partie de l'environnement, } S \text{ sinon} \\ T^\star = skip \text{ si } evB \text{ fait partie de l'environnement, } T \text{ sinon} \end{array}}{\begin{array}{l} \textsc{when}\ E_{k-1} = \varnothing\ \wedge\ E_k = CTRL \\ \qquad \textsc{then}\ S^\star ; T^\star \parallel E_p := \varnothing \parallel E_{p+1} := CTRL\ \textsc{end} \end{array}}$$

On applique cette règle de façon répétitive en commençant par $(k, m, p) = (0, 0, 1)$ puis $(0, 1, 2)$ et enfin $(0, 2, 3)$.

Dans le modèle B, des événements (ceux de l'arbitre) se déclenchaient entre la deuxième et la quatrième période. Pour que cette implantation de l'ordonnancement séquentiel soit correcte il faut que la fréquence de l'horloge soit calculée (par l'outil de synthèse) de façon à laisser le temps à l'arbitre de faire ses calculs.

Le modèle BHDL final d'un contrôleur est donné sur la figure 5. À partir de ce modèle, les traducteurs développés par KeesDA [2] sont capables de produire automatiquement des modèles VHDL ou SystemC d'un contrôleur. La correction des traducteurs a été prouvée dans [1]. Par rapport à un modèle B, une machine BHDL a les deux nouvelles clauses INPUTS et OUTPUTS qui contiennent respectivement les entrées et les sorties. On a comme condition qu'une entrée ne doit être que lue et qu'une sortie doit être toujours écrite. Le type BIT est simplement l'ensemble $\{0, 1\}$, les entrées et les sorties sont initialisées de manière non déterministe.

On peut constater que dans un modèle BHDL, comme dans nos modèles B, il n'est pas fait distinction entre les fils et les registres du circuit. Cette distinction se fait automatiquement lors de la traduction vers VHDL ou SystemC : un registre est une variable qui est lue avant d'être écrite alors qu'un fil est modélisé par une variable qui est n'est jamais lue avant d'être écrite.

Ce modèle BHDL est obtenu à partir du modèle d'un système B dans lequel figurait le modèle d'un arbitre. Le bon fonctionnement du contrôleur obtenu est dépendant de l'arbitre : celui-ci doit être conforme au modèle abstrait donné dans le système B. On peut par exemple obtenir un arbitre en continuant le raffinement de façon à obtenir le code implantable d'un arbitre.

6 Autres approches

Notre approche de la modélisation de circuits utilisant des variables d'ordonnancement est une généralisation de [5] qui autorise la modélisation de systèmes

Fig. 5. Modèle BHDL d'un contrôleur

```
MACHINE controler
INPUTS bus_used, bReq, in_c, binp
OUTPUTS out_c, cbus
VARIABLES bbus
INVARIANT
  bus_used ∈ BOOL ∧ bReq ∈ BIT ∧ in_c ∈ BIT∧
  binp ∈ BIT ∧ out_c ∈ BIT ∧ cbus ∈ BIT ∧ bbus ∈ BIT
INITIALISATION
  bus_used :∈ BOOL || bReq :∈ BIT || in_c :∈ BIT ||
  binp :∈ BIT || out_c :∈ BIT || cbus :∈ BIT || bbus := 0
EVENTS
                                         BEGIN
                                           IF bbus = 0 THEN
BEGIN                                        IF in_c = 1 THEN
  IF bus_used = FALSE THEN                     bbus := 1
    IF bReq = 1 THEN                         END ||
      out_c := 1                             cbus := 1
    ELSE                                ;  ELSE
      out_c := 0                             IF msg_c = 0 THEN
    END                                        cbus := binp
  ELSE                                       ELSE
    out_c := 0                                 bbus := 0||cbus := 1
  END                                        END
END                                        END
                                         END
```

constitués de plus d'un composant. Dans [5], Abrial décrit un "ping-pong" entre un événement modélisant l'environnement et un événement modélisant le circuit. Notre approche permet également d'exprimer de la concurrence ou de la séquentialité entre les différents composants du système.

Dans sa thèse [20], Proch propose également une méthode de modélisation de systèmes électroniques basée sur le B événementiel. La modélisation consiste en un ensemble de tâches dépendantes les unes des autres. Il synchronise les tâches en utilisant des variables d'ordonnancement. Proch propose une traduction vers SystemC dans laquelle les variables d'ordonnancement ne disparaissent pas mais sont implantées (par des signaux) dans le système dans lequel chaque tâche est modélisée par un processus.

L'approche proposée par Boulanger, Aljer et Mariano dans [4] est inverse à la nôtre. Ils proposent, à partir de VHDL, de traduire en B (non événementiel) les composants de base des circuits électroniques. Une porte logique est modélisée par une machine constituée d'une fonction par entrée et d'une fonction pour récupérer le signal de sortie. Cette approche ne permet pas la modélisation de systèmes électroniques à très haut niveau d'abstraction puisque les éléments utilisés pour la description de circuits sont déjà au niveau d'implantation. Il nous

semble que le travail présenté dans [4] décrit plutôt de la synthèse de circuits en B ; les auteurs considèrent d'ailleurs qu'un multiplexeur est un composant déjà complexe.

7 Conclusion

Nous avons présenté dans cet article une méthode formelle de modélisation et de développement de circuits électroniques synchrones. Nous avons exposé les principes de la modélisation événementielle nécessaires à la modélisation de circuits.

Le système est d'abord modélisé de façon abstraite, modèle orienté par les propriétés qu'on souhaite spécifier. Une fois toutes les propriétés introduites, on raffine en direction d'une implantation en introduisant le cycle d'horloge, nous avons montré comment modéliser ce cycle en le divisant en périodes successives et en utilisant pour cela des variables d'ordonnancement. Une période étant un moment de concurrence entre les composants du système, les communications se font d'une période à l'autre. Il faut ensuite séparer les composants en faisant en sorte que chacun ne modifie que son propre ensemble de variables. À ce niveau, le non déterminisme restant modélise implicitement des entrées qui sont finalement introduites explicitement pour obtenir un modèle totalement déterministe.

Une fois obtenu un modèle implantable, on regroupe les événements et on implante l'ordonnancement de façon à ce que le modèle du composant souhaité soit constitué d'un seul événement. Cet événement constitue le modèle BHDL du composant, à partir duquel des traducteurs permettent d'obtenir automatiquement du VHDL ou du SystemC.

Le travail présenté dans ce papier peut se poursuivre dans diverses perspectives. Une possibilité serait d'étudier la possibilité de raffiner les propriétés en même temps que les modèles. En effet, les propriétés systèmes qui s'expriment simplement par des invariants à des niveaux abstraits sont souvent difficiles à écrire au niveau de l'implantation. On pourrait imaginer récupérer, en fin de raffinement, un ensemble de propriétés systèmes vérifiables au niveau de l'implantation. Ces propriétés peuvent être utilisées pour faire du test ou pour développer des "IP de vérification", c'est à dire des composants intégrés au système permettant de vérifier son bon fonctionnement pendant son exécution.

Une autre perspective possible serait l'écriture de patrons de développement, permettant par exemple d'automatiser l'introduction et la gestion des variables d'ordonnancement ou la transformation des ensembles en booléens.

Enfin, des travaux ont été commencés au sein de la société KeesDA pour étudier la modélisation TLM (Transaction Level Modeling) en B. Il reste encore du travail pour faire en sorte que la modélisation TLM soit plus simple et propose des obligations de preuve moins fastidieuses.

Références

1. Zimmermann, Y. : Modélisation et développement formel de circuits électroniques. PhD thesis, Université Henri Poincaré - Nancy 1 (2006) http ://zyan.free.fr.

2. KeesDA : 2 avenue de Vignate, 38610 Gières, Grenoble - www.keesda.com.
3. Abrial, J.R. : The B-Book – Assigning programs to meanings. Cambridge University Press (1996)
4. Boulanger, J.L., Aljer, A., Mariano, G. : Formalization of digital circuits using the B method. In : CompRAIL VIII. (2002) 10 pages
5. Abrial, J.R. : Event Driven Electronic Circuit Construction. Unpublished (August 2001)
6. Abrial, J.R. : Guidelines to Formal System Studies. (November 2000)
7. Abrial, J.R. : Extending B without changing it (for developing distributed systems). In : Proc. of the 1st B Conference, Nantes, France. (1996)
8. Abrial, J.R., Mussat, L. : Introducing dynamic constraints in B. In Bert, D., ed. : B'98 : Recent Advances in the development and Use of the B Method. Volume 1393 of lncs., sv (1998)
9. Bert, D. : Preuve de propriétés d'équité en B : étude du protocole du bus SCSI-3. In : Actes de l'Atelier AFADL-2001. (2001) 221–241
10. Cansell, D., Gopalakrishnan, G., Jones, M., Méry, D., Weinzoepflen, A. : Incremental Proof of the Producer/Consumer Property for the PCI Protocol. In Bert, D., ed. : ZB 2002. Volume 2272 of lncs. 22–41
11. Drusinsky, D., Harel, D. : Using statecharts for hardware description and synthesis. In : Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on. Volume 8 (7). (Jul 1989) 798–807
12. Plosila, J., Seceleanu, T. : Synchronous action systems. Technical Report TUCS-TR-192, Turku Centre for Computer Science, Finland (1998)
13. Plosila, J., Sere, K. : Action systems in pipelined processor design. In : Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, IEEE Computer Society Press (1997) 156–166
14. Brock, B., Kaufmann, M., Moore, J. : ACL2 theorems about commercial microprocessors. In M. Srivas, A. Camilleri, eds. : First international conference on formal methods in computer-aided design. Volume 1166., Palo Alto, CA, USA, Springer Verlag (1996) 275–293
15. Coupet-Grimal, S., Jakubiec, L. : Coq and hardware verification : A case study. In von Wright, J., Grundy, J., Harrison, J., eds. : TPHOLs'96, Turku, Finland, 26–30 Aug 1996. Volume 1125. Springer-Verlag, Berlin (1996) 125–139
16. Bardoult, F., Quinton, P., Rajopadhye, S., Risset, T. : Synthesis of data-flow interfaces for regular parallel programs. Technical-Report RR-3760, Inria (1999)
17. Zimmermann, Y., Hallerstede, S., Cansell, D. : Formal modelling of electronic circuits using event-B, case study : SAE J1708 serial communication link (2004) KeesDA, 85 pages, confidential.
18. Projet PUSSEE : IST-2000-30103, www.keesda.com/pussee.
19. Abrial, J.R. : Event Driven Sequential Program Construction. (October 2000)
20. Proch, C. : Assistance au développement incrémental et prouvé de systèmes enfouis. PhD thesis, Université Henri Poincaré (2006)

Debugging Event-B Models using the ProB Disprover Plug-in ⋆

Olivier Ligot[1], Jens Bendisposto[2] and Michael Leuschel[2]

[1] Facultés Universitaires Notre-Dame de la Paix Namur
Namur, Belgium
olivier.ligot@student.fundp.ac.be
[2] Softwaretechnik und Programmiersprachen
Heinrich-Heine Universität Düsseldorf
Universitätsstr. 1, 40225 Düsseldorf, Germany
{bendisposto,leuschel}@cs.uni-duesseldorf.de

Abstract. The B-method, as well as its offspring Event-B, are both tool-supported formal methods used for the development of computer systems whose correctness is formally proven. However, the more complex the specification becomes, the more proof obligations need to be discharged. While many proof obligations can be discharged automatically by recent tools such as the RODIN platform, a considerable number still have to be proven interactively. This can be either because the required proof is too complicated or because the B model is erroneous. In this paper we describe a disprover plugin for RODIN that utilizes the ProB animator and model checker to automatically find counterexamples for a given problematic proof obligation. In case the disprover finds a counterexample, the user can directly investigate the source of the problem (as pinpointed by the counterexample) and should not attempt to prove the proof obligation. We also discuss under which circumstances our plug-in can be used as a prover, i.e., when the absence of a counterexample actually is a proof of the proof obligation.
Keywords: RODIN, ProB, Event-B, B-Method, Autoprover.

1 Introduction

The B-method introduced by J.-R. Abrial [1] is a theory and methodology for formal development of computer systems which is based on the notion of *abstract machines* and *refinement*. B is used in industry, mainly for safety critical applications. It is supported by several industrial strength tools such as AtelierB [22], the B Toolkit [6] or Click'n Prove [3] for proving correctness and code generation and tools for animation, model checking and model based testing such as ProB [13] or the BZTT [5].

However, classical B is lacking certain dynamic constraints (temporal logic constraints, liveness constraints) that can be used to model how a system can

⋆ This research is being carried out as part of the EU funded research projects: IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems).

evolve. This shortcoming was one of the reasons to extend B to Event-B [2, 4] which enables the specification of reactive systems without abandoning the notion of refinement.

An Event-B specification is written as an abstract machine that consists of variables which define its state and some events. An event decomposes into a predicate, the guard, that specifies under which circumstances it might occur and some generalized substitutions called actions. For instance, if the state s of an abstract machine is $(x = 2, y = 7)$ and there is an event e with the guard $true$ and the action $x := y$, then a successor state of s might be $(x = 7, y = 7)$.

A notable recent development is the EU funded research project IST 511599 RODIN, which aims to develop an open tool platform based on Eclipse that supports Event-B. The objective of RODIN is to create a unified methodology and supporting tools for cost-effective, rigorous development of software systems.

A rigorous software development requires to reason about the correctness of the formal specification. For example, one should verify that an Event-B model does not violate its invariant. Other correctness conditions are related to refinement or the properties associated with constants. The proof obligations that need to be discharged in order to establish correctness can be mechanically extracted from an Event-B model. For instance, one proof obligation will stipulate that the initialisation of an Event-B model must establish the invariant. The RODIN platform comes with a tool, the proof obligation generator, that extracts proof obligations from a model (see Figure 1).

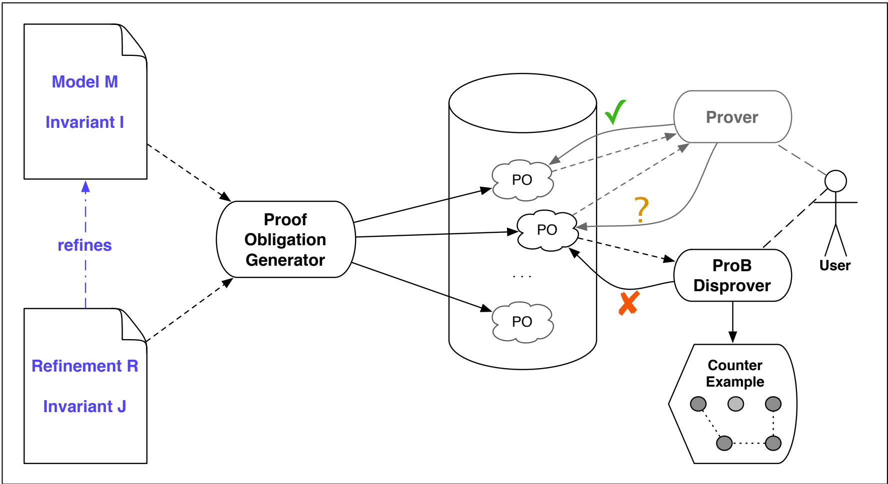


Fig. 1. Overview of proof activities and the role of the disprover

The RODIN platform also comes with some automatic provers, which can discharge a considerable number proof obligations automatically. Obviously, due

to incompleteness, not all proofs can be done automatically. In that circumstance the user is left wondering:

- Is the proof obligation valid and the proof simply too complicated for the automated prover? In other words, should I start up the interactive prover and try to prove the goal manually?
- Or is there a problem within the specification and should I spend time looking for the error and then correct it?

Pursuing either path can lead to considerable wasted effort. In this paper we present a tool which helps the user in this common situation: a disprover which tries to find a counterexample for the particular problematic proof obligation (see also Figure 1).

- If the disprover finds a counterexample we know that it is futile to spend time with the interactive prover. Also, the counterexample will give us a handle on the problem and help us find the error in the specification more quickly.
- If the disprover finds no counterexample, we know that—in certain circumstances at least—the proof obligation seems to be valid. Of course, we are still not sure whether the proof obligation is true in all circumstances; but we have at least gained some additional confidence about its validity.

As an example, suppose we want to prove the theorem, that every finite undirected graph has at least two nodes of the same degree.[3] Our Event-B model will contain the basic set $NODES$ and a graph consists of a set $V \subseteq NODES$ of vertices as well as a symmetric binary relation E representing the edges. The degree of a vertex v is simply $card(\{v\} \lhd E) = card(E[\{v\}])$. A predicate representing our theorem might be something like the following:

$$V \subseteq NODES \wedge E \in NODES \leftrightarrow NODES \wedge E = E^{-1} \wedge card(V) \in \mathbb{N}$$
$$\Rightarrow$$
$$\exists x \exists y : x \in V \wedge y \in V \wedge x \neq y \wedge card(\{x\} \lhd E) = card(\{y\} \lhd E)$$

However, this theorem is not provable since we made two mistakes in the definition. While it might not be obvious which mistakes we made, the disprover plug-in finds counterexamples that will help us to identify the problems and to correct the theorem. A trivial counterexample the tool finds is the empty graph, another counterexample found by our tool with 5 vertices that contains self loops is shown in Fig. 2. As can be seen, all vertices have a different degree. So we need to strengthen the left side of our implication to disallow self loops and graphs with less than two nodes, after which the disprover can no longer find a

[3] This example is inspired by a talk given by Leslie Lamport at B'2007.

counterexample:

$$V \subseteq NODES \land E \in NODES \leftrightarrow NODES \land E = E^{-1} \land card(V) \in \mathbb{N} \land$$
$$card(V) > 1 \land id(NODES) \cap E = \varnothing$$
$$\Rightarrow$$
$$\exists x \exists y : x \in V \land y \in V \land x \neq y \land card(\{x\} \lhd E) = card(\{y\} \lhd E)$$

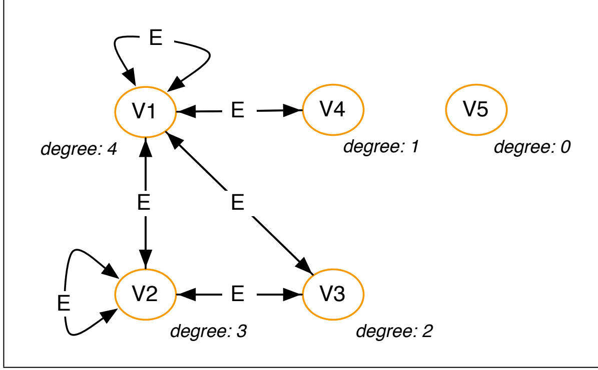


Fig. 2. Counterexample found by the disprover

RODIN can be extended by third parties, in particular it is possible to add external proving tools. We have thus developed a prover plug-in, which works as a disprover. Our plug-in is based on the Prolog based animator and model checker ProB [13]. The ProB animator is fully automatic and does not require the user to guess the right values for the operation arguments or choice variables. The undecidability of animating B is overcome by restricting animation to finite sets and integer ranges, while efficiency is achieved by delaying the enumeration of variables as long as possible.

The main idea of our work is to use the ProB animator to find counterexamples for an individual proof obligation. Therefore, we translate this proof obligation into a B machine as described in Section 3. Of course, one could have used the ProB model checker itself on the whole Event-B model. This is an alternate validation option, but this will "only" find sequences of operations which violate the invariant starting from some valid initialisation; i.e., it will not detect problems if the invariant is too weak (see [13]). Furthermore, by restricting our attention to a single, problematic proof obligation we can increase the likelihood of the disprover finding counterexamples.

The rest of the paper is structured as follows. First we provide some background on proof in Event-B in Section 2. Then we present the underlying methodology of our disprover plug-in in Section 3, before discussing the actual imple-

mentation in Section 4. We conclude with remarks on how to use the disprover as a prover and other future work in Section 5.

2 The Event-B proving subsystem

This section gives an introduction to proofs in Event-B. We will also discuss the architecture of the RODIN proving subsystem, its Kernel prover and how external reasoners work.

Proving correctness A proof in Event-B is constructed using (a slight variation of the) sequent calculus [19]. A *sequent* in Event-B is of the form $\Gamma \vdash \Sigma$ where Γ is a finite set of predicates called *hypotheses* and Σ is a single predicate called *goal*. A sequent basically means that the goal should be a logical consequence of the hypotheses Γ. Proofs of sequents are carried out using an *inference rule*. An inference rule contains a finite set of sequents A — the *antecedent* — and a single sequent C — the *consequent*. An inference rule means: if we can prove all sequents within A, then C has also been proven. It is also possible that a rule has the empty set as antecedent, this means that C has been proven. A proof for a sequent can thus be viewed as a finite tree. Each node of this *proof tree* contains a sequent s as well as an inference rule r whose consequent is s. The children of a node are the sequents in the antecedent of its rule r. Leaf nodes are those nodes where the associated inference rule has the empty set as antecedent. To actually discharge a proof obligation *po*, we need to find a finite proof tree whose root node is labeled with *po*.

Note that the antecedent of sequents might contain a subset called *type environment*, where the predicates only carry type information such as *x is an integer* ($x \in \mathbb{Z}$) or *y is a member of a basic set Y* ($y \in Y$). Sequents of the type environment can be statically checked by a type checker and thus have the empty set as antecedent (unless there is a typing error). In RODIN terms, a sequence of inference rules that discharge a certain type of sequents is called a *proof tactic*.

The RODIN proving subsystem In RODIN, a considerable number of proofs can be done automatically by the proving subsystem that consists, as shown in Fig. 3, of the proof obligation generator and the Event-B Kernel Prover [18]. The proof obligation generator extracts all proof obligations from the Event-B model that need to be discharged in order to prove correctness of the model and stores them in a XML database file. After all POs have been generated, the kernel prover tries to discharge valid POs automatically, i.e. in background, without user interactions.

As shown in Fig. 3, the Event-B kernel decomposes into the proof manager and a set of prover plugins. While the proof manager is responsible for storage, traversal, composition and re-usage of proofs, the prover plugins try to generate valid inferences in order to discharge the proof obligations. The proof manager

also maintains the state of current proofs for all proof obligations and decides if they have to be discharged and calls external provers if they are non-interactive. There are also interactive reasoners that require the user to apply them, the ProB disprover plug-in is such an interactive plug-in. In the next section we show the underlying principles of our plug-in, before showing in Section 4 how it was integrated into the RODIN platform.

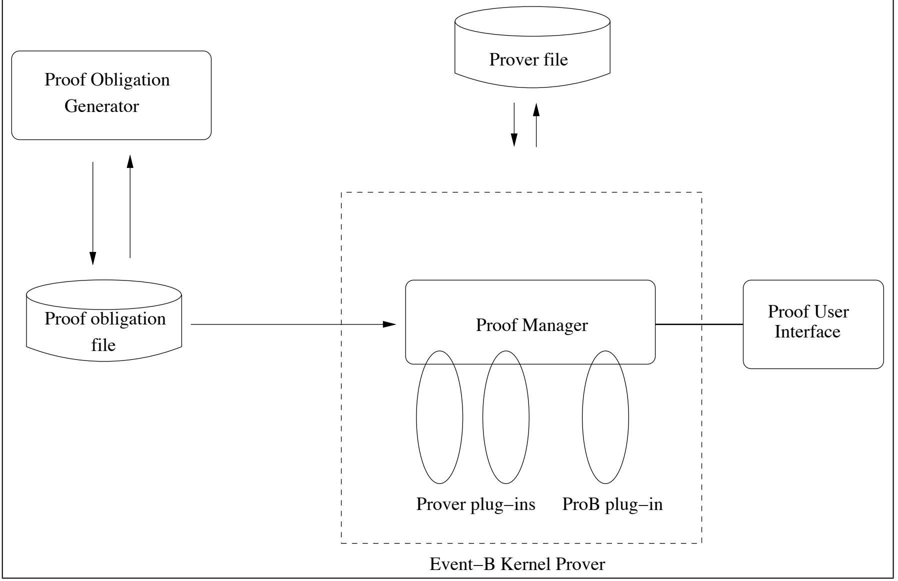


Fig. 3. Architecture of the RODIN proving subsystem

3 The principle of disproving using ProB

In the following, we will explain, how a sequent can be translated into a B machine that can be used with ProB.

Finding counterexamples Let $G(x_1, ..., x_k)$ be the goal of a sequent s and let $H_1(x_1, \ldots, x_k), \ldots, H_n(x_1, \ldots, x_k)$ be the hypotheses. To find a counterexample for s, we need to check if the predicate

$$\exists x_1, ..., x_k : (H_1(x_1, ..., x_k) \wedge ... \wedge H_n(x_1, ..., x_k)) \wedge \neg G(x_1, ..., x_k) \qquad (1)$$

holds. If it does, then we can extract a concrete counterexample by finding a valuation for $x_1, ..., x_k$ which makes the implication true. Finding values that

satisfy a propositional boolean formula is NP complete, and for the first order logic formulas that can occur within sequents, the problem is undecidable. To overcome this difficulty, we have to restrict sets to relatively small, finite domains. As a consequence, we know that in principle it is not possible to guarantee that a disproving algorithm can automatically find a counterexample if one exists. In other words, the absence of a counterexample does not mean in general, that a proof obligation is valid. (There are, however, certain cases where the absence of a counterexample discharges a proof obligation. We discuss these cases in section 5.)

Transforming sequents into classical B machines ProB can be used to find a counterexample for a given sequent, but it needs a classical B machine that encodes the sequent as its input. Fortunately this encoding is — at least in principle — not difficult to obtain. We create a B machine, that contains an operation *disproveHypotheses* with the predicate from equation (1) as its guard. The operation is enabled, if and only if ProB can find a counterexample.

In order to construct this stand-alone machine, we need to extract some information from the original Event-B specification such as axioms[4], carrier sets, parameters, variables (including type information) and constants. Furthermore, we need information about the sequent to be (dis-)proved, such as the hypotheses and the goal. The translation of these information is in most cases straightforward, for example we construct the `SETS` clause of the machine by enumerating the set definitions from the original Event-B specification. In some cases the translation is less obvious. For instance, we translate the axioms together with the type information of the constants into the `PROPERTIES` clause. We generate new definitions called `TypeEnvironment` and `Hypotheses` inside the `DEFINITIONS` clause. The `TypeEnvironment` is a subset of the hypotheses that only contains predicates dealing with type information. A schema of the B machine constructed from a given sequent $H_1, H_2, \ldots H_n \vdash G$ is shown in Listing 1.1.

Selecting Hypotheses The RODIN proving subsystem allows the user to select a subset of hypotheses that are in the database, these hypotheses are either directly derived from the specification or previously proven. Obviously if a subset of H proves G, then H also proves G.

$$H' \subseteq H \wedge H' \vdash G \Rightarrow H \vdash G$$

Thus a user can restrict the hypotheses in a sequent to an arbitrary subset of so-called *selected hypotheses*, by removing hypotheses that are not relevant for the proof. By default, a particular set of hypotheses which deals with the involved variables are automatically selected by RODIN. The user can also decide to hide a particular subset of hypotheses, this subset are called *hidden hypotheses*. In fact, there are thus two alternatives:

[4] An axiom is treated like a sequent $true \vdash A$

– run the external disprover with the *selected* hypotheses or
– run it with *all* hypotheses except the hidden ones.

In any case, the user can choose which alternative to apply (our plug-in provides two buttons) and change his mind later.

Listing 1.1. Schema of an abstract machine constructed from a sequent

```
1  MACHINE Disprove
2  DEFINITIONS
3      TypeEnvironment == H_1(x_1,...,x_k) & ... & H_i(x_1,...,x_k);
4      Hypotheses == TypeEnvironment &
           H_{i+1}(x_1,...,x_k) & ... & H_n(x_1,...,x_k);
5      Goal == G(x_1,...,x_k)
6  OPERATIONS
7      disprove(x_1,...,x_k) =
8         PRE Hypotheses & not(Goal)
9         THEN skip
10        END
11 END
```

4 Implementation of the ProB disprover plug-in

In previous work [7], we have developed a version of ProB that integrates with Eclipse. Its main component is the Eclipse ProB plug-in as shown in Fig. 4. It allows third party tools to use ProB for several tasks, thus it can be seen as a Java abstraction layer for the Prolog part of ProB. The disprover uses this core plug-in to find counterexamples. Therefore it creates — when applied to a node of the proof tree — a B machine as described in Section 3 and starts animating this machine. If the operation `disprove` is enabled, we have found a counterexample.

Our disprover plug-in consists of a user interface (UI) that displays the results of a proof and a core component that encapsulates the proof logic. The UI is an extension to the RODIN proving user interface. It allows the user to select a node in the proof tree, that he wants to check with ProB. The core plug-in provides a way to apply the ProB disprover. Its role is to:

– Translate the sequent into a B machine.
– Call ProB through the Eclipse ProB core plug-in.
– Return results to the user interface.
– Handle failures, time outs and user cancel requests.

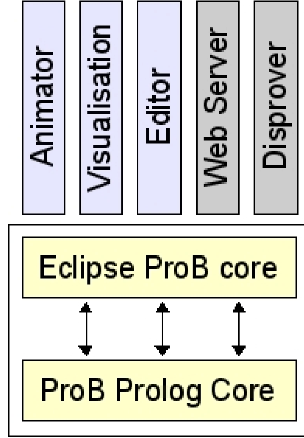


Fig. 4. Architecture of the ProB Eclipse Version

Displaying counterexamples A first approach was to display counterexamples in a separate window. This solution was however not very useful because there is no connection between the counterexample and the proof obligations. We thus take another approach to resolve this problem by applying a case distinction [1] to the node in the proof tree.[5] As seen in the previous section, a counterexample can be described by a predicate

$$C_p \equiv x_1 = e_1, ..., x_k = e_k$$

Now we apply a case distinction to the node. This results in two child nodes with the sequents

1. $H, C_p \vdash G$
2. $H, \neg C_p \vdash G$

The first sequent is the case where the counterexample was found (C_p makes G false). The second one is the remaining case, where the particular counterexample is explicitly disallowed. This second case can be used to find further counterexamples by re-running the disprover on it.

Figure 5 shows the tool displaying a counterexample. The goal of the generated proof obligation was to prove that

$$counter - 1 \in \mathrm{dom}(hst)$$

holds. Except information on the types, there are no other hypotheses. After lauching the disprover, the proof tree view shows, that the counterexample $hst = \{\} \wedge counter = -5$ has been found[6].

[5] Original idea by Farhad Mehta.

[6] The value -5 is actually the lower bound for the integers; this can be modified in the preference of the plug-in.

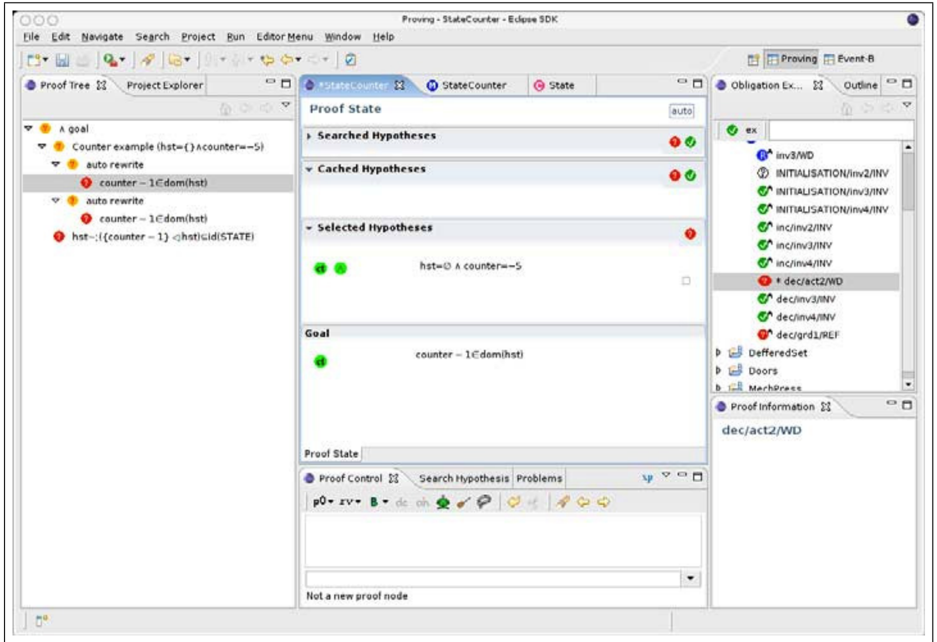


Fig. 5. Screenshot showing the display of a counterexample

5 Future Work and Conclusion

Using ProB as a prover If the ProB disprover fails to find a counterexample for a particular proof obligation, we cannot infer that the proof obligation is true. This is due to two reasons:

- **deferred sets:** If a B machine uses deferred sets (i.e., sets which are not explicitly enumerated in the SETS clause), then the cardinality of those sets is not a priori fixed; the set could even be infinite. ProB, however, will check the proof obligation only for some finite cardinalities of the deferred sets, and thus may fail to find existing counterexamples. For example, ProB will fail to find a counterexample for the formula $\exists n.(n : \mathbb{N} \wedge card(S) < n)$, where S is a deferred set without further restrictions.
- **integers:** If an integer variable occurs inside a proof obligation, whose value is *not* determined by the rest of the proof obligation, ProB will enumerate the variable only within a finite interval (between user determined MININT and MAXINT). Again, ProB may thus fail to find counterexamples for integer values which lie outside of MININT..MAXINT.

However, if a B machine contains neither deferred sets nor integer variables, the ProB disprover can actually also be used as a *prover*. This condition can

be easily checked statically, in which case our disprover can inform the Rodin platform that the PO has actually been proven. Some practical B specifications fall into this category. For example, the Volvo vehicle function used in [13]. Another example is a Hamming encoder [9], for which Dominique Cansell has used ProB to prove some essential theorems (which would have been extremely tedious to prove by hand).[7]

In future work, we are planning to implement a static analysis which safely infers intervals for the integer variables. If all variables can be proven to lie within a finite range, ProB could be used as a prover on this larger class of specifications (provided MININT and MAXINT cover all those ranges).

More future work An empiric evaluation of the use of ProB as a disprover is required if one wants to see if the plug-in is efficient or can be more optimized. A number of tests have already be done but more benchmark tests on several operating systems would be welcome.

When using relations or functions in the Event-B, the possible values for the variables of a sequent grows extremely large. For example, given $r : A \leftrightarrow A$, where A has a cardinality of 4, we have $2^{4*4} = 65536$ possibilities for r. Given $x : \mathbb{P}(A \leftrightarrow A)$ we even have $2^{2^{4*4}} = 2^{65536}$ possibilities for x. ProB has to investigate these possibilities in order to search for a counterexample. Symmetry reduction is one way to ease this task, and we plan to check whether we can make use of ProB's recent developments [14, 15] in that area for our disprover plug-in. Another option is to partion the configuration space into several areas, and let different instances of ProB running in parallel take care of the corresponding exploration.

Related work A very popular tool for validating models and finding counterexamples is Alloy [12], which makes use SAT solvers (rather than constraint solving). The specification language of Alloy is first-order and thus cannot be applied "out of the box" to Event-B models. However, Event-B models could be compiled (for fixed finite base sets) into Alloy or directly into SAT formulas. In future work we plan to investigate this avenue of research.

Earlier related work are the model generators FINDER [20] and MACE [16] which can also be used to find counterexamples. The prover Isabelle now also has a quick check function [8], looking randomly for counterexamples. There are many more related works, such as the more recent [21], and even several CADE and IJCAR workshops on disproving have been organized. There is also considerable work on combining model checking [10] with theorem proving in general (e.g., [17, 11]).

Conclusion In summary, we have presented a method to use the existing model checker ProB as a tool for proof support, by trying to find counterexamples for individual proof obligations. We have also discussed under which circumstances

[7] Private communication by Dominique Cansell.

the model checker can be used as a prover. We have presented the implementation within Eclipse, using the Rodin Event-B platform and have shown how this has enabled to use the model checker in a very targeted and convenient way. We believe that a model checker can provide a very valuable support for the B developer, avoiding unnecessary time spent trying to prove a false proof obligation.

References

1. J.-R. Abrial. *The B book : assigning programs to meanings.* Cambridge University Press, 1996.
2. J.-R. Abrial. Extending B without changing it. (For Distributed System). Proc. of 1st Conf. on B Method. pages 169–191, 1996.
3. J.-R. Abrial and D. Cansell. Click'n prove: Interactive proofs within set theory. In *TPHOL*, 2003.
4. J.-R. Abrial and L. Mussat. Introducing dynamic constraints in B. In *B '98: Proceedings of the Second International B Conference on Recent Advances in the Development and Use of the B Method*, pages 83–128, London, UK, 1998. Springer-Verlag.
5. F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, M. Utting, and N. Vacelet. BZ-testing-tools: A tool-set for test generation from Z and B using constraint logic programming. In *Proceedings of FATES'02, Formal Approaches to Testing of Software*, pages 105–120, August 2002. Technical Report, INRIA.
6. U. B-Core (UK) Limited, Oxon. *B-Toolkit, On-line manual*, 1999. Available at `http://www.b-core.com/ONLINEDOC/Contents.html`.
7. J. Bendisposto. Integration of the ProB modelchecker into Eclipse. Bachelor's thesis, July 2006.
8. S. Berghofer and T. Nipkow. Random Testing in Isabelle/HOL. In *SEFM*, pages 230–239. IEEE Computer Society, 2004.
9. D. Cansell, S. Hallerstede, and I. Oliver. UML-B specification and hardware implementation of a hamming coder/decoder. In J. Mermet, editor, *UML-B Specification for Proven Embedded Systems Design.* Kluwer Academic Publishers, Nov 2004. Chapter 16.
10. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.
11. E. L. Gunter and D. Peled. Model checking, testing and verification working together. *Formal Asp. Comput.*, 17(2):201–221, 2005.
12. D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11:256–290, 2002.
13. M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.
14. M. Leuschel, M. Butler, C. Spermann, and E. Turner. Symmetry Reduction for B by Permutation Flooding. In *Proceedings of the 7th International B Conference (B2007)*, LNCS 4355, pages 79–93, Besancon, France, 2007. Springer-Verlag.
15. M. Leuschel and T. Massart. Efficient Approximate Verification of B via Symmetry Markers. *Proceedings International Symmetry Conference*, pages –, Januar 2007.
16. W. McCune. MACE 2.0 Reference Manual and Guide. *CoRR*, cs.LO/0106042, 2001.

17. S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.
18. RODIN Consortium. Rodin deliverable D10 - Specification of basic tools and platform. Technical report, 2005. Available at http://rodin.cs.ncl.ac.uk/deliverables/rodinD10.pdf.
19. RODIN Consortium. Rodin deliverable D7 - Event B language. Technical report, 2005. Available at http://rodin.cs.ncl.ac.uk/deliverables/rodinD7.pdf.
20. J. K. Slaney, E. L. Lusk, and W. McCune. SCOTT: Semantically Constrained Otter System Description. In A. Bundy, editor, *CADE*, volume 814 of *Lecture Notes in Computer Science*, pages 764–768. Springer, 1994.
21. G. Steel, A. Bundy, and E. Denney. Finding counterexamples to inductive conjectures and discovering security protocol attacks. In *Foundations Of Computer Security Workshop*, 2002.
22. F. Steria, Aix-en-Provence. *Atelier B, User and Reference Manuals*, 1996. Available at `http://www.atelierb.societe.com/index_uk.html`.

Dérivation d'algorithmes sans verrou à partir d'une spécification atomique

Loïc Fejoz* et Stephan Merz

INRIA Lorraine & LORIA, Nancy, France
Loic.Fejoz@loria.fr Stephan.Merz@loria.fr

Résumé Pour gérer les accès de plusieurs processus à des données partagées, on utilise souvent un verrou global. Ici nous nous intéressons aux algorithmes sans verrou qui permettent un accès simultané en lecture et écriture. Malgré une littérature récente abondante, il y a peu de preuves de ces algorithmes.

Nous proposons une méthode modulaire qui permettra de dériver des algorithmes sans verrou à partir d'une spécification atomique qui décrit la fonctionnalité des opérations élémentaires sur une structure de données. Cette méthode peut être utilisée dans un style « à la B » (approche top-down), mais elle permet aussi d'ajouter des points de linéarisation aux algorithmes, en adaptant le style de la méthode « assume-guarantee ». Dans le présent article, on donne une formalisation de la méthode, et on explique son utilisation sur des exemples simplifiés. Nous comparons cette méthode à des méthodes plus classiques telles que B et TLA$^+$.

1 Introduction

Dans un environnement concurrent, l'approche la plus usitée pour gérer l'accès aux ressources partagées est l'utilisation d'un verrou global. Les algorithmes en jeu utilisent donc des primitives de synchronisation (mutexes, sémaphores) qui assurent un accès non-concurrent (*i.e.* en section critique). Malheureusement ceci amène de nombreux problèmes : l'utilisation d'un verrou global qui contrôle la structure entière empêche l'accès concurrent à des parties disjointes de la structure. L'introduction de verrous plus fins complique souvent la logique des applications et est en proie à des problème de blocages dûs à l'attente de la libération d'un ou plusieurs verrous.

Un exemple typique serait donné par deux processus qui s'échangent des messages par l'intermédiaire d'une liste, le premier ajoutant des messages en tête de liste et le second prélevant les messages en queue. Avec un verrou global, seul un des deux processus accède à la liste. L'utilisation d'un verrou plus fin (par exemple sur les deux premiers et les deux derniers éléments de la liste) ne résout pas le problème de famine dans le cas où l'un des processus venait à mourir en section critique. Seule une solution sans verrou assurerait l'indépendance des deux processus.

* This work was supported by Microsoft Research through its European PhD Scholarship Programme.

Les algorithmes sans verrou se basent sur des primitives atomiques fournies par les processeurs modernes. Ces primitives, dont la plus connue est le "Compare-and-swap" (CAS), n'ont pas toujours été disponibles sur toutes les plates-formes. La large diffusion de matériel supportant l'exécution parallèle explique le regain d'intérêt [4,5,11,14,15,16] pour les algorithmes sans verrou. Malgré la complexité de ces algorithmes, peu de travaux sont accompagnés de preuves formelles de correction. Nous nous proposons de définir une méthode spécifique pour la dérivation d'algorithmes sans verrou. En effet, ces algorithmes utilisent souvent des solutions types qui s'avèrent difficiles à formaliser dans des méthodes formelles généralistes telles que B, TLA$^+$ ou l'approche « rely-guarantee ». Notre méthode permettra de disposer d'une bibliothèque de solutions qui pourront être adaptées à un algorithme donné.

Le chapitre 2 introduit les propriétés de correction associées aux algorithmes sans verrou et présente la primitive CAS et ses dérivées. Notre méthode pour le développement conjoint d'un ensemble d'algorithmes opérant sur une même structure de données est décrite dans le chapitre 3. Nous l'appliquons sur des exemples simples mais illustratifs dans le chapitre 4. Enfin, le chapitre 5 montre en quoi notre méthode nous semble plus adaptée au problème que les méthodes TLA$^+$ [8], B [1] et « rely-guarantee » [3].

2 Algorithmes sans verrou

2.1 Propriétés désirables

Une structure de données est manipulée par plusieurs algorithmes. En effet, on en trouve autant que de méthodes supportées par la structure (dans le cas de la liste : ajout, suppression, insertion, lecture, etc). Ces algorithmes peuvent être exécutés de manière séquentielle ou parallèle par plusieurs processus (ou threads).

Afin de simplifier la compréhension des algorithmes par l'utilisateur, il est préférable que chacune des méthodes apparaisse à l'exécution comme étant atomique. Cette propriété est appelée *linéarisabilité* et s'énonce comme suit : une trace d'exécution concurrente est linéarisable s'il existe une trace séquentielle qui permet d'observer le même comportement extérieur. Il est donc intéressant d'avoir une méthode qui ait pour première spécification la description de l'effet atomique des méthodes.

Par ailleurs, les verrous (bloquants) sont très souvent remplacés par une boucle utilisant les primitives évoquées en introduction. Il se pose alors la question de la terminaison. Trois notions sont souvent évoquées dans la littérature.

1. **wait-free** : tous les processus progressent même s'il y a du délai ;
2. **lock-free** : certains processus progressent ;
3. **obstruction-free** : les processus exécutés isolément progressent.

Cependant, nous ne considérons pas ces notions de vivacité dans le présent article.

2.2 CAS

De nombreuses primitives atomiques ont été proposées et peuvent être exploitées pour la construction d'algorithmes sans verrou, comme Load-Link/Store-conditionnal (LL/SC), Test-and-Set (TAS), Fetch-and-Add et la famille des Compare-and-Swap (CAS). Nous ne les détaillerons pas toutes ici. En effet, toutes ne sont pas universelles comme l'a montré Herlihy [6]. Nous nous focaliserons sur la primitive CAS car elle est universelle, dans le sens où l'on peut construire les autres primitives à partir du CAS. La primitive de base, dénotée CAS_1, prend en paramètre un pointeur a et deux valeurs o, n. Si la valeur pointée par a est égal au deuxième argument o, on donne à la valeur pointée la valeur du troisième argument n, sinon, la valeur référencée par a n'est pas modifiée. Autrement dit, on exécute atomiquement la fonction suivante (décrite en C) :

```
word_t CAS1(word_t *a, word_t o, word_t n) {
  old = *a;
  if (old == o) {
    *a = n;
  }
  return old;
}
```

Dans la suite, on utilisera une version modifiée appelée CAS_1. Elle diffère de CAS_1 par le fait qu'au lieu de retourner la valeur précédente du pointeur, elle retourne True si l'affectation a eu lieu, False sinon.

2.3 RDCSS

Harris et al. [5] ont montré comment la primitive CAS_1 peut servir de base à l'implémentation des primitives CAS opérant sur plusieurs valeurs à la fois ; ces variantes sont à la base d'algorithmes sans verrou sur des structures de données complexes. Une étape clef dans leur construction est la réalisation des opérations RDCSS (*restricted double compare single swap*) et RDCSSRead dont les spécifications atomiques sont les suivantes :

```
word_t RDCSS(word_t *a1, word_t o1, word_t *a2, word_t o2, word_t n2) {
  r = *a2;
  if ((r == o2) && (*a1 == o1)) {
    *a2 = n2;
  }
  return r;
}

word_t RDCSSRead(word_t *a2) {
  return *a2;                                                    10
}
```

L'opération RDCSS prend en paramètre deux pointeurs a_1 et a_2 et trois valeurs o_1, o_2 et n_2. Si les valeurs référencées par a_1 et a_2 sont égales aux valeurs

```
word_t RDCSS(RDCSSDescriptor_t *d) {
  do {
    r = CAS1(d->a2, d->o2, d); // C1
    if (IsDescriptor(r)) Complete(r); // H1
  } while (IsDescriptor(r)); // B1
  if (r == d->o2) Complete(d);
  return r;
}

word_t RDCSSRead(addr_t *addr) {                                10
  do {
    r = *addr; // R1
    if (IsDescriptor(r)) Complete(r); //H2
  } while (IsDescriptor(r)); // B2
  return r;
}

void Complete(RDCSSDescriptor_t *d) {
  v = *(d->a1); // R2
  if (v == d->o1) {                                             20
    CAS1(d->a2,d,d->n2); // C2
  } else {
    CAS1(d->a2,d,d->o2); // C3
  }
}
```

Fig. 1. Algorithme implémentant RDCSS

o_1 et o_2, alors on donne à la valeur pointée par a_2 la valeur n. L'opération RDCSSRead renvoie simplement la valeur référencée par l'adresse donnée en argument.

L'implémentation de RDCSS et RDCSSRead proposée par Harris et al. apparaît dans la figure 1. Comme beaucoup de tels algorithmes, le code pour le RDCSS installe d'abord un descripteur qui contient tous les éléments nécessaires pour effectuer l'opération. Ce descripteur est installé à la place du résultat. Le premier processus opérant sur cette adresse de la mémoire effectue l'opération indiquée par le descripteur, avant de faire sa propre opération. (On suppose qu'un descripteur est différent de toute autre valeur.) Dans cette approche, un processus peut être amené à effectuer une opération à la place d'un autre processus. Le processus original verra une valeur différente de son descripteur, signalant que l'opération a déjà été effectuée.

Nous nous posons comme objectif de concevoir une méthode pour démontrer la correction d'algorithmes tels que celui de la figure 1. La méthode doit pouvoir être composable. C'est-à-dire permettre de réutiliser des spécifications effectuées

par ailleurs. Une fois outillée, elle doit pouvoir interagir avec des prouveurs automatiques ou, à défaut, interactifs.

3 Spécification et vérification d'algorithmes parallèles

Nous considérons un système constitué d'un ensemble arbitraire de processus, chacun exécutant un algorithme parmi un nombre fixe d'algorithmes accédant à la structure partagée. Chaque processus possède un espace mémoire propre (environnement local) et interagit avec les autres via une mémoire partagée (environnement global). Ceci nous amène donc à spécifier le système en décrivant les algorithmes executés par les processus. Pour chacun des algorithmes nous allons démontrer qu'il est équivalent, dans le contexte des autres algorithmes, à sa spécification atomique, afin d'assurer la linéarisabilité.

3.1 Formalisation des algorithmes

Un algorithme est modélisé comme un système de transitions ; les noeuds étant appelé places. Les transitions sont supposées être prises atomiquement. Une place est donc un point où des transitions d'autres algorithmes peuvent être effectuées (interleaving semantics). On associe alors à chaque place un invariant local. Celui-ci s'applique à la fois sur l'environnement local et global. Pour avoir une méthode compositionnelle, on associe aussi un prédicat indiquant ce que doivent satisfaire les actions exécutées en parallèle.

Definition 1. *Un contrôle de flot* $CF \equiv (Place, p_{init}, \Delta)$ *consiste en :*
- *un ensemble fini de places Place,*
- *une place initiale* p_{init} *appartenant à Place et*
- *un ensemble de transitions* $\Delta \subseteq Place \times Place$ *sur les places.*

Un algorithme est décrit par le contrôle de flot sous-jacent et des prédicats qui spécifient l'invariant local et la condition de garantie associés à chaque place, et la garde et l'effet de chaque transition. Les environnements global et locaux associent des valeurs aux variables globales et locales ; pour simplifier l'exposition nous représentons tous les environnements par un même type E.

Definition 2. *Un algorithme* $A = (CF, local_invariant, rely, guard, effect)$ *est décrit par un quintuplet où :*
- CF *est un contrôle de flot* CF*,*
- $local_invariant : Place \to E \times E \to \mathbb{B}$ *associe un invariant local (portant sur un environnement global et local) à chaque place,*
- $rely : Place \to E \times E \to E \times E \to \mathbb{B}$ *décrit, pour chaque place, une condition de garantie entre les environnements locaux et globaux d'avant et après,*
- $guard : \Delta \to E \times E \to \mathbb{B}$ *décrit la garde pour chaque transition et*
- $effect : \Delta \to E \times E \to E \times E \to \mathbb{B}$ *décrit l'effet des transitions.*

Notons que pour encoder les algorithmes paramétrés (comme RDCSS avec a_1, a_2, o_1, o_2 et n_2), on peut inclure les paramètres dans l'environnement local.

Nous associons des obligations de preuve à un algorithme : toute transition, qu'elle soit due à l'algorithme lui-même ou à son environnement décrit par la condition *rely*, doit respecter les invariants associés aux places. Aussi, nous exigeons que toute transition soit possible à partir des états décrits par l'invariant associé à la place source.

Definition 3. *Un algorithme* $A = (CF, local_invariant, rely, guard, effect)$ *est* valide *s'il vérifie les conditions suivantes.*

- *Chaque transition* $\delta = (p_1, p_2) \in \Delta$ *respecte l'invariant local : pour tous environnements* $genv_1$, $lenv_1$, $genv_2$, $lenv_2$ *l'implication*
$$\begin{aligned} & local_invariant(p_1, genv_1, lenv1) \\ & \wedge\ guard(\delta, genv_1, lenv_1) \wedge effect(\delta, genv_1, lenv_1, genv_2, lenv_2) \\ \Rightarrow\ & local_invariant(p_2, genv_2, lenv_2) \end{aligned}$$
est valide.
- *Chaque transition décrite par le prédicat rely respecte l'invariant local : pour toute place* p *et tous environnements* $genv_1$, $lenv_1$, $genv_2$, $lenv_2$ *l'implication*
$$\begin{aligned} & local_invariant(p, genv_1, lenv1) \wedge rely(p, genv_1, lenv_1, genv_2, lenv_2) \\ \Rightarrow\ & local_invariant(p, genv_2, lenv_2) \end{aligned}$$
est valide.
- *Chaque transition* $\delta \in \Delta$ *est possible : pour tous environnements* $genv_1$ *et* $lenv_1$ *il existe des environnements* $genv_2$ *et* $lenv_2$ *tels que*
$$\begin{aligned} & local_invariant(p_1, genv_1, lenv1) \wedge guard(\delta, genv_1, lenv_1) \\ \Rightarrow\ & effect(\delta, genv_1, lenv_1, genv_2, lenv_2) \end{aligned}$$
est valide.

Le nombre de processus étant variable, il convient de faire la preuve en faisant abstraction de ce nombre. Une spécification est donc seulement constituée d'une liste d'algorithmes. Car ces algorithmes s'exécutent en parallèle, y compris plusieurs instances d'un même algorithme, nous exigeons que chaque transition d'un algorithme satisfasse les garanties *rely* de tous les algorithmes, y compris lui même.

Definition 4. *Une spécification est une liste d'algorithmes* $Spec = [A_1, \ldots, A_n]$.

Une spécification $Spec \equiv [A_1 \ldots A_n]$ *est valide si elle vérifie les conditions suivantes, où* $A_i = (CF_i, local_invariant_i, rely_i, guard_i, effect_i)$.

- *Chaque algorithme* A_i *est valide et*
- *chaque transition d'un algorithme vérifie les conditions rely de tous les algorithmes. Pour tous* $i, j \in \{1, \ldots, n\}$, *toute place* $p_i \in Place_i$, *toute transition* $\delta_j = (p_j, p'_j) \in \Delta_j$ *et tous environnements* $genv$, $genv'$, $lenv_i$, $lenv_j$, $lenv'_j$ *:*
$$\begin{aligned} & local_invariant_i(p_i, genv, lenv_i) \wedge local_invariant_j(p_j, genv, lenv_j) \\ & \wedge\ guard_j(\delta_j, genv, lenv_j) \wedge effect_j(\delta_j, genv, lenv_j, genv', lenv'_j) \\ \Rightarrow\ & rely_i(p_i, genv, lenv_i, genv', lenv_i). \end{aligned}$$

3.2 Raffinement

Comme l'on veut que chaque algorithme implémenté soit équivalent à un appel atomique d'une méthode, on va travailler par raffinement. En d'autres termes, on va écrire des spécifications intermédiaires qui introduisent de plus en plus de détails.

La spécification d'un algorithme atomique est composée de deux places et d'une transition, pour laquelle l'annotation *effect* explicite la relation pré/post-conditions. Souvent, les places ne contiendront pas d'invariants locaux ni de *rely*.

Formellement, nous définissons une relation de raffinement entre deux algorithmes $\underline{A}$ et $\overline{A}$ aux niveaux concret et abstrait. Pour faire le lien entre les places abstraites et les places concrètes, on utilise une relation de raffinement $mapp : \underline{Place} \times \overline{Place}$. Nous indiquerons cette relation de raffinement souvent par des flêches pointillées (voir par exemple les figures 2 et 3). La relation entre les environnements abstraits et concrets est décrite par un invariant de collage de la forme $gluing_invariant : E \times E \to E \times E \to \mathbb{B}$ – afin de simplifier l'exposition, nous ne distinguons pas ici entre les environnements globaux et locaux.

Definition 5. *Un algorithme concret* $\underline{A}$ *raffine un algorithme abstrait* $\overline{A}$ *sous* $mapp$ *et* $gluing_invariant$ *si :*

- $(\underline{p}_{init}, \overline{p}_{init}) \in mapp$, *c'est à dire la place initiale concrète raffine la place initiale abstraite,*
- *chaque place concrète raffine une place abstraite : pour tout* $\underline{p} \in \underline{Place}$ *il existe* $\overline{p} \in \overline{Place}$ *avec* $(\underline{p}, \overline{p}) \in mapp$,
- *l'invariant local initial concret implique l'invariant local initial abstrait :*

$$\begin{aligned} & \underline{local_invariant}(\underline{p}_{init}, \underline{genv}, \underline{lenv}) \\ & \wedge\ gluing_invariant(\underline{genv}, \underline{lenv}, \overline{genv}, \overline{lenv}) \\ \Rightarrow\ & \overline{local_invariant}(\overline{p}_{init}, \overline{genv}, \overline{lenv}), \end{aligned}$$

- *chaque pas concret correspond à un pas abstrait ou bien à un bégaiement : pour toute transition concrète* $\underline{\delta} = (\underline{p}, \underline{p}') \in \underline{\Delta}$ *et toute place abstraite* $\overline{p} \in \overline{Place}$ *avec* $(\underline{p}, \overline{p}) \in mapp$, *et pour tous environnements* $\underline{genv}$, $\underline{lenv}$, $\underline{genv}'$, $\underline{lenv}'$, $\overline{genv}$, $\overline{lenv}$ *il existe des environnements abstraits* $\overline{genv}'$ *et* $\overline{lenv}'$ *tels que*
  - *soit il existe une place abstraite* $\overline{p}' \in \overline{Place}$ *avec* $(\underline{p}', \overline{p}') \in mapp$ *et* $\overline{\delta} = (\overline{p}, \overline{p}') \in \overline{\Delta}$ *telle que*

$$\begin{aligned} & \underline{local_invariant}(\underline{p}, \underline{genv}, \underline{lenv}) \\ & \wedge\ \underline{guard}(\underline{\delta}, \underline{genv}, \underline{lenv}) \wedge \underline{effect}(\underline{\delta}, \underline{genv}, \underline{lenv}, \underline{genv}', \underline{lenv}') \\ & \wedge\ \underline{local_invariant}(\underline{p}', \underline{genv}', \underline{lenv}') \\ & \wedge\ gluing_invariant(\underline{genv}, \underline{lenv}, \overline{genv}, \overline{lenv}) \\ \Rightarrow\ & gluing_invariant(\underline{genv}', \underline{lenv}', \overline{genv}', \overline{lenv}') \\ & \wedge\ \overline{local_invariant}(\overline{p}', \overline{genv}', \overline{lenv}') \\ & \wedge\ \overline{guard}(\overline{\delta}, \overline{genv}, \overline{lenv}) \wedge \overline{effect}(\overline{\delta}, \overline{genv}, \overline{lenv}, \overline{genv}', \overline{lenv}'), \end{aligned}$$

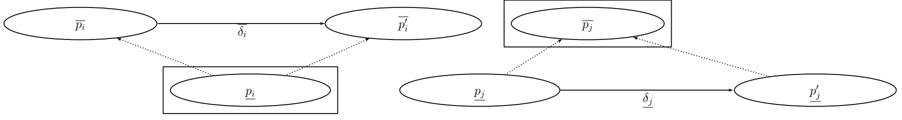


Fig. 2. $\underline{A_j}$ effectue un pas abstrait de $\overline{A_i}$.

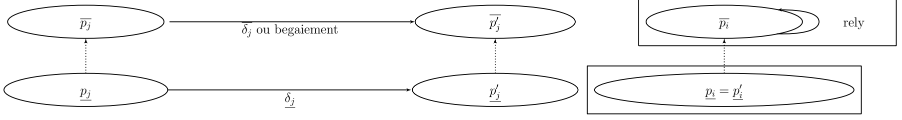


Fig. 3. $\underline{A_j}$ n'a pas d'effet $\overline{A_i}$.

– *soit la transition est invisible au niveau abstrait :*

$$
\begin{aligned}
& \underline{local_invariant}(\underline{p}, \underline{genv}, \underline{lenv}) \\
& \wedge\ \underline{guard}(\underline{\delta}, \underline{genv}, \underline{lenv}) \wedge \underline{effect}(\underline{\delta}, \underline{genv}, \underline{lenv}, \underline{genv}', \underline{lenv}') \\
& \wedge\ \underline{local_invariant}(\underline{p}', \underline{genv}' \underline{lenv}') \\
& \wedge\ gluing_invariant(\underline{genv}, \underline{lenv}, \overline{genv}, \overline{lenv}) \\
\Rightarrow\ & gluing_invariant(\underline{genv}', \underline{lenv}', \overline{genv}', \overline{lenv}') \\
& \wedge\ \overline{local_invariant}(\overline{p}', \overline{genv}', \overline{lenv}') \\
& \wedge\ \overline{rely}(\overline{genv}, \overline{lenv}, \overline{genv}', \overline{lenv}')
\end{aligned}
$$

Pour définir le raffinement entre spécifications, il faut prendre en compte les relations entre algorithmes différents. Or, comme expliqué dans le paragraphe 2.3 pour l'algorithme RDCSS, un algorithme peut effectuer un pas abstrait d'un autre. Il faut évidement que quand l'algorithme original reprend pour finir son action (qui a déjà été effectuée par un autre) l'action ne doit avoir aucun effet. On autorise donc un algorithme concret $\underline{A_j}$ à effectuer un pas d'un algorithme abstrait $\overline{A_i}$ dont il n'est pas un raffinement. Les deux situations possibles sont illustrées sur les figures 2 et 3. C'est à dire que soit le pas concret $\underline{\delta_j}$ est un bégaiement pour $\overline{A_j}$ et peut correspondre à un pas $\overline{\delta_i}$, soit $\underline{\delta_j}$ correspond à $\overline{\delta_j}$ et donc garantit la condition $\overline{rely_i}$ de la place considérée. Finalement, on obtient la définition suivante d'un raffinement valide.

Definition 6. *Un raffinement entre les spécifications* $\underline{Spec} = [\underline{A_1}, \ldots, \underline{A_n}]$ *et* $\overline{Spec} = [\overline{A_1}, \ldots, \overline{A_m}]$ *sous mapp et gluing_invariant est dit valide ssi :*

– $\underline{Spec}$ *et* $\overline{Spec}$ *sont valides,*
– $n = m$, i.e. $\underline{Spec}$ *et* $\overline{Spec}$ *sont composés du même nombre d'algorithmes,*
– *Pour tout* $i \in \{1, \ldots, n\}$, *l'algorithme concret* $\underline{A_i}$ *raffine l'algorithme abstrait* $\overline{A_i}$
– *Un algorithme concret* $\underline{A_j}$ *peut aussi effectuer un pas abstrait d'un autre algorithme* $\overline{A_i}$ *: pour tous* $i, j \in \{1, \ldots, n\}$, *transitions* $\underline{\delta_j} = (\underline{p_j}, pc_j')$, *places*

$\underline{p}_i$, $\overline{p}_i$, $\overline{p}_j$ avec $(\underline{p}_j, \overline{p}_j) \in mapp$ et $(\underline{p}_i, \overline{p}_i) \in mapp$ et tous environnements $\underline{genv}$, $\underline{genv}'$, $\underline{lenv}_j$, $\underline{lenv}'_j$, $\underline{lenv}_i$, $\overline{genv}$, $\overline{lenv}_j$ et $\overline{lenv}_i$ il existe des places $\overline{p}'_j$ et $\overline{p}'_i$ avec $(\underline{p}'_j, \overline{p}'_j) \in mapp$ et $(\underline{p}'_i, \overline{p}'_i) \in mapp$, ainsi que des environnements $\overline{genv}'$, $\overline{lenv}'_j$ et $\overline{lenv}'_i$ avec

$$
\begin{aligned}
& gluing_invariant(\underline{genv}, \underline{lenv}_j, \overline{genv}, \underline{lenv}_j) \\
& \wedge\ gluing_invariant(\underline{genv}, \underline{lenv}_i, \overline{genv}, \underline{lenv}_i) \\
& \wedge\ local_invariant_j(\underline{p}_j, \underline{genv}, \underline{lenv}_j) \wedge local_invariant_i(\underline{p}_i, \underline{genv}, \underline{lenv}_i) \\
& \wedge\ \underline{guard}_j(\underline{\delta}_j, \underline{genv}, \underline{lenv}_j) \wedge \underline{effect}_j(\underline{\delta}_j, \underline{genv}, \underline{lenv}_j, \underline{genv}', \underline{lenv}'_j) \\
\Rightarrow\ & \overline{p}'_i = \overline{p}_i \wedge ((\overline{p}_j, \overline{p}'_j) \in \overline{\Delta}_j \vee \overline{p}'_j = \overline{p}_j) \\
& \wedge\ \overline{lenv}'_i = \overline{lenv}_i \wedge \overline{rely}_i(\overline{p}_i, \overline{genv}, \overline{lenv}_i, \overline{genv}', \overline{lenv}'_i) \\
& \vee\ (\overline{p}_i, \overline{p}'_i) \in \overline{\Delta}_i \wedge \overline{p}'_j = \overline{p}_j \\
& \overline{guard}_i((\overline{p}_i, \overline{p}'_i), \overline{genv}, \overline{lenv}_i) \wedge \overline{effect}_i((\overline{p}_i, \overline{p}'_i), \overline{genv}, \overline{lenv}_i, \overline{genv}', \overline{lenv}'_i)
\end{aligned}
$$

Ces conditions de raffinement nous permettent de reconstruire une trace abstraite de $\overline{Spec}$ pour toute trace concrète de $\underline{Spec}$ de telle manière que les places concrètes et abstraites soient toujours liées par la relation *mapp* et que les environnements soient liés par *gluing_invariant*.

Theorem 1. *Soient mapp, gluing_invariant, $\underline{Spec}$ et $\overline{Spec}$ tels que $\underline{Spec}$ raffine $\overline{Spec}$ sous mapp et gluing_invariant, alors pour toute trace concrète de $\underline{Spec}$ il existe une trace abstraite de $\overline{Spec}$ reliée par mapp et gluing_invariant.*

La reconstitution de la trace abstraite se fait par induction. En effet, pour chaque transition concrète $\underline{\delta}_j$, on cherche à voir si elle effectue une transition $\overline{\delta}_j$ ou alors une transition $\overline{\delta}_i$ d'un autre algorithme abstrait. Dans le deuxième cas la transition est permise car $\overline{\delta}_i$ respecte la condition $\overline{rely}_j$ et on peut construire la trace abstraite correspondante. Cela nous indique au passage que la place $\underline{p}_i$ est aussi un raffinement de $\overline{p}'_i$ donc lorsque la transition $\underline{\delta}_i$ raffinant naturellement la transition abstraite $\overline{\delta}_i$ sera prise, il existera bien un environnement abstrait suivant satisfaisant la condition $\overline{rely}_i$.

Ces définitions et ce théorème nous permettent donc d'assurer la correction de la méthode. Dans le cas d'une spécification abstraite atomique elle nous assure la linéarisabilité de l'implémentation.

4 Exemples

Le premier exemple illustre un raffinement très courant. Il a été developpé dans l'assistant à la preuve Isabelle/HOL [10]. Le deuxième illustre la possibilité pour un algorithme d'effectuer le pas abstrait d'un autre. Pour cet exemple, on a écrit un générateur d'obligations de preuve qui ont ensuite été prouvés automatiquement par Isabelle (en utilisant la tactique auto). Il est prévu par la suite de soumettre ces obligations à un prouveur automatique.

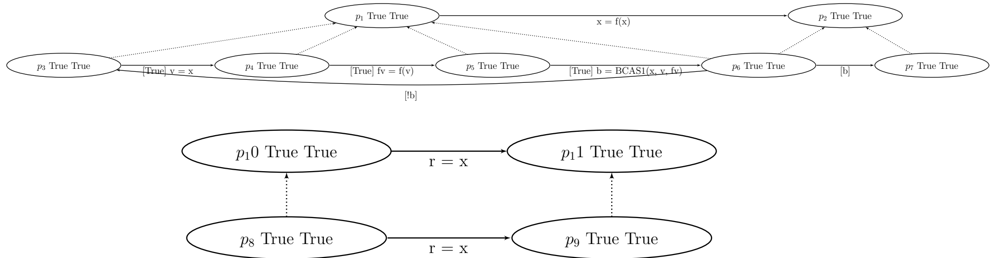


Fig. 4. Raffinement des algorithmes d'affectation.

4.1 Premier exemple : affectation

Pour l'exemple, on va appliquer la méthode sur un autre schéma très utilisé. Le premier algorithme effectue la mise à jour

```
x = f(x);
```

pour une fonction `f` non spécifiée. Le deuxième algorithme accède à la variable `x` en lecture. Comme la fonction `f` peut être complexe et la variable `x` est partagée entre plusieurs processus, on ne peut pas se contenter dans une implémentation de lire `x`, calculer `f(x)`, puis changer `x`. On va alors effectuer le calcul en local et utiliser la commande CAS_1 pour effectuer l'affectation de manière atomique :

```
b = false;
do {
  v = x;
  fv = f(v);
  b = BCAS1(x, v, fv);
} while(!b);
```

Le deuxième algorithme reste inchangé dans le raffinement. Les relations de raffinement entre ces algorithmes concrets et abstraits apparaissent dans la figure 4, et il est facile à vérifier que toutes les conditions des définitions 5 et 6 sont bien vérifiées.

4.2 Autre exemple

Notre deuxième exemple illustre le fait qu'un algorithme puisse effectuer un pas d'un autre. En effet, l'algorithme 2 (voir figure 6) attend une condition a qui est positionnée par l'algorithme 1 (voir figure 5). Dans l'algorithme concret, l'algorithme 1 commence par annoncer qu'il va effectuer l'action en positionnant b. L'algorithme 2 peut alors effectuer l'action lui-même. Cela fonctionne sur cet exemple car l'action a' effectuée deux fois a le même effet que lorsqu'elle

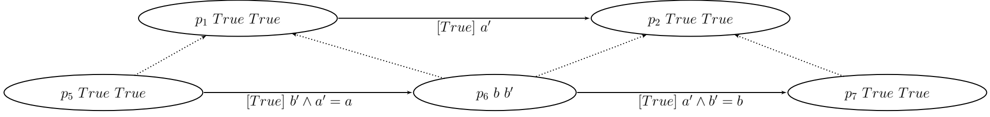


Fig. 5. Algorithmes 1, abstrait et concret.

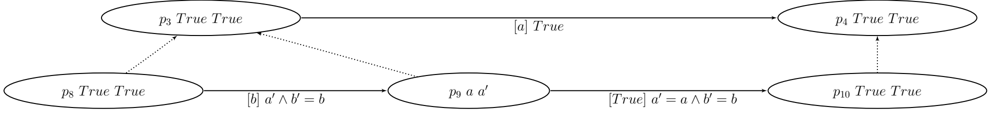


Fig. 6. Algorithmes 2, abstrait et concret.

est effectué une seule fois, en d'autres termes $a' \cdot a' \equiv a'$. Aussi, on vérifie aisément que les versions concrètes de ces deux algorithmes raffinent les versions abstraites.

5 Comparaisons

Il existe de nombreuses méthodes formelles pour prouver et/ou développer des algorithmes et des systèmes. Trois méthodes très connues et liées à nos travaux sont B, TLA$^+$ et l'approche «rely-guarantee». Dans cette section nous mettons en relief les différences entre ces méthodes et celle présentée ici. Les similitudes sont nombreuses puisque nous avons tout d'abord cherché à utiliser ces méthodes avant d'écrire la notre.

5.1 B

Abrial [1] a développé la méthode B ainsi que sa variante B événémentiel [2]. Elle permet de spécifier un système et d'en développer des raffinements. La méthodes est bien outillée et très utilisée dans le domaine industriel. Malheureusement, son orientation système rend complexes et peu lisibles les spécifications d'algorithmes.

En effet, il faut explicitement parler des processus et des places. De plus l'utilisation d'un unique invariant global le rend peu lisible. En effet, pour reconstruire l'invariant global, on a besoin de prédicats de place. Il faut alors compléter *effect* afin d'indiquer le contrôle de flot et de combiner les invariants locaux en un invariant global de taille importante.

Un inconvénient plus conceptuel de la méthode B réside dans le fait que le raffinement se fait toujours par événement. Ceci nous oblige d'anticiper les situations où un algorithme peut effectuer l'action d'un autre algorithme abstrait. Autrement dit, on est obligé d'indiquer les points de linéarisabilité ce qui peut rendre les spécifications assez obscures.

5.2 TLA$^+$ et $^+$CAL

Le langage de spécification TLA$^+$ et le langage $^+$CAL ont été développés par Lamport [8,9]. Les spécifications $^+$CAL sont en fait transformées en TLA$^+$ afin d'être interprétées et analysées par TLC, le model checker de TLA$^+$. Le langage $^+$CAL ressemble a du pseudo-code et a été développé afin de vérifier des algorithmes parallèles et distribués. Malheureusement, $^+$CAL souffre de deux problèmes. Le premier est qu'il est impossible de spécifier librement le niveau d'atomicité car il est souvent imposé par le langage (par exemple dans les branches d'une conditionnelle). Le deuxième problème est que $^+$CAL et TLA$^+$ ne sont outillés que d'un model checker. Or les spécifications qui nous intéressent sont difficilement traitables par un model checker.

5.3 Rely-Guarantee en Isabelle

L'approche «rely-guarantee» a été conçue par Jones [7], voir aussi le livre [3]. Son objectif est de permettre une vérification compositionnelle de programmes parallèles. La méthode étend les triplets de Hoare en rajoutant deux formules : la première caractérisant ce que peuvent faire les autres systèmes (comme notre condition *rely*), la deuxième caractérisant ce que fait le système (*guarantee*). On peut alors décrire dans *guarantee* l'effet atomique de l'algorithme. Mais ceci ne permet pas de vérifier complètement le système car il pourrait exécuter plusieurs fois l'action décrite par le prédicat *guarantee*. On pourrait rajouter des variables «fantômes» mais cela complique la spécification. Il n'est pas prévu non plus de développer conjointement des algorithmes par raffinement.

Une librairie «rely-guarantee» [12,13] pour Isabelle a été développée par Prensa Nieto. Cette formalisation nous permettrait de vérifier nos algorithmes à condition de rajouter les primitives atomiques. C'est ce que nous avons fait en travail préliminaire. Mais la méthode ne supporte pas le raffinement.

6 Conclusion et travail futur

On a présenté une méthode permettant de vérifier la spécification d'une famille d'algorithmes. Cette méthode se veut surtout adaptée à la vérification d'algorithmes sans verrou. Elle est fondée sur la description de plusieurs algorithmes qui sont exécutés par un nombre quelconque de processus et qui opérent sur une mémoire partagée. Nous avons formalisé les concepts d'algorithmes et de systèmes et défini les obligations de preuve pour s'assurer de la cohérence d'une spécification à un certain niveau d'abstraction. Une notion de raffinement d'algorithmes et de systèmes permet le développement conjoint de plusieurs algorithmes. Elle admet notamment qu'une opération d'un certain algorithme puisse être effectuée par un algorithme différent ; cette technique est souvent utilisée dans les algorithmes qui nous intéressent comme nous avons vu dans le cas du RDCSS (section 2.3). Or, ce type de raffinement est souvent difficile à justiier dans des méthodes généralistes comme B ou TLA$^+$.

Il nous reste à valider notre méthode en l'appliquant à des études de cas plus conséquents que les simples exemples présentés ici, notamment au cas de l'algorithme RDCSS puis CAS_n de Harris et al. [5]. Par ailleurs, nous sommes en train de justifier la méthode en la formalisant et en démontrant sa correction dans l'assistant de preuve Isabelle/HOL.

Afin de mettre à la disposition un environnement outillé de preuve et de développement aux concepteurs d'algorithmes sans verrou, nous étudierons l'instanciation de la méthode à une syntaxe concrète. Nous avons observé que, bien que représentant des systèmes à états infinis, les algorithmes n'effectuent pas en général des opérations complexes sur leurs données. C'est pourquoi nous souhaitons concevoir un générateur d'obligations de preuve qui pourra s'interfacer avec des outils de preuve automatiques. Afin de traiter des algorithmes plus complexes, il conviendra de prévoir le remplacement d'une transition par un sous-algorithme dont la correction a déjà été démontrée.

Références

1. Abrial, J.-R. *The B-book : assigning programs to meanings.* Cambridge University Press, New York, NY, USA, 1996.
2. Abrial, J.-R., Cansell, D., and Méry, D. Refinement and Reachability in Event_B. In *Formal Specification and Development in Z and B (ZB 2005)* (Guilford, UK, 2005), H. Treharne, S. King, and M. Henson, Eds., vol. 3455 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 222–241.
3. de Roever, W.-P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., and Zwiers, J. *Concurrency Verification : Introduction to Compositional and Noncompositional Methods.* Cambridge University Press, 2001.
4. Gao, H. *Design and verification of lock-free parallel algorithms.* PhD thesis, University of Groningen, Apr. 2005.
5. Harris, T. L., Fraser, K., and Pratt, I. A. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Symposium on Distributed Computing* (2002).
6. Herlihy, M. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems 13*, 1 (January 1991), 124–149.
7. Jones, C. B. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems 5*, 4 (Oct. 1983), 596–619.
8. Lamport, L. *Specifying Systems.* Addison-Wesley, Boston, Mass., 2002.
9. Lamport, L. Checking a multithreaded algorithm with +cal. In *20th Intl. Symp. Distributed Computing (DISC 2006)* (Stockholm, Sweden, 2006), S. Dolev, Ed., vol. 4167 of *Lecture Notes in Computer Science*, pp. 151–163.
10. Nipkow, T., Paulson, L., and Wenzel, M. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic.* No. 2283 in Lecture Notes in Computer Science. Springer-Verlag, 2002.
11. Parkinson, M., Bornat, R., and O'Hearn, P. Modular verification of a non-blocking stack. *SIGPLAN Not. 42*, 1 (2007), 297–302.

12. Prensa Nieto, L. *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL*. PhD thesis, Technische Universität München, 2002.

13. Prensa Nieto, L. The rely-guarantee method in Isabelle/HOL. In *European Symposium on Programming (ESOP 2003)* (Warsaw, Poland, 2003), P. Degano, Ed., vol. 2618 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 348–362.

14. Sundell, H., and Tsigas, P. Lock-free and practical deques using single-word compare-and-swap. Tech. Rep. 2004-02, Computing Science, Chalmers University of Technology, Mar. 2004.

15. Vafeiadis, V., Herlihy, M., Hoare, T., and Shapiro, M. Proving correctness of highly-concurrent linearisable objects. In *PPoPP '06 : Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2006), ACM Press, pp. 129–136.

16. Valois, J. D. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems* (Las Vegas, NV, 1994), pp. 64–69.

Contrôler le contrôle d'accès : Approches formelles

Mathieu Jaume and Charles Morisset

SPI - LIP6 - Université Paris 6
104 av. du Président Kennedy
F-75016 Paris, France

Abstract. Un des aspects de la sécurité en informatique concerne le contrôle des accès aux données d'un système pour lequel différentes politiques de sécurité peuvent être mises en application. Toutefois, rien ne sert de mettre en place une politique de sécurité pour gérer un système si les programmes chargés de garantir le bon fonctionnement de cette politique ne sont pas fiables. Ne pas apporter de garanties fortes sur la correction de tels programmes reviendrait à construire un château fort avec une porte en papier. Cet article rend compte de manière informelle de différentes expériences permettant d'obtenir des développements formels de politiques de contrôle d'accès. Ces développements nous conduisent à introduire un "cadre sémantique" dans lequel il est possible de spécifier, implanter et comparer des politiques de contrôle d'accès.

Keywords: Contrôle d'accès, Méthodes formelles

1 Introduction – Motivations

La protection des informations d'un système informatique est une préoccupation majeure. L'apparition de systèmes informatiques de plus en plus grands, la dissémination de l'information et le développement des réseaux, permettent dorénavant des attaques depuis l'extérieur et rendent la protection des informations de plus en plus complexe. Comme dans toutes les autres disciplines scientifiques, le besoin de recourir à des modèles et formalismes mathématiques se fait ressentir pour mieux comprendre et analyser les problèmes liés à l'informatique. C'est de ce besoin que viennent les méthodes formelles. Dans [3], P. Amey définit la "chose formelle" comme une "chose soutenue par une rigueur mathématique". Ainsi, les méthodes formelles peuvent être vues comme des "méthodes soutenues par une rigueur mathématique" dont l'absence d'ambiguïté permet de spécifier et dans certains cas d'implanter un système en garantissant que certaines propriétés sont respectées. Lorsqu'il s'agit de systèmes logiciels critiques, ces propriétés peuvent être vitales.

Dans cet article, nous utilisons les méthodes formelles pour étudier certaines des propriétés classiques de sécurité des systèmes informatiques. Nous nous intéressons plus particulièrement au contrôle d'accès. Il s'agit de régir et

de gérer les accès effectués selon certains modes (lecture, écriture, ...) par des sujets, les entités actives (processus, programmes, utilisateurs, ...) sur des objets, les entités passives (données, fichiers, programmes, ...). Dans ce cadre, on distingue essentiellement trois propriétés : la confidentialité, qui assure que les données ne sont lues que par les personnes autorisées, l'intégrité, qui assure que les données ne sont modifiées que par les personnes autorisées et la disponibilité, qui assure que les données sont accessibles aux personnes autorisées.

La disponibilité semble être la propriété la plus difficile à garantir, car elle ne dépend pas uniquement du système en question, mais également de son environnement et de ses ressources (par exemple, il suffit de couper l'alimentation électrique d'un serveur pour que celui-ci ne soit plus disponible).

Deux procédés sont habituellement utilisés pour assurer la confidentialité et l'intégrité : l'utilisation de protocoles cryptographiques pour les communications au sein du système, ainsi que depuis ou vers l'extérieur, et la mise en place d'un moniteur de référence qui va gérer au sein du système l'accès aux données. Les mécanismes de ces deux approches sont différents (chacun est plus ou moins sensible à certaines attaques) :

- la cryptographie assure la protection *a posteriori* : n'importe qui peut accéder aux données, mais seuls les utilisateurs possédant la clé peuvent les lire ou les modifier,
- un moniteur de référence assure la protection *a priori* : seuls les utilisateurs autorisés peuvent accéder à l'information.

Dans cet article nous nous intéressons à l'approche “moniteur de référence” pour gérer les accès effectués dans un système. Les Critères Communs (recueil de normes définies par des agences gouvernementales) fournissent une méthodologie permettant d'atteindre des hauts niveaux de sécurité. Ils définissent à la fois un cadre de travail pour la conception et la réalisation de logiciels et une référence pour les utilisateurs de ces logiciels. Les hauts niveaux de sûreté des Critères Communs [1] (EAL 5 à 7) requièrent l'utilisation de méthodes formelles dans les étapes de spécification et de conception du logiciel. Selon les Critères Communs, un système est vu comme une installation donnée de technologies de l'information, avec un objectif et un environnement opérationnel particuliers. Une politique de sécurité est un ensemble de règles qui précisent comment gérer, protéger ou distribuer les informations ou ressources du système. Un moniteur de référence est une machine abstraite qui applique les politiques de contrôle d'accès d'un système, ces politiques étant un sous-ensemble des politiques de sécurité. Toujours selon ces mêmes Critères Communs, un moniteur de référence doit posséder les trois caractéristiques suivantes :

- Des sujets non sûrs ne peuvent pas interférer avec son fonctionnement, i.e. il est à l'épreuve d'une intrusion physique.
- Des sujets non sûrs ne peuvent pas court-circuiter les contrôles qu'il effectue, i.e. il est systématiquement appelé.
- Il est suffisamment simple pour être analysé et pour comprendre son comportement, i.e. sa conception est simple.

Ces trois caractéristiques, introduites dans [4], sont connues sous l'acronyme *NEAT*, pour "*Non-bypassable*" (il n'est pas possible d'éviter les fonctions de sécurité), "*Evaluatable*" (les fonctions de sécurité sont suffisamment simples pour être mathématiquement vérifiées et évaluées), "*Always Invoked*" (les fonctions de sécurité sont tout le temps appelées) et "*Tamperproof*" (les fonctions de sécurité ne peuvent pas être altérées). Cet acronyme est défini dans le cadre de *MILS* (*Multiple Independent Levels of Security*), une approche de développement de systèmes sécurisés (`http://www.ois.com/MILS/`).

A plus long terme, notre objectif est d'obtenir une bibliothèque certifiée de moniteurs de référence mettant en application différentes politiques de sécurité. En effet, le développement logiciel d'un moniteur de référence n'a de sens que s'il permet de garantir les propriétés de sécurité pour lesquelles il a été conçu. Pour atteindre de hauts niveaux de certification, il est nécessaire de fournir un modèle formel du système permettant d'obtenir des preuves formelles mécanisées. Nous présentons ici plusieurs expériences menées pour atteindre cet objectif. Comme nous allons le voir, trois difficultés sont à prendre en compte dans ce travail.

La première est classique et provient de l'activité même de formalisation : le passage de l'informel au formel nécessite d'identifer les hypothèses implicites et d'expliciter totalement le système à modéliser. Cette difficulté est illustrée dans la section 2.

Afin de valider la formalisation obtenue, il s'agit alors de la "mécaniser" (i.e. de l'implanter). C'est la deuxième difficulté : certaines preuves "triviales" à obtenir sur le papier le sont beaucoup moins avec un assistant à la preuve. La section 3 présente deux développements formels de politiques de contrôle d'accès.

Quoi qu'il en soit, conduire un développement formel de cette nature est une activité chronophage en temps. Il faut, dans la mesure du possible, factoriser les spécifications et les preuves formelles afin de faciliter la réutilisation de ces développements. Bien sûr, l'utilisation d'un atelier de développement formel muni de mécanismes facilitant l'écriture modulaire de spécifications, de définitions et de preuves permet d'atteindre une certaine "réutilisabilité" des développements conduits. Ce n'est toutefois pas suffisant. Il est en effet souhaitable de concevoir un cadre formel uniforme dans lequel puissent s'exprimer les modèles de contrôle d'accès que nous envisageons. C'est la troisième difficulté : il s'agit à la fois d'identifier les "ingrédients" communs aux politiques de contrôle d'accès, d'exprimer les propriétés génériques qu'ils vérifient, d'en prouver certaines et de formaliser les politiques envisagées comme des instances du cadre générique. La section 4 décrit les grandes lignes d'un tel cadre.

La suite de cet article illustre ces difficultés en considérant deux politiques classiques de contrôle d'accès :

- la politique de Bell et LaPadula [18, 5], initialement conçue pour les militaires, repose sur un ensemble partiellement ordonné de niveaux de sécurité associés aux objets et aux sujets (il s'agit plus précisément d'un treillis);
- la politique de la Muraille de Chine [8], permettant d'éviter les conflits d'intérêt et conçue pour le monde des consultants, repose sur un partitionnement des objets.

Cet article a pour objectif de fournir une présentation synthétique des divers travaux que nous avons réalisés sur la formalisation et l'implantation des politiques de contrôle d'accès dans un cadre formel [14–17].

2 Formaliser pour mieux comprendre

Des détails qui n'en sont pas ...

Le passage de l'informel au formel permet souvent de détecter des erreurs. Nous présentons ici un exemple de manque de précision dans la formalisation d'une politique de contrôle d'accès qui à première vue peut sembler sans conséquence mais qui permet finalement de violer la politique de sécurité modélisée.

La politique de Bell et LaPadula est habituellement décrite par une machine à états. Elle dépend d'un ensemble $\mathcal{S}$ de sujets, d'un ensemble $\mathcal{O}$ d'objets, d'un ensemble $\mathcal{A}$ de modes d'accès et d'un treillis fini $(\mathcal{L}, \preceq, \curlyvee, \curlywedge)$ de niveaux de sécurité. Un état du système est un tuple (m, D, f_s, f_o) où m est l'ensemble des accès courants, D est l'ensemble des droits d'accès et $f_s : \mathcal{S} \rightarrow \mathcal{L}$ (resp. $f_o : \mathcal{O} \rightarrow \mathcal{L}$) est une fonction associant un niveau de sécurité aux sujets (resp. aux objets). Les éléments de m (resp. de D) sont des accès représentés par des triplets de la forme (s, o, a) exprimant qu'un sujet s accède (resp. dispose des droits discrétionnaires pour accéder) à un objet o selon le mode d'accès a. Les trois propriétés de sécurité définies dans la politique de Bell et LaPadula sont les suivantes.

- Propriété DAC (*Discretionnary Access Control*) : La propriété DAC exprime simplement que tout accès courant est conforme aux droits d'accès. Plus formellement, cette propriété s'écrit :

$$m \subseteq D$$

- Propriétés MAC et MAC* (*Mandatory Access Control*) :
  - La propriété MAC connue sous le nom de "*no read-up property*" exprime qu'un sujet ne peut accèder en lecture à un objet que si son niveau de sécurité est supérieur à celui de l'objet accédé. Plus formellement, cette propriété s'écrit :

$$(s, o, \text{read}) \in m \Rightarrow f_s(s) \succeq f_o(o)$$

  - La propriété MAC* connue sous le nom de "*no write-down property*" permet d'éviter qu'un sujet "malicieux" recopie de l'information sensible à un niveau de sécurité inférieur. Plus formellement, cette propriété s'écrit :

$$((s, o_1, \text{read}) \in m \wedge (s, o_2, \text{write}) \in m) \Rightarrow f_o(o_1) \preceq f_o(o_2) \qquad (1)$$

Dans [19], McLean introduit une "algèbre de sécurité" à partir de laquelle il formalise une version enrichie de la politique de Bell et LaPadula, essentiellement

en considérant la notion d'accès conjoints. Cette notion vient du fait que dans certains systèmes, il existe des opérations qui doivent être effectuées par plusieurs sujets en même temps pour pouvoir être autorisées. L'exemple le plus classique vient du monde militaire, où pour lancer un missile, il faut que des personnes habilitées appuient sur un bouton en même temps. La propriété de sécurité $\mathsf{MAC}^{\star}$ est alors (re)définie comme suit :

> [19] "*a state is $\star$-secure if for any subjects S_1, S_2 and objects o_1, o_2, if $(S_1, o_1, read) \in m$ and $(S_2, o_2, write) \in m$ and the classification of o_1 dominates that of o_2, then $S_1 \cap S_2 = \emptyset$* "

Ici, S_1 et S_2 sont des ensembles de sujets et un accès est un triplet (S, o, a) exprimant que les sujets présents dans l'ensemble S accèdent conjointement à l'objet o selon le mode a. La formalisation de cet énoncé permet d'obtenir la spécification suivante :

$$((S_1, o_1, \text{read}) \in m \wedge (S_2, o_2, \text{write}) \in m \wedge f_o(o_2) \prec f_o(o_1)) \Rightarrow S_1 \cap S_2 = \emptyset$$

Par contraposition, on obtient finalement :

$$((S_1, o_1, \text{read}) \in m \wedge (S_2, o_2, \text{write}) \in m \wedge S_1 \cap S_2 \neq \emptyset) \Rightarrow \neg(f_o(o_2) \prec f_o(o_1)) \quad (2)$$

Or, si on se limite au cas où S_1 et S_2 sont réduits à des singletons, c'est-à-dire si on ne considère pas les accès conjoints, les deux propriétés (1) et (2) ne sont pas équivalentes. En effet, pour que ces deux propriétés soient équivalentes il faudrait que l'ordre sur les niveaux de sécurité soit total. Or, il ne s'agit que d'un ordre partiel puisque l'ensemble des niveaux de sécurité est muni d'une structure de treillis. Malheureusement, la propriété (2) n'exprime alors pas la propriété souhaitée comme l'illustre l'exemple suivant. En effet, comme le montre la figure 1, si l'on se contente de respecter la propriété (2), il est possible qu'un sujet s_1 accède simultanément en lecture à un objet dont le niveau de sécurité est l_1 et en écriture à un objet dont le niveau de sécurité est l_2 tel que l_1 et l_2 ne soient pas comparables. Un sujet s_2 peut alors lire ce dernier objet de niveau l_2 et écrire simultanément dans un objet de niveau l_3 tel que l_3 et l_2 ne soient pas comparables mais tel que l_3 soit inférieur à l_1. Il y a alors une "fuite d'information vers le bas" puisqu'il devient ainsi possible de recopier les informations d'un objet de niveau l_1 dans un objet de niveau $l_3 \preceq l_1$.

L'activité de formalisation permet de mettre en lumière de tels problèmes. Ils peuvent être découverts en essayant de prouver des "énoncés faux" (par exemple en essayant de prouver l'équivalence entre (1) et (2) lorsque S_1 et S_2 sont des singletons), mais ils peuvent aussi être découverts en utilisant des méthodes de test (par exemple en testant une propriété assurant qu'il n'existe pas de séquence d'accès permettant de copier de l'information de niveau élevé dans des objets de niveaux inférieurs).

Objets informels et définitions formelles

La politique de la Muraille de Chine [8] a été créée pour résoudre les problèmes de conflit d'intérêt dans le monde des consultants. Chaque objet du système

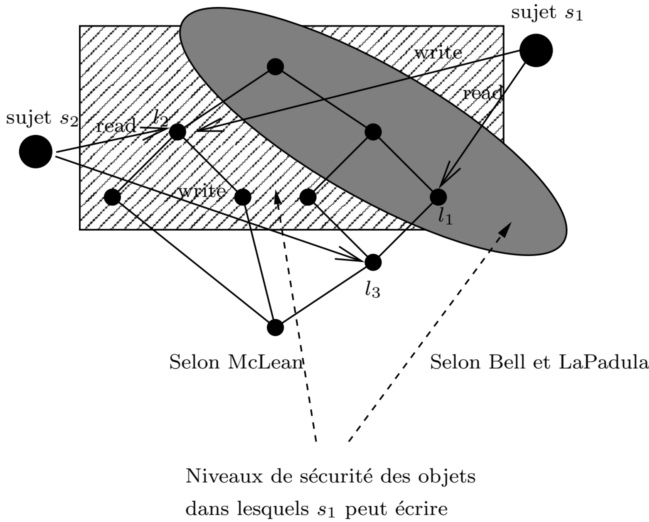


Fig. 1. Violation de la politique de sécurité

appartient à un ensemble de données d'une compagnie, et chaque compagnie appartient à une unique classe de conflit d'intérêt. Ces classes correspondent à des milieux professionels distincts, comme par exemple les banques ou les compagnies pétrolières. La politique de sécurité consiste à dire qu'un consultant peut travailler en même temps pour une banque et une compagnie pétrolière, mais pas pour deux banques ou deux compagnies pétrolières. A cela il faut ajouter l'existence d'une classe de conflit spéciale qui ne contient qu'une seule compagnie et qui contient les informations "sanitisées", c'est-à-dire qu'elles peuvent être lues par tout le monde sans provoquer de conflit d'intérêt. La politique est énoncée comme la combinaison de deux propriétés de sécurité. La première est la règle de sécurité-simple :

> [8] "*Access is only granted if the object requested: (a) is in the same company dataset as an object already accessed by that subject, i.e. within the Wall, or (b) belongs to an entirely different conflict of interest class.*"

La deuxième est celle de sécurité-$\star$:

> [8]"*Write access is only permitted if (a) access is permitted by the simple security rule, and (b) no object can be read which is in a different company dataset to the one for which write access is requested and contains unsanitized information.*"

Ces deux propriétés sont volontairement répétées en anglais, afin de permettre au lecteur de mieux percevoir le niveau de formalisation de la description originale de cette politique. Ces deux propriétés sont difficilement compréhensibles à

la première lecture et également difficiles à formaliser. En effet, quel est le statut exact d'un objet qui "peut être lu" ? Est ce un objet que le sujet est en train de lire ? ou bien un objet qu'il lira dans l'avenir ? ou encore est ce que cela signifie qu'il a les droits pour le faire ? Dans ce dernier cas, cela signifie que lorsqu'un sujet veut écrire dans un objet, il doit révoquer tous les droits qu'il a sur des objets appartenant à une autre classe de conflit. D'autre part, dans l'article original, il n'est mentionné nulle part que les accès peuvent être relâchés. Est ce que cela signifie qu'il n'est pas possible de le faire, et qu'un accès est éternel ? Dans ce cas, le modèle de la Muraille de Chine est très restrictif, puisqu'une fois qu'un sujet a écrit dans un objet, il ne peut plus avoir d'accès en lecture ou en écriture à un objet d'une classe de conflit différente. Pour formaliser ces deux propriétés de sécurité, il faut faire des choix quant à l'interprétation de ces propriétés. Le danger provient de ces choix qui permettent plusieurs formalisations non équivalentes de cette politique.

Enfin, l'étape de formalisation permet de distinguer clairement les propriétés de sécurité souhaitées du moniteur de référence chargé de les faire respecter. En effet, avec la présentation informelle de la politique de la Muraille de Chine, il est tentant de penser que la description des propriétés de sécurité constitue une description algorithmique de la fonction d'autorisation des accès ... ce qui n'est pas le cas.

3 Mécaniser la formalisation

3.1 Formalisation de la politique de Bell et LaPadula avec Coq

Nous considérons ici la formalisation complète d'une politique de contrôle d'accès, celle de Bell et LaPadula [18, 5], à l'aide du système Coq [23], une implantation d'un λ-calcul typé avec types dépendants et types inductifs permettant d'obtenir des preuves formelles de manière interactive.

Ce développement est décrit en détail dans [14] et nous n'en esquissons ici que les grandes lignes. Il est paramétré par un ensemble $\mathcal{S}$ de sujets, un ensemble $\mathcal{O}$ d'objets, un ensemble $\mathcal{A}$ de modes d'accès et un treillis fini $(\mathcal{L}, \preceq, \curlyvee, \curlywedge)$ de niveaux de sécurité. Ce treillis est en fait obtenu à partir de deux paramètres : un ensemble $\mathcal{K}$ de "domaines" (connu sous le nom de "*needs-to-know*"), comme par exemple { nucléaire, médical , ... }, et un ensemble $\mathcal{C}$ de classifications, comme par exemple { Top-secret , secret , public , ... }, muni d'une relation d'ordre total. $\mathcal{L}$ est alors défini comme le treillis produit $\mathcal{C} \times T_k$ où $T_k = (\wp(\mathcal{K}), \subseteq, \cup, \cap)$ est le treillis des parties de $\mathcal{K}$.

A partir de la notion d'états du système, les fonctions de transition sont définies comme des fonctions de $\mathcal{R} \times \Sigma$ dans $\mathcal{D} \times \Sigma$ où $\mathcal{R}$ est un ensemble de requêtes, $\mathcal{D} = \{\text{yes}, \text{no}\}$ contient les réponses possibles et Σ est l'ensemble des états. Une fonction de transition τ correspond à la mise en application d'une politique de sécurité et sera qualifiée de fonction "sûre" si et seulement si elle préserve les propriétés de sécurité (i.e. ssi elle transforme chaque état vérifiant les propriétés DAC, MAC et $\mathsf{MAC}^\star$ en états vérifiant encore ces propriétés). Le

développement obtenu consiste en une implantation dans Coq de la fonction de transition τ_{BLP} introduite par Bell et LaPadula ainsi que de la preuve mécanisée du célèbre "*Basic Security Theorem*" [5] affirmant que τ_{BLP} est "sûre". Le mécanisme d'extraction de Coq a permis, à partir de cette preuve, d'obtenir un programme certifié qui implante τ_{BLP}.

Ce développement nous a permis de spécifier, implanter et raisonner dans un cadre formel et d'atteindre un niveau de confiance élevé sur le programme obtenu. Toutefois, si elle remplit ses objectifs en termes de garantie de correction, une telle approche peut conduire en revanche à des développements difficilement réutilisables. En effet, la moindre modification tant sur la spécification de la politique que sur son implantation conduit à modifier une preuve de plus d'un millier de lignes. Il s'agit là d'un exercice particulièrement chronophage.

Pour faciliter la réutilisation, il est souhaitable de se placer dans un cadre formel caractérisant les éléments communs aux politiques de contrôle d'accès et permettant d'en dériver différentes implantations. Il est d'autre part préférable d'utiliser un environnement de développement qui facilite l'implantation de ces différents traits. Le système Coq dispose à présent de mécanismes permettant de conduire des développements formels de manière modulaire, mais nous avons choisi par la suite d'utiliser l'atelier Focal développé au sein de notre équipe. Le paragraphe suivant présente une expérience suivant cette approche.

3.2 Développement de la politique de Bell et LaPadula vue comme une instance de l'algèbre de McLean avec l'atelier Focal

Afin d'éviter les inconvénients du développement présenté dans la section 3.1, nous avons choisi de définir, dans l'environnement de développement Focal [22, 12], l' "algèbre de sécurité" introduite par McLean [19]. Ce cadre générique pour le contrôle d'accès peut ensuite être instancié afin d'en dériver une implantation de la politique de Bell et LaPadula, qui peut à son tour être utilisée pour gérer les accès à une base de données relationnelle. Nous décrivons ici l'architecture de l'ensemble de ces dévelopements qui sont détaillés dans [15, 16, 7].

Implantation de l' "algèbre de sécurité" de McLean avec Focal

L'atelier Focal [22, 12, 20], et la méthodologie sous-jacente, ont initialement été appliqués au calcul formel. C'est en fait ce domaine qui a servi de modèle et donné les lignes directrices du développement de l'atelier. Grâce à l'atelier Focal, R. Rioboo a pu construire une bibliothèque de calcul formel assez conséquente [24] comprenant des algorithmes complexes et dont l'efficacité est comparable à celle des meilleurs systèmes de calcul formel. Aujourd'hui, Focal est aussi utilisé dans le domaine de la sécurité : outre les développements que nous effectuons avec Focal sur les politiques de contrôle d'accès, Focal a été utilisé avec succès pour formaliser la politique de sécurité au sol d'un aéroport [10].

Focal est un environnement de programmation basé sur un langage muni de traits objets (héritage multiple, liaison tardive, redéfinition, ...) permettant non seulement de structurer un dévelopement de manière à le rendre facilement

réutilisable, mais aussi d'obtenir un logiciel par raffinements successifs en passant progressivement de la spécification à l'implantation. En effet, les constructions de ce langage autorisent l'écriture de spécifications, de programmes et de preuves. D'autre part, l'atelier Focal dispose à présent d'un démonstrateur automatique, Zenon, permettant d'obtenir de nombreuses preuves de manière automatique.

Toutes ces caractéristiques font de Focal un bon candidat pour implanter l'"algèbre de sécurité" introduite par McLean. Enfin, le choix de Focal est aussi motivé par notre expérience dans le domaine de la réutilisation et le "management" de preuves dans cet atelier [11, 21].

Pour profiter pleinement de la puissance des constructions de Focal en terme de réutilisabilité, nous avons choisi de ne pas développer directement telle ou telle politique de sécurité mais plutôt de considérer cette politique comme une instance d'un modèle plus général. Afin de factoriser le travail de formalisation et de développement, il est en effet souhaitable de se placer dans un cadre abstrait générique permettant de considérer chacune des politiques envisagées comme une instance. La littérature sur ce domaine est encore relativement restreinte et nous avons choisi d'implanter l'"algèbre des modèles de sécurité" introduite par J. McLean en 1988 [19]. Cette algèbre fournit un cadre dans lequel peuvent se définir certains des concepts que beaucoup de politiques de sécurité partagent. D'autre part, l'existence d'une importante bibliothèque de calcul formel [24] permet d'envisager aisément l'implantation dans Focal de cette algèbre qui fait appel à de nombreuses structures algébriques classiques (treillis, algèbres de Boole, ...).

Le cadre introduit par McLean permet la description d'une politique de contrôle d'accès à trois niveaux de spécification différents (figure 2) : les frameworks, les modèles et les systèmes.

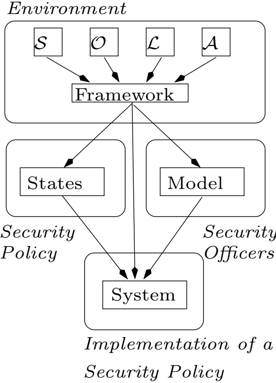


Fig. 2. Algèbre de Sécurité

Un framework décrit l'environnement : il est paramétré par un ensemble $\mathcal{S}$ de sujets, un ensemble $\mathcal{O}$ d'objets, un ensemble $\mathcal{A}$ de modes d'accès, un treillis $\mathcal{L}$ de niveaux de sécurité et spécifie quels sont les ensembles de sujets qui pourront conjointement demander à accéder à un objet ou demander à changer le niveau de sécurité d'un sujet ou d'un objet (sans toutefois spécifier comment sera traitée cette demande). Dans le cas le plus général, il s'agit des parties non vides de $\mathcal{S}$ (toute opération est initiée par au moins un sujet) mais il peut exister des contraintes sur ces ensembles. Pour certaines politiques, il est par exemple spécifié que si un groupe de sujets peut conjointement soumettre une requête, alors tout sous-ensemble de ce groupe peut également soumettre une requête. Dans le cas le plus courant pour lequel les accès conjoints ne sont pas pris en compte, les seuls ensembles autorisés à soumettre une requête sont les singletons construits à partir de $\mathcal{S}$. Une hiérarchie de frameworks est alors définie selon les propriétés vérifiées par ces ensembles de sujets.

Introduite par J.McLean afin de pallier les problèmes posés par des systèmes de contrôle d'accès parfois trop rudimentaires, la notion de modèle permet de fixer les règles régissant le changement de niveau de sécurité d'un sujet ou d'un objet. La notion de modèle est paramétrée par celle de framework. Un modèle spécifie quels sont les ensembles de sujets qui seront effectivement autorisés à modifier les niveaux de sécurité. Ici, encore, une hiérarchie de modèles est définie selon les propriétés vérifiées par ces ensembles de sujets. Certaines relations et opérations sur les modèles sont définies et permettent d'implanter l'ensemble des modèles comme une instance de treillis distributif.

Enfin, la notion d'état d'un système est définie. Les états décrivent les informations relatives aux divers niveaux de sécurité associés aux sujets et aux objets ainsi que les accès courants. Une fois la notion d'état définie, il est possible de spécifier par un prédicat quels sont les états sûrs, c'est-à-dire quelle est la politique de sécurité appliquée.

C'est à partir de la notion d'état et de modèle qu'il est alors possible de définir la notion de système qui spécifie ce qu'est une fonction de transition et quelles sont les propriétés qu'elle doit vérifier. Une fonction de transition entre états permet de décrire les changements d'états produits lors des requêtes d'accès émises par les sujets.

En définissant, c'est-à-dire en instanciant, chacune des entités spécifiées dans le cadre obtenu, nous avons défini la politique de Bell et LaPadula et avons fourni une implantation de la fonction de transition τ_{BLP} évoquée dans la secion 3.1.

Application au contrôle des accès d'une base de données

L'implantation de la politique de Bell et LaPadula obtenue dans le cadre décrit précédemment reste encore relativement générique. Par exemple, elle ne spécifie ni les sujets, ni les objets. En effet, un programme qui implante une certaine politique de contrôle d'accès est *a priori* indépendant du système sur lequel il doit s'appliquer (bases de données, systèmes de gestion de fichiers, ...). Il possède des paramètres destinés à prendre en compte l'environnement concret dans lequel il va s'exécuter. Nous allons maintenant illustrer succintement l'intégration de

tels programmes dans des systèmes “concrets”. Il s’agit ici d’une mise à l’épreuve du terrain de la méthode toute entière.

Afin d’illustrer l’utilisation du programme Focal implantant le système de Bell et LaPadula, nous avons mis en application ce programme pour le contrôle des accès dans une base de données relationnelle (MySQL). Cette base de données contient à la fois les données des utilisateurs, et les données relatives à la sécurité (droits d’accès, accès courants, niveaux de classification, “*needs-to-know*”). Les paramètres du système de Bell et LaPadula obtenu dans le paragraphe précédent sont instanciés par des objets stockés dans la partie ”sécurité” de la base de données et les états de ce système sont construits à partir de ces données. Deux tables particulières concernant les sujets et les objets sont présentes dans la zone dédiée aux données relatives à la sécurité de la base de données. La table `sujets` contient pour chaque sujet son identifiant unique, son login, son mot de passe, ainsi que son niveau de sécurité. De même, la table `objets` contient pour chaque objet son identifiant unique, le nom de la table à laquelle il correspond ainsi que son niveau de sécurité. Ainsi, les objets du sytème d’information sont les tables de la base de données. Une granularité plus fine pourrait être envisagée mais soulèverait des problèmes connus dans le domaine des bases de données pour lesquels des solutions existent mais sortent du cadre de notre prototype. Par exemple, considérer les tuples d’une base de données comme des objets conduit à un phénomène de polyinstanciation [25] lorsque deux tuples ont la même clé primaire, mais des informations de niveaux de sécurité différents ([6] propose des solutions à ce problème).

L’architecture de l’implantation se décompose en trois parties : la politique de sécurité, le gestionnaire de base de données et le moniteur de référence. La partie concernant la politique de sécurité est entièrement définie en Focal et repose sur le modèle de Bell et LaPadula. L’accès au gestionnaire de base de données s’effectue grâce à une interface définie en Focal proposant des fonctions de connexion/déconnexion à la base de données, ainsi que des fonctions permettant l’exécution d’une requête et la récupération du résultat. Ces fonctions reposent sur la librairie *ocaml-mysql* [2], et sont donc spécifiques au gestionnaire de base de données MySQL. La figure 3 illustre le traitement des requêtes effectué par l’application obtenue. Etant donnée une requête SQL soumise par un sujet (authentifié), le programme d’analyse de requêtes SQL que nous avons développé fait appel au système de Bell et LaPadula et selon la réponse obtenue traite ou refuse l’exécution de la requête soumise. Pour cela, nous avons défini pour chaque requête SQL, un ensemble de requêtes pour le système de Bell et LaPadula, donnant en quelque sorte une sémantique en termes d’accès aux requêtes SQL. On remarquera que les requêtes SQL soumises par l’utilisateur utilisent la syntaxe standard de SQL, rendant ainsi transparente pour l’utilisateur la mise en oeuvre de notre application, qui peut être vue comme un filtre entre l’utilisateur et le système de gestion de la base de données.

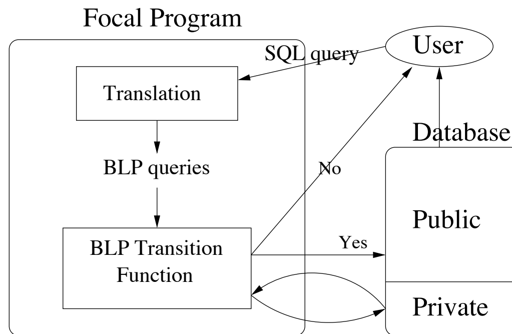


Fig. 3. Contrôle d'accès pour un SGBD

4 Vers un cadre sémantique pour le contrôle d'accès

A l'issue des travaux que nous venons de décrire, nous avons souhaité obtenir un développement certifié de la politique de la Muraille de Chine. Bien sûr, comme nous l'avons vu dans la section 2, les mêmes problèmes liés au passage de l'informel au formel sont apparus. D'autre part, lors de la formalisation de cette politique, le cadre de McLean s'est révélé à la fois trop contraignant (dans le contexte de la Muraille de Chine nous ne disposons pas d'un treillis de niveaux de sécurité mais d'un ensemble de classes de conflits) et trop peu expressif (en termes de propriétés de simulation permettant une plus grande réutilisabilité). Nous avons alors été amenés à concevoir un cadre sémantique pour le contrôle d'accès permettant de répondre à nos objectifs. Il ne s'agit pas de la définition d'un langage mais plutôt d'une spécification abstraite de ce qui constitue une politique de contrôle d'accès.

Afin de réutiliser le plus facilement possible le développement en Focal de la politique de Bell et LaPadula, nous souhaitons obtenir une implantation certifiée de la politique de la Muraille de Chine basée sur les travaux de R.S. Sandhu [27, 26] proposant une interprétation de la Muraille de Chine qui repose sur un treillis de niveaux de sécurité. Nous souhaitons également "comparer" cette implantation avec celle de la politique de Bell et LaPadula. Pour adopter cette approche, il est nécessaire de formaliser complètement, non seulement cette interprétation, mais aussi les notions de comparaison et d'interprétation d'une politique par une autre politique. Le cadre sémantique que nous proposons répond à ces objectifs :

- d'un point de vue pratique, il permet de comprendre certains mécanismes de réutilisation de politiques de sécurité
- d'un point de vue plus théorique, il permet d'introduire un préordre entre ces politiques.

Nous présentons ici les grandes lignes de ce cadre. Le détail d'une version préliminaire de ce cadre est décrit dans [17].

Une politique de contrôle d'accès repose sur des paramètres de sécurité (par exemple un treillis des niveaux de sécurité pour la politique de Bell et LaPadula ou des ensembles de classes de conflit et de compagnies pour la politique de la Muraille de Chine). Si l'on note ρ les paramètres de sécurité sur lesquels elle repose, une politique de contrôle d'accès $\mathbb{P}[\rho]$ est définie par un tuple :

$$\mathbb{P}[\rho] = (\mathcal{S}, \mathcal{O}, \mathcal{A}, \Sigma, \Omega)$$

où $\mathcal{S}$ est un ensemble de sujets, $\mathcal{O}$ est un ensemble d'objets, $\mathcal{A}$ est un ensemble de modes d'accès, Σ est l'ensemble des états du système et Ω est un prédicat sur Σ caractérisant les états sûrs (Ω correspond à la spécification des propriétés de sécurité de la politique et on note $\Sigma_{|\Omega}$ le sous-ensemble de Σ contenant les états sûrs). Une politique de contrôle d'accès correspond essentiellement à la description de ce qu'est un état "sûr".

Dans un système, les utilisateurs (i.e. les sujets) soumettent des requêtes pour accéder et/ou modifier des informations. On note $\mathcal{R}$ le langage de requêtes utilisé et on désigne par :

$$\llbracket \mathcal{R} \rrbracket_\Sigma \subseteq \mathcal{R} \times \Sigma$$

la sémantique de ce langage. Il s'agit d'une relation permettant de décrire de manière plus ou moins fine certaines propriétés vérifiées par un état obtenu après application d'une requête. Etant donnée une requête R et un état σ, l'énoncé $(R, \sigma) \in \llbracket \mathcal{R} \rrbracket_\Sigma$ permet de caractériser les propriétés qu'un état σ doit satisfaire quand il est obtenu à partir d'un état en appliquant avec succès la requête R.

La donnée d'une politique $\mathbb{P}[\rho]$ et d'un langage de requêtes $\mathcal{R}$ muni d'une sémantique $\llbracket \mathcal{R} \rrbracket_\Sigma$ constitue un modèle :

$$\mathbb{M}[\rho] = (\mathbb{P}[\rho], \llbracket \mathcal{R} \rrbracket_\Sigma)$$

Implanter un modèle de contrôle d'accès $\mathbb{M}[\rho] = (\mathbb{P}[\rho], \llbracket \mathcal{R} \rrbracket_\Sigma)$ consiste à définir une paire (τ, Σ_I) où $\tau : \mathcal{R} \times \Sigma \to \mathcal{D} \times \Sigma$ est une fonction de transition entre états ($\mathcal{D}$ est l'ensemble des réponses possibles) et où Σ_I est l'ensemble des états initiaux possibles (il s'agit bien sûr d'un sous-ensemble des états sûrs). A ce niveau, il est possible de spécifier les propriétés attendues sur (τ, Σ_I). Les deux propriétés principales sont les propriétés de correction vis-à-vis de la politique (τ transforme tout état sûr en état sûr) et de la sémantique des requêtes (les états obtenus en appliquant τ à une requête R satisfont la relation $\llbracket \mathcal{R} \rrbracket_\Sigma$). Si l'on note $\Gamma_\tau(E)$ l'ensemble des états atteignables à partir des états contenus dans l'ensemble E par application de la fonction de transition τ, une implantation (τ, Σ_I) est dite :

- $\mathbb{P}[\rho]$-*correcte* ssi chaque état atteignable est sûr : $\Gamma_\tau(\Sigma_I) \subseteq \Sigma_{|\Omega}$
- $\llbracket \mathcal{R} \rrbracket_\Sigma$-*correcte* ssi :

$$\forall \sigma_1, \sigma_2 \in \Sigma \quad \forall R \in \mathcal{R} \quad \tau(R, \sigma_1) = (\text{yes}, \sigma_2) \Rightarrow (R, \sigma_2) \in \llbracket \mathcal{R} \rrbracket_\Sigma$$

D'autres propriétés permettent de caractériser plus finement les implantations d'une politique de contrôle d'accès en spécifiant les liens qui existent entre les

états accessibles selon les types de requêtes qui permettent de passer de l'un à l'autre.

Un modèle admet plusieurs implantations correctes. Par exemple, la fonction de transition qui retourne (no, σ) pour tout état σ et toute requête est une implantation trivialement correcte (dont l'intérêt est cependant assez limité). Aussi, il peut être intéressant de comparer les implantations d'une même politique afin, par exemple, de déterminer si une implantation est plus restrictive qu'une autre. On définit un préordre sur les implantations : intuitivement une implantation $I = (\tau, \Sigma_I)$ est plus restrictive qu'une implantation $I' = (\tau', \Sigma'_I)$, ce que l'on note $I \sqsubseteq I'$, ssi les états accessibles avec I le sont également avec I' et si I' permet de faire de "plus petits pas" que I et d'agir avec une granularité plus fine sur les états (la définition formelle de cette condition nécessite la définition d'un préordre sur les états que nous ne détaillons pas ici).

Le cadre sémantique que nous venons d'introduire nous permet de formaliser les concepts nécessaires pour envisager la comparaison de modèles de contrôle d'accès. Ces concepts reposent sur la notion classique de relation de simulation. Etant donnés deux modèles de contrôle d'accès $\mathbb{M}_1[\rho_1] = (\mathbb{P}_1[\rho_1], [\![\mathcal{R}]\!]_{\Sigma_1})$ et $\mathbb{M}_2[\rho_2] = (\mathbb{P}_2[\rho_2], [\![\mathcal{R}]\!]_{\Sigma_2})$ (partageant les mêmes ensembles de sujets, d'objets et de modes d'accès), l'implantation $I_2 = (\tau_2, \Sigma_I^2)$ de $\mathbb{M}_2[\rho_2]$ simule l'implantation $I_1 = (\tau_1, \Sigma_I^1)$ de $\mathbb{M}_1[\rho_1]$, ce que l'on note $I_1 \overset{\kappa_\Sigma}{\preceq} I_2$, ssi il existe une relation $\kappa_\Sigma \subseteq \Sigma_1 \times \Sigma_2$ qui relie tout état initial de I_1 à au moins un état initial de I_2 et qui soit une relation de simulation de τ_1 par τ_2, c'est-à-dire :

$$\begin{array}{l} \forall \sigma_1, \sigma'_1 \in \Sigma_1 \ \forall \sigma_2 \in \Sigma_2 \ \forall R \in \mathcal{R} \ \forall a \in \mathcal{D} \\ \quad ((\sigma_1, \sigma_2) \in \kappa_\Sigma \wedge \tau_1(R, \sigma_1) = (a, \sigma'_1)) \\ \Rightarrow (\exists \sigma'_2 \in \Sigma_2 \ (\sigma'_1, \sigma'_2) \in \kappa_\Sigma \wedge \tau_2(R, \sigma_2) = (a, \sigma'_2)) \end{array}$$

Interpréter un modèle de contrôle d'accès par un autre consiste alors à définir une relation de simulation satisfaisant de bonnes propriétés. Concrètement, si l'on note $\mathbb{M}_{BLP}[\rho_{BLP}] = (\mathbb{P}_{BLP}[\rho_{BLP}], [\![\mathcal{R}]\!]_{\Sigma_{BLP}})$ le modèle de Bell et LaPadula et $\mathbb{M}_{CW}[\rho_{CW}] = (\mathbb{P}_{CW}[\rho_{CW}], [\![\mathcal{R}]\!]_{\Sigma_{CW}})$ le modèle de la Muraille de Chine, donner une interprétation de $\mathbb{M}_{CW}[\rho_{CW}]$ basée sur un treillis de niveaux de sécurité consiste tout d'abord à définir un treillis de niveaux de sécurité ρ_{BLP} à partir de ρ_{CW} puis à définir une relation $\kappa_\Sigma \subseteq \Sigma_{CW} \times \Sigma_{BLP}$ telle que l'implantation classique $I_{CW} = (\tau_{CW}, \Sigma_I^{CW})$ de la Muraille de Chine soit simulable par une implantation correcte mais non standard $I'_{BLP} = (\tau'_{BLP}, \Sigma_I^{'BLP})$ du modèle de Bell et LaPadula. Bien sûr, I'_{BLP} s'obtient en réutilisant l'implantation standard de Bell et LaPadula.

D'un point de vue plus théorique, les notions introduites dans cette section permettent de définir un préordre sur les modèles de contrôle d'accès. On dira qu'un modèle $\mathbb{M}_1[\rho_1]$ est plus restrictif qu'un modèle $\mathbb{M}_2[\rho_2]$, noté $\mathbb{M}_1[\rho_1] \trianglelefteq \mathbb{M}_2[\rho_2]$, si et seulement si toute implantation correcte de $\mathbb{M}_1[\rho_1]$ peut être simulée par une implantation correcte de $\mathbb{M}_2[\rho_2]$.

Avec une telle définition, comparer deux modèles nécessite de considérer toutes les implantations d'un modèle ce qui dans la pratique pose un problème

évident. Toutefois, si la relation de simulation qui permet d'envisager la comparaison vérifie des propriétés supplémentaires, ce problème peut être évité.

En effet, c'est le cas lorsque la relation de simulation construite est injective et fonctionnelle puisque dans ce cas il suffit seulement de montrer que toutes les implantations maximales[1] peuvent être simulées. On montre dans ce cas que si une implantation I est simulable, alors toute implantation I' telle que $I' \sqsubseteq I$ est aussi simulable.

C'est aussi le cas lorsque l'on sait définir une relation de simulation qui préserve à la fois les propriétés de sécurité des politiques et la sémantique des requêtes. On montre en effet directement dans ce cas que $\mathbb{M}_1[\rho_1] \trianglelefteq \mathbb{M}_2[\rho_2]$.

Ce préordre nous a permis de montrer que le modèle de la Muraille de Chine est strictement plus restrictif que le modèle de Bell et LaPadula. En effet, on a bien $\mathbb{M}_{CW}[\rho_{CW}] \trianglelefteq \mathbb{M}_{BLP}[\rho_{BLP}]$, ce qui se prouve en montrant que l'implantation standard de la Muraille de Chine est l'unique implantation maximale de ce modèle. D'autre part, il est possible de trouver une implantation du modèle de Bell et LaPadula qui ne soit pas simulable par une implantation du modèle de la Muraille de Chine.

5 Conclusion – Perspectives

La sécurité, et plus particulièrement le contrôle d'accès, sont des problématiques actuelles en informatique. En effet, il devient aujourd'hui important de pouvoir contrôler les flots d'informations dans les réseaux et dans les systèmes d'information. Il convient de développer au sein des systèmes informatiques des mécanismes permettant de filtrer les accès afin de ne laisser passer que ceux autorisés. Il s'agit pour cela de définir une politique de sécurité, c'est-à-dire la caractérisation des accès permis. Le programme chargé de mettre en application cette politique, le moniteur de référence, est souvent considéré comme l'une des clés de voûte de la sécurité d'un système. Sa conception et son développement doivent être menés de manière à garantir sa fiabilité et sa sûreté. En effet, toute faille au sein de ce programme pourrait entraîner des violations de la politique de sécurité. L'emploi des méthodes formelles dans le développement d'un moniteur de référence permet de garantir que certaines propriétés sont toujours respectées.

Dans cet article nous avons rendu compte de manière informelle de quelques expériences formelles que nous avons faites dans cette direction. Comme nous l'avons vu, les problèmes posés par une approche formelle du contrôle d'accès ne sont pas aussi simples qu'ils paraissent.

L'ensemble des travaux décrits ci-dessus méritent à présent d'être poursuivis. Tout d'abord, l'indispensable travail de formalisation a mis en relief certaines confusions : plusieurs définitions non équivalentes d'une même propriété coexistent dans la littérature. Nous envisageons de continuer notre étude en formalisant d'autres politiques de sécurité (par exemple en considérant RBAC [13], déjà formalisé dans [28]). D'autre part, nous souhaitons enrichir le cadre sémantique

[1] L'implantation I d'un modèle $\mathbb{M}[\rho]$ est maximale, s'il n'existe pas d'implantation I' (avec $I' \neq I$) de ce même modèle telle que $I \sqsubseteq I'$.

proposé afin de permettre d'envisager la composition de politiques de contrôle d'accès.

Nous souhaitons aussi "comparer" les "comparaisons de modèles de contrôle d'accès". En effet, quelques travaux ont déjà eu lieu sur ce sujet mais ils sont encore parcellaires: [29] envisage la comparaison de politiques de contrôle d'accès en terme de puissance d'expression, [9] utilise des techniques de simulation pour comparer des modèles de contrôle d'accès discrétionnaire, [30] envisage la comparaison de politiques sous l'angle de la combinaison de politiques. Toutes ces approches portent sur le même objet et méritent d'être reconsidérées et étendues dans un cadre uniforme afin d'en étudier les liens et d'en dégager des techniques de réutilisation. Une telle étude doit permettre de mieux identifier les "ingrédients" présents dans une politique de contrôle d'accès, et d'en comprendre le rôle dans une implantation.

A plus long terme, nous souhaitons implanter le cadre obtenu dans l'atelier Focal et en dériver une bibliothèque certifiée de moniteurs de référence. Parallèlement, nous envisageons d'étudier et de caractériser les fonctionnalités qu'un environnement integré de développement (IDE), comme Focal, doit offrir non seulement pour produire des logiciels en conformité avec les exigences requises pour atteindre de hauts niveaux de confiance (EAL 5 6 7), mais aussi pour faciliter le processus d'évaluation de ces logiciels selon les standards (IEC61508, CC, ...). C'est d'ailleurs la thématique d'un récent projet de l'ANR, le projet SSURF (*Safety and Security under Focal*).

Remerciements. Nous remercions vivement Thérèse Hardin avec qui nous avons de nombreuses conversations fructueuses autour de ce travail.

References

1. Common Criteria for Information Technology Security Evaluation, `http://www.commoncriteriaportal.org/`.
2. Ocaml-mysql v. 1.0.4: `http://raevnos.pennmush.org/code/ocaml-mysql/`.
3. P. Amey. Dear sir, yours faithfully: an everyday story of formality. In *Practical Elements of Safety, Proc. of the Twelfth Safety-critical Systems Symposium.* Spinger-Verlag, 2004.
4. J. P. Anderson. Computer security technology planning study. ESD-TR-73-51, vol 1 AD-758 206, ESD/AFSC Hanscom, AFB Bedford, Mass, 1972.
5. D. Bell and L. LaPadula. Secure Computer Systems: a Mathematical Model. Technical Report MTR-2547 (Vol. II), MITRE Corp., Bedford, MA, May 1973.
6. E. Bertino and R.S. Sandhu. Database security-concepts, approaches, and challenges. *IEEE Trans. Dependable Sec. Comput*, 2(1), 2005.
7. J. Blond and C. Morisset. Formalisation et implantation d'une politique de sécurité d'une base de données. In INRIA, editor, *17ème Journées Francophones des Langages Applicatifs, JFLA'2006*, pages 71–86, 2006.
8. D. F. C. Brewer and M. J. Nash. The chinese wall security policy. In *Proc. IEEE Symposium on Security and Privacy*, pages 206–214, 1989.
9. A. Chander, J.C. Mitchell, and D. Dean. A state-transition model of trust management and access control. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop CSFW*, pages 27–43. IEEE Computer Society Press, 2001.

10. D. Delahaye, J.F. Étienne, and V. Donzeau-Gouge. Certifying airport security regulations using the Focal environment. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, 2006, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*, pages 48–63. Springer, 2006.
11. C. Dubois, J. Grandguillot, and M. Jaume. Réutilisation de preuves formelles : Une étude pour le système FoC. In INRIA, editor, *14ème Journées Francophones des Langages Applicatifs, JFLA'2003*, pages 63–75, 2003.
12. C. Dubois, T. Hardin, and V. Viguié Donzeau Gouge. Building certified components within Focal. In *Symposium on Trends in Functional Programming*, 2004.
13. S. Gavrila and J. Barkley. Formal specification for RBAC user/role and role/role relationship management. In *Proceedings of the 3rd ACM Workshop on Role-Based Access Control (RBAC-98)*, pages 81–90. ACM Press, 1998.
14. E. Gureghian, Th. Hardin, and M. Jaume. A full formalisation of the Bell and Lapadula security model. Technical Report 2003-007, Univ. Paris 6, LIP6, 2003.
15. M. Jaume and C. Morisset. Formalisation and implementation of access control models. In *Information Assurance and Security (IAS'05) International Conference on Information Technology, ITCC*, pages 703–708. IEEE CS Press, 2005.
16. M. Jaume and C. Morisset. A formal approach to implement access control. *Journal of Information Assurance and Security*, 2:137–148, 2006.
17. M. Jaume and C. Morisset. Towards a formal specification of access control. In P. Degano, R. Kusters, L. Vigano, and S. Zdancewic, editors, *Proceedings of the Joint Workshop on Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis, FCS-ARSPA'06*, pages 213–232, 2006.
18. L.J. LaPadula and D.E. Bell. Secure Computer Systems: A Mathematical Model. *Journal of Computer Security*, 4:239–263, 1996.
19. McLean. The algebra of security. In *Proc. IEEE Symposium on Security and Privacy*, pages 2–7. IEEE Computer Society Press, 1988.
20. V. Prevosto and D. Doligez. Algorithms and proof inheritance in the Foc language. *Journal of Automated Reasoning*, 29(3-4):337–363, dec 2002.
21. V. Prevosto and M. Jaume. Making proofs in a hierarchy of mathematical structures. In *Proceedings of the 11th Calculemus Symposium*, Rome, sep 2003.
22. Focal project. *Focal, version 0.3.1 Tutorial and reference manual.* LIP6 – INRIA – CNAM, sept 2006. Distribution available at: `http://focal.inria.fr`.
23. Logical Project. *The Coq Proof Assistant Reference Manual.* INRIA-Rocquencourt, 2002.
24. R. Rioboo. *Programmer le Calcul Formel, des Algorithmes à la Sémantique, Mémoire d'Habilitation.* Mémoire d'habilitation, Université Pierre et Marie Curie, Paris, France, 2002.
25. Sandhu and Chen. The multilevel relational (MLR) data model. *ACMTISS: ACM Transactions on Information and System Security*, 1, 1998.
26. R. S. Sandhu. A lattice interpretation of the chinese wall policy. In *Proc. 15th NIST-NCSC National Computer Security Conference*, pages 329–339, 1992.
27. R. S. Sandhu. Lattice-Based Access Control Models. *IEEE Computer*, 26(11):9–19, November 1993.
28. H. Toussaint. Formalisation des politiques de contrôle d'accès. Master's thesis, FUNDP, Namur, Belgium, 2006.
29. M.V. Tripunitara and N. Li. Comparing the expressive power of access control models. In *SIGSAC: 11th ACM Conference on Computer and Communications Security*. ACM SIGSAC, 2004.

30. M.C. Tschantz and S. Krishnamurthi. Towards reasonability properties for access-control policy languages. In D.F. Ferraiolo and I. Ray, editors, *SACMAT 2006,11th ACM Symposium on Access Control Models and Technologies, Proceedings*, pages 160–169. ACM, 2006.

A System to check Operational Properties of Logic Programs

François Gobert and Baudouin Le Charlier

Université Catholique de Louvain, Belgium,
{gobert,blc}@info.ucl.ac.be
This work is supported by the Belgian FNRS fund.

Résumé. Un analyseur statique de programmes logiques est présenté. Le système est construit dans un cadre unifié d'interprétation abstraite, permettant l'intégration de plusieurs domaines et analyses servant à la vérification et à l'optimisation automatiques des programmes. L'analyseur vérifie de nombreuses propriétés désirables de programmes logiques, en tenant compte de la recherche en profondeur d'abord, et des autres caractéristiques spécifiques de Prolog. Les propriétés opérationnelles vérifiées concernent les types, les modes, la linéarité, le partage de variables entre les termes, la preuve de terminaison, la non nécessité de l'*occur-check*, le succès ou l'échec certain, le déterminisme des procédures logiques, les relations linéaires entre les tailles des termes en entrée et en sortie, ainsi que le nombre de solutions pour une requête donnée. L'article met en évidence l'interopérabilité entre les différents domaines qui coopèrent entre eux. Cette contribution mutuelle entre les domaines est nécessaire pour la vérification des propriétés sus-mentionnées.

Mots-clés. Détection statique d'erreurs, interprétation abstraite, vérification automatique, Prolog, analyse statique.

1 Introduction

Static analysis of a logic program aims at deriving general information about its execution without actually running it. Applications are numerous and closely related to the kind of properties which are derived. Various code optimizations are possible when information about type, mode, reference chain length, liveness and sharing is available. Frameworks performing global optimizations of logic programs have been proposed [3, 14, 17]. Static analysis can also be oriented towards verification. The confidence in the code correctness of the program is much increased when some operational properties are checked. For instance, frameworks are designed to prove termination [6, 12, 17, 23, 25], to prove the non-failure of execution [5, 10], to prove that a procedure is deterministic [15, 24], or to prove that unifications without occur-check are sound [9]. Static analyses are also useful to assist methodologies for constructing correct and efficient logic programs [1, 11]. For instance, the Deville's methodology [11] consists of three main steps: elaboration of a specification, construction of a logic description, and derivation of a correct and efficient Prolog procedure. The method involves a number of transformation and verification steps. Some of them are systematic but tedious to perform by hand. Static analyses can help to support the automatable aspects of this methodology.

We present an implemented analyser which integrates the essential of the techniques of static analyses of logic programs. It is based on a unified abstract

interpretation framework [8], which allows us to master the combination of several domains in a single global analysis. The abstract information is statically collected at each program-point in so-called *abstract sequences*. An abstract sequence models accurately the execution of a goal, a clause or a procedure, by maintaining the mode, type, sharing in input/output terms, the conditions on sure success/failure of execution, the number of solutions in function of the size of input terms, as well as the relation between the size of input/output terms. A language for specifying such operational properties has been defined. The analyser can verify that a procedure respects its specification, and it can automatically perform some source to source transformation like safe introduction of cuts and removal of useless literals. In this paper, we only concentrate on the verification aspect of the analyser, and we illustrate how the various domains cooperate together and may improve each other. For more details, the reader is referred to [7, 20, 22] and to the web page of the analyser *http://www.info.ucl.ac.be/∼gobert*.

The rest of the paper is organised as follows. Section 2 describes the set of operational properties the analyser can prove, in the context of Deville's methodology. The specification language of the analyser is introduced. Section 3 presents the abstract domains and the abstract execution. Section 4 illustrates the benefit of performing a global analysis where the different abstract domains cooperate together. Section 5 reports on experimental results.

2 Specifying Operational Properties

In this section, we choose to illustrate the kind of operational properties desirable for a procedure in the context of Deville's methodology for constructing correct and efficient logic programs [11]. An important step of the methodology consists in verifying that a Prolog procedure complies with a given specification. Subsection 2.1 recalls the specification format of a logic procedure as proposed by Deville's methodology. Subsection 2.2 introduces the specification language of the analyser, which describes the operational properties the system is able to prove. The presentation explores to which extent our system can formally express the specification schema from the methodology.

2.1 Specifying with Deville's Methodology

Deville's methodology [11] proposes a standard schema for specifying a procedure. It contains among other things the name and the formal parameters of the procedure, and pre-post conditions on its execution. For instance, the procedure `select/3` can be specified in this way:

procedure `select`(X_1, X_2, X_3)
Type: X_1: term
X_2, X_3: lists
Relation: X_3 is list X_2 where one occurrence of term X_1 was removed.
Application conditions:
in(ground,ground,ground):**out**(ground,ground,ground) $\langle 0, * \rangle$
in(var,ground,var):**out**(ground,ground,ground) $\langle 0, * \rangle$
in(ground,var,ground):**out**(ground,ground,ground) $\langle 0, * \rangle$
in(var,var,var):**out**(var,ngv,any) $\langle 0, \infty \rangle$

The *application conditions* (or *directionalities*) describe the possible uses of the procedure: each directionality specifies the allowed modes of the parameters before the execution (the **in** part), and the corresponding modes after the call (the `out` part). Type information (a type is here defined as a set of ground terms) acts as pre-post conditions too. The cardinality information $\langle Min, Max\rangle$ specify the minimum and maximum lengths of the sequence of computed answer substitutions returned by a procedure call respecting the **in** part of the directionality. The symbol $*$ (resp. ∞) expresses that there is a finite (resp. infinite) number of solutions.

2.2 Specifying with the Analyser

Our system is useful to express (and verify) that a Prolog code respects many aspects of a Deville's specification. It can even be more expressive for some aspects. The formal specification language used by the analyser is introduced progressively, by refining step by step the operational properties the system is able to prove. Consider the context where the programmer has built a Prolog code for `select/3` as follows:

```
select(X,[X|T],T).
select(X,[H|T],[H|Ts]) :- select(X,T,Ts).
```

One can use the analyser to automatically prove operational properties expressed in the Deville's specification of the previous section. More generally, for each kind of input calls (i.e., for each input directionality), the programmer can query the system for proving termination, checking the number of solutions, verifying the conditions of sure success or sure failure, the correct-typing of procedure calls, and the occur-check freeness of the program. We only consider here the second and third directionalities proposed in section 2.1.

Checking input/output modes. The analyser is able to check that the two considered directionalities hold by providing the following formal specifications:

```
select/3                              select/3
  in(X1:var,X2:gr,X3:var)               in(X1:gr,X2:var,X3:gr)
  out(gr,_,gr)                          out(_,gr,_)
```

The `in` part specifies conditions on the input arguments. The symbol `var` (resp. `gr`) denotes that the argument is bound to a free variable (resp. a ground term). The `out` part describes the form of the arguments at the end of the execution, when the execution succeeds. The symbol '_' is used when we do not provide refined information about an argument. Other mode information can be specified (e.g., `gv` denotes a ground term or a variable, `ngv` denotes a non-ground term which is not a variable).

Checking correct-typing. The above specifications can be refined by describing the input/output types of the arguments as follows (`any` denotes a term with no specific type, and `list` denotes the type of (non-necessarily ground) lists):

```
select/3                                select/3
  in(X1:any,X2:list,X3:any)               in(X1:any,X2:any,X3:list)
  out(_,_,list)                           out(_,list,_)
```

Modes and types can be combined together as follows:

```
select/3                                select/3
  in(X1:var,X2:gr list,X3:var)            in(X1:gr,X2:var,X3:gr list)
  out(gr,_,gr list)                       out(_,gr list,_)
```

Checking linearity, nosharing and occur-check freeness. Linearity and nosharing are needed to prove occur-check freeness of the procedure.

```
select/3                                 select/3
  in(X1:any,X2:list,X3:any;                in(X1:gr,X2:var,X3:gr list)
     noshare(<X1,X2,X3>);                  out(_,gr list,_)
     linear(X2))
  out(_,_,list)
```

The first specification expresses that the input arguments do not share any variable, and that the input argument is bound to a linear list. The analyser is then able to prove that the procedure is not subject to the occur-check.

Checking cardinality information and proving termination. The system can check information about the number of solutions, and can prove termination.

```
select/3                                 select/3
  in(X1:var,X2:gr list,X3:var)             in(X1:gr,X2:var,X3:gr list)
  out(gr,_,gr list)                        out(_,gr list,_)
  srel(X2_in = X3_out+1)                   srel(X2_out = X3_in+1)
  sol(0 <= sol <= X2_in)                   sol(sol = X3_in+1)
  sexpr(X2)                                sexpr(X3)
```

The `srel` part describes a linear relation between the size of input and output terms. The list-length size measure is used. The size of an input (resp. output) term `T` is denoted by `T_in` (resp. `T_out`). The symbol `sol` denotes the number of solutions, which depends on the value of the input terms. In the first case, the number of solutions is checked to be between zero and the length of the input list. Sure success is indeed not guaranteed, because the input arguments `X1` and `X2` may possibly share variable (the `noshare` tag is not present), and because `X2` may be an empty list. Sure success is proved for the second specification. Termination is proved for the two specifications. The `sexpr` part specifies the induction parameter decreasing through recursive calls.

Proving sure success/failure, and refining cardinality information. Refined conditions for sure success and sure failure can be specified, and the analyser is then able to prove more accurate cardinality information.

```
select/3
  in(X1:var,X2:gr list,X3:var;
     noshare(<X1,X3>))
  ref(_,[_|_],_)
  out(gr,_,gr list)
  srel(X2_in = X3_out+1)
  sol(sol = X2_in) ; sexpr(X2)
```

```
select/3
  in(X1:gr,X2:var,X3:gr list)
  out(_,[gr|gr list],_)
  srel(X2_out = X3_in+1)
  sol(sol = X3_in+1)
  sexpr(X3)
```

The first specification considers input calls where `X1` and `X3` are two distinct variables (presence of the `noshare` tag). The `ref` part specifies necessary conditions on the input arguments to obtain success (at least one solution). In the first case, the `select` predicate succeeds only if input `X2` is a non-empty ground list. (If it is an empty list, then the execution surely fails.) Information that can be implicitly deduced need not be explicitly written (the symbol '_' is used).

The system is well-suited to support a methodology for constructing correct logic programs. Our specification language is expressive enough to capture various behavioural properties, which can be checked by the analyser. Such properties include the operational aspects proposed by Deville but they are often more precise and expressive since, for instance, the number of solutions can be explicitly related to the size of input terms, and since the information on sure success and failure can be expressed with respect to a refinement of the information on the input terms (to give only two examples).

3 Abstract Interpretation Framework

This section presents the abstract interpretation framework of the analyser (see [20] for a more theoretical presentation). The verification analysis is modular and compositional. An input to our Prolog program verifier is a module, i.e., a file containing a set of Prolog procedures interleaved with additional information about the program. The additional information consists of formal specifications, type declarations, and norm declarations. A module can import other modules. To the contrary to Mercury, the additional information is not mandatory but the more information is given about the program the more precise and useful the verification can be. The checker verifies a procedure against its specification (or against several specifications), assuming that the specifications hold for the subproblems (defined in the module itself or in some imported modules). An important feature of the analyser is thus the capability of analysing Prolog procedures without the code of the subprocedures. Instead, it uses formal specifications to pass from one execution point to the next one. This allows the construction of procedures from others, using only their specification and not their code.

Subsection 3.1 illustrates the use of type and norm declarations. Subsection 3.2 describes the domains of abstract substitutions (which maintain the structure and some useful properties about the terms and subterms of the program substitutions) and abstract sequences (which model the execution of a goal, a clause or a procedure). Subsection 3.3 explains how a procedure is checked in a single global analysis.

3.1 Type and Norm declarations

The user can declare polymorphic recursive types, and can specify norms on the defined types. A non-polymorphic type is intended to represent some set of Prolog terms (non-necessarily ground). Polymorphic types denotes infinite families of non-polymorphic ones and are given a name. They can be arbitrarily instantiated to produce so called *type expressions* that denote sets of Prolog terms. The system also considers a number of primitive types, namely `any`, `gr`, `int`, `float`, `var` which denote the sets of any terms, ground terms, integers, floats, and variables respectively. Actually, modes are particular types since we are not limited to ground types. Beside primitive types the user can declare its own types. Several primitive norms exist (e.g., the list-length and the general term sizes). The user is also able to declare norms associated to defined types.

Consider for instance the program `flattree(T,L)`. It describes a relation where `L` is the list of the elements of the tree `T` in prefixed order.

```
flattree(void, []).
flattree(t(LT,X,RT), [X|Xs]) :-
  flattree(LT, LLT), flattree(RT, LRT), append(LLT, LRT, Xs).
```

The polymorphic type for tree structures used in the program `flatten/2` can be declared recursively as follows:

```
tree(T) ::= void | t(tree(T),T,tree(T))
```

The type name is `tree`, and the type parameter is `T`. Such a type cannot be used in specifications directly, but arbitrary type expressions built from polymorphic and primitive type names can be used instead. They may not contain type variables. For instance, `tree(list(int))` is a type expression denoting the set of binary trees whose elements are lists of integers. We can specify a size measure for a tree, viz. *elems*, which is the number of its elements:

```
(elems)void        =  0
(elems)t(T1,X,T2)  =  1+(elems)T1+(elems)T2
```

The analyser is able to check the following behaviours:

```
flattree                                   flattree
  in(T:tree(any),L:var;noshare(<T,L>))       in(T:var, L:list(int))
  out(_, list(any))                          out(tree(int), _)
  srel( (elems)T_in = (list)L_out )          srel( (elems)T_out = (list)L_in )
  sol( sol = 1 )                             sol( sol >= 1 )
  sexpr((elems)T)                            sexpr((list)L)
```

The way we combine type and mode information is similar to the integrated types of G. Janssens and M. Bruynooghe [18] but in order to avoid combinatorial explosion problems arising in [18], the user is restricted to declare disjoint types (i.e., principal functors of two types must be distinct). Our use of parametric types also has similarities with the type system of Mercury [27]. However, our approach is not strictly comparable: our integration of modes in the type declarations, and the use of qualified type expressions is more powerful but we do not allow polymorphic specifications.

3.2 Abstract Substitutions and Sequences

The system is based on an abstract interpretation framework which integrates various domains. The checker realizes a global analysis to collect (and verify) simultaneously each domain information. The connection and cooperation between the various domains is implemented through the use of indices. An index represents an (abstract) input/output term. Every domain maps each index to some abstract property (e.g., mode, type, size, etc). The domains are combined in *abstract substitutions* and *abstract sequences*.

The domain of *abstract substitutions* is an instantiation to modes, types, linearity and possible sharing of the generic abstract domain $\mathtt{Pat}(\Re)$ described in [7]. An abstract substitution represents a set of program substitutions of the form $\{X_1/t_1, ..., X_n/t_n\}$, where the X_i's are program variables, and the t_i's are terms. An abstract substitution β is a triple of the form $\langle sv, frm, \alpha\rangle$. The *same-value* and *frame* components provide information about the structure of terms and subterms of them. Each term described in β is denoted by an index. The sv component maps each program variable X to its corresponding index. Hence, the equality $sv(X) = sv(Y)$ means that variables X and Y are bound to the same term. The frm component describes the pattern of some indices, by giving their functor name and the indices of their composing subterms. The *alpha tuple* α is the generic part of the domain. It provides extra information about all terms and subterms of interest (represented by the indices). In the current framework, α is of the form $\langle mo, ty, ps, lin, E\rangle$. The mo component maps each index to its mode (e.g., `gr`, `var`). The ty component maps each index to its type expression (e.g., `list(int), list(any)`). The ps component is a binary relation over indices, and expresses the possible sharing between two terms. Pairs of indices that do not belong to ps surely do not share a variable. The lin component contains all indices that are surely linear (i.e., they do not contain several occurrences of the same variable). The E component is a linear relation between the size of terms (several norms can be combined).

The domain of *abstract sequences* models the operational behaviour of a Prolog procedure. Abstract sequences describe sets of pairs $\langle\theta, S\rangle$ where θ is a program substitution and S is the sequence of answer substitutions resulting from executing a procedure (a clause, a goal, etc.) with input substitution θ. An abstract sequence B is a tuple of the form $\langle\beta_{in}, \beta_{ref}, \beta_{fail}, U, \beta_{out}, E_{ref_out}, E_{sol}\rangle$ where β_{in} is an abstract substitution describing the class of accepted input calls; β_{ref} is an abstract substitution describing an over approximation of the successful input calls, i.e., those which produce at least one solution; β_{fail} is an abstract substitution describing an under approximation of the input calls that surely fail; U describes the set of input terms that are untouched (non-instantiated) during execution; β_{out} is an abstract substitution describing an over approximation of the set of output substitutions; E_{ref_out} is a linear relation between the size of the input/output terms; and E_{sol} is a linear relation between the number of solutions and the size of the input terms.

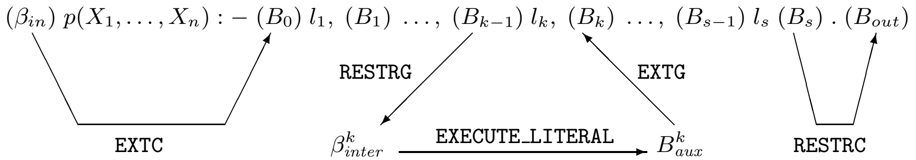
A *behaviour* for a procedure is a formalization of the specification of behavioural properties provided by the user (Section 2.2). A behaviour Beh_p for a

procedure p is a finite set of pairs $\{\langle B_1, se_1\rangle, \dots, \langle B_m, se_m\rangle\}$ where $B_1, \dots, B_m$ are abstract sequences, and $se_1, \dots, se_m$ are sequences of linear expressions whose variables are the sizes of the input arguments.

3.3 Abstract Execution

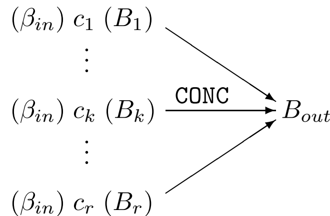
The abstract execution follows a standard top-down verification technique. For a given program, it analyses each pair procedure/specification; for a given procedure/specification, it analyses each clause; for a given clause, it analyses each atom. If an atom in the body of a clause is a procedure call, the analyser looks at the given specifications to infer information about its execution.

The execution of a clause $c ::= p(X_1, ..., X_n) : -l_1, ..., l_s$ with respect to a formal specification $Beh_{spec} = \langle B_{spec}, se\rangle$ can be depicted as follows:



The operation EXTC extends the input substitution β_{in} of B_{spec} to an initial abstract sequence B_0, where the local variables of the clause are considered. Then we compute iteratively B_k from B_{k-1} and l_k $(1 \leq k \leq s)$. Finally we restrict B_s to the variables in the head of c through the operation RESTRC. To compute B_k from B_{k-1}, we first restrict the abstract substitution β_{out} of B_{k-1} to the variables $X_{i_1}, \dots, X_{i_n}$ of l_k and we rename them into $X_1, \dots, X_n$ through the operation RESTRG. Then we execute the literal l_k with β^k_{inter} which returns an abstract sequence B^k_{aux}. Finally, we propagate and extend this result on B_{k-1} by computing $B_k = \mathtt{EXTG}(l_k, B_{k-1}, B^k_{aux})$. The execution of a literal depends of its form (either a unification, or a call to a sub-predicate, or a negation, or a cut, etc.). If l_k is a non-recursive call $q(X_{i_1}, \dots, X_{i_m})$ (i.e., $q \neq p$) then the analyser looks at the set of specifications, to find an abstract sequence general enough to give information about this call. If l_k is a recursive call $p(X_{i_1}, \dots, X_{i_n})$ then the analyser checks whether the induction parameter se strictly decreases.

When each clause of a procedure is executed, then the analyser concatenates the output abstract sequences through the CONC operation:



The verification succeeds if the resulting B_{out} is covered by the formal specification B_{spec}. Next Section illustrates the abstract verification and the benefit of performing a global analysis where the various domains cooperate together.

4 Cooperating Abstract Domains

The global analysis collects simultaneously the information over every abstract domain. Each abstract domain component may help to improve the precision of each other, making the global analysis accurate and efficient. This section illustrates the cooperation of domains during the abstract execution of the `mergesort(Xs,Ys)` program, which sorts the list of integers `Xs` into list `Ys`. The `split(Xs,Ys,Zs)` procedure divides the list `Xs` into two lists `Ys` and `Zs` whose lengths differ at most by one element. The `merge(Xs,Ys,Zs)` predicate combines in ascending order the two lists `Xs` and `Ys` into the list `Zs`.

```
mergesort([], []).
mergesort([X], [X]).
mergesort([X,Y|Xs], Ys) :- split([X,Y|Xs], X1s, X2s),
                           mergesort(X1s, Y1s), mergesort(X2s, Y2s),
                           merge(Y1s, Y2s, Ys).
split([], [], []).
split([X|Xs] ,[X|Ys], Zs) :- split(Xs, Zs, Ys).

merge([], Ys, Ys).
merge(Xs, [], Xs) :- not(Xs=[]).
merge([X|Xs], [Y|Ys], [X|R]) :- X<Y, merge(Xs,[Y|Ys],R).
merge([X|Xs], [Y|Ys], [Y|R]) :- X>Y, merge([X|Xs],Ys,R).
merge([X|Xs], [X|Ys], [X,X|R]) :- merge(Xs,Ys,R).
```

The specification of `mergesort` corresponding to its usual directionality is:

```
mergesort/2
  in(Xs:list(int), Ys:var); srel(Ys_out = Xs_in)
  out(_, list(int))       ; sol(sol = 1)         ; sexpr(Xs)
```

At each procedure point, the system derives information which capture the behaviour of the procedure execution, until that point. The system maintains such information in *abstract sequences*. The analyser computes a number of sequences: one for every prefix of every clause of the procedure, one for every clause, and finally, one for the complete procedure.

The analyser begins to execute separately the three clauses. The execution of the first and second clauses yields abstract sequences `B1` and `B2`, respectively:

```
B1: in(Xs:list(int), Ys:var)             B2: in(Xs:list(int), Ys:var)
    ref(Xs:[], _)                            ref(Xs:[_], _)
    out(Xs:[], Ys:[])                        out(Xs:[X], Ys:[X])
    srel(Xs_ref = Ys_out = 0)                srel(Xs_ref = Ys_out = 1)
    sol(sol = 1)                             sol(sol = 1)
```

In sequence `B1`, the `ref` component describes conditions for sure success: the first clause surely succeeds if and only if `Xs` is an empty list. Similarly, sequence `B2` expresses that second clause surely succeeds if and only if `Xs` is a one element list. The `ref` and `out` patterns information serve to construct the `srel` system.

The third clause is executed as follows. Before the `split` call, the analyser was able to verify sequence `B3` (local program variables `X1s`, `X2s`, `Y1s` and `Y2s` are maintained in the `out` part):

```
B3: in(Xs:list(int), Ys:var)
    ref(Xs:[X,Y|_], _)
    out(Xs:[X,Y|_], Ys:var, X1s:var, Y1s:var, X2s:var, Y2s:var ;
        noshare(<Ys,X1s,Y1s,X2s,Y2s>))
    srel(Xs_ref >= 2) ;  sol(sol = 1)
```

The next literal to execute is a call to `split` whose specification is:

```
  split/3
    in(As:list(int), Bs:var, Cs:var; noshare(<Bs,Cs>))
    out(_,list(int),list(int))
    srel(As_in = Bs_out+Cs_out, 0 <= Bs_out-Cs_out <= 1)
    sol(sol = 1)     ; sexpr(As)
```

At this point, the `out` part of `B3` informs us that `Xs` is a list of integers, and that `X1s`, `Y1s` are two distinct variables. This satisfies the input conditions of the `split` specification, such that the call is proved to be correctly typed. The `split` specification serves to derive the sequence `B4`, situated at the next program point. In particular, `X1s` and `Y1s` become lists of integers, and the size equations `Xs_out=X1s_out+Y1s_out` and `0<=X1s_out-Y1s_out<=1` hold:

```
B4: in(Xs:list(int), Ys:var)
    ref(Xs:[X,Y|_], _)
    out(Xs:[X,Y|_], Ys:var, X1s:list(int), Y1s:list(int),
        X2s:var, Y2s:var ; noshare(<Ys,X2s,Y2s>))
    srel(Xs_ref>=2, Xs_out=X1s_out+Y1s_out, 0<=X1s_out-Y1s_out<=1)
    sol(sol = 1)
```

The system is then able to prove that the next two recursive calls are correctly typed. In order to ensure the termination of `mergesort`, the analyser checks that the current size of `X1s_out` and of `X2s_out` are less than the size of the input `Xs_ref` (first argument is the induction parameter). The analyser can prove it, thanks to the cooperation between the `ref` pattern and the `srel` part. The abstract sequence `B5` before the call to `merge` is derived from the `mergesort` specification:

```
B5: in(Xs:list(int), Ys:var)
    ref(Xs:[X,Y|_], _)
    out(Xs:[X,Y|_], Ys:var,
        X1s:list(int), Y1s:list(int), X2s:list(int), Y2s:list(int))
    srel(Xs_ref>=2, Xs_out=X1s_out+Y1s_out, 0<=X1s_out-Y1s_out<=1,
         X1s_out=X2s_out, Y1s_out=Y2s_out)
    sol(sol = 1)
```

The specification of the `merge` procedure is as follows:

```
merge/3
  in(As:list(int), Bs:list(int), Cs:var) ; out(_,_,list(int))
  srel(Cs_out = As_in+Bs_in)             ; sol(sol = 1)  ; sexpr(As+Bs)
```

The call to `merge` is operationally correct, and `Ys` becomes a completely instantiated list of integers whose size is the sum of the size of `X2s` and `Y2s`. By injecting that information into abstract sequence `B5`, and by removing the local variables, the analyser is able to derive the abstract sequence `B6`, which models the entire execution of the third clause.

```
B6: in(Xs:list(int), Ys:var)      ; ref(Xs:[X,Y|_], _)
    out(Xs:[X,Y|_], Ys:list(int)) ; srel(Xs_ref>=2, Xs_out=Ys_out)
    sol(sol = 1)
```

Finally, the analyser detects that the `ref` parts of the three clauses (i.e., `B1`, `B2` and `B6`) are exclusive: each `ref` component differs from one another. Furthermore, the system verifies that each sequence implies the global specification of `mergesort`. The sure success of the whole program can be proved, because the upper bound of the three `ref` components is an exact (i.e., not approximated) union, which is exactly the same as the intended `ref` part. The analyser concludes from the above automatic reasoning that `mergesort` satisfies its specification.

5 Experimental Results

Table 1 reports on the verification of some classical programs, borrowed from [1, 11, 28]. The *In* (resp. *Out*) column describes the input (resp. output) modes, types, and patterns of a predicate execution. The following notations are used: 'a' denotes any term, 'g' denotes any ground term, 'v' denotes a variable, 'i' denotes an integer, 'l' denotes a (non-necessarily ground) list, 'gl' denotes a ground list, 'il' denotes a list of integers, 't' denotes a binary tree.

The analyser was able to prove that every predicate in the table are not subject to the occur-check. The *Card* column provides information about the number of solutions. The list-length (resp. the integer and the general) norm of the kth input predicate argument is denoted by `Xk` (resp. $\mathtt{Xk}^i$ and $\mathtt{Xk}^g$). The analyser is often able to verify the most precise cardinality; in particular it may prove either determinacy ($sol \leq 1$), full determinacy ($sol = 1$), sure success ($sol \geq 1$) or sure failure ($sol = 0$). The number of solutions is checked according to the `ref` part of the specifications (not shown in the table); for instance, the `select` program does not fail only if its second argument is a non-empty list.

The analyser can prove termination for all the programs reported on the table. The induction parameter is given in the *Term* column. For non-recursive procedure, there is obviously no need to give any induction parameter. Note that the induction is sometimes required to be a linear combination of the size of input terms (e.g., `merge/3`). For some programs, the programmer has to specify a sequence of size expressions that decreases lexicographically (e.g., `ack/3`).

The Parma Polyhedra Library `PPL` [2] is interfaced with the analyser in order to handle the linear relations between the size of the input/output terms. Due to the lack of space, the size relations verified by the system are not shown. Analysis execution time is given in seconds (tested on an PC; 1,5 GHz Pentium; 1 GB RAM). If one is only interested in checking that calls to procedures are correctly typed, or in verifying that programs are occur-check free (without proving termination nor checking the number of solutions), then the analyser no longer needs to maintain size constraints. Therefore, all polyhedral operations can be removed. In such case, the analysis execution time without considering the size components is given in the *NoPol* column.

6 Related Work

Analysing properties of logic programs for correctness or optimisation has attracted so many researchers that giving a fair account of those works is completely impossible here. We refer to [14, 20, 21] for a discussion of closely related

Table 1. Results of the analyser illustrated on some classical programs. All procedures and their specifications are available at *http://www.info.ucl.ac.be/∼gobert*.

Predicate	**In**	**Out**	**Card**	**Term**	**PPL**	**NoPol**
append	(gl,gl,gl)	(gl,gl,gl)	$sol \leq 1$	X1	0,391	0,222
	(gl,gl,v)	(gl,gl,gl)	$sol = 1$	X1	0,458	0,144
	(v,v,gl)	(gl,gl,gl)	$sol = X3+1$	X3	0,684	0,228
append_dl	(gl,a,gl,gl,a,a)	(gl,gl,gl,gl,gl,gl)	$sol \leq 1$	X1	0,266	0,151
flatten	(g,v)	(g,gl)	$sol \geq 0$	X1	0,694	0,300
flatten_dl	(g,v,gl)	(g,gl,gl)	$sol \geq 0$	X1	0,681	0,254
qs	(il,v)	(il,il)	$sol = 1$	X1	0,743	0,273
part	(i,il,v,v)	(i,il,il,il)	$sol = 1$	X2	1,274	0,431
qs_dl	(il,v,il)	(il,il,il)	$sol = 1$	X1	0,569	0,145
rev_naive	(gl,v)	(gl,gl)	$sol = 1$	X1	0,545	0,219
rev_acc	(gl,gl,v)	(gl,gl,gl)	$sol = 1$	X1	0,491	0,176
rev_dl	(gl,v,gl)	(gl,gl,gl)	$sol = 1$	X1	0,412	0,132
efface	(g,gl,v)	(g,[g|gl],gl)	$sol \leq 1$	X2	0,715	0,425
	(g,v,gl)	(g,[g|gl],gl)	$1 \leq sol \leq X3+1$	X3	0,870	0,257
max_list	(gl,gl,v)	(gl,gl,gl)	$sol = 1$	X1	0,986	0,492
ack	(i,i,v)	(i,i,i)	$sol = 1$	$\langle X1^i, X2^i \rangle$	1,146	0,600
close	(a)	(l)	$sol \leq 1$	X1	0,254	0,105
select	(g,v,gl)	(g,[g|gl],gl)	$sol = X3+1$	X3	0,644	0,246
	(g,gl,v)	(g,[g|gl],gl)	$sol \leq X2$	X2	0,664	0,241
	(v,gl,v)	(g,[g|gl],gl)	$sol = X2$	X2	0,930	0,250
perm_1	(v,gl)	(gl,gl)	$sol \geq X2, 1$	X2	0,506	0,215
	(gl,gl)	(gl,gl)	$sol \geq 0$	X2	0,269	0,109
perm	(gl,v)	(gl,gl)	$sol \geq X1, 1$	X1	0,069	0,059
	(v,gl)	(gl,gl)	$sol \geq X2, 1$	X2	0,104	0,022
length	(l,v)	(l,g)	$sol = 1$	X1	0,692	0,260
mergesort	(il,v)	(il,il)	$sol = 1$	X1	1,030	0,342
split	(il,v,v)	(il,il,il)	$sol = 1$	X1	0,506	0,121
merge	(il,il,v)	(il,il,il)	$sol = 1$	X1+X2	1,822	0,736
ins_sort	(il,v)	(il,il)	$sol = 1$	X1	0,415	0,230
insert	(i,il,v)	(i,il,il)	$sol = 1$	X2	0,840	0,384
tree_member	(v,gt)	(g,t(g,gt,gt))	$1 \leq sol$	$X2^g$	3,067	0,982
isotree	(gt,gt)	(gt,gt)	$sol \geq 0$	$X1^g$	1,316	0,521
	(gt,v)	(gt,gt)	$sol \geq 1$	$X1^g$	1,253	0,316
substitute	(i,i,it,v)	(i,i,it,it)	$sol = 1$	$X3^g$	1,841	0,677
polynomial	(g,g)	(g,g)	$sol \geq 1$	$X1^g$	0,942	0,493
derivative	(g,v)	(g,g)	$sol = 1$	$X1^g$	2,205	1,007
hanoi	(g,g,g,g,v)	(g,g,g,g,gl)	$sol \leq 1$	$X1^g$	1,090	0,438
flattree	(gt,v)	(gt,gl)	$sol = 1$	$X1^g$	0,815	0,278
Mean					0,845	0,331

work on program construction and verification, and we concentrate on a comparison with some implemented systems for logic program analysis.

The FOLON environment [16] was designed to partially support Deville's methodology of logic program construction [11]. Our analyser greatly improves on the FOLON analyser, since it allows us to prove, among others, termination, cardinality, occur-check freeness, which were not covered by FOLON.

The Ciao preprocessor CiaoPP [17] is a powerful static analyser based on abstract interpretation, which features many analyses similar to ours and other ones. The system can infer and/or check properties like regular types, modes, sharing, non-failure and determinacy, bounds on computational cost, bounds on sizes of terms in the program, and termination. It can perform automatic optimizations such as source-to-source transformation, specialization, partial evaluation of programs, program parallelization. In that system, procedures can be optionally annotated by *assertions* [26], which partially corresponds to specifications of our system. For instance, the second directionality of procedure `select` proposed in section 2.1 can be described by the following assertion.

```
:- check pred select(X1,X2,X3)
          : ( var(X1), list(X2,gnd), var(X3), indep(X1,X3) )
         => ( gnd(X1), list(X3,gnd),
              size(X2,length(X2)), size(X3,length(X2)-1) )
          + ( terminates, possibly_fails, steps_ub(length(X2)+1)).
```

Conditions on input arguments are placed between the `:` and `=>` tags, and conditions on output arguments are placed after the `=>` tag. Complex properties such as termination and non-failure can be described after the `+` tag. The above assertion expresses the following. If input `X1` and `X3` are two independent variables and input `X2` is a ground list, then after success of the execution, `X1` becomes a ground term and `X3` becomes a ground list whose size (list-length) is equal to the size of `X2` minus one. The procedure terminates, may fail, and the upper bound of resolution steps is the list-length of `X2` plus one. Cardinality analysis is restricted to determinacy analysis, sure success, and sure failure (e.g., `is_det`, `not_fails`, `fails` properties). Unlike our system, the assertion language of CiaoPP does not allow the user to describe the number of solutions in function of the size of input terms and does not provide a notion of refined description of input terms, which allows us to be more precise, expressive, and informative about sure success and failure (the `ref` part in formal specifications 2.2). We plan to establish a detailed comparison with the CiaoPP environment. At the time of writing, CiaoPP is still under development (beta-version 1.13), and our first attempts to use it for checking programs assertions similar to the ones reported in Table 1 have failed.

The Mercury programming language [27] also is associated with a whole range of analysis tools for optimisation and verification purposes. In Mercury also, the programmer has to annotate the program with information about modes, types, success and determinacy. A main difference is that not all logic programs are accepted by Mercury (only limited forms of unification are allowed). So our approach is more appealing to programmers who are willing to keep the full power of logic programming. Nevertheless, substantial work remains to be done to make our system usable in a "real life" context.

Termination has been the subject of many works (e.g., consider implemented provers like TerminWeb [4, 6, 13], TermiLog [23], cTI [19, 25]). Those systems perform inference of termination, which generalizes termination checking. With termination inference, annotations such as the parameter induction have not to be provided by the user. It should be desirable to improve our system with such

automatic techniques. Nevertheless, the ability to explicitly specify decreasing size-expressions will be maintained since it is sometimes desirable that the user can provide such information to the system.

7 Conclusion and Future Work

We have illustrated an implemented analyser of Prolog programs which integrates various state-of-the-art techniques of static analyses in a unified framework. The system is useful for optimization and verification. Another interest of the tool is the support of methodologies for constructing correct and efficient logic programs [1, 11]. It was shown how the information collected during the global static analysis is used in order to prove operational properties like termination, determinacy, correct typing, occur-check freeness, etc. The analyser distribution and documentation is available at *http://www.info.ucl.ac.be/∼gobert*.

Implementing a complete analyser is a long-term project, and we will continue this work along the following lines. Presently, each procedure has to be explicitely annotated with a formal specification. This task may appear to be too cumbersome for the programmer. For instance, it should be desirable to improve our system to automatically infer the induction parameter when proving termination. We plan to build an hybrid system where procedures may be specified, partially specified, or not specified at all. A fixpoint algorithm must be implemented, extending the one in [21], and including the definition of widening operators for the abstract domains. Furthermore, the current framework has to be extended for proving sure non-termination (see, for instance, [21]), and for proving that a predicate has an infinite number of solutions.

References

1. K.R. Apt. *From Logic Programming to Prolog.* Prentice Hall, 1997.
2. R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In *M. Hermenegildo and G. Puebla, ed, Static Analysis: Proceedings of the 9th International Symposium*, vol 2477 of LNCS, p. 213-229, Madrid, 2002. Springer-Verlag, Berlin.
3. M. Bruynooghe, B. Demoen, A. Callebaut and G. Janssens. *Abstract Interpretation: Towards the Global Optimization of Prolog Programs.* In Proc. Fourth IEEE Symposium on Logic Programming, San Francisco, CA, Sep. 1987.
4. M. Bruynooghe, M. Codish, J. Gallagher, S. Genaim, and W. Vanhoof, *Termination Analysis through Combination of Type Based Norms*, ACM TPLS. 2005.
5. F. Bueno, P. López-García, M. Hermenegildo. *Multivariant Non-Failure Analysis via Standard Abstract Interpretation.* 7th Int. Symposium on Functional and Logic Programming (FLOPS 2004), LNCS, vol. 2998, p. 100-116, Springer-Verlag, 2004.
6. M. Codish and C. Taboch. A semantic basis for termination analysis of logic programs. *JLP*, 41(1):103-123, 1999.
7. A. Cortesi, B. Le Charlier, P. Van Hentenryck. *Combination of abstract domains for logic programming: open product and generic pattern construction.* Science of Computer Programming 38:27–71, Elsevier Science 2000.
8. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of Fourth ACM Symposium on Programming Languages (POPL'77)*, pages 238–252, Los Angeles, California, January 1977.

9. L. Crnogorac, A.D. Kelly, and H. Søndergaard. *A Comparison of Three Occur-Check Analysers.* SAS. Pages 159-173. 1996.
10. S.K. Debray, P. López García, M. Hermenegildo. *Non-Failure Analysis for Logic Programs.* International Conference on Logic Programming, pages 48-62, MIT Press, Cambridge, MA, June 1997.
11. Y. Deville. *Logic Programming:Systematic Program Development.* A. Wesley, 1990.
12. J. Fischer. *Termination analysis for Mercury using convex constraints.* Honours report, Computer Science and Software Engineering, Melbourne, 2002.
13. S. Genaim, M. Codish, J. Gallagher, and V. Lagoon. *Combining Norms to Prove Termination*, 3rd Int. Workshop on Verification, Model Checking and Abstract Interpretation, Ed. A. Cortesi, LNCS 2294:126–138, Springer-Verlag, Jan. 2002.
14. T. W. Getzinger. *The Costs and Benefits of Abstract Interpretation-driven Prolog Optimization.* In SAS'94, LNCS 864, Springer-Verlag, pages 1–25, 1994.
15. F. Henderson, Z. Somogyi and T. Conway. *Determinism analysis in the Mercury compiler.* In Australian Computer Science Conference, pages 337–346, 1996.
16. J. Henrard and B. Le Charlier. FOLON: An Environment for Declarative Construction of Logic Programs (Extended Abstract). In M. Bruynooghe and M. Wirsing, editors, *4th Int. Workshop on (PLILP'92)*, LNCS, Leuven, 1992. Springer-Verlag.
17. M. Hermenegildo, G. Puebla, F. Bueno, P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). Science of Computer Programming, Vol. 58, Num. 1-2, pages 115-140, Elsevier Science, October 2005.
18. G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by Means of Abstract Interpretation. *JLP*, 13(2-3):205–258, 1992.
19. V. Lagoon, F. Mesnard, P.J. Stuckey. *Termination analysis with types is more accurate.* In Catuscia Palamidessi, editor, Proceedings of the 19th International Conference on Logic Programming, LNCS, Springer-Verlag, 2003.
20. B. Le Charlier, C. Leclère, S. Rossi, and A. Cortesi. *Automated Verification of Prolog Programs. JLP* 39:3–42, Elsevier Science 1999.
21. B. Le Charlier, S. Rossi, and P. Van Hentenryck. Sequence-Based Abstract Interpretation of Prolog. *TPLP* 2(1): 25-84 (2002)
22. B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM TOPLAS*, 16(1):35–101, 1994.
23. N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. *TermiLog*: A system for checking termination of queries to logic programs. In O. Grumberg, editor, *CAV, 9th International Conference*, vol. 1254 of *LNCS*, 63-77. Springer Verlag, June 1997.
24. P. López-García, F. Bueno, M. Hermenegildo. *Determinacy Analysis for Logic Programs Using Mode and Type Information.* Proc. 14th Int. Symp. on Logic-based Program Synthesis and Transformation, LNCS, vol. 3573, p. 19-35, 2005.
25. F. Mesnard and U. Neumerkel. Applying static analysis techniques for inferring termination conditions of logic programs. In P. Cousot, editor, SAS 2001, vol. 2126 of *LNCS*, pages 93-110. Springer Verlag, 2001.
26. G. Puebla, F. Bueno, M. Hermenegildo. *An Assertion Language for Constraint Logic Programs.* Analysis and Visualization Tools for Constraint Programming, LNCS, Num. 1870, pages 23-61, Springer-Verlag, September 2000.
27. Z.Somogyi, F.Henderson, T.Conway.The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language.*JLP*, 29(1–3):17–64, 1996.
28. L. Sterling and E. Shapiro. *The Art of Prolog, Advanced Programming Techniques.* Second Edition. The MIT Press. 1994. Cambridge, Massachusetts. London.

Différentiation automatique et formes de Taylor en analyse statique de programmes numériques

Alexandre Chapoutot et Matthieu Martel

CEA LIST, Laboratoire Modélisation et Analyse de Systèmes en Interaction
Boîte courrier 94, F91191 Gif-sur-Yvette France
{alexandre.chapoutot,matthieu.martel}@cea.fr

Résumé Des travaux récents sur l'analyse statique de programmes numériques ont montré que les techniques d'interprétation abstraite étaient adaptées à la validation de la précision des calculs en arithmétique flottante. L'utilisation des intervalles comme domaine numérique, même avec des méthodes de subdivision, induit une sur-approximation des résultats en particulier par l'existence de l'effet enveloppant (wrapping effect). Une solution utilisée pour éviter ce problème est la définition de domaines relationnels étroitement liés aux propriétés à valider. Nous allons montrer dans cet article, comment l'utilisation les techniques de différentiation automatique peuvent être utilisées pour définir des formes de Taylor permettant de définir une nouvelle analyse statique.

1 Introduction

Les règles de conception des logiciels embarqués critiques imposent, en particulier dans le domaine de l'avionique, d'apporter des garanties sur le "bon comportement" des programmes. Or, l'arithmétique à virgule flottante est de plus en plus utilisée dans ces applications, notamment à cause de la présence d'unités de calcul dédiées, ce qui nécessite la mise au point de méthodes de validation appropriées. Les techniques de test restent difficilement applicables à la détection de problèmes de précision numérique (les pires erreurs peuvent survenir pour des jeux de données très particuliers qui ne correspondent pas par exemple à des cas limites tels que les bornes du domaine d'une variable) et plusieurs travaux de recherche récents ont porté sur la validation par analyse statique de la précision de ces calculs [10,16,9]. Dans cet article, nous présentons une nouvelle technique pour l'étude de la précision numérique, adaptée à l'analyse statique par interprétation abstraite.

Plusieurs méthodes ont été proposées pour étudier la précision numérique des calculs réalisés par un programme [1,15]. Cependant, les objectifs ne sont pas, en général, la validation de comportements numériques mais la représentation des valeurs réelles en machine, c'est-à-dire l'amélioration de la précision des résultats. L'arithmétique d'intervalles permet d'encadrer une valeur réelle par deux nombres machines. Les calculs effectués avec cette arithmétique sont garantis puisqu'à chaque opération le pire cas est considéré, d'où l'apparition de l'*effet enveloppant* cause de sur-approximations. La méthode CESTAC [3] (Contrôle et Estimation STochastique des Arrondis de Calculs) consiste à exécuter plusieurs fois un programme en changeant aléatoirement la manière d'arrondir les opérations, ce qui permet d'estimer le résultat réel d'un calcul. Comme

toute méthode statistique, elle ne garantit les résultats qu'à probabilité près. Un autre exemple d'étude de la précision numérique qui a pour objectif la représentation des réels est la méthode CENA [14] (Correction des Erreurs Numériques d'Arrondi). Elle utilise des informations données par les dérivées des fonctions calculées par un programme pour compenser les erreurs d'arrondi et approcher au mieux les calculs dans les réels. En général, ces méthodes, ne sont pas bien adaptées à la validation des comportements numériques des programmes.

La théorie de l'interprétation abstraite [4] permet, dans certains cas, d'apporter formellement ces garanties. Elle a été utilisée avec succès pour prouver l'absence d'erreurs à l'exécution de programmes embarqués critiques de très grande taille [2] et s'est avérée être adaptée à l'analyse de la précision numérique comme l'ont montré les travaux [7,9,10] qui utilisent une arithmétique flottante avec erreurs permettant de calculer la distance (i.e. l'erreur) séparant le résultat flottant du résultat réel, et de tracer l'origine des erreurs. Ces informations supplémentaires sont très utiles pour la phase de correction.

Les travaux présentés dans cet article ont pour objectif d'améliorer le calcul des erreurs dans l'arithmétique flottante avec erreurs. Pour cela, nous allons utiliser la méthode de la différentiation automatique qui permet de calculer les dérivées d'une fonction et ainsi de mettre au point des approximations polynomiales de la fonction. Le couplage de cette méthode avec l'arithmétique d'intervalles nous permet d'une part de garantir les résultats des approximations et, d'autre part, de limiter l'influence de l'effet enveloppant dans le calcul des erreurs et ainsi limiter le nombre de fausses alarmes dans l'analyse de la précision numérique par interprétation abstraite. La combinaison de l'arithmétique flottante avec erreurs et de la méthode de différentiation automatique est à la base de la nouvelle arithmétique présentée dans cet article : *l'arithmétique de l'erreur différentiée*.

Le reste de cet article est organisé comme suit : La Norme IEEE 754, l'arithmétique flottante avec erreurs et la méthode de la différentiation automatique sont introduites à la section 2. La nouvelle arithmétique est définie à la section 3 ; elle a été implantée dans un prototype et des résultats expérimentaux sont présentés à la section 4.

2 Méthodes numériques utilisées

2.1 Norme IEEE 754

Dans cet article, nous supposons que l'arithmétique utilisée en machine est conforme à la norme IEEE 754 [12,6] dont les principaux points sont brièvement présentés ci-dessous. Cette norme définit l'arithmétique à virgule flottante en base 2 qui est implantée dans la majorité des processeurs actuels. Elle caractérise complètement la représentation mémoire des éléments de cette arithmétique, appelés nombres à virgule flottante, nombres flottants ou flottants. Un flottant est composé en mémoire (au niveau du bit) de trois parties : un signe s valant 1 ou -1, un exposant e compris entre e_{min} et e_{max} et une mantisse m. Le réel r associé à un nombre flottant est donné par la relation $r = s.m.2^e$.

La norme propose plusieurs types de flottants dont les plus importants sont le type simple précision et le type double précision. La précision p d'un nombre flottant corres-

pond au nombre de bits qui compose sa mantisse. De plus, pour une précision donnée les valeurs de e_{min} et e_{max} sont fixées. Par exemple, en double précision la mantisse est codée sur 53 bits, $e_{max} = 1023$ et $e_{min} = -e_{max} - 1$ ce qui correspond à un codage sur 10 bits de l'exposant.

Les opérations $+, -, \times, \div$ et $\sqrt{}$ sont définies mathématiquement par la norme ce qui rend leur implantation indépendante de la machine. La propriété partagée par ces opérations est celle de l'arrondi exact, c'est-à-dire que le résultat r d'une opération flottante o est équivalent au résultat r' obtenu par le calcul de o en précision infinie puis arrondi. La norme définit plusieurs modes d'arrondi dont les principaux sont : vers $+\infty$ qui donne un flottant plus grand que le réel, vers $-\infty$ qui donne un flottant plus petit que le réel et au plus près qui prend le flottant le plus proche du réel.

Une information importante sur les flottants est l'*ulp* (Unit in the Last Place) dont la valeur donne l'ordre de grandeur du dernier bit de la mantisse. L'ulp permet d'estimer la distances séparant deux nombres flottants, c'est-à-dire de majorer l'erreur d'arrondi. Pour un flottant f, $ulp(f) = 2^{-p+e-1}$ avec p la précision du flottant et e son exposant. La majoration de l'erreur dépend du mode d'arrondi : pour les arrondis vers les infinis l'erreur d'arrondi est majorée par $ulp(f)$ tandis qu'avec le mode au plus près la majoration est de $\frac{1}{2}ulp(f)$.

La norme définit aussi des valeurs spéciales qui ont pour but de ne pas interrompre le flot de calcul lors de l'apparition de cas particuliers. Pour cela, il existe les valeurs $+\infty$ et $-\infty$ pour matérialiser les limites de la représentation, par exemple avec l'opération $1/0$ et la valeur NaN (Not a Number) pour les valeurs non représentable, par exemple pour $\sqrt{-1}$.

2.2 Arithmétique flottante avec erreurs

Dans cette section, nous présentons la sémantique de l'erreur globale, notée $[\![.]\!]_{\mathbb{E}}$, qui est une des deux composantes de la nouvelle arithmétique introduite dans cette article. Nous présentons tout d'abord la sémantique concrète qui permet de calculer exactement la distance séparant un résultat flottant du résultat réel, puis la sémantique abstraite qui est basée sur l'arithmétique d'intervalles et qui permet, en pratique, de valider la précision numérique d'un programme.

Dans la suite de cet article, toutes les sémantiques que nous allons définir s'appuient sur le langage des expressions arithmétiques donné à la figure 1. Ce langage est composé de valeurs réelles r, de variables x, et des opérations $+, -, \times, \div$. Chaque élément d'une expression arithmétique est muni d'une étiquette unique ℓ permettant de l'identifier. Ces étiquettes représentent les points de contrôle, c'est-à-dire les points importants du programme pour l'analyse de la précision numérique.

$$a^{\ell} ::= r^{\ell} \mid x^{\ell} \mid a_0^{\ell_0} +^{\ell} a_1^{\ell_1} \mid a_0^{\ell_0} -^{\ell} a_1^{\ell_1} \mid a_0^{\ell_0} \times^{\ell} a_1^{\ell_1} \mid a_0^{\ell_0} \div^{\ell} a_1^{\ell_1}$$

FIG. 1. *Grammaire des expressions arithmétiques étiquetées.*

Nous présentons la sémantique de l'erreur globale qui permet de calculer globalement l'erreur due aux arrondis pour chaque variable du programme.

L'objectif de l'arithmétique flottante avec erreurs est de mesurer la qualité d'un calcul flottant par rapport au même calcul réalisé avec l'arithmétique réelle. Une valeur réelle r est représentée dans par un couple (f, e) avec f la valeur flottante et e l'erreur d'arrondi tel que $r = f + e$. e représente la distance séparant le flottant f du réel r. Ainsi, plus la valeur de e est petite et plus la qualité du calcul est grande. Cette arithmétique est définie suivant la structure d'une expression arithmétique a par $[\![a]\!]^{\natural}_{\mathbb{E}} = (\mathcal{F}^{\natural}(a), \mathcal{E}^{\natural}(a))$. La fonction $\mathcal{F}^{\natural}$, donnée à la figure 2(a), correspond à l'arithmétique flottante telle qu'elle est définie dans la norme IEEE 754. La fonction $\mathcal{E}^{\natural}$, donnée à la figure 2(b), permet de calculer l'erreur globale générée par l'expression arithmétique considérée.

Les fonctions $\mathcal{F}^{\natural}$ et $\mathcal{E}^{\natural}$ utilisent deux fonctions auxiliaires qui sont $\uparrow_{\circ}\colon \mathbb{R} \to \mathbb{F}$ et $\downarrow_{\circ}\colon \mathbb{R} \to \mathbb{R}$. La fonction $\uparrow_{\circ}$ associe à un réel r le nombre flottant f correspondant suivant le mode d'arrondi $\circ$ choisi. Cette fonction définit la partie représentable de r dans les flottants. La fonction $\downarrow_{\circ}$ calcule la partie non représentable de r dans les flottants et est définie par : $\downarrow_{\circ}(r) = r - \uparrow_{\circ}(r)$.

La fonction $\mathcal{F}^{\natural}$ est l'implantation directe de la norme IEEE 754. La fonction $\mathcal{E}^{\natural}$ permet de propager les erreurs entre les différents éléments d'une expression arithmétique. De plus, conformément à la norme IEEE 754, chaque opération introduit une nouvelle erreur issue de l'arrondi du résultat de cette opération. Par exemple, comme on peut le voir à la figure 2(b), l'erreur pour une addition est $\mathcal{E}^{\natural}(a+b) = e_a + e_b + \downarrow_{\circ}(f_a + f_b)$: les erreurs sur les deux opérandes sont additionnées et l'erreur $\downarrow_{\circ}(f_a + f_b)$ due à l'addition flottante $\uparrow_{\circ}(f_a + f_b)$ est elle-même ajoutée au résultat. La sémantique de la soustraction est similaire à celle de l'addition. Quant à celle de la multiplication, elle découle du développement de $(f_a + e_a) \times (f_b + e_b)$. La définition du calcul de l'erreur pour l'opération de l'inverse est plus compliquée puisqu'elle fait intervenir une décomposition en série entière [16,15].

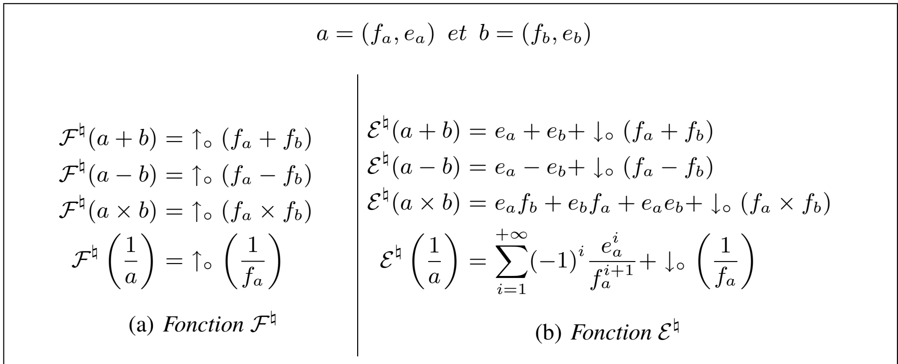


FIG. 2. *Définition des fonctions $\mathcal{F}^{\natural}$ et $\mathcal{E}^{\natural}$.*

Cette sémantique permet de calculer le couple flottant et erreur à chaque point de contrôle du programme. Une instabilité numérique est détectée quand la valeur de l'erreur est importante.

L'objectif de la sémantique abstraite, notée $[\![.]\!]^\sharp_{\mathbb{E}}$, est d'étudier la précision numérique d'un programme pour des classes d'exécution. Nous voulons valider le comportement numérique du programme pour des plages d'entrées possibles. Pour cela, nous abstrayons les ensembles d'entrées réelles R possibles par un couple d'intervalles $([f],[e])$. $[f]$ est une sur-approximation de l'ensemble des flottants représentant R en machine et $[e]$ est une sur-approximation de l'ensemble des erreurs $\downarrow_\circ (x)$, $x \in R$. Dans la suite de cet article, tout symbole entre crochets [] dénotera un intervalle. L'arithmétique d'intervalles est plus longuement définie dans [17,1].

Nous étendons la fonction $\downarrow_\circ$ à des valeurs d'intervalles grâce à la fonction $\downarrow_\circ^I$ définie par l'équation (1). Cette fonction est paramétrée par le mode d'arrondi qui détermine l'intervalle d'erreur. Pour un intervalle $[a,b]$ avec a la borne inférieure et b la borne supérieure, l'erreur d'arrondi maximale η est donnée par la valeur de l'ulp de la plus grande borne, telle que définie à la section 2.1. L'ensemble des erreurs d'arrondi est alors donné par l'intervalle $[-\eta,\eta]$.

$$\downarrow_\circ^I ([a,b]) = ulp(\max(|a|,|b|)) \times [-1,1] \quad (1)$$

La sémantique $[\![.]\!]^\sharp_{\mathbb{E}}$ est définie sur la structure d'une expression arithmétique a par $[\![a]\!]^\sharp_{\mathbb{E}} = (\mathcal{F}^\sharp(a), \mathcal{E}^\sharp(a))$. $\mathcal{F}^\sharp$ et $\mathcal{E}^\sharp$ sont les extensions des fonctions $\mathcal{F}^\natural$ et $\mathcal{E}^\natural$ à l'arithmétique d'intervalles.

Exemple 1 *Soit l'expression arithmétique tirée de [9]* $p = x - ax$ *avec* $x = 1$ *dans les flottants et avec une erreur comprise dans l'intervalle* $[-0.5, 0.5]$)*, c'est-à-dire que* x *varie dans les réels entre* $[0.5, 1.5]$. *Nous fixons* $a = 0.65$ *et nous considérons qu'aucune erreur n'est attachée à* a. *Nous considérons, pour simplifier, que les calculs flottants n'engendrent pas de nouvelle erreur.*

$$\begin{aligned}
\mathcal{F}^\sharp(p) &= \mathcal{F}^\sharp(x) - \mathcal{F}^\sharp(ax) \\
&= [1,1] - [0.65, 0.65] \times [1,1] \\
&= [1,1] \times [0.65, 0.65] \\
&= [0.65, 0.65] \\
\mathcal{E}^\sharp(p) &= \mathcal{E}^\sharp(x) - \mathcal{E}^\sharp(ax) \\
&= [-0.5, 0.5] - ([0,0] \times [1,1] + [-0.5, 0.5] \times [0.65, 0.65] + [0,0] \times [-0.5, 0.5]) \\
&= [-0.5, 0.5] - [-0.325, 0.325] \\
&= [-0.825, 0.825]
\end{aligned}$$

Le résultat réel est alors dans l'intervalle $[-0.175, 1.475]$ *alors que le flottant est* 0.65 *il y a donc une erreur importante.*

L'exemple précédent montre que les dépendances entre les valeurs affectent les résultats de l'analyse. La méthode de différentiation automatique permet de combler ce manque.

2.3 Différentiation automatique

Cette partie présente la différentiation automatique [11] qui est la deuxième composante de la nouvelle arithmétique définie à la section 3.

L'idée de la différentiation automatique est de considérer un programme comme une fonction mathématique définie par composition de fonctions élémentaires. En nous restreignant aux expressions arithmétiques, les fonctions élémentaires sont les opérations $+, -, \times, \div$ et les fonctions étudiées sont toutes les compositions possibles de ces opérations. Cette méthode s'appuie sur la règle de dérivation en chaîne : si f et g sont deux fonctions différentiables alors $f \circ g$ est différentiable telle que $(f \circ g)' = f'(f \circ g')$. Cette règle donne la façon de calculer la dérivée de fonctions composées, par application des règles mathématiques usuelles de dérivation. La différentiation est réalisée suivant les variables du programme et l'on distingue deux types de variables : les *variables indépendantes* et les *variables dépendantes*. Les variables indépendantes sont la plupart du temps les variables d'entrée du programme et elles sont utilisées pour la différentiation. Les variables dépendantes sont celles pour lesquelles nous voulons calculer les dérivées et sont en général les sorties du programme. De plus, les variables sont dites *actives* si elles sont accompagnées d'une information de dérivée.

Nous définissons la sémantique $[\![.]\!]_{\mathbb{D}}^{\natural}$ associée à cette méthode suivant la structure d'une expression arithmétique a telle que $[\![a]\!]_{\mathbb{D}}^{\natural} = (\mathcal{F}^{\natural}(a), \mathcal{D}^{\natural}(a))$. $\mathcal{F}^{\natural}$ est la fonction définie à la figure 2(a) implantant la norme IEEE 754 et $\mathcal{D}^{\natural}$, définie à la figure 3, correspond au calcul des dérivées en un point. Une valeur réelle est représentée par un couple (f, d) avec f la valeur flottante et d la valeur de la dérivée au point f. Comme nous l'avons mentionné précédemment, la fonction $\mathcal{D}^{\natural}$ est définie par les règles de dérivation mathématique.

$$a = (r_a, d_a) \ \ et \ \ b = (r_b, d_b)$$

$$\begin{aligned}
\mathcal{D}^{\natural}(a + b) &= d_a + d_b \\
\mathcal{D}^{\natural}(a - b) &= d_a - d_b \\
\mathcal{D}^{\natural}(a \times b) &= d_a f_b + d_b f_a \\
\mathcal{D}^{\natural}\left(\frac{1}{a}\right) &= -\frac{d_a}{f_a^2} \ \ \text{si } f_a \neq 0
\end{aligned}$$

FIG. 3. *Définition de la fonction* $\mathcal{D}^{\natural}$.

Cette méthode permet, par le biais des valeurs de la différentielle, une analyse de dépendances entre les variables présentes dans le programme. En effet, la différentielle est composée de dérivées partielles calculées par rapport aux variables actives du programmes. Si une dérivée partielle p associée à la variable x est nulle pour une variable active v alors nous pouvons conclure que v ne dépend pas de x. A l'inverse, si p a la plus grande valeur non nulle parmi les dérivées partielles composant la différentielle de v alors la variable x est la plus influente dans le calcul de v.

Exemple 2 *Reprenons l'exemple de la section 2.2 avec* $p = x - ax$ *et calculons la dérivée de* p *par rapport à* x*.* $a = (0.65, 0)$ *avec* 0.65 *la valeur flottante et* 0 *la valeur de la dérivée de* a *par rapport à* x*.* $x = (1, 1)$ *où le premier* 1 *est la valeur flottante et le second* 1 *représente la valeur de la dérivée de* x *par rapport à* x*.*

$$\begin{aligned} \mathcal{D}^{\natural}(p) &= \mathcal{D}^{\natural}(x) - \mathcal{D}^{\natural}(ax) \\ &= 1 - (0.65 \times 1 + 0 \times 1) \\ &= 0.35 \end{aligned}$$

x *a une influence dans le calcul de* p *et nous en déduisons qu'une erreur initiale* e *sur* x *induit une approximation du résultat de* a *dans les flottants de* $0.35e$*.*

3 Arithmétique de l'erreur différentiée

Dans cette section, nous présentons une nouvelle sémantique, notée $[\![.]\!]_{\mathbb{ED}}$, basée sur les méthodes de différentiation automatique et de l'erreur globale introduites aux sections 2.2 et 2.3. Nous introduisons tout d'abord à la section 3.1 une première sémantique, notée $[\![.]\!]^1_{\mathbb{ED}}$ et permettant de calculer des approximations linéaires de l'erreur, puis nous montrerons à la section 3.2 comment cette sémantique peut être étendue pour réaliser des approximations polynomiales basées sur des polynômes de Taylor. Nous détaillerons une extension basée sur des polynômes de Taylor du second ordre, notée $[\![.]\!]^2_{\mathbb{ED}}$.

Conformément à la théorie de l'interprétation abstraite [4,5], nous allons tout d'abord définir des sémantiques concrètes pour $[\![.]\!]^1_{\mathbb{ED}}$ et $[\![.]\!]^2_{\mathbb{ED}}$, c'est-à-dire des sémantiques spécifiant exactement le comportement d'un programme au cours d'une exécution, puis nous donnerons des versions abstraites de ces sémantiques dans lesquelles les ensembles de valeurs sont abstraits par des intervalles afin de pouvoir raisonner, au cours d'une analyse statique, sur des ensembles d'exécutions d'un programme.

3.1 Différentiation au premier ordre

Sémantique concrète Nous présentons maintenant la sémantique qui permet d'approximer linéairement la fonction $\mathcal{E}^{\natural}$ pour chaque variable du programme. Nous partons du constat qu'à chaque point de contrôle d'une expression arithmétique (c.f. figure 1) une nouvelle erreur est introduite. L'erreur introduite au point de contrôle ℓ sera notée o^{ℓ}. Elle correspond soit à l'erreur de représentation d'une constante, soit à l'erreur d'arrondi du résultat d'une opération. $\mathcal{E}^{\natural}$ (c.f. figure 2(b)) est une fonction des variables o^{ℓ} dont le résultat est l'erreur globale. Nous allons alors appliquer la méthode de différentiation automatique à la fonction $\mathcal{E}^{\natural}$ par rapports aux variables o^{ℓ}. Dans la sémantique concrète $[\![.]\!]^{1\natural}_{\mathbb{ED}}$, une valeur réelle r est représentée par un triplet (f, e, d) avec f la valeur flottante, e l'erreur globale et d la différentielle de e par rapport aux variables o^{ℓ}. $[\![.]\!]^{1\natural}_{\mathbb{ED}}$ est définie suivant la structure d'une expression arithmétique a telle que

$$[\![a]\!]^{1\natural}_{\mathbb{ED}} = (\mathcal{F}^{\natural}(a), \mathcal{E}^{\natural}(a), \mathcal{D}^{1\natural}_{\mathcal{E}}(a))$$

avec $\mathcal{F}^{\natural}$ l'implantation de la norme IEEE 754 (c.f. figure 2(a)) et $\mathcal{E}^{\natural}$ la fonction de calcul de l'erreur globale (c.f. figure 2(b)). La fonction $\mathcal{D}^{1\natural}_{\mathcal{E}}$ est la composition de la

fonction $\mathcal{E}^\natural$ et de la fonction $\mathcal{D}^\natural$ dont la définition est donnée à la figure 4, elle calcule le gradient de $\mathcal{E}^\natural$. On obtient ainsi un gradient dont les composantes sont de la forme $\partial\mathcal{E}^\natural/\partial o^\ell$.

La sémantique d'une constante réelle c au point de contrôle ℓ est donnée par le triplet $(\uparrow_\circ(c), \downarrow_\circ(c), \frac{\downarrow_\circ(c)}{\partial o^\ell})$. $\uparrow_\circ(c)$ est la partie représentable de c dans les flottants, $\downarrow_\circ(c)$ est l'erreur de représentation et $\frac{\downarrow_\circ(c)}{\partial o^\ell}$ est le vecteur dont toutes les composantes sont nulles sauf la ℓ-ième qui vaut 1. Comme pour la fonction $\mathcal{D}^\natural$, la définition de la fonction $\mathcal{D}_\mathcal{E}^{1\natural}$ s'appuie sur les règles de dérivation mathématique. Pour les opérations d'addition et de soustraction nous obtenons des combinaisons linéaires des vecteurs de dérivées partielles des opérandes auxquelles nous ajoutons le vecteur associé à la nouvelle erreur d'arrondi. Le calcul pour l'opération de multiplication se base sur la formule $ax+by+xy$ avec a et b des constantes et, x et y les variables suivant lesquelles nous différentions. Nous obtenons la différentielle $ax'+by'+x'y+xy' = x'(a+y)+y'(b+x)$ et, en posant $a = f_a$, $b = f_b$, $x = e_a$, $y = e_b$, $x' = d_a$ et $y' = d_b$, nous obtenons la définition de la figure 4.

$$a = (f_a, e_a, d_a) \;\; et \;\; b = (f_b, e_b, d_b)$$

$$\begin{aligned}
\mathcal{D}_\mathcal{E}^{1\natural}(a+b) &= d_a + d_b + \frac{\partial\downarrow_\circ(f_a+f_b)}{\partial o^{\ell_i}}\\
\mathcal{D}_\mathcal{E}^{1\natural}(a-b) &= d_a - d_b + \frac{\partial\downarrow_\circ(f_a-f_b)}{\partial o^{\ell_i}}\\
\mathcal{D}_\mathcal{E}^{1\natural}(a\times b) &= d_a(f_b+e_b) + d_b(f_a+e_b) + \frac{\partial\downarrow_\circ(f_a\times f_b)}{\partial o^{\ell_i}}\\
\mathcal{D}_\mathcal{E}^{1\natural}\left(\frac{1}{a}\right) &= \sum_{i=1}^{+\infty}(-1)^i\frac{1}{f^{i+1}}e^{i-1}d_a + \frac{\partial\downarrow_\circ(\frac{1}{f_a})}{\partial o^{\ell_i}}
\end{aligned}$$

FIG. 4. *Définition de la fonction* $\mathcal{D}_\mathcal{E}^{1\natural}$.

Dans cette sémantique, le calcul des dérivées partielles ne permet qu'une analyse de dépendances puisque le terme d'erreur est calculé dans les réels. Par contre, nous nous servirons de leurs valeurs dans la sémantique abstraite pour calculer des ensembles d'erreurs, représentés par des intervalles, mais en réduisant de manière significative l'effet enveloppant.

Sémantique abstraite La sémantique abstraite, notée $[\![.]\!]^{1\sharp}_{\mathbb{ED}}$, permet comme la sémantique abstraite $[\![.]\!]^{\sharp}_{\mathbb{E}}$ (c.f. figure 2.2) d'évaluer la précision numérique pour des classes d'exécutions. Nous présentons dans cette partie la manière dont nous calculons des approximations linéaires garanties de l'erreur avant de définir formellement $[\![.]\!]^{1\sharp}_{\mathbb{ED}}$.

L'analyse par intervalles [13] introduit la notion de *fonctions d'inclusion*. A une fonction mathématique μ est associée une fonction informatique ι calculée à l'aide de l'arithmétique d'intervalles. Cette fonction ι est une fonction d'inclusion si l'image

d'un intervalle $[x]$ par μ est contenu dans l'image de $[x]$ par ι. La fonction d'inclusion naturelle est celle définie directement à partir des opérations de l'arithmétique d'intervalles. Nous notons $[f]_n$ la fonction d'inclusion naturelle associée à la fonction mathématique f. Cependant, il est possible de définir d'autres fonctions d'inclusion.

Le théorème des accroissements finis dit : "Soit f une fonction différentiable sur l'intervalle $[a, b]$, de différentielle g, alors il existe c dans $]a, b[$ tel que $f(b) = f(a) + g(c)(b - a)$". En généralisant, nous obtenons :

$$\forall x \in [a, b],\ \exists c \in]a, b[,\ f(x) = f(m) + g(c)(x - m) \tag{2}$$

où m est le milieu de l'intervalle $[a, b]$. Ces théorèmes nous affirment l'existence de cette valeur c mais pas le moyen de la calculer. En utilisant l'arithmétique d'intervalles pour calculer g nous obtenons l'ensemble des dérivées de f sur $[a, b]$ et nous obtenons la relation :

$$\forall x \in [a, b],\ f(x) \in f(m) + [g]_n([a, b])(x - m) \tag{3}$$

Par extension, nous obtenons :

$$f([a, b]) \subseteq f(m) + [g]_n([a, b])([a, b] - m) \tag{4}$$

Nous pouvons alors définir la fonction d'inclusion centrée, notée $[f]_c$ pour la fonction mathématique f, définie pour toute fonction f différentiable sur un intervalle $[x]$ dont le milieu est noté m et de différentielle g. Nous avons :

$$[f]_c([x]) = f(m) + [g]_n([x])([x] - m) \tag{5}$$

Cette définition peut être étendue sans difficulté à des fonctions manipulant des valeurs vectorielles. Grâce à cette nouvelle fonction d'inclusion nous pouvons définir un nouveau calcul d'ensemble d'erreurs en prenant comme fonction f la fonction $\mathcal{E}^\natural$ et comme intervalle $[x]$ le vecteur d'intervalles d'erreurs élémentaires, noté $\boldsymbol{O}$.

Dans la sémantique abstraite, un ensemble de valeurs réelles R est représenté par un triplet $([f], m, [d])$ avec $[f]$ l'intervalle de flottants, m le centre de l'intervalle d'erreurs et $[d]$ le vecteur de dérivées partielles. La sémantique abstraite est définie sur la structure d'une expression arithmétique a telle que $[\![a]\!]^{1\sharp}_{\mathbb{ED}} = (\mathcal{F}^\sharp(a), \mathcal{E}^\sharp_m(a), \mathcal{D}^\sharp_\mathcal{E}(a))$. La fonction $\mathcal{F}^\sharp$ est l'extension de la fonctions $\mathcal{F}^\natural$ à l'arithmétique d'intervalles. La fonction $\mathcal{E}^\sharp_m$, définie à la figure 5, est une adaptation de la fonction $\mathcal{E}^\sharp$ au calcul de la valeur centrée de l'erreur globale. Elle suit les mêmes règles de calcul que $\mathcal{E}^\sharp$ mais en convertissant systématiquement tous les intervalles en une valeur unique qui est leur centre. Nous notons mid la fonction, qui calcule le centre d'un intervalle, définie par $mid([a, b]) = \frac{a+b}{2}$.

La fonction $\mathcal{D}^\sharp_\mathcal{E}$ est une extension de la fonction $\mathcal{D}^\natural_\mathcal{E}$ à l'arithmétique d'intervalles où chaque occurrence d'une variable d'erreur e est remplacée par un calcul de l'approximation linéaire définie par la relation (6). Cette substitution permet de calculer l'ensemble des dérivées premières de l'intervalle d'erreur et ainsi permet de garantir les approximations. $mid(\boldsymbol{O})$ représente l'application de la fonction mid sur toutes les

$$a = ([f_a], m_a, [d_a]) \;\; et \;\; b = ([f_b], m_b, [d_b])$$

$$\mathcal{E}^\sharp_m(a+b) = m_a + m_b + mid(\downarrow^I_\circ ([f_a] + [f_b]))$$
$$\mathcal{E}^\sharp_m(a-b) = m_a - m_b + mid(\downarrow^I_\circ ([f_a] - [f_b]))$$
$$\mathcal{E}^\sharp_m(a \times b) = mid(m_a[f_b]) + mid(m_b[f_a]) + m_a m_b + mid(\downarrow^I_\circ ([f_a] \times [f_b]))$$
$$\mathcal{E}^\sharp_m\left(\frac{1}{a}\right) = \sum_{i=1}^{+\infty}(-1)^i \frac{m_a^i}{[f_a]^{i+1}} + mid\left(\downarrow^I_\circ \left(\frac{1}{[f_a]}\right)\right)$$

FIG. 5. *Définition de la fonction $\mathcal{E}^\sharp_m$.*

composantes de $\boldsymbol{O}$. L'ensemble réel R est obtenu par la relation $R = [f] + E$. Ce calcul permet d'obtenir l'ensemble des dérivées de l'intervalle d'erreurs.

$$E \subseteq m + [d](\boldsymbol{O} - mid(\boldsymbol{O})) \tag{6}$$

Exemple 3 *Reprenons l'exemple de la section 2.2 avec $p = x - ax$, $x = ([1,1], 0, [1,1]_x)$ où 0 est le milieu de l'intervalle d'erreur initiale $[-0.5, 0.5]$, et $[1,1]$ est la valeur de la dérivée de e_x par rapport à l'erreur initiale. La valeur de a est $a = ([0.65, 0.65], 0, [1,1]_a)$ (erreur initiale nulle). Nous considérons pour simplifier que les calculs flottants n'engendrent pas de nouvelles erreurs. La notation x_y représente une valeur d'erreur ou de dérivée x associée à la variable y.*

$$\begin{aligned}
\mathcal{E}^\sharp_m(p) &= \mathcal{E}^\sharp_m(x) - (mid(\mathcal{E}^\sharp_m(a) \times \mathcal{F}^\sharp(x)) + mid(\mathcal{E}^\sharp_m(x) \times \mathcal{F}^\sharp(a)) + \mathcal{E}^\sharp_m(a) \times \mathcal{E}^\sharp_m(x)) \\
&= 0 - (mid(0 \times [1,1]) + mid(0 \times [0.65, 0.65]) + 0 \times 0 \\
&= 0 \\
\mathcal{D}^{1\sharp}_{\mathcal{E}}(p) &= \mathcal{D}^{1\sharp}_{\mathcal{E}}(x) - (\mathcal{D}^{1\sharp}_{\mathcal{E}}(a) \times \mathcal{F}^\sharp(x) + \mathcal{D}^{1\sharp}_{\mathcal{E}}(x) \times \mathcal{F}^\sharp(a) + \mathcal{D}^{1\sharp}_{\mathcal{E}}(x) \times \mathcal{E}^\sharp(a) + \mathcal{D}^{1\sharp}_{\mathcal{E}}(a) \times \mathcal{E}^\sharp(x)) \\
&= [1,1]_x - ([1,1]_a \times [1,1] + [1,1]_x \times [0.65, 0.65] + [1,1]_x \times 0 + [1,1]_a \times [-0.5, 0.5]) \\
&= [1,1]_x - ([-0.5, 0.5]_a, [0.65, 0.65]_x) \\
&= ([-0.5, 0.5]_a, [0.65, 0.65]_x)
\end{aligned}$$

Nous savons par le calcul des dérivées que p a une erreur qui dépend de l'erreur sur a et de l'erreur sur x. Nous obtenons alors un ensemble d'erreurs avec $\otimes$ le produit scalaire et $(0_a, [-0.5, 0.5]_x)$ le vecteur d'erreurs initiales :

$$\begin{aligned}
E &= \mathcal{E}^\sharp_m(p) + \mathcal{D}^{1\sharp}_{\mathcal{E}}(a) \otimes (0_a, [-0.5, 0.5]_x) \\
&= 0 + ([-0.5, 0.5]_a, [0.65, 0.65]_x) \otimes (0_a, [-0.5, 0.5]_x) \\
&= [-0.325, 0.325]
\end{aligned}$$

Nous avons alors un ensemble réel R compris dans l'intervalle $[0.325, 0.975]$ avec le flottant toujours égal à 0.65.

3.2 Différentiation d'ordre supérieur

La différentiation d'ordre supérieur permet de prendre en compte beaucoup plus d'informations de dépendance, ce qui permet de définir des approximations plus pré-

cises. Nous montrons les possibilités d'extension de la sémantique précédente par application successive de la différentiation automatique afin de calculer des dérivées d'ordre n.

La fonction $\mathcal{D}^\natural$ peut-être appliquée sur la fonction $\mathcal{D}_\mathcal{E}^{1\natural}$ pour donner la fonction $\mathcal{D}_\mathcal{E}^{2\natural}$ calculant les dérivées secondes de la fonction $\mathcal{E}^\natural$ par rapport aux o^ℓ. En appliquant à nouveau la fonction $\mathcal{D}^\natural$ sur la fonction $\mathcal{D}_\mathcal{E}^{2\natural}$ nous pouvons calculer les dérivées troisièmes et ainsi de suite pour calculer les dérivées n-ième.

Grâce à la connaissance des dérivées de tous ordres, nous pouvons définir une fonction d'inclusion basée sur des formes de Taylor. Le théorème des accroissements finis se généralise aux fonction différentiables n fois par la formule de Taylor-Lagrange. Et, en procédant de la même manière qu'à la section 3.1, nous pouvons définir une fonction d'inclusion de Taylor, notée $[f]_T$ telle que :

$$[f]_T([x]) = f(m) + f'(m)([x]-m) + \ldots + f^{n-1}(m)\frac{([x]-m)^{n-1}}{(n-1)!} + [f^n]_n([x])\frac{([x]-m)^n}{n!} \tag{7}$$

Cette définition s'étend aussi sans difficulté à des valeurs vectorielles. L'avantage d'une telle fonction d'inclusion est de n'utiliser l'arithmétique d'intervalle qu'à l'ordre n, en général sur des petits ensembles de valeurs, et donc le limiter grandement l'effet enveloppant.

Nous présentons une extension à l'ordre 2 de la sémantique définie à la section 3.1, notée $[\![.]\!]^2_{\mathbb{ED}}$. Une valeur réelle r dans la sémantique concrète est représentée par un quadruplet (f, e, j, h) où f et e représentent respectivement la partie représentable et non représentable de r et j et h représentent les vecteurs de dérivées partielles au premier et au second ordre de l'erreur globale suivant les o^ℓ. La sémantique $[\![.]\!]^{2\natural}_{\mathbb{ED}}$ est définie sur la structure d'une expression arithmétique a par $[\![a]\!]^{2\natural}_{\mathbb{ED}} = (\mathcal{F}^\natural(a), \mathcal{E}^\natural_m(a), \mathcal{D}^{1\natural}_\mathcal{E}(a), \mathcal{D}^{2\natural}_\mathcal{E}(a))$. Les fonctions $\mathcal{F}^\natural(a)$, $\mathcal{E}^\natural_m(a)$ et $\mathcal{D}^{1\natural}_\mathcal{E}$ sont les mêmes que précédemment et sont respectivement définies aux figures 2(a), 2(b) et 4. La fonction $\mathcal{D}^{2\natural}_\mathcal{E}$ est définie à la figure 6. Sa définition est obtenue de la même manière que la définition de la fonction $\mathcal{D}^{1\natural}_\mathcal{E}$. La difficulté vient du calcul des dérivées secondes à partir du produit de deux vecteurs de dérivées premières. Il faut réaliser une multiplication matricielle entre les deux vecteurs afin d'obtenir toutes les combinaisons linéaires possibles, d'où la transposition de vecteurs.

Comme pour la sémantique $[\![.]\!]^1_{\mathbb{ED}}$, le calcul des dérivées premières et secondes ne permettent que de faire une analyse de dépendance. Les dérivées d'ordre 2 permettent de connaître l'influence des combinaisons d'erreurs élémentaires, introduites par l'opération de multiplication en particulier, sur la valeur de l'erreur globale. Ceci, nous permet d'étudier les erreurs d'ordre supérieur, erreurs résultats du produit de deux erreurs, qui ont dans certains cas une influence non négligeable sur le calcul de l'erreur.

La sémantique abstraite, notée $[\![.]\!]^{2\sharp}_{\mathbb{ED}}$, utilise la fonction d'inclusion de Taylor d'ordre 2 pour calculer des ensembles d'erreurs. Cette fonction est donnée par l'équation 8 et elle induit un changement dans le calcul des dérivées premières définies à la section 3.1 puisque ces dérivées ne doivent plus être calculées sur la totalité de l'intervalle d'erreur mais uniquement au centre. Nous définissons à la figure 7 la fonction $\mathcal{D}^{1\sharp}_{m\mathcal{E}}$ calculant les dérivées d'ordre 1 au niveau du centre de l'intervalle d'erreur, c'est-à-dire suivant

$$a = (f_a, e_a, j_a, h_a) \;\; et \;\; b = (f_b, e_b, j_b, h_b)$$

$$\begin{aligned}
\mathcal{D}^{2\natural}_{\mathcal{E}}(a+b) &= h_a + h_b \\
\mathcal{D}^{2\natural}_{\mathcal{E}}(a-b) &= h_a - h_b \\
\mathcal{D}^{2\natural}_{\mathcal{E}}(a\times b) &= h_a f_b + h_b f_a + {}^t j_b j_a + e_a j_b + {}^t j_a j_b + e_b j_a \\
\mathcal{D}^{2\natural}_{\mathcal{E}}\left(\frac{1}{a}\right) &= \sum_{i=1}^{+\infty} (-1)^i \frac{i}{f_a^{i+1}} \left(h_a e_a^{i-1} + (i-1) j_a^2 e^{i-2}\right)
\end{aligned}$$

FIG. 6. *Définition de la fonction* $\mathcal{D}^{2\natural}_{\mathcal{E}}$.

les valeurs de la fonction $\mathcal{E}^{\sharp}_m$. Cette définition est construite de la même manière que la fonction $\mathcal{E}^{\sharp}_m$ (c.f. figure 5) et le résultat de chaque opération mettant en jeu un intervalle est réduit en son centre grâce à la fonction mid.

$$[f]_{T2}([x]) = f(m) + f'(m)([x]-m) + \frac{1}{2}([x]-m)[f'']_n([x])([x]-m) \qquad (8)$$

Un ensemble de valeurs réelles R est représenté dans la sémantique abstraite $[\![.]\!]^{2\sharp}_{\mathbb{ED}}$ par un quadruplet $([f], m, j, [h])$ avec $[f]$ l'ensemble des valeurs flottantes associées à R, m la valeur centrée de l'erreur, j le vecteur de dérivées partielles premières calculée au point m et h la matrice de dérivées secondes. Les dérivées secondes sont représentées sous forme de matrice où les lignes et les colonnes représentent les points de contrôle et où la valeur de la case d'indices i, j représente la dérivée seconde de la fonction $\mathcal{E}^{\natural}$ par rapport aux erreurs élémentaires o^{ℓ_i} et o^{ℓ_j}. Cette dérivée représente la contribution du produit de o^{ℓ_i} et de o^{ℓ_j} dans la valeur de l'erreur globale.

$$a = ([f_a], m_a, j_a, [h_a]) \;\; et \;\; b = ([f_b], m_b, j_b, [h_b])$$

$$\begin{aligned}
\mathcal{D}^{1\sharp}_{m\mathcal{E}}(a+b) &= j_a + j_b + \frac{\partial \downarrow_\circ (a+b)}{\partial o^{\ell_i}} \\
\mathcal{D}^{1\sharp}_{m\mathcal{E}}(a-b) &= j_a - j_b + \frac{\partial \downarrow_\circ (a-b)}{\partial o^{\ell_i}} \\
\mathcal{D}^{1\sharp}_{m\mathcal{E}}(a\times b) &= mid(j_a[f_b]) + mid(j_b[f_a]) + m_a j_b + m_b j_a + \frac{\partial \downarrow_\circ (a\times b)}{\partial o^{\ell_i}} \\
\mathcal{D}^{1\sharp}_{m\mathcal{E}}\left(\frac{1}{a}\right) &= \sum_{i=1}^{+\infty} (-1)^i mid\left(\frac{i}{[f_a]^{i+1}}\right) m^{i-1} j_a + \frac{\partial \downarrow_\circ (\frac{1}{a})}{\partial o^{\ell_i}}
\end{aligned}$$

FIG. 7. *Définition de la fonction* $\mathcal{D}^{1\sharp}_{m\mathcal{E}}$.

La sémantique $[\![.]\!]^{2\sharp}_{\mathbb{ED}}$ est définie sur la structure d'une expression arithmétique a telle que $[\![a]\!]^{2\sharp}_{\mathbb{ED}} = (\mathcal{F}^\sharp(a), \mathcal{E}^\sharp_m(a), \mathcal{D}^{1\sharp}_{m\mathcal{E}}(a), \mathcal{D}^{2\sharp}_{\mathcal{E}}(a))$. Comme pour la sémantique abstraite $[\![.]\!]^{1\sharp}_{\mathbb{ED}}$, la fonction $\mathcal{D}^{2\sharp}_{\mathcal{E}}$ est une extension de la fonction $\mathcal{D}^{2\natural}_{\mathcal{E}}$ à l'arithmétique d'intervalle où chaque occurrence d'une variable d'erreur est remplacée par le calcul de l'approximation polynomiale de l'erreur. Cette définition nous permet de calculer l'ensemble des dérivées partielles secondes sur l'intervalle d'erreur et elle nous permet d'avoir des approximations garanties.

4 Résultats expérimentaux

Nous avons implanté la sémantique de l'erreur différentiée à l'ordre 1 et 2 dans un prototype écrit en OCaml[1] permettant d'analyser un noyau de langage impératif basé sur le langage C. Nous obtenons des résultats qui confirment l'intérêt de la composition de sémantiques dans l'étude de la précision numérique. Nous illustrons les bonnes propriétés de la sémantique $[\![.]\!]_{\mathbb{ED}}$ par deux exemples. Le premier exemple permet d'apprécier le gain de précision des résultats entre un calcul d'erreur dans les intervalles et un autre avec notre nouvelle sémantique. Cet exemple est basé sur un calcul de racine carré par une méthode de Newton. Le second exemple a été construit, à partir de la suite $x_{n+1} = x_n - ax_n$, et met en évidence l'intérêt d'une différentiation de l'erreur au second ordre.

Dans ces deux exemples, nous avons analysé les programmes en utilisant une arithmétique flottante multi-précision pour calculer les erreurs. L'arithmétique multi-précision est une implantation logicielle de l'arithmétique flottante permettant de fixer arbitrairement le nombre de bits de la mantisse. De plus, afin d'atteindre un certain niveau de précision dans les résultats des analyses, nous avons déplié les boucles avant d'appliquer l'opérateur d'élargissement, quasi indispensable en interprétation abstraite, mais difficile à définir en analyse de la précision numérique. En effet, une méthode courante pour définir cet opérateur utilise un ensemble de valeurs paliers [2]. Dans notre cas, il n'est pas possible de définir a priori des paliers servant pour l'élargissement des termes d'erreur. La technique alors adoptée est la dégradation de la précision. A chaque application de l'opérateur d'élargissement, nous diminuons le nombre de bits servant à représenter les erreurs. Cette diminution a pour effet d'augmenter la valeur de l'*ulp* et permet ainsi d'avoir un ensemble dynamique de paliers.

Le premier programme implante une méthode de Newton calculant la racine carrée d'un nombre a strictement positif ; le code source est donné à la figure 8(a). Ce calcul s'appuie sur la récurrence définie par l'équation :

$$x_{n+1} = \frac{x_n}{2}\left(3 - ax_n^2\right)$$

Le résultat du programme, c'est-à-dire la racine carrée r d'un nombre a, est $r = x_p \times a$ avec x_p la valeur du point fixe de la fonction. Nous avons analysé les propriétés numériques de ce programme avec les sémantiques $[\![.]\!]^\sharp_{\mathbb{E}}$ et $[\![.]\!]^{1\sharp}_{\mathbb{ED}}$ et un dépliage initial de la boucle de 10 itérations. Notre sémantique de l'erreur différentiée permet d'améliorer

[1] http ://caml.inria.fr/

```
double xn = 0.1;
double xn1 = 0.0;
double a = [25, 25]_[-0.01, 0.01];
int cond = 0;
double temp = 0.0;
double res = 0.0;

while (cond < 1) {
  xn1 = 0.5 * xn * (3.0 - a * xn * xn);
  temp = xn1 - xn;
  if (temp < 1e-12) {
    cond = 2;
  }
  if (temp > -1e-12) {
    cond = 2;
  }
  xn = xn1;
  res = xn1 * a;
}
```

(a) *Racine carrée par la méthode de Newton.*

```
double a = [0.8,0.9];
double b = 0.35;
double x = [0.79,0.99]_[-0.1,0.1];

while (1) {
  x = a * x * x - b * x * x;
}
```

(b) *Suite géométrique quadratique en x.*

FIG. 8. *Code source des exemples.*

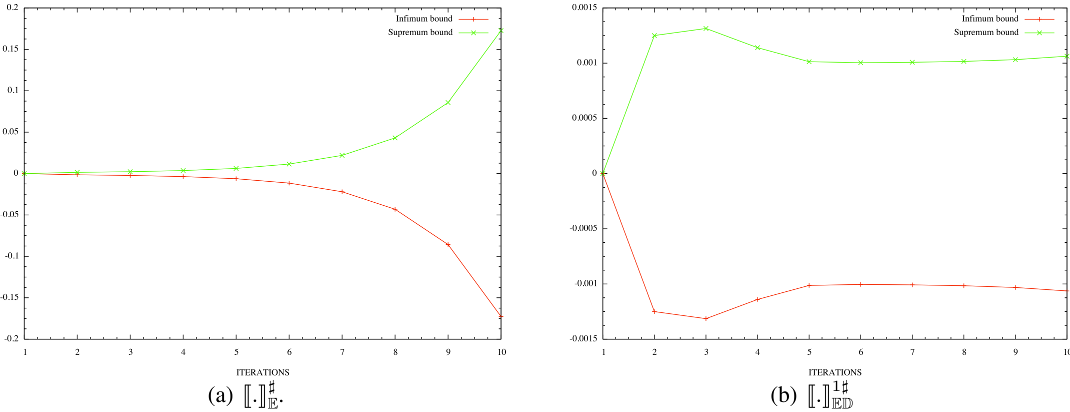
le calcul des erreurs en prenant en compte les relations qui les lient. Mais elle n'agit pas sur les valeurs flottantes. Afin d'assurer la stabilité de l'algorithme dans les flottants, nous avons choisi de calculer la racine carrée d'une valeur simple 25.0 mais entachée d'une erreur $[-0.01, 0.01]$.

L'analyse avec la sémantique $[\![.]\!]^\sharp_{\mathbb{E}}$ qui utilise l'arithmétique d'intervalles pour le calcul du flottant et de l'erreur ne permet pas d'apprécier la qualité du résultat flottant qui est $[5.0, 5.0]$ alors que l'évolution de l'intervalle d'erreur grandit. La figure 9(a) montre cette évolution en fonction des itérations. La sémantique $[\![.]\!]^{1\sharp}_{\mathbb{ED}}$ permet, grâce à la prise en compte des relations entre erreurs, de calculer un terme d'erreur qui converge. L'évolution de l'erreur calculée par approximation linéaire est donnée à la figure 9(b). Grâce à ces informations nous pouvons conclure que le résultat flottant est très proche du résultat réel.

Le second exemple est une suite géométrique régie par l'équation $x_{n+1} = ax_n^2 - bx_n^2$. Le programme associé à cette suite est donné à la figure 8(b). Nous avons comme valeur initiale x_0 l'intervalle flottant $[0.79, 0.99]$ et celle-ci est entachée d'une erreur comprise dans l'intervalle $[-0.1, 0.1]$, c'est-à-dire que les valeurs réelles sont dans l'intervalle $[0.69, 1.09]$. La constante a est dans l'intervalle $[0.8, 0.9]$ et la constante b est égale à 0.35. Naturellement, cette suite converge positivement vers 0 quand n tend vers $+\infty$.

Une analyse par la sémantique $[\![.]\!]^\sharp_{\mathbb{E}}$ conduit à une valeur flottante qui converge vers 0 mais un terme d'erreur qui rapidement explose (à partir de la 13-ième itération) pour être de la forme $[-MAX_FLOAT, MAX_FLOAT]$ avec $MAX_FLOAT = 1.7976931348623157e^{308}$. Ce résultat est uniquement dû à l'arithmétique d'intervalles qui ne permet pas de détecter que la soustraction utilise le même x_n^2.

Par contre, une analyse par la sémantique $[\![.]\!]^{1\sharp}_{\mathbb{E}}$ permet de calculer un terme d'erreur plus proche du modèle mathématique. L'évolution de l'erreur est donnée à la figure 10(a). Nous obtenons alors un terme d'erreur convergeant vers 0 au bout d'une douzaine



(a) $[\![.]\!]^{\sharp}_{\mathbb{E}}$. (b) $[\![.]\!]^{1\sharp}_{\mathbb{ED}}$

FIG. 9. *Intervalle d'erreur pour chaque itération du programme 8(a).*

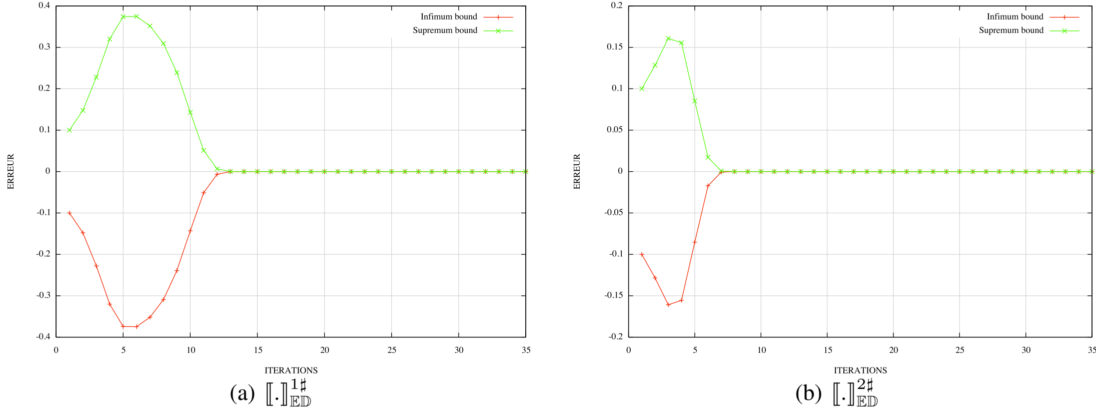
d'itérations avec une amplitude de $[-0.3742, 0.3742]$. Le passage à l'ordre 2 dans le calcul de l'erreur en utilisant la sémantique $[\![.]\!]^{2\sharp}_{\mathbb{E}}$ permet d'affiner encore ce résultat comme le montre l'évolution de l'erreur donnée à la figure 10(b). Cette sémantique montre une convergence vers 0 en 7 itérations et avec un intervalle d'erreur qui est au maximum de $[-0.161, 0.161]$.

L'élévation au carré des x_n engendre des erreurs d'ordre supérieur qui sont de plus en plus prépondérantes dans le calcul de l'erreur globale. L'application directe de l'arithmétique d'intervalles ne permet pas de détecter les relations entre les erreurs et donc fournit un mauvais résultat. La différentiation au premier ordre capte les relations entre erreurs élémentaires mais pas entre erreurs d'ordre supérieur ce qui conduit à un bon résultat mais qui peut encore être amélioré, ce que fait la sémantique à l'ordre deux.

5 Conclusion

Dans cet article, nous avons présenté une nouvelle arithmétique dédiée à l'étude de la précision numérique dans le cadre de la validation de programmes numériques. Cette arithmétique est issue de la combinaison de deux méthodes numériques existantes : l'erreur globale et la différentiation automatique. Elle permet de calculer plus finement les erreurs d'arrondi en diminuant l'effet enveloppant dans les calculs et ainsi permet d'obtenir des analyses plus précises.

Nous nous sommes concentrés sur l'amélioration du calcul des erreurs d'arrondi. Or les termes d'erreurs dépendent aussi des valeurs flottantes, en particulier dans le cas de la multiplication. Une sur-approximation de cet ensemble de flottants se répercute dans le calcul de l'erreur même avec la technique présentée ici. Une solution envisagée est d'utiliser des formes relationnelles qui introduisent des contraintes linéaires pour affiner le calcul d'ensemble de valeurs flottantes [8,9] et ainsi réduire l'effet enveloppant au niveau de la partie flottante. [9] définit une sémantique relationnelle basée sur des



(a) $[\![.]\!]^{1\sharp}_{\mathbb{ED}}$ (b) $[\![.]\!]^{2\sharp}_{\mathbb{ED}}$

FIG. 10. *Intervalle d'erreur pour chaque itération du programme 8(b).*

formes affines permettant de limiter l'effet enveloppant dans le calcul flottant mais pas dans le calcul des erreurs. Une combinaison entre cette sémantique et la sémantique de l'erreur différentiée conduirait à des analyses très précises prenant en considération les dépendances entre flottants et erreurs. [8] propose aussi de prendre en compte toutes ces dépendances, en ajoutant à la sémantique de [9] des formes affines entre les erreurs. Une comparaison de cette sémantique complètement relationnelle avec la combinaison de la sémantique de [9] et la notre est envisagée.

Par ailleurs, la connaissance des dérivées de la fonction $\mathcal{E}^{\natural}$, nous laisse entrevoir la possibilité de définir un opérateur d'élargissement adapté pour l'étude de la précision numérique. L'opérateur d'élargissement, élément important de la théorie de l'interprétation abstraite, permet d'accélérer la convergence du calcul de point fixe. Nous examinons la possibilité de définir un tel opérateur en se basant sur l'évolution des dérivées.

Références

1. J-C. Bajard, O. Beaumont, J-M. Chesneaux, M. Daumas, J. Erhel, D. Michelucci, J-M. Muller, B. Philippe, N. Revol, J-L. Roch, and J. Vignes. *Qualité des Calculs sur Ordinateurs. Vers des arithmétiques plus fiables ?* Masson, 1997.
2. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *Programming Language Design and Implementation (PLDI'03)*, ACM, pages 196–207. ACM Press, 2003.
3. J.M. Chesneaux and J. Vignes. Sur la robustesse de la méthode cestac. *Comptes Rendus de l'Académie des Sciences, Paris*, 307(1) :855–860, 1988.
4. P. Cousot and R. Cousot. Abstract Interpretation : a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Principles of Programming Languages (POPL'77)*, ACM, pages 238–252. ACM Press, 1977.
5. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Symbolic Computation*, 2(4) :511–547, 1992.

6. D. Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, 23(1) :5–48, 1991.
7. E. Goubault. Static Analyses of Floatting-Point Operations. In *Static Analysis Symposium (SAS '01)*, volume 2126 of *LNCS*. Springer Verlag, 2001.
8. E. Goubault and S. Putot. Weakly relational domains for floating-point computation analysis. In *First International Workshop on Numerical and Symbolic Abstract Domains*, 2005.
9. E. Goubault and S. Putot. Static Analysis of Numerical Algorithms. In *Static Analysis Symposium (SAS '06)*, LNCS. Springer Verlag, 2006.
10. E. Goubault, S. Putot, and M. Martel. Some Future Challenges in the Validation of Control Systems. In *Embedded Real Time Software (ERTS'06)*. SIA, 2006.
11. A. Griewank. *Evaluating Derivatives : Principles and Techniques of Algorithmic Differentiation*. Society for Industrial and Applied Mathematics, 2000.
12. IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floatting-Point Arithmetic*. IEEE, New York, 1985.
13. L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied Interval Analysis*. Springer, 2001.
14. P. Langlois. Automatic Linear Correction of Rounding Errors. Technical report, INRIA, 1999.
15. M. Martel. An Overview of Semantics for the Validation of Numerical Programs. In *Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *LNCS*. Springer Verlag, 2005.
16. M. Martel. Semantics of roundoff error propagation in finite precision calculations. *Journal of Higher Order and Symbolic Computation*, 19 :7–30, 2006.
17. R.E. Moore. *Methods and Applications of Interval Arithmetic*. Studies in Applied Mathematics. SIAM, 1979.

Présentations d'outils

Formal Requirements Modelling and Early Verification & Validation of Critical Systems

C. Ponsard, P. Massonet, J.F. Molderez and G. Dallons

CETIC Research Center, Charleroi (Belgium) - {cp,phm,jfm,gd}@cetic.be

Abstract. The FAUST toolset supports the formal layer of the KAOS goal oriented methodology for requirements engineering. It enables the early formalisation of requirements, their validation using animation techniques, the verificaiton of consitency/completness using model-checking techniques and the generation of acceptance test cases.

1 The FAUST toolset

Achieving assurance requires quality throughout the whole development lifecycle: from requirements to specification, architecture, code and tests. Among those, it is widely recognised that the main cause of project failure still remains the poor quality of requirements. Our focus is on the requirements problem and its relation to the rest of the lifecycle. More precisely we want to answer the following questions:

- Validation: do we have the right requirements ?
- Verification: are the requirements right ?
- Acceptance: is the deliverable right ?

The FAUST toolset [1][4] supports the KAOS goal-oriented approach[7]. A goal model captures system and environment properties as well as agent capabilities/responsibilities. The model is first structured semi-formally, with extended UML notations. The system parts identified as critical can then be formalized using a real-time temporal logic. FAUST enables deep analysis of the system behaviours to be carried out at that early stage using the tools described hereafter.

2 Validation using the Requirements Animator

In order to validate the system requirements, the animator can simulate and display behaviours of the future system [6]. The simulation process relies on finite state machines (FSM) which are generated from a scoped subset of properties, enabling incremental validation. The user interface can display FSM in UML notations and also with graphical animations familiar to the validating user. The animator can also be used for model debugging. For this purpose, a monitor automatically checks for the violation of all properties in the animation scope.

3 Verification using the Early Analyser

The early analyzer can perform a variety of checks on the model in order to provide the formal assurance that abstract goals are correctly refined into more concrete ones, that system operations do enforce the requirements, that obstacles are not present, etc. Through the use of model-checking, the checks are fully automated and produce explanatory counter-examples on failure.

4 Acceptance Test Case Generation

FAUST also supports the generation of complex artefacts such as tests cases [2]. Bridges to recognised formal industrial development processes are also addressed, such as B[3] and others are being considered, such as AADL. Those formal objects can always be traced to informal statements, ensuring the communication with end-users and enabling impact analysis.

5 An Integrated Toolset

FAUST is based on the Objectiver requirements engineering platform [5] and is now also partially integrated within Eclipse through standard model-based technologies. It is also opening to external tools such as Rodin (Event-B) and Topcased (AADL). This tool presentation will emphasise the latest developments, especially the Eclipse integration and the test cases generation tool.

Acknowledgement

This work is financially supported by the European Union (ERDF and ESF) and the Walloon Region (DGTRE).

References

1. FAUST, *http://faust.cetic.be*.
2. J.F. Molderez and C. Ponsard, *Deriving acceptance tests from goal requirements*, 2nd International Mozart/Oz Conference, Charleroi (Belgium), September 2004.
3. C. Ponsard and E. Dieul, *From requirements models to formal specifications in B*, in Proc. CAISE Workshops, Regulations Modelling and their Validation and Verification (REMO2V), Luxembourg, June 2006.
4. C. Ponsard, P. Massonet, A. Rifaut, J.F. Molderez, A. van Lamsweerde, and H. Tran Van, *Early verification and validation of mission critical systems*, 8th FMICS Workshop, Linz (Austria), July 2004.
5. The Objectiver Tool, *http://www.objectiver.com*.
6. H. Tran Van, A. van Lamsweerde, P. Massonet, and C. Ponsard, *Goal-oriented requirements animation*, 12th IEEE Int.Req.Eng.Conf., Kyoto, September 2004.
7. A. van Lamsweerde, *Goal-oriented requirements engineering: A guided tour*, Proc. RE'01 - 5th IEEE International Symposium on Requirements Engineering, 2001.

Intégration du support OCL dans Kermeta
Spécifiez la sémantique statique de vos méta-modèles

Jean-Marie Mottu - Olivier Barais - Mark Skipper – Didier Vojtisek - Jean-Marc Jézéquel

Projet Triskell/IRISA
Campus de Beaulieu.
F - 35 042 Rennes Cedex
prenom.nom}@irisa.fr

Abstract. Ce document présente succinctement l'intégration du support OCL pour le langage de méta-modélisation exécutable Kermeta. L'accent dans cette courte présentation est mis sur les cas d'utilisation possibles de cette brique logicielle et l'intérêt des choix techniques retenus : un support natif du paradigme de Conception par Contrats (Design by Contract) dans Kermeta et une transformation de modèle du méta-modèle OCL au méta-modèle Kermeta pour le support de la syntaxe concrète OCL
Keywords : Ingénierie Dirigée par les Modèles, Sémantique statique, Méta-modélisation, logique du premier ordre, Conception par contrats

1. Motivations

La méta-modélisation dans le domaine de l'informatique se définit comme la mise en évidence d'un ensemble de concepts pour un domaine particulier. Un modèle est une abstraction d'un phénomène du monde réel tandis qu'un méta-modèle est une abstraction d'ordre supérieur mettant en évidence les concepts utilisés pour définir le modèle. Dans ce domaine, on peut parler d'un modèle conforme à son méta-modèle comme on peut parler d'un programme conforme à sa grammaire. L'OMG au travers du MOF [1] propose un langage standardisé afin de permettre la définition de nouveaux méta-modèles. Kermeta [2], développé au sein de l'équipe Triskell, est un langage de méta-modélisation exécutable construit comme une extension du MOF. Il permet de définir un méta-modèle auquel on associe une sémantique opérationnelle pour permettre la simulation des modèles. Le MOF ainsi que Kermeta permettent de définir un certain nombre de contraintes sur le modèle représenté, portant entre autres sur la cardinalité des relations entre les concepts. Ces contraintes peuvent servir à vérifier la cohérence d'un modèle par rapport à son méta-modèle. Cependant, un langage comme le MOF manque, au même titre qu'UML, d'expressivité pour représenter un certains nombre de contraintes dans les futurs méta-modèles. Un exemple consiste à regarder la définition d'un méta-modèle qui permet d'écrire des modèles de composants possédant des ports et des opérations fournies et requises au niveau de ses ports, deux composants étant reliables grâce à un concept de connecteur entre leurs ports. Il est impossible avec un langage comme le MOF ou Kermeta de

définir la sémantique statique de ce méta-modèle, entre autres de définir dans ce méta-modèle ce qu'est un assemblage de composants valides.

OCL (Object Constraint Language) incorpore la notion de conception par contrats [4] en UML. Largement étendu depuis la première version datant de 1999, la version 2 du langage [3] propose actuellement une syntaxe déclarative simple fondée sur une logique du premier ordre permettant à la fois d'associer des contraintes à un méta-modèle et de définir des requêtes sur des modèles. Sa proximité avec UML est, en outre, moins forte autorisant son utilisation dans le cadre de la méta-modélisation. L'utilisation d'OCL dans le cadre de la méta-modélisation offre un moyen simple d'exprimer la sémantique statique d'un méta-modèle dans le cas où celle-ci est exprimable avec une logique du premier ordre. Nous proposons dans la présentation d'exposer le support d'OCL en Kermeta et d'expliquer l'intérêt de ce support pour un futur concepteur de méta-modèles.

```
operation foo(param : String) : String
  pre myprecondition is param != ""
  post apostcondition is result.size > param.size
  is do
     result := param + " world"
  end
```

Fig. 1. Syntaxe Kermeta pour la définition des pré et des post-conditions

2. Architecture

L'intégration d'OCL dans Kermeta est construite à l'aide de deux briques de base.

- L'ajout de la notion de conception par contrats « à la Eiffel » dans Kermeta. La définition des invariants et des pré-post conditions se fait alors à l'aide de la syntaxe concrète Kermeta. (voir exemple de la Figure 1)
- Le support de la syntaxe concrète d'OCL en fournissant une transformation de modèles entre l'AST d'OCL et l'AST de Kermeta. Cette transformation est écrite en Kermeta. Le modèle résultant de la transformation est automatiquement associé avec le méta-modèle auquel il s'applique grâce à un mécanisme de composition appelé aussi merge de modèles. Le schéma de la figure 2 illustre l'architecture générale de la transformation.

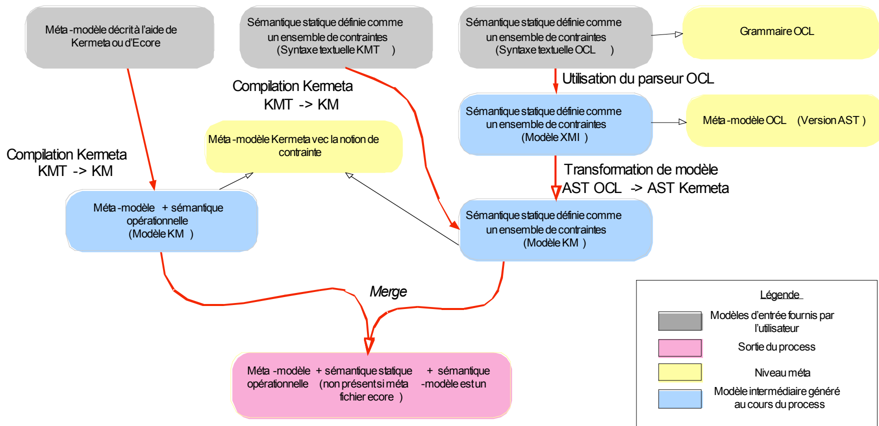


Fig 1. Vue globale de l'architecture

3. Intérêts et atouts

Le support d'OCL dans Kermeta apporte au moins trois principaux avantages pour un concepteur de méta-modèle.

- Pour les concepteurs de méta-modèle qui utilisent déjà Kermeta, ce support améliore la compatibilité avec les standards existants et permet l'intégration de contraintes définies en OCL. Ce mécanisme permet alors dans tous les cas de définir simplement la sémantique statique d'un méta-modèle. Kermeta offre ainsi un moyen de vérifier à la demande la correction d'un modèle avec sa sémantique statique au cours d'une simulation.
- D'un point de vue méthodologique pour le concepteur de méta-modèle, cette architecture permet une bonne séparation des préoccupations entre la description du méta-modèle, la description de la sémantique statique et la description de la sémantique opérationnelle. Chacune de ces sémantiques étant en effet définies à l'aide d'une syntaxe dédiée dans une unité dédiée : avec un fichier ecore par exemple pour le méta-modèle, un fichier Kermeta pour la sémantique opérationnelle et un fichier OCL pour la sémantique statique.
- Enfin, pour les utilisateurs d'OCL, l'implémentation d'OCL en Kermeta offre un moyen très simple de mettre en œuvre une extension d'OCL. En effet, OCL offre la possibilité d'appeler des méthodes définies dans des méta-classes du méta-modèle à partir du moment où ces méthodes n'ont pas d'effet de bord sur le modèle. La définition en Kermeta de la sémantique opérationnelle de ces méthodes permet une mise en œuvre et une évaluation de ces nouvelles primitives du langage.

[1] OMG, MOF 2.0 Core Final Adopted Specification. 2004.
[2] P-A. Muller, F. Fleurey, and J-M Jézéquel. -- Weaving executability into object-oriented meta-languages. -- In S. Kent L. Briand, editor, Proceedings of MODELS/UML'2005, volume 3713 of LNCS, pages 264--278, Montego Bay, Jamaica, October 2005. Springer.
[3] OMG, UML 2.0 Object Constraint Language (OCL) Final Adopted specification. 2003.
[4] B. Meyer. Applying "Design by Contract". Computer 25, 10 (Oct. 1992), 40-51.

haRVey: satisfaisabilité et théories

Diego Caminha B. de Oliveira (Univ. Rio Grande do Norte, Brésil)
David Déharbe* (Univ. Rio Grande do Norte, Brésil),
Pascal Fontaine** (LORIA – Université de Nancy)

Univ. Rio Grande do Norte, Brésil / LORIA – Université de Nancy

Le problème de la satisfaisabilité de formules logiques est au centre des méthodes formelles: les modèles formels (par exemple la méthode B [1], et TLA$^+$ [6, 7]) et les raffinements sont validés au moyen d'un générateur de conditions de vérification, la preuve de ces formules étant déléguée à un module particulier de l'atelier de modélisation. Nous présentons ici le prouveur de formules haRVey.

Le logiciel haRVey, comme la plupart des solveurs SMT (Satisfiability Modulo Theories [10]), s'appuie, pour la gestion de la structure booléenne des formules, sur un solveur SAT (voir par exemple [12, 4]). Ces logiciels décident efficacement le problème de la satisfaisabilité booléenne. Un exemple simple de formule pouvant être réfuté au moyen d'un solveur SAT est la suivante:

$$\neg\big[\,(p \Rightarrow q) \Rightarrow \big[\,(\neg p \Rightarrow q) \Rightarrow q\big]\big].$$

Les solveurs SMT élèvent le langage accepté par les solveurs SAT à un langage toujours décidable, mais plus expressif que la logique booléenne. Le logiciel haRVey, comme nombre de ses concurrents, accepte l'égalité, et les symboles non interprétés, en utilisant l'algorithme de clôture de congruence [5, 9]. Il est donc capable par exemple de reconnaître que la formule

$$a = b \wedge \big[f(a) \neq f(b) \vee (p(a) \wedge \neg p(b))\big]$$

est inconsistante. Les techniques de combinaison de théories à la Nelson-Oppen [8, 11] permettent d'intégrer le raisonnement de divers théories décidables. Par exemple, il est possible, à partir de la fermeture de congruence, et d'une procédure de décision pour l'arithmétique linéaire, de construire une procédure de décision qui comprends à la fois les symboles non-interprétés, et les symboles de l'arithmétique linéaire, afin d'analyser une formule du type:

$$a \leq b \wedge b \leq a + x \wedge x = 0 \wedge \big[f(a) \neq f(b) \vee (p(a) \wedge \neg p(b + x))\big].$$

Nous montrons aussi deux capacités originales de l'outil: son aptitude à accepter dans une combinaison, des théories du premier ordre fixées par l'utilisateur (par un ensemble d'axiomes, voir par exemple [2]), et sa capacité à traiter certains opérateurs ensemblistes, comme dans:

$$a = b \wedge f(a) \in (A \cap B) \wedge \big[f(a) \in A \setminus B \vee f(b) \notin B\big].$$

* david@dimap.ufrn.br
** Pascal.Fontaine@loria.fr

Le but de cette démonstration est de donner à la communauté des méthodes formelles une idée précise du langage accepté par l'outil, et de lui permettre d'apprécier les performances du logiciel. Nous voulons identifier les méthodes de modélisation et de développement prouvé pouvant bénéficier, pour la vérification des formules générées pendant le processus, de l'assistance de ce logiciel de preuve automatique. Enfin, les besoins des utilisateurs pourront guider le développement de la version future du logiciel [3].

On peut télécharger le logiciel, et trouver les publications le décrivant, sur le site `http://harvey.loria.fr`.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. A. Armando, S. Ranise, and M. Rusinowitch. A rewriting approach to satisfiability procedures. *Information and Computation*, 183(2):140–164, 2003.
3. D. Déharbe and P. Fontaine. haRVey: combining reasoners. In *Sixth International Workshop on Automated Verification of Critical Systems*, 2006.
4. N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.
5. P. Fontaine. *Techniques for verification of concurrent systems with invariants*. PhD thesis, Institut Montefiore, Université de Liège, Belgium, Sept. 2004.
6. L. Lamport. *Specifying Systems*. Addison-Wesley, Boston, Mass., 2002.
7. S. Merz. On the logic of TLA^+. *Computers and Informatics*, 22:351–379, 2003.
8. G. Nelson and D. C. Oppen. Simplifications by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, Oct. 1979.
9. G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, Apr. 1980.
10. S. Ranise and C. Tinelli. The SMT-LIB standard : Version 1.2, Aug. 2006.
11. C. Tinelli and M. T. Harandi. A new correctness proof of the Nelson–Oppen combination procedure. In F. Baader and K. U. Schulz, editors, *Frontiers of Combining Systems (FroCoS)*, Applied Logic, pages 103–120. Kluwer Academic Publishers, Mar. 1996.
12. L. Zhang and S. Malik. The quest for efficient Boolean satisfiability solvers. In A. Voronkov, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Computer Science*, pages 295–313. Springer-Verlag, 2002.

VeSTA : Vérification de la préservation des propriétés d'un composant lors de son intégration dans un système temporisé

Jacques Julliand, Hassan Mountassir, and Emilie Oudot

LIFC - Laboratoire d'Informatique de l'Université de Franche-Comté
16, route de Gray, 25030 Besançon Cedex
{julliand,mountass,oudot}@lifc.univ-fcomte.fr

L'intégration de composants est une méthode de développement incrémental utilisée dans le cadre des systèmes à base de composants. Il s'agit de vérifier qu'un composant s'intègre correctement dans un environnement, i.e., que les propriétés du composant sont préservées par l'intégration. Lors d'une vérification par model-checking, cela permet de contourner le problème d'explosion combinatoire en menant la vérification sur des modèles plus petits. Dans [1], nous avons défini une τ-simulation pour les systèmes temporisés, appelée *τ-simulation temporisée sensible à la divergence et respectant la stabilité*, qui préserve les propriétés MITL (Metric Interval Temporal Logic)[2], le non-zénonisme fort, et l'absence de blocage. La propriété de composabilité exprime que tout composant A simule sa composition avec d'autres composants. Elle garantit donc la préservation des propriétés de A lors d'une intégration. La simulation définie assure la composabilité vis-à-vis de l'opérateur de composition *non bloquant* de [3] (sous certaines hypothèses simples), mais pas vis-à-vis de l'opérateur classique de composition pour les systèmes temporisés. Ainsi, pour garantir la préservation lors d'une intégration avec cet opérateur, il faut vérifier algorithmiquement la simulation. C'est l'objet de l'outil VeSTA[1] (**Ve**rification of **S**imulations for **T**imed **A**utomata).

Présentation générale. VeSTA considère donc des systèmes temporisés à base de composants, modélisés par des automates temporisés [4] avec l'opérateur de composition classique noté $\|$. La fonctionnalité principale de VeSTA est de vérifier si les propriétés locales d'un composant A sont préservées quand A est intégré dans un environnement E. Pour ce faire, VeSTA vérifie si A simule l'automate $A\|E$ réalisant l'intégration. La simulation peut être vérifiée soit totalement, pour assurer la préservation de toutes les propriétés pouvant être exprimées sur A, soit de manière partielle, pour vérifier la préservation de propriétés spécifiques exprimées sur le composant. Cette seconde approche permet d'optimiser la vérification de la simulation, mais garantit uniquement la préservation des propriétés spécifiées. Dans les deux cas, si la vérification échoue, VeSTA fournit un diagnostic graphique, composé d'une trace de $A\|E$ qui n'est simulée par aucune trace de A, ainsi que de la trace de A à laquelle il est attendu qu'elle corresponde.

[1] VeSTA est disponible à l'adresse : `http://lifc.univ-fcomte.fr/~oudot/VeSTA`

Eléments techniques. VeSTA a été développé en JAVA (pour l'interface utilisateur) et en C (pour les modules de vérification de la simulation). Il utilise la bibliothèque SMI (Symbolic Model Interface), qui fournit une représentation symbolique basée sur les diagrammes de décision, pour des modèles finis décrits par des réseaux d'automates. Ce choix a été guidé par l'outil Open-Kronos [5], dont nous avons adopté pour VeSTA une architecture similaire. La partie centrale de l'outil comporte trois modules. Le module `translator` traduit les deux AT A et $A||E$ en un fichier C, contenant les structures de données et fonctions permettant de générer un graphe symbolique pour chaque AT. Ce fichier est de plus créé de manière à pouvoir le connecter à la plate-forme de vérification Open-Caesar [6]. Les modules `simul` et `Profounder` (ce dernier provient d'Open-Kronos) permettent alors de vérifier la simulation et de retourner les diagnostics.

Expérimentations. Les premiers résultats obtenus sont prometteurs. Nous avons notamment traité l'exemple de la cellule de production de [7]. Nous avons identifié des propriétés locales à certains composants de ce système, et comparé la vérification classique (directement sur le modèle complet du système, avec le model-checker KRONOS[8]) et l'approche par intégration de composants (vérification locale avec KRONOS et vérification de la préservation avec VeSTA). Il s'avère que, même sur ce système de taille raisonnable, la méthode classique nécessite un temps de calcul d'environ 20 secondes, tandis que la méthode par préservation nécessite moins d'une seconde. Les résultats détaillés sont présentés dans [9].

Références

1. Bellegarde, F., Julliand, J., Mountassir, H., Oudot, E. : On the contribution of a τ-simulation in the incremental modeling of timed systems. In : FACS'05. Volume 160 of ENTCS., Macao, Macao, Elsevier (2005) 97–111
2. Alur, R., Feder, T., Henzinger, T. : The benefits of relaxing punctuality. Journal of the ACM **43** (1996) 116–146
3. Bornot, S., Sifakis, J. : An Algebraic Framework for Urgency. Information and Computation **163** (2000) 172–202
4. Alur, R., Dill, D. : A theory of timed automata. Theoretical Computer Science **126** (1994) 183–235
5. Tripakis, S. : The analysis of timed systems in practice. PhD thesis, Universite Joseph Fourier, Grenoble, France (1998)
6. Garavel, H. : OPEN/CAESAR : An Open Software Architecture for Verification, Simulation and Testing. In : TACAS'98, Lisboa, Portugal (1998)
7. Burns, A. : How to verify a safe real-time system : The application of model-checking and timed automata to the production cell case study. Real-Time Systems Journal **24** (2003) 135–152
8. Yovine, S. : KRONOS : A verification tool for real-time systems. Journal of Software Tools for Technology Transfer **1** (1997) 123–133
9. Bellegarde, F., Julliand, J., Mountassir, H., Oudot, E. : Experiments in the use of τ-simulations for the components-verification of real-time systems. In : SAVCBS'06, Portland, USA (2006) Available on ACM Digital Library.

Test fonctionnel pour Focal

Matthieu Carlier & Catherine Dubois

ENSIIE - CEDRIC
18 allée Jean Rostand, 91025 Évry Cedex, France
{carlier, dubois}@ensiie.fr

Introduction

L'atelier Focal (voir [DHG04] et la bibliographie proposée sur le site Focal http ://focal.inria.fr), développé conjointement par des chercheurs des équipes SPI (LIP6), CPR (Cedric) et de l'Inria, est un atelier intégré de construction modulaire de logiciels certifiés. L'atelier permet l'écriture de composants comprenant des déclarations, des définitions, des propriétés et des preuves. Les déclarations peuvent être raffinées en définitions et les propriétés en preuves, par passage progressif de la spécification à l'implantation, à l'aide de mécanismes d'héritage, de liaison retardée et d'instanciation de paramètres. Les définitions sont ici des fonctions à la ML (appelées aussi méthodes dans le langage Focal), les propriétés des énoncés du 1er ordre.

L'atelier incorpore un outil de preuve, Zenon [Dol], démonstrateur au style déclaratif, du 1er ordre, fondé sur la méthode des tableaux. Il offre également un compilateur qui traduit les modules Focal en code exécutable OCaml et en code Coq pour les besoins de certification. Il peut aussi fournir automatiquement une documentation dans un format à la XML.

Nous présentons ici un nouvel outil intégré à l'atelier Focal, l'outil de test automatique appelé FocTest. Cet outil prend en entrée une ou plusieurs propriétés, supposées exécutables, et vérifie que ces propriétés sont satisfaites sur un ensemble de données générées aléatoirement. On ne peut utiliser cet outil que si les fonctions utilisées dans les propriétés ont été définies afin de pouvoir exécuter les tests. Néanmoins le code des fonctions n'est pas utilisé. FocTest est par conséquent un outil de test fonctionnel, nous parlons ici de *test de propriétés*.

Dans un environnement orienté preuve tel que Focal, les raisons de faire appel à FocTest sont diverses :
- Avant de se lancer dans la preuve d'une propriété, il peut s'avérer intéressant de la tester. L'outil peut générer un ou plusieurs contre-exemples. Si c'est le cas, le code ne satisfait pas la propriété. De même au cours d'une preuve, on peut utiliser l'outil FocTest pour tester un lemme intermédiaire jugée utile dans la preuve.
- On peut aussi avoir recours à FocTest avec un logiciel certifié (produit avec Focal) pour valider ce dernier vis à vis de propriétés nouvelles (c'est-à-dire des propriétés qui n'ont pas été introduites pendant les phases de spécification et de conception).
- Il est possible d'importer dans un composant Focal des fonctions écrites en

OCaml. Les propriétés qui spécifient ce code ne peuvent être prouvées et sont donc irrémédiablement admises. Il convient, pour avoir une plus grande confiance, de tester ces propriétés.

Un aperçu de la technique mise en œuvre

Les propriétés acceptées par FocTest sont de la forme $\forall X_1 \dots X_n . \alpha_1 \Rightarrow \dots \alpha_n \Rightarrow (A_1^1 \vee \dots \vee A_{n_1}^1) \wedge \dots \wedge (A_1^m \vee \dots \vee A_{n_m}^m)$, c'est-à-dire des propriétés en forme prénexe et sans quantificateur existentiel. Les α_i sont des formules avec les connecteurs $\vee$, $\wedge$ et $\neg$, les A représentent des appels de méthodes.

Pour les besoins du test, une propriété de cette forme est réécrite en un ensemble de propriétés plus simples, appelées *formes élémentaires* dont la conjonction est équivalente à la propriété initiale. Les formes élémentaires sont de la forme $A_1 \Rightarrow \dots \Rightarrow A_n \Rightarrow \dots \Rightarrow B_1 \vee \dots \vee B_m$. Chacune de ces formes élémentaires constitue un couple pré-condition/conclusion, la pré-condition est la conjonction des A_i et la conclusion est la prémisse droite de la suite d'implications $B_1 \vee \dots \vee B_m$.

L'approche adoptée par FocTest est de prendre une à une les formes élémentaires et de générer des jeux de test puis d'appliquer ceux-ci sur l'implantation donnée par l'utilisateur. La phase de test se découpe donc en deux parties : la génération de jeux de test et la soumission de jeux de test avec calcul du verdict et écriture d'un rapport de test au format XML.

Pour qu'un jeu de test soit considéré comme valide, il faut que celui-ci satisfasse la pré-condition. En effet dans le cas contraire, le jeu de test n'est pas pertinent car il ne permet pas d'apprécier la conclusion.Ainsi, FocTest génère pseudo-aléatoirement des données de test et les rejette si elles ne valident pas la pré-condition. Sinon, on a un jeu de test (validé) qui est alors soumis à l'implantation. Puis FocTest évalue la conclusion. Si celle-ci est vérifiée, le jeu de test *passe*. Sinon, le jeu de test *échoue* et nous avons alors produit un contre-exemple.

Conclusion

L'outil présenté dans cet article a été utilisé sur la librairie de calcul formel implantée avec Focal (incorporée à la distribution Focal courante) et a permis de relever une erreur.

Afin d'améliorer la génération des jeux de test (dans le cas de préconditions très contraignantes), nous étudions la possibilité de générer les données en analysant les spécifications et le code des fonctions intervenant dans les préconditions.

Références

[DHG04] C. Dubois, T. Hardin, and V. Viguié Donzeau Gouge. Building certified components within FOCAL. In H.-W. Loidl, editor, *Symposium on Trends in Functional Programming (TFP04)*, pages 33–48. Intellect, 2004.

[Dol] Damien Doligez. Zenon : an automatic theorem prover for first-order logic. http ://focal.inria.fr/download.

Tobias-2 : un outil pour la maîtrise de tests combinatoires

Yves Ledru, Sébastien Ville, Elodie Rose, Lydie du Bousquet, Frédéric Dadeau

Laboratoire d'Informatique de Grenoble
BP 72, 38402 St Martin d'Hères cedex, France
Yves.Ledru@imag.fr

Tobias est un générateur de tests basé sur l'approche combinatoire. Il part du constat qu'une campagne de tests comprend souvent de nombreux tests similaires qui peuvent être générés par un dépliage combinatoire. Il met ainsi à la disposition de l'ingénieur de test un outil qui lui permet d'exprimer ses tests de façon concise, de déplier cette description en un ensemble de tests abstraits, puis de concrétiser ces tests pour une technologie cible (par exemple Java/Junit).

Une première version de l'outil a été développée au cours du projet RNTL COTE (2000-02). Elle a donné lieu à plusieurs applications couronnées de succès notamment sur une mini-application bancaire pour la société Gemplus [2], et sur le test d'un moteur d'interface multi-modale [3]. D'autres expérimentations ont montré que les suites de test générées avec Tobias étaient généralement plus complètes et avaient un meilleur pouvoir de détection d'erreurs que des suites de tests produites manuellement [6].

Cependant, si l'approche combinatoire permet de trouver plus d'erreurs, elle souffre intrinsèquement de l'explosion combinatoire du nombre de tests. Des solutions ont été proposées pour maîtriser l'explosion combinatoire : filtrage des tests générés à partir de prédicats [5], génération combinatoire par paires [1], réduction de suites de test sur base d'une mesure de couverture [4]. Parmi celles-ci, seule une partie du filtrage a été mise en œuvre dans la version originale de Tobias.

Tobias-2 est un redéveloppement de l'outil Tobias. Il conserve les points forts de l'outil, notamment l'approche combinatoire et la génération de tests abstraits qui peuvent être concrétisés en fonction du contexte technologique. Ce nouveau développement ajoute les objectifs suivants :

- la définition d'une architecture où les solutions à la maîtrise de l'explosion combinatoire prennent la forme de “plug-ins” de filtrage et de sélection des cas de test.
- l'implantation d'algorithmes à faible consommation de mémoire pour pouvoir générer jusqu'à un million de cas de tests.
- des langages d'entrée et de sortie plus expressifs, qui d'une part simplifient et unifient les concepts de l'outil original, et d'autre part permettent un meilleur support des constructions des langages à objets.

Le cœur de l'outil est écrit en Java et manipule des fichiers XML en entrée et sortie. Ce moteur de dépliage est intégré dans l'environnement Eclipse qui lui offre une interface plus agréable et automatise l'enchaînement des actions (Fig. 1). L'architecture du moteur permet également de l'intégrer comme un service web, ou de lui fournir une interface utilisateur dédiée.

La démonstration proposée à AFADL comprendra une présentation des principaux concepts de l'outil, et une illustration de divers plug-ins de filtrage et de sélection.

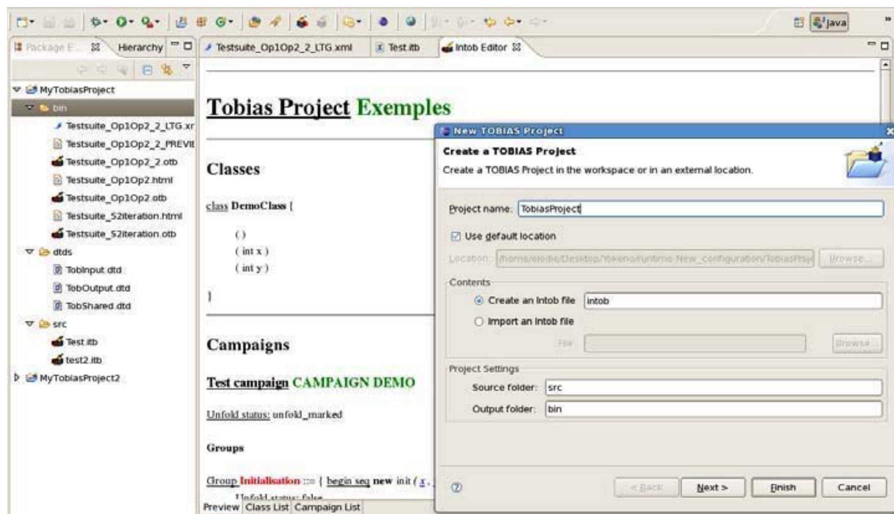


Fig. 1. La perspective TOBIAS d'Eclipse

Remerciements Ce travail a été financé en partie par le projet RNTL COTE, et le projet RNTL POSE (ANR-05-RNTL-01001).

Bibliographie

1. David M. Cohen, Siddhartha R. Dalal, Jesse Parelius, and Gardner C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 13(5):83–88, 1996.
2. Lydie du Bousquet, Yves Ledru, Oliver Maury, Catherine Oriat, and Jean-Louis Lanet. Case study in JML-based software validation. In *ASE 2004*. IEEE CS Press, 2004.
3. Sophie Dupuy-Chessa, Lydie du Bousquet, Jullien Bouchet, and Yves Ledru. Test of the ICARE platform fusion mechanism. In *DSVIS'05*, volume 3941 of *LNCS*. Springer, 2006.
4. Mary Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, 1993.
5. Yves Ledru, Lydie du Bousquet, Olivier Maury, and Pierre Bontron. Filtering Tobias combinatorial test suites. In *FASE 2004*, volume 2984 of *LNCS*. Springer, 2004.
6. Olivier Maury, Yves Ledru, Pierre Bontron, and Lydie du Bousquet. Using TOBIAS for the automatic generation of VDM test cases. In *3rd VDM Workshop (in conjunction with FME'02)*, 2002.

B2EXPRESS : Un animateur de modèles B événementiels

Idir Aït-Sadoune

LISI / ENSMA - Téléport 2 - 1, avenue Clément Ader - B.P. 40109.
86960 Futuroscope Cedex - France.
idir.aitsadoune@ensma.fr

Principe. L'outil B2EXPRESS est un animateur de modèles B événementiel [1] fondé sur l'instanciation de modèles de données exprimés dans le langage EXPRESS [2]. Chaque construction de modèle B (substitutions, invariant, ensembles, variables, etc) est transformée en une construction de modèle EXPRESS (entité, attribut, regèles locales et globales) [3]. L'animation consiste à définir des instances des différentes entités du modèle EXPRESS obtenu et de contrôler les contraintes associées.

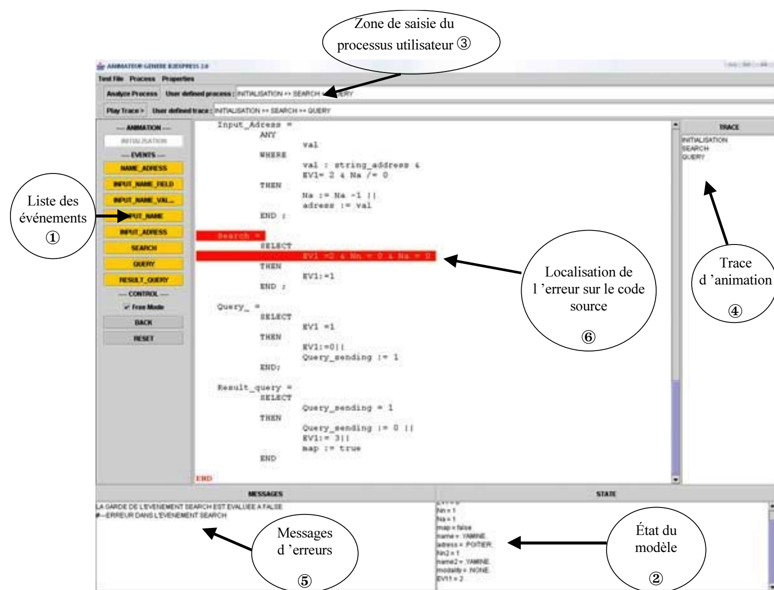


Fig. 1. l'interface de B2EXPRESS

Présentation. Les événements de B sont déclenchés sur l'interface graphique (figure 1) de l'animateur par des clics souris. ECCO Toolkit [4] vérifie la conformité des instances par rapport au modèle EXPRESS issu de la transforma-

tion et renvoie d'éventuels messages d'erreur. Ces messages sont capturés par B2EXPRESS qui, à son tour, les interprète en termes d'éléments du modèle B événementiel. Ainsi sur la figure 1 apparaissent, en ① les événements, en ② les variables et leurs valeurs courantes et en ④ la trace des événements déclenchés. En cas de blocage au niveau des événements ou de détection d'un invariant non satisfait, l'outil affiche un message d'erreur ⑤ et localise l'erreur sur le code source du modèle B en entrée ⑥.

Deux modes d'animation sont offerts par B2EXPRESS.

- *Le mode gardé.* Seuls les événements dont la garde est correcte peuvent être déclenchés. Ils apparaissent en vert sur l'interface, les autres sont en rouge.
- *Le mode libre.* Il laisse le choix à l'utilisateur de déclencher n'importe quel événement, même si sa garde est fausse. Tous les événements apparaissent alors en couleur orange.

En combinaison avec ces deux modes, B2EXPRESS offre la possibilité de décrire par une expression d'une algèbre de processus, à la CCS, les traces désirées ③. Cela permet au développeur de valider des exigences exprimées par des comportements. Il offre également la possibilité de choisir un contrôle de tous les éléments du modèle (invariant, assertions, gardes des événements et expression de trace) ou bien d'inhiber le contrôle de certains de ces éléments.

Enfin B2EXPRESS est également en mesure d'exécuter des séquences de test, générées par d'autres outils, sous forme d'instances du modèle EXPRESS qui correspond au modèle B. Cette démarche a été suivie avec l'outil Casting [5].

Notons que pour une substitution ANY, une boite de dialogue est ouverte pour demander à l'utilisateur de donner les valeurs des variables locales de cet événement. Nous n'avons pas suivi l'approche qui consiste à fixer un nombre fini de valeurs aux ensembles manipulés dans le modèle B.

Des animations vidéos décrivant le fonctionnement de B2EXPRESS peuvent être visualisées sur le lien http ://www.lisi.ensma.fr/members/idir/index.html.

Références

1. Abrial, J. : The B Book. Assigning Programs to Meanings. Cambridge University Press (1996)
2. IS010303.02 : Product data representation and exchange - part 2 : Express reference manual. ISO (055) (1994)
3. Aït-Ameur, Y., Aït-Sadoune, I., Mota, J.M., Van-Aertryck, L. : Validation d'IHM3 par animation de modèles B : LOT 4. Technical report, Projet RNRT Verbatim, LISI / ENSMA and SILICOMP / AQL (2006)
4. Staub, G., Maier, M. : ECCO Tool Kit - an environnement for the evaluation of express models and the development of step based it applications. User Manual (1997)
5. Van-Aertryck, L. : Une méthode et un outil pour l'aide à la génération de jeux de tests de logiciels. PhD thesis, Thèse de l'Université de Rennes I (1998)

Application de critères structurels en présence d'appels de fonctions pour la sélection et génération de tests

Patricia Mouy[*,1], Nicky Williams[1], Pascale Le Gall[2]

[1] CEA/Saclay LIST/LSL
{patricia.mouy,nicky.williams}@cea.fr
[2] Laboratoire IBISC, Université d'Évry Val d'Essonne
pascale.legall@ibisc.univ-evry.fr

Les méthodes de test structurel sont largement reconnues pour leur rigueur. Dans la classe des critères structurels, le critère tous-les-chemins est le critère le plus rigoureux mais il risque l'explosion combinatoire du nombre de chemins. Les techniques de test structurel sont peu appliquées en pratique à cause de la complexité du passage à l'échelle pour traiter des fonctions réalistes complexes (nombre de chemins), de la difficulté de construire des jeux de test satisfaisant les critères considérés (problème des chemins infaisables, présence de boucles, appels de fonction, ...) et de la mise en œuvre des tests (bouchons couramment utilisés de façon ad hoc mais pas toujours justifiés). Nous présentons ici une méthode et un prototype adressant ce problème pour les méthodes de test dynamiques structurelles.

L'origine de ces travaux est de trouver une gestion efficace des appels de fonctions pour la méthode de test PathCrawler[WMMR05], [Mou04]. Celle-ci est une méthode automatique de génération de cas de test structurels pour le langage C comme de la méthode InKa[GBR00]. Le code source sous test est instrumenté pour récupérer les traces d'exécution à chaque cas de test (méthode dynamique). Le prédicat de chemin calculé à partir de ces traces permet de déterminer les données du prochain cas de test (méthode adaptative) ce qui permet de faciliter la détection des chemins infaisables [WMMR05]. Cette stratégie de sélection des cas de test revient à parcourir le CFG de la fonction sous test selon une exploration en profondeur d'abord : le critère appliqué est le critère tous-les-

* Travaux de thèse en informatique de l'Université d'Évry, contrat de thèse CEA (LIST/LSL Laboratoire de Sûreté de Logiciels) avec le soutien de la Région Île-de-France et de TNI-Valiosys

chemins ou des k-chemins[3]. La stratégie de résolution de contraintes utilise la programmation logique avec contraintes (PLC) et l'environnement PLC d'Eclipse [WNJ97] et est déjà utilisée dans [MA00] et [GDGM01].

Dans notre approche de gestion des appels de fonction, nous partons de l'hypothèse que les spécifications des fonctions appelées fournies par l'utilisateur ont été validées lors d'un test unitaire précédent. Elles sont exprimées dans un langage de spécification correspondant à un langage du premier ordre sur domaines finis. Nous avons choisi d'exprimer les spécifications des fonctions sous forme de couples de precondition/postcondition. La spécification d'une fonction est un ensemble fini de couples pre/post composés chacun d'un ensemble de contraintes sur les entrées caractérisant un sous-domaine en entrée de la fonction (désigné comme la precondition) et d'un ensemble de contraintes sur les entrées et sorties (désigné comme la postcondition) caractérisant la relation entrées/sorties attendue pour le sous-domaine associé. Les spécifications fournies doivent être complètes sur le domaine de définition de la fonction, déterministes et composées de couples pre/post à domaines exclusifs. Nous construisons un graphe connexe orienté et étiqueté à unique entrée et sortie pour chaque fonction appelée en s'appuyant sur la spécification associée. Nous appelons cette représentation d'une fonction appelée le graphe abstrait de la fonction. Le graphe abstrait est étiqueté des contraintes sur les entrées des couples pre/post de la spécification et de prédicats existentielles sur les sorties de la fonction vérifiant la spécification. Lors de la construction du graphe de contrôle de la fonction sous test, les instructions d'appel de la fonction sont modélisées par le graphe abstrait associé. Nous obtenons ainsi le graphe mixte de la fonction sous test composé d'informations structurelles de la fonction sous test et fonctionnelles des fonctions appelées.

L'application de la stratégie de sélection de PathCrawler aux graphes mixtes des fonctions sous test avec appels nous a amené à définir deux nouveaux critères de test. Le premier critère correspond à une transposition directe du critère tous-les-chemins dans le cadre

[3] Critère tous-les-chemins restreint aux chemins contenant maximum k itérations par structure répétitive

des graphes mixtes. Le second critère permet de supprimer les cas de test structurels redondants pour la fonction sous test c'est-à-dire les cas de test consistant à couvrir plusieurs fois le même chemin structurel de la fonction sous test.

Un prototype est en cours de finalisation. Des exemples académiques ont montré l'efficacité de la méthode et une validation sur un exemple plus industriel également. Nous gagnons en combinatoire des chemins par rapport à une méthode "inlining" selon l'hypothèse raisonnable qu'il existe au moins un domaine fonctionnel auquel correspond deux ou plus chemins structurels de l'implantation de la fonction appelée. Contrairement à l'utilisation de bouchons fonctionnels, nous maintenons la couverture de la fonction sous test.

Les expérimentations ont montré des résultats concluants. Nous envisageons de valider la méthode sur d'autres exemples de fonctions issus de contextes industriels et d'étendre le sous-ensemble du langage C considéré. Nous envisageons également la mise en place d'une méthode de test en contexte lors de la soumission du graphe mixte à notre premier critère. Cela nous permettrait de lever une de nos hypothèses de base à savoir des fonctions appelées déjà testées et validées unitairement. Notre objectif final est un passage à l'échelle au monde industriel : nous espérons, en effet, que notre travail facilitera la mise en oeuvre pratique des techniques prometteuses d'automatisation de test encore trop peu employées à ce jour.

Références

[GBR00] A. Gotlieb, B. Botella, and M. Rueher. A CLP framework for computing structural test data. *Lecture Notes in Computer Science*, 1861 :399–413, July 2000.

[GDGM01] S. Gouraud, A. Denise, M. Gaudel, and B. Marre. A new way of automating statistical testing methods. 2001.

[MA00] B. Marre and A. Arnould. Test sequences generation from Lustre descriptions : GATeL. In *Proc. ASE 2000*, pages pp 229–237, Grenoble, September 2000. IEEE Computer Society Press.

[Mou04] P. Mouy. Vers une méthode de génération de test boîte grise "à la volée". In *Proc. AFADL 2004*, Besançon, France, juin 2004.

[WMMR05] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler : Automatic generation of path tests by combining static and dynamic analysis. In *Proc. EDCC 2005*, pages 281–292, Budapest, Hungary, April 2005.

[WNJ97] M. Wallace, S. Novello, and J.Schimpf. *ECLiPSe : A platform for Constraint Logic Programming.* IC-Parc, Imperial College, London, August 1997.

Utilisation des outils IBM Rational pour le développement orienté modèle

Éric Cattoir

Consultant Rational, IBM Belgique

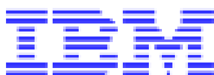
Résumé

De plus en plus on aperçoit une pression sur la réactivité des équipes de développement. Une façon d'améliorer la productivité est d'évoluer vers une approche d'automatisation au travers du développement orienté modèle.

L'objectif de cette session est de vous donner une idée sur les possibilités de ces méthodologies au travers d'un exemple pratique. La session suppose une connaissance de base en UML et en programmation Java.

IBM Rational Software Architect

IBM Rational Software Architect est un outil intégré de conception et de développement. Il s'appuie sur le développement par modèle en langage UML pour créer des applications et des services bien structurés et il contribue également à la compréhension et à l'évolution des applications au sein des équipes.



Index des auteurs

Abrial, J.-R. 15
Ait-Ameur, Y. 21, 295
Ait-Sadoune, I. 295
Ausbourg, B. d' 21
Barais, O. 109, 283
Bendisposto, J. 199
Bosc, J.-M. 57
Bousquet, L. du 293
Brat, G. 11
Carlier, M. 291
Castel, C. 57
Cattoir, E. 301
Chapoutot, A. 261
Colin, S. 91
Cortier, A. 21
Dadeau, F. 161, 293
Dallons, G. 281
Darfeuil, P. 57
Déharbe, D. 287
Dubois, C. 291
Duchien, L. 109
Dutuit, Y. 57
Faella, M. 75
Fejoz, L. 213
Focone, E. 57
Fontaine, P. 287
Gobert, F. 245
Groz, R. 39
Haddad, A. 161
Humbert, S. 57
Jaume, M. 227
Julliand, J. 289
Jézéquel, J.-M. 283
Lanoix, A. 91
Laurent, O. 13
Le Charlier, B. 245
Ledru, Y. 293
Le Gall, P. 297
Legay, A. 75
Leuschel, M. 199
Ligot, O. 199
Li, K. 39
Madani, L. 125
Martel, M. 261
Massonet, P. 281
Merz, S. 213
Molderez, J.-F. 281
Morisset, C. 227
Mottu, J.-M. 283
Mountassir, H. 289

Moutet, T. 161

Mouy, P. 297

Oliveira, D. Caminha B. de 287

Oudot, E. 289

Parissis, I. 125, 143

Plouzeau, N. 109

Ponsard, C. 281

Rose, E. 293

Saudrais, S. 109

Seguin, C. 57

Seljimi, B. 143

Shahbaz, M. Muzammil ... 39

Skipper, M. 283

Souquières, J. 91

Ville, S. 293

Vojtisek, D. 283

Williams, N. 297

Wolper, P. 17

Zimmermann, Y. 181