Actes des 16èmes journées

AFADL

Approches Formelles dans l'Assistance au Développement de Logiciels

Les 14 et 15 Juin 2017 à Montpellier

Edités par Akram Idani et Nikolai Kosmatov

Avant-propos

⋆ ⋆ ⋆

Depuis 1997, année de son lancement, l'atelier AFADL a toujours été un lieu d'échanges fructueux autour des méthodes formelles pour le développement des logiciels. Plusieurs communautés se retrouvent à l'occasion de cet événement francophone, couvrant ainsi un large spectre du monde rigoureux des méthodes formelles : preuve, test, analyse de code, raffinement... Aujourd'hui les techniques, outils et démarches de raisonnement formel ont atteint un niveau de maturité élevé, permettant leur applicabilité tout au long du cycle de développement de logiciels. Ces actes en présentent diverses facettes au travers de travaux allant de la spécification d'un cahier des charges aux activités de V&V et de production de code sûr.

AFADL est un lieu d'échanges qui se veut ouvert et accessible grâce aux différentes catégories de soumissions acceptées : articles courts, démonstrations d'outils, résumés longs, présentations de projets et contributions de doctorants. Cette année encore l'atelier AFADL est co-localisé avec les journées du GDR-GPL (Génie de la Programmation et du Logiciel) et la conférence CIEL. Le programme de l'édition 2017 est très riche avec un ensemble de vingt-trois présentations faisant la jonction entre le cœur d'AFADL et les groupes de travail du GDR-GPL :

— IE : Ingénierie des Exigences
— LTP : Langages, Types et Preuves
— MFDL : Méthodes Formelles dans le Développement de Logiciels
— MTV2 : Méthodes de Test pour la Validation et la Vérification

L'implication des coordinateurs de ces groupes ainsi que des membres du comité de programme a été un gage de qualité de ces actes. Nous les remercions d'avoir minutieusement examiné les diverses contributions. Grâce à leurs efforts et à la participation des auteurs, les communautés francophones des méthodes formelles continuent, pour la 16ème édition, à se retrouver autour de cet atelier. Enfin, un grand merci aux organisateurs locaux qui ont beaucoup travaillé pour que cet événement soit un succès.

⋆ ⋆ ⋆

Le 23 Mai 2017

Akram IDANI et Nikolai KOSMATOV

Comité de programme

Présidents du comité de programme

Akram IDANI	LIG / Grenoble INP, Grenoble
Nikolai KOSMATOV	CEA, Saclay

Membres du comité de programme

Yamine Ait Ameur	IRIT / INPT, Toulouse
Sandrine Blazy	IRISA / Univ. Rennes 1, Rennes
Frédéric Boniol	ONERA, Toulouse
Pierre Casteran	LaBRI, Bordeaux
Frédéric Dadeau	FEMTO-ST, Besançon
David Deharbe	ClearSy System Engineering, Aix-en-Provence
Lydie Du Bousquet	LIG / UGA, Grenoble
Catherine Dubois	ENSIIE / CEDRIC, Evry
Christele Faure	Saferiver, Paris
Aurélie Hurault	IRIT / INPT, Toulouse
Akram Idani	LIG / Grenoble INP, Grenoble
Jacques Julliand	FEMTO-ST, Besançon
Florent Kirchner	CEA, Saclay
Nikolai Kosmatov	CEA, Saclay
Regine Laleau	LACL / Univ. Paris-Est, Créteil
Jean-Louis Lanet	LHS / INRIA, Rennes
Arnaud Lanoix	Univ. de Nantes, Nantes
Pascale Le Gall	Centrale-Supelec, Paris
Yves Ledru	LIG / UGA, Grenoble
Nicole Levy	CNAM, Paris
Delphine Longuet	LRI / Univ. Paris-Sud, Paris
Ioannis Parissis	LCIS / Grenoble INP, Valence
Pascal Poizat	LIP6 / Univ. Paris Ouest, Paris
Marie-Laure Potet	VERIMAG / Grenoble INP, Grenoble
Marc Pouzet	LIENS, Paris
Antoine Rollet	LaBRI / Univ. de Bordeaux, Bordeaux
Vlad Rusu	INRIA, Lille
Nicolas Stouls	INRIA / INSA Lyon, Lyon
Safouan Taha	Centrale-Supelec, Paris
Sylvie Vignes	Télécom ParisTech, Paris
Laurent Voisin	Systerel
Virginie Wiels	ONERA / DTIM, Toulouse
Fatiha Zaidi	LRI / Univ. Paris-Sud, Paris

Table des matières

⋆ ⋆ ⋆ AFADL 2017 ⋆ ⋆ ⋆

Articles courts

Contextualisation et dépendance en Event-B*

Souad Kherroubi et Dominique Méry

Université de Lorraine - LORIA, Campus scientifique - BP 239 - 54506
Vandœuvre-lès-Nancy, France.

Résumé

Dans ce document, nous introduisons, d'une part une nouvelle définition de ce qu'est un contexte dans le cadre du développement en Event-B, et d'autre part une relation entre deux modèles Event-B appelée dépendance. La contextualisation des modèles Event-B s'appuie en partie sur des connaissances issues du domaine et sur une décomposition de ces connaissances en contraintes, hypothèses ou dépendances selon leur véracité dans la preuve. La dépendance entre deux modèles permet de structurer les développements de modèles de système, en organisant des phases identifiées dans le processus analysé. Ces idées sont illustrées sur un petit exemple de comptabilisation et ont été validées sur un développement de patrons de conception pour les protocoles de vote.

1 Introduction

La conception de systèmes s'appuie sur différentes méthodes et techniques informelles, semi-formelles et formelles. Les techniques fondées sur l'abstraction [9] facilitent cette conception, en donnant des vues plus ou moins précises de ces systèmes. L'acquisition pertinente des connaissances du domaine d'étude permet une meilleure compréhension et communication du problème à traiter. En effet, un modèle conceptuel doit intégrer l'intention du concepteur et donner une vue claire correcte et complète par une sémantique non ambigüe. Le contextualisme met précisément l'accent sur les détails de chaque application particulière, ainsi que sur le processus de modélisation lui-même. Ces détails définissent un contexte et constituent l'identité unique de chaque tâche de modélisation. La contextualisation est un mécanisme d'abstraction qui offre une séparation entre les données collectées. Elle permet de résoudre les problèmes liés aux différences de perception entre les différents acteurs dans le système favorisant ainsi l'organisation et la rationalisation des perspectives d'une même réalité.

*This work was supported by grant ANR-13-INSE-0001 (The IMPEX Project `http://impex.gforge.inria.fr`) from the Agence Nationale de la Recherche (ANR)

L'intérêt du contexte dans la vérification formelle des systèmes est manifeste : alors que l'utilisation du contexte dans les phases de développement en amont permet de situer dans l'espace et dans le temps les systèmes développés [10], son utilisation durant les phases en aval du processus de développement améliore la validation des systèmes produits [4]. Une telle validation se détermine par la mesure de la relation entre la représentation du système réel et un modèle confronté à des cas d'utilisations (configurations et scénarios) prévues du modèle.

Dans cette optique, la logique et les ontologies sont les deux grands axes de recherche où le contexte prend une grande place. Les travaux de Barlatier [3] montrent que cette notion n'est pas absolue et est toujours définie par rapport à un focus, une activité ou un concept intentionnel i.e. une action, ... Les contextes sont perçus comme des objets formels construits de façon incrémentale à partir de contextes. Ce principe est appelé *context lifting* (cf. McCarthy [7]). Les *situations* ou *états* apparaissent comme un paramètre dans les prédicats. Ceux-ci deviennent donc *dépendants* d'une situation.

Au niveau de la preuve, le contexte permet d'établir et de valider des relations de confiance qui fournissent des interprétations valides dans un domaine précis à des fins de certification. Nous nous intéressons à la contextualisation et au contexte de preuve dans le formalisme Event-B [1]. La méthode Event-B est un cadre de structuration qui spécifie, au travers de son langage, une description formelle des systèmes *réactifs*. Les spécifications des comportements dans ce formalisme suivent une approche assertionnelle décrite au moyen de *contextes* Event-B et de *machines* Event-B. Plutôt que de considérer les contextes comme des objets formels à la McCarthy, nous adoptons l'approche de Barlatier [3] qui considère le contexte comme connaissance minimale faisant partie d'un système centré sur la preuve, en assimilant les modèles Event-B à des concepts au sens ontologique. Le contexte se trouve alors comme partie intégrante de la preuve et contraint sa sémantique. Nous proposons de répartir le contexte en Event-B en *contraintes*, *hypothèses* et *dépendances*, selon que les connaissances sont acquises, admises ou déduites dans un système de preuve. Nous définissons ce nouveau mécanisme de dépendance entre les modèles Event-B, où la propriété de terminaison stable est une condition nécessaire pour son établissement. Une correspondance avec la théorie des situations est établie : une situation correspond à l'état du système en Event-B et les faits établissent une interprétation logique qui définit l'état du système. Ceux-ci représentent les valeurs dans les ensembles (le contenu des ensembles Event-B), les valeurs des constantes, ainsi que les valeurs des variables définies dans les machines. Les contraintes correspondent aux propriétés statiques définies dans les contextes Event-B. Ce mécanisme se définit alors par une combinaison d'états et de propriétés statiques de contextes Event-B. La dépendance est une relation mesurable prenant des valeurs à partir des faits existants dans une situation qui nécessite la définition d'une nouvelle obligation de preuve.

Le document est organisé comme suit. Nous présentons la modélisation Event-B et les concepts liés au contexte. Puis nous expliquons la dépendance de modèles Event-B et donnons un exemple de dépendance. Enfin nous concluons.

2 Modélisation en Event-B

Un modèle formel en Event-B se bâtit à partir de trois principaux ingrédients : un langage de spécification fondé sur la théorie des ensembles et la logique des prédicats ; un système de vérification fondé sur la génération d'un ensemble d'obligations de preuves ; et un mécanisme de raffinement qui permet un développement incrémental correct-par-construction. Un modèle Event-B $\mathcal{M}$ est composé de contextes et de machines. Un contexte $\mathcal{T}h$ spécifie la partie statique d'un modèle et comprend des ensembles supports s, des constantes c, des axiomes et théorèmes $\mathcal{P}$ qui établissent des contraintes et des propriétés des éléments statiques. Une machine décrit la dynamique du système au moyen d'une liste finie de variables x décrivant l'état du système, éventuellement modifiés par une liste d'événements $\{e_0, ..., e_n\}$. Le changement d'état doit préserver des propriétés appelées *invariants* $\mathcal{I}nv$ qui doivent être maintenues à chaque observation des événements dans le système. Un événement est défini par : une condition (garde) sous laquelle l'événement peut être observé et des actions qui définissent l'évolution des valeurs des variables d'état dans le modèle pour l'état suivant.

Dans ce qui suit, nous adoptons la terminologie de Barlatier [3] et nous expliquons la correspondance avec notre cadre de modélisation. La notion centrale dans ses travaux est un concept ontologique dénoté par un type. Dans ses travaux, un *fait* dénote les objets en relation ; une *situation* $\mathcal{S}$ est décrite par une collection de faits $f_i, i \in 1..n$, telle que $\mathcal{S} = \{f_1, ..., f_n\}$. Une situation $\mathcal{S}$ est appelée état dans Event-B, et un état comprend les valeurs variables, ainsi que toutes les valeurs des ensembles et des constantes.

Définition 1 *(Contexte) Etant donné le système de preuve Event-B, le contexte* $\mathcal{C}xt$ *est la connaissance minimale qui affecte le comportement de toute action dans le système satisfaisant les propriétés de sûreté dans une situation donnée.*

Les connaissances minimales se répartissent selon trois catégories explicitées dans la suite : *i)* Le contexte comme contraintes ; *ii)* le contexte comme hypothèses *iii)* et le contexte comme dépendances. En raison du manque d'espace, les deux premières catégories ne sont pas abordées dans le présent document.

Le contexte comme dépendances - Les travaux de McCarthy montrent que les prédicats qui définissent le contexte sont paramétrés par des situations. Alors que le point de vue ontologique démontre qu'un contexte est un "*moment universel*" [1] [5]. Barlatier a établi une correspondance avec la théorie des ensembles, à savoir, les concepts ontologiques (i.e. les types) sont interprétés comme des ensembles d'éléments et les rôles comme des relations entre les éléments de ces différents concepts. Partant de ce constat, nous pouvons exprimer ce type de contexte par une relation de dépendances entre les modèles Event-B qui résultent de la combinaison des situations (des valeurs des états dans le formalisme Event-B) et de contraintes exprimées dans les contextes Event-B, en confondant ce *moment universel* à la termi-

1. un moment est un individu qui dépend existentiellement d'autres individus.

naison qui nécessite une stabilité. Ce type de contexte peut parfaitement être illustré par les systèmes de vote (voir [6]). On notera les dépendances des contraintes sous la forme de $D - AX$ et $D - TM$ pour illustrer les axiomes et les théorèmes dépendants dans les contextes d'un modèle Event-B.

Nous avons donné une ébauche de la contextualisation en Event-B et nous allons décrire la notion de modèles Event-B dépendants afin d'enrichir les techniques de structuration des modèles Event-B.

3 Dépendance de modèles

La dépendance entre deux modèles Event-B $\mathcal{M}_1$ et $\mathcal{M}_2$ se définit ainsi : *i)* les contextes Event-B $\mathcal{M}_1$ et $\mathcal{M}_2$ sont communs ie : font partie du contexte Event-B $\mathcal{T}h(s_2, c_2)$; *ii)* une transformation d'un certain nombre de variables d'un modèle source $\mathcal{M}_1$ en des constantes dans le modèle cible $\mathcal{M}_2$; *iii)* le prédicat caractérisant la terminaison du premier modèle satisfait les contraintes définies dans le contexte Event-B du deuxième composant. Dans ce cas, il n'y a pas de partage de variables, et un composant correspond à un modèle Event-B abstrait et éventuellement un ensemble de modèles qui raffinent ce modèle abstrait, et l'on demande à ce que le niveau de raffinement pour chaque phase soit suffisamment élaboré pour que l'ensemble des constantes dans le contexte Event-B de la phase qui suit la phase en cours trouve son correspondant (variable) dans le dernier raffinement de cette phase dont il dépend. Dans ce qui suit, nous adoptons les notations de Méry [8], qui expriment des modèles relationnels et leurs extensions [2] aux aspects temporels à la TLA.

3.1 Modèles dépendants pour Event-B

Soient deux modèles Event-B $\mathcal{M}_i$, avec $i \in 1..2$ définis par :

$$\mathcal{M}_i \mathrel{\hat{=}} (\mathcal{T}h_i(s_i, c_i), x_i, Val_i, \mathcal{I}nit_i(x_i), \{e_{i,0}, ..., e_{i,n_i}\})$$

— $\mathcal{T}h_i(s_i, c_i)$ est le contexte qui définit les éléments et les propriétés statiques d'un modèle $\mathcal{M}_i$;
— l'espace d'états est l'ensemble des valeurs possibles des variables Val_i ;
— $\mathcal{P}_i(s_i, c_i) \mathrel{\hat{=}} AX(s_i, c_i) \cup TM(s_i, c_i)$ les propriétés statiques ou les *contraintes* du modèle $\mathcal{M}_i$ sur les ensembles s_i et les constantes c_i. Nous noterons $\mathcal{C}_i(s_i, c_i)$ pour exprimer les contraintes dépendantes définies dans un contexte Event-B exprimant les axiomes $D - AX(s_i, c_i)$ et les théorèmes $D - TM(s_i, c_i)$ dépendants ;
— un ensemble d'états initiaux $\mathcal{I}nit_i$;
— un ensemble d'événements $\{e_{i,0}, ..., e_{i,n_i}\}$.

$$\mathcal{S}pec(\mathcal{M}_i) \mathrel{\hat{=}} \mathcal{I}nit_i(x_i) \wedge \Box[NEXT_i]_{x_i} \wedge L_i$$

— avec $NEXT_i \mathrel{\hat{=}} \exists e.e \in \{e_{i,0}, ..., e_{i,n_i}\} \wedge BA(e)(x_i, x_i')$; x_i' est la valeur des variables après l'observation de l'événement e ; $\Box[NEXT_i]$ signifie

que toute paire d'états satisfait la relation $NEXT_i$ et que les valeurs des variables restent les mêmes ou changent ; et L_i est une conjonction de contraintes d'équité faibles et fortes sur des combinaisons d'événements e_i du modèle $\mathcal{M}_i$;

Une trace d'exécution équitable $tfair(\mathcal{M}_i, L_i)$ engendrée par un modèle $\mathcal{M}_i$ est définie par une suite infinie de valeurs :

$$\mathcal{T}race(\mathcal{M}_i) \hat{=} \{\sigma_0\sigma_1\sigma_2...\sigma_j\sigma_{j+1}... \mid \sigma_0 \in \mathcal{I}nit_i \wedge \forall j \in \mathbb{N}.(\sigma_j, \sigma_{j+1}) \in \ NEXT_i\}$$

satisfaisant la spécification $\mathcal{S}pec(\mathcal{M}_i)$ et notée par $\sigma \vDash \mathcal{S}pec(\mathcal{M}_i)$.

Pour définir la propriété de terminaison stable, il faut ajouter à la définition de l'opérateur *leadsto* ou $\rightsquigarrow$ [2] la propriété de stabilité qui suit : il s'agit d'exprimer qu'on arrivera, à partir de P, fatalement à un futur où la propriété Q sera vérifiée et restera vraie sur tous les états de la trace qui suivent.

Définition 2 *(Terminaison stable) Une spécification $\mathcal{S}pec(\mathcal{M}_i)$ d'un modèle $\mathcal{M}_i$ sous les hypothèses d'équité L_i satisfait la propriété de terminaison en Q, si la propriété suivante est vérifiée pour toutes les traces $\sigma \in \mathcal{T}race(\mathcal{M}_i)$*

$$\forall i.(i \geqslant 0 \wedge P(\sigma_i) \Rightarrow \exists j.(j \geqslant i \wedge Q(\sigma_j) \wedge \forall k.(k \geqslant j \wedge Q(\sigma_k))))$$

Q sera le prédicat qui caractérise l'ensemble des états terminaux du modèle $\mathcal{M}_i$ noté $\mathcal{T}_i$ dans la suite. La stabilité implique que le prédicat qui caractérise les états finaux restera vrai pour tous les états suivants. Une preuve de progression, en définissant un variant sur un ordre bien fondé est nécessaire pour montrer la convergence dans le système [2].

Définition 3 *(Dépendance entre deux modèles Event-B)*
Les deux modèles $\mathcal{M}_i$ ($i \in 1..2$) sont dépendants, et on dit que le modèle $\mathcal{M}_2$ dépend contextuellement du modèle $\mathcal{M}_1$ par rapport à une propriété de terminaison $\mathcal{T}_1$ et on notera $Dep(\mathcal{M}_1, \mathcal{T}_1, \mathcal{M}_2, v_1, c_{21})$, si :

1. *$\mathcal{S}pec(\mathcal{M}_1) \vDash \mathcal{I}nit_1 \rightsquigarrow \mathcal{T}_1$ où $\mathcal{T}_1$ est le prédicat caractérisant l'ensemble les états terminaux stables du modèle $\mathcal{M}_1$*
2. *$\mathcal{S}pec(\mathcal{M}_1) \vDash (\forall x_1, x_1'.(\mathcal{T}_1(x_1) \wedge NEXT_1(x_1, x_1') \Rightarrow \mathcal{T}_1(x_1')))$*
3. *le contexte dépendant étend le contexte source i.e. :*
   $\mathcal{T}h_2(s_2, c_2)\ extends\ \mathcal{T}h_1(s_1, c_1)$;
4. *il existe un sous-ensemble non vide v_1 de l'ensemble des variables x_1 ($v_1 \subseteq x_1$) du modèle $\mathcal{M}_1$ tel que $\mathcal{T}h_2(s_2, c_2) \vDash (\mathcal{T}_1(x_1) \wedge v_1 = c_{21} \wedge c_2 = c_{21} \cup c_1 \cup c_{22}) \Rightarrow \mathcal{C}_2(s_2, c_2)$,*

Concrètement, un modèle $\mathcal{M}_2$ dépend d'un autre modèle $\mathcal{M}_1$, si le prédicat caractérisant l'ensemble des états terminaux du modèle $\mathcal{M}_1$ permet de satisfaire la

2. Leads to : Sous les hypothèses d'équité L du modèle $\mathcal{M}$, la spécification du modèle $\mathcal{S}pec(\mathcal{M})$ satisfait la propriété $P \rightsquigarrow Q$, si pour toutes les traces $\sigma \in tfair(\mathcal{M}, L)$, la propriété suivante est vérifiée : $\forall i.(i \geqslant 0 \wedge P(\sigma_i) \Rightarrow \exists j.(j \geqslant i \wedge Q(\sigma_j)))$

"configuration initiale" du modèle $\mathcal{M}_2$ définie par le contenu des ensembles porteurs, ainsi que les valeurs des constantes du contexte $\mathcal{T}h_2(s_2, c_2)$. $\mathcal{T}h_2(s_2, c_2)$ est la structure définissant le contexte Event-B du second modèle dans sa totalité i.e., toutes les propriétés (axiomes et théorèmes) statiques en conjonction qui établissent le typage ainsi que les contraintes dépendantes et indépendantes des situations, alors que $\mathcal{C}_2$ sont les contraintes qui doivent être satisfaites par l'ensemble des états finaux du premier modèle i.e., qui sont dépendantes des situations. Celles-ci s'ajoutent donc dans le contexte du second modèle Event-B en conjonction avec les propriétés indépendantes des situations liées au premier modèle. Cette approche reflète le fait qu'à la stabilisation de la première phase, aucune modification ne peut être faite sur ces éléments en tant que variables, puisque ces dernières dans le premier composant conservent leurs valeurs à la terminaison. On applique alors à l'opérateur définissant le prédicat des contraintes statiques qui prend la liste des ensembles et des constantes définis dans le contexte dépendant les nouvelles valeurs des variables définies dans le premier modèle. La consistance en Event-B est définie par l'ensemble des obligations de preuve. L'obligation de preuve relative au mécanisme de dépendance entre deux modèles $\mathcal{M}_1$ et $\mathcal{M}_2$ qui doit être prouvée est définie comme suit.

Définition 4 *(Obligation de preuve de dépendance entre $\mathcal{M}_1$ et $\mathcal{M}_2$)*

$$\mathcal{T}h_1(s_1, c_1), Inv(s_1, c_1, x_1), v_1 \subseteq x_1, c_{21} \subseteq c_2, \mathcal{T}_1(x_1) \vdash \mathcal{C}_2(s_2, c_2)(v_1/c_{21})$$

Les contextes [3] deviennent en relation partonomique [4] telle que définie par Barlatier où un contexte $\mathcal{C}xt_1$ est une partie d'un autre contexte $\mathcal{C}xt_2$, si $\mathcal{C}xt_1$ contient au moins toutes les contraintes vérifiées dans $\mathcal{C}xt_2$. Le second modèle devient dépendant du premier.

Exemple - L'exemple donné ici concerne la gestion de stocks par achats et ventes d'articles, et une comptabilisation des recettes et des dépenses en différé. Cette modélisation est une représentation simplifiée et ne décrit pas tous les détails de gestion dans un système ERP. Le premier modèle $\mathcal{M}\_av$ est décrit par un contexte *cxt_achat_vente* et une machine *machine_achat_vente*. Le système gère les achats (*acheter*) et les ventes (*vendre*) d'articles. Les ventes s'effectuent selon les prix fixés dans le contexte, alors que les achats se font à des prix fixés par le marché. L'imputation des écritures comptabilisées en différé permet de ne comptabiliser les opérations qu'au moment de l'exécution des opérations de transfert. Le journal des écritures comptables pour les opérations de ventes et d'achats est configuré en différé selon une période de clôture notée *periode_diff*. Les achats et les ventes peuvent se réaliser durant cette période (*grd3*). Cette période est décrue par l'événement convergent *avancer_temps* qui décroît l'expression du variant définie dans cette première machine, et est soumis à une hypothèse d'équité faible. L'impression des différentes opérations de ventes et d'achats se fait à travers les deux

3. On parlera indifféremment de contextes ou modèles Event-B

4. Les relations partonomiques ou Partie-Tout sont issues des méréologies. Dans ses travaux, Barlatier ne s'intéresse qu'aux relations partonomiques qui s'appuient sur les dépendances

variables *recettes* et *depenses* respectivement. La *clôture* d'une période d'opérations implique que la comptabilisation des recettes et des dépenses peut *commencer* lorsque la valeur de la variable *periode* vaudra *periode_diff* : i.e., à la terminaison stable du premier composant, exprimant la relation de dépendance entre les deux modèles.

CONTEXT cxt_achat_vente
sets $ARTICLES$
constants $periode\_diff, prix\_art$
axioms
$periode\_diff \in \mathbb{N}1$
$prix\_art \in ARTICLES \rightarrow \mathbb{N}1$...
MACHINE machine_achat_vente
sees cxt_achat_vente
invariants
$inv1 : periode \in 0 \mathinner{..} periode\_diff$
$inv2 : art\_vendus \subseteq ARTICLES$
$inv3 : recettes \in art\_vendus \rightarrow \mathbb{N}$
$inv4 : art\_achetes \subseteq ARTICLES$
$inv5 : depenses \in art\_achetes \rightarrow \mathbb{N}$
$inv6 : art\_achetes \cap art\_vendus = \varnothing$
$inv7 : \forall art, p \cdot (art \mapsto p \in recettes \Rightarrow art \mapsto p \in prix\_art)$
variant $periode\_diff - periode$
events avancer_temps $\hat{=}$ **convergent when**
$grd1 : periode \in 0 \mathinner{..} (periode\_diff - 1)$ **then**
$act1 : periode := periode + 1$ **end**

acheter $\hat{=}$ **any** art, p **where**
$grd1 : art \in ARTICLES \wedge p \in \mathbb{N}_1$
$grd2 : art \notin art\_achetes \wedge art \notin art\_vendus$
$grd3 : periode \in 0 \mathinner{..} (periode\_diff - 1)$ **then**
$act1 : art\_achetes := art\_achetes \cup \{art\}$
$act2 : depenses(art) := p$ **end**
vendre $\hat{=}$ **any** art **where**
$grd1 : art \in ARTICLES$
$grd2 : art \notin art\_vendus \wedge art \notin art\_achetes$
$grd3 : periode \in 0 \mathinner{..} (periode\_diff - 1)$ **then**
$act1 : art\_vendus := art\_vendus \cup \{art\}$
$act2 : recettes(art) := prix\_art(art)$ **end** ... **end**
MACHINE machine_compt /*machine dépendante*/
sees cxt_compt /*contexte dépendant*/
invariants
$inv1 : solde \in \mathbb{Z} \wedge comptabilise \in BOOL$
$inv2 : periode = periode\_diff \wedge comptabilise = FALSE \Rightarrow solde = 0$
events comptabiliser $\hat{=}$ **where**
$grd : comptabilise = FALSE$ **then**
$act1 : solde := total(recettes) - total(depenses)$
$act2 : comptabilise := TRUE$ **end**...

La comptabilisation est décrite par le modèle $\mathcal{M}\_compt$ défini par la machine *machine_compt* qui voit le contexte *cxt_compt* dépendant de la machine *machine_achat_vente*. Ce contexte étend le premier contexte *cxt_achat_vente* et contient toutes les constantes définies comme variables dans cette dernière machine (*machine_achat_vente*) avec la valeur de la constante *periode* qui vaut *periode_diff* dans le contexte dépendant. On note la dépendance entre les deux modèles par : *Dep*($\mathcal{M}\_av$, *periode=periode_diff*, $\mathcal{M}\_compt$, v_1, c_{21}), avec : v_1 sont toutes les variables définies par les invariants $inv1$,...,$inv7$ dans la première machine ; c_{21} correspond à ces mêmes éléments mais définies comme étant des constantes dans le contexte dépendant *cxt_compt*, auxquelles nous ajoutons la contrainte suivante : $axm11 : periode = periode\_diff \Rightarrow (art\_achetes \neq \varnothing \vee art\_vendus \neq \varnothing)$. La contrainte dépendante dans cet exemple est $periode = periode\_diff \wedge axm11$. La comptabilisation des opérations consiste à calculer à travers la variable $solde$, initialisée à 0, la différence entre les recettes et les dépenses via la constante *total*.

4 Conclusion

Les relations partonomiques entre les contextes dans les travaux de Barlatier expriment un changement de structures ce qui correspond à un changement de modèles dans notre cas. Nous avons appliqué ce mécanisme aux protocoles de vote que nous avons développés séparément [6]. La spécification du fonctionnement des protocoles de vote passe par la modélisation de plusieurs composants correspondant aux différentes phases citées dans ce qui précède, chacun de ces composants

vérifiant un certain nombre de propriétés de sûreté. Dans ce cadre, nous définissons un système de vote comme des compositions de modèles Event-B via le mécanisme de dépendance. Les modèles dans notre développement sont décrits au moyen de contextes et de machines B liées par raffinement décrivant les contraintes et la dynamique du système.

Références

[1] Jean-Raymond Abrial. *Modeling in Event-B : System and Software Engineering*. Cambridge University Press, 2010.

[2] Manamiary Bruno Andriamiarina. *Développement d'algorithmes répartis corrects par construction*. Thèse, Université de Lorraine ; Loria & Inria Grand Est, October 2015.

[3] Richard Dapoigny and Patrick Barlatier. Modeling contexts with dependent types. *Fundam. Inform.*, 104(4) :293–327, 2010.

[4] Philippe Dhaussy and Frédéric Boniol. Mise en œuvre de composants MDA pour la validation formelle de modèles de systèmes d'information embarqués. *Ingénierie des Systèmes d'Information (Lavoisier)*, 12(5) :133–157, 2007.

[5] P. Dockhorn Costa, J.P. Andrade Almeida, L. Ferreira Pires, G. Guizzardi, and M.J. van Sinderen. Towards conceptual foundations for context-aware applications. In T.R. Roth-Berghofer, S. Schulz, and D.B. Leake, editors, *AAAI Workshop on Modeling and Retrieval of Context 2006*, volume WS-06- of *AAAI Technical Report*, pages 54–58, Menlo Park, CA, USA, 2006. AAAI Press.

[6] J. Paul Gibson, Souad Kherroubi, and Dominique Méry. Applying a Dependency Mechanism for Voting Protocol Models Using Event-B. In *FORTE 2017 - 37th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, LNCS. Springer, 2017. to appear.

[7] John McCarthy. Notes on formalizing context. In *Proceedings of the 13th International Joint Conference on Artifical Intelligence - Volume 1*, IJCAI'93, pages 555–560, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[8] Dominique Méry. Playing with state-based models for designing better algorithms. *Future Generation Comp. Syst.*, 68 :445–455, 2017.

[9] J. Mylopoulos. Information modeling in the time of the revolution. *Information systems*, 23(3) :127–155, 1998.

[10] A. Sutcliffe, S. Fickas, and M.M. Sohlberg. Pc-re : a method for personal and contextual requirements engineering with some experience. *Requirements Engineering*, 11(3) :157–173, 2006.

Du cahier des charges à sa spécification

Imen Sayar et Jeanine Souquières

LORIA – CNRS UMR 7503 – Université de Lorraine
Campus Scientifique, BP 239
F-54506 Vandœuvre lès Nancy cedex

Firstname.Lastname@loria.fr

Résumé. Le cahier des charges est un document de référence tout au long du développement d'un système. Il s'agit tout d'abord de le comprendre et de le structurer. Les liens entre ce document et la spécification en Event-B définie en termes de ses différents raffinements nous amène à utiliser des outils de vérification et de validation. Nous présentons quelques leçons sur le développement de la machine d'hémodialyse, avec l'introduction de patrons.

Mots clés. Cahier des charges, développement, spécification, raffinement, patron, validation, vérification, Event-B, outils.

1 Introduction

La compréhension des besoins est indispensable pour démarrer un processus formel de développement de logiciels. La qualité du cahier des charges, appelé *CdC* dans la suite du papier, affecte celle du logiciel obtenu. Ce document est souvent peu utilisable et difficile à utiliser par les parties prenantes tout au long du processus. Les rapports du Standish Group dont le premier date de 1994 [1], montrent qu'une mauvaise qualité des exigences entraîne des difficultés dans le développement et mène à des échecs et à des coûts importants en temps et en argent. Un soin accordé au *CdC* aide à réduire l'écart entre ce document et sa spécification formelle. Il améliore la qualité du logiciel final.

Notre approche commence par une étape de re-structuration des besoins. Celle-ci a pour objectif l'obtention d'un document lisible. Nous travaillons sur le *CdC* du client et le ré-écrivons sous forme de phrases courtes étiquetées en utilisant les recommandations d'Abrial [13] et l'outil ProR [8]. Différentes sortes de diagrammes sont disponibles pour aider à la compréhension des besoins [11]. La notion de patron avec ses paramètres est introduite au niveau du développement. Elle permet de décrire un sous-problème identifié et sa solution en réutilisant des connaissances acquises par l'expérience [7].

L'activité de vérification a pour objectif d'assurer la correction du modèle formel développé. La validation sert à montrer que le modèle satisfait les besoins du client. Dans notre approche, ces activités aident à améliorer la qualité du *CdC* et sa spécification formelle en Event-B [4]. Nous prenons en compte la validation en tant que processus rigoureux préparé dès le traitement des besoins et tout au long du développement de la spécification formelle [12]. Nous introduisons des termes formels dans le *CdC* structuré avec ProR, plug-in de la plateforme Rodin [1], [5].

La section 2 décrit rapidement notre approche et les outils utilisés. Un exemple de patron de développement est présenté dans la section 3. La section 4 présente quelques leçons retirées de plusieurs études de cas [2] avec utilisation du patron présenté dans la section précédente. Nous les illustrons par un système d'hémodialyse [10]. La section 5 conclut et décrit la suite de ce travail.

2 Notre approche

Un système informatique, voir figure 1, est composé de :

1. Rapport CHAOS du Standish Group (https ://www.standishgroup.com)
2. `http://dedale.loria.fr`

- son *CdC*, décrit à l'aide de l'outil ProR,
- sa spécification formelle, décrite en Event-B et
- les liens entre ce *CdC* et sa spécification formelle.

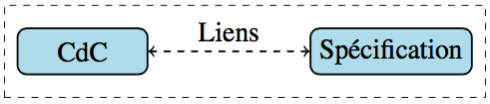


FIGURE 1 – Etat du système

L'ensemble *CdC/Spécification* évolue en permanence. Il est mis à jour grâce aux liens entre le *CdC* et la *Spécification* qui sont affinés. Cette mise à jour est réalisée grâce aux différents outils disponibles.

2.1 *CdC*

Le cahier des charges, informel au démarrage du développement, est ré-écrit sous forme d'une suite de phrases courtes et étiquetées. Une phrase peut contenir des termes formels intégrés dans le texte informel. Ces termes proviennent de la spécification formelle associée au *CdC*. Ses opérations sont celles des types utilisés, suites, ensembles et produit cartésien noté CP.

CdC is Sequence(*Sentence*)

Sentence is CP [ID: *TEXT*,
[Order: *INTEGER*,]
Description : *Set*(*Term*),
[*Sequence*(*Sentence*),]
[*Set*(*Term_ Type*)]]

Term is Informal_ Term | *Formal_ Term*

Term_ Type is Fact | *Functionality* | *Behavior* | *Obligation*

La figure 2 présente deux exemples d'exigences du cahier des charges de l'étude de cas de la machine d'hémodialyse. L'exigence *R-8-1-2* est un fils de *R-8-1*.

ID	*Description*
R-8-1	The software shall control the pressure at the AP transducer during initiation
R-8-1-2	The initiation is a working phase in the BEP component

FIGURE 2 – Exemple d'exigences avec ProR

2.2 Spécification

La spécification est définie en Event-B en utilisant le raffinement. La correspondance entre le *CdC* et sa spécification est définie par :

- *ID*. Il s'agit d'un commentaire.
- *Order*. L'ordre entre les événements dans la spécification est décrit par leur garde.
- *Informal_ Term*. Il s'agit d'un commentaire.
- *Formal_ Term*. Un terme formel introduit dans le *CdC* correspond à un élément d'Event-B ; il s'agit du nom d'une variable, d'une constante, d'un ensemble ou d'un événement.
- *Term_ Type*. Le type d'un terme est :
  - *Fact* pour des constantes, des ensembles et des variables,
  - *Functionality* pour des événements,

- *Obligation* pour des axiomes, invariants, théorèmes et gardes,
- *Behavior* pour l'animation et la simulation de machines.

2.3 Liens entre le *CdC* et sa spécification

Pour mémoriser le cahier des charges, nous prenons en compte ses différentes mises à jour et explicitons sa place dans le processus de développement. Pour cela, un ensemble de liens entre besoins et spécifications a été identifié et réalisé avec ProR. Ces liens sont automatiquement mis à jour et disponibles tout au long du développement. Initialement, nous n'avons pas de spécification formelle et il n'y a pas de terme formel dans le *CdC*. Au cours du développement, des termes formels sont intégrés dans le *CdC*.

Les informations pour la validation sont extraites du *CdC* en tenant compte du type de chaque terme de chaque phrase dans le modèle formel associé. Un terme formel introduit dans la spécification est introduit dans le *CdC*. Nous avons défini :

- un lien entre un terme formel et sa définition informelle dans le *CdC*. Il s'agit d'un élément du glossaire ;
- un lien entre ce terme formel, dénoté entre crochets dans le *CdC*, et sa définition dans la spécification Event-B. Ce lien est géré par ProR via une adresse.

2.4 Outils

Gestion des exigences. L'outil ProR, *plug-in* de Rodin, permet d'exprimer de manière hiérarchique des exigences [8]. Il commence par l'élicitation des exigences initiales et les hypothèses du domaine. Il ne propose pas de notation particulière mais un classement des artefacts, voir figure 2.

Les liens entre exigences et éléments du modèle Event-B en cours de construction sont créés manuellement et peuvent être annotés. Pour gérer la traçabilité entre exigences et modèles formels, ProR permet :

- de définir manuellement des liens depuis la spécification Event-B vers les exigences texte. Initialement, ces exigences sont informelles,
- d'insérer des éléments formels dans les exigences, ces éléments étant issus de la spécification Event-B.

La vérification. Nous cherchons à formaliser les propriétés du système. Ces propriétés sont abstraites et générales au début du développement puis elles se précisent. Tout au long du développement de la spécification formelle [2], nous prouvons sa correction via des preuves formelles. Les obligations de preuve, OPs, sont générées automatiquement ou semi-automatiquement par Rodin.

La validation. Cette activité est utilisée tout au long du processus de développement avec l'outil ProB [9] sous Rodin. Des informations nécessaires à la validation de la future spécification sont extraites de chaque exigence. Elles sont mises à jour par des termes formels au fur et à mesure du développement. La validation s'effectue par la preuve de propriétés. Il s'agit de raisonner en termes de propriétés auxquelles le futur système doit obéir.

3 Un exemple de patron de développement

La lecture du texte de référence du *CdC* [3] nous invite à étudier la notion de patrons pour réutiliser une solution dans ce développement. La présentation des besoins de la machine de l'hémodialyse utilise une forme particulière. Elle nous a conduit à définir plusieurs patrons [7] ou modèles génériques adaptés à notre approche dans laquelle nous manipulons des triplets *<CdC, Liens, Spec>*. Chaque patron générique est mémorisé ainsi que ses paramètres. La défintion du patron utilise le raffinement en Event-B, les preuves correspondantes et précise les éléments restant à décrire.

Exemple. Regardons le besoin *R-8* :

> *R-8 During initiation, if the software detects that the pressure at the AP Arterial Pressure transducer falls below the lower pressure limit, then the software shall stop the BP Blood Pumping and execute an alarm signal.*

Ce besoin a la même forme que la plupart des exigences de ce cahier des charges. Nous définissons un patron pour faciliter le développement de ce dernier.

3.1 Forme générique de ce besoin

R-i During actual phase, if the software detects an error on p [for more/less than n seconds], then the software shall stop the BP Blood Pumping and execute an alarm signal.

Ce patron a un paramètre p qui désigne l'élément physique de la machine d'hémodialyse.

3.2 Présentation du patron

Le besoin *R-i* paramétré par p a la même forme qu'un besoin *R-j* décrit précédemment et paramétré par *prev_p*. Le patron de développement associé à *R-i* crée sa spécification *R-i_Mch*, voir figure 3. Cette construction s'effectue en fonction du besoin *R-j* et de sa spécification *R-j_Mch* associée. Les éléments dénotés par "..." sont à définir par le développeur.

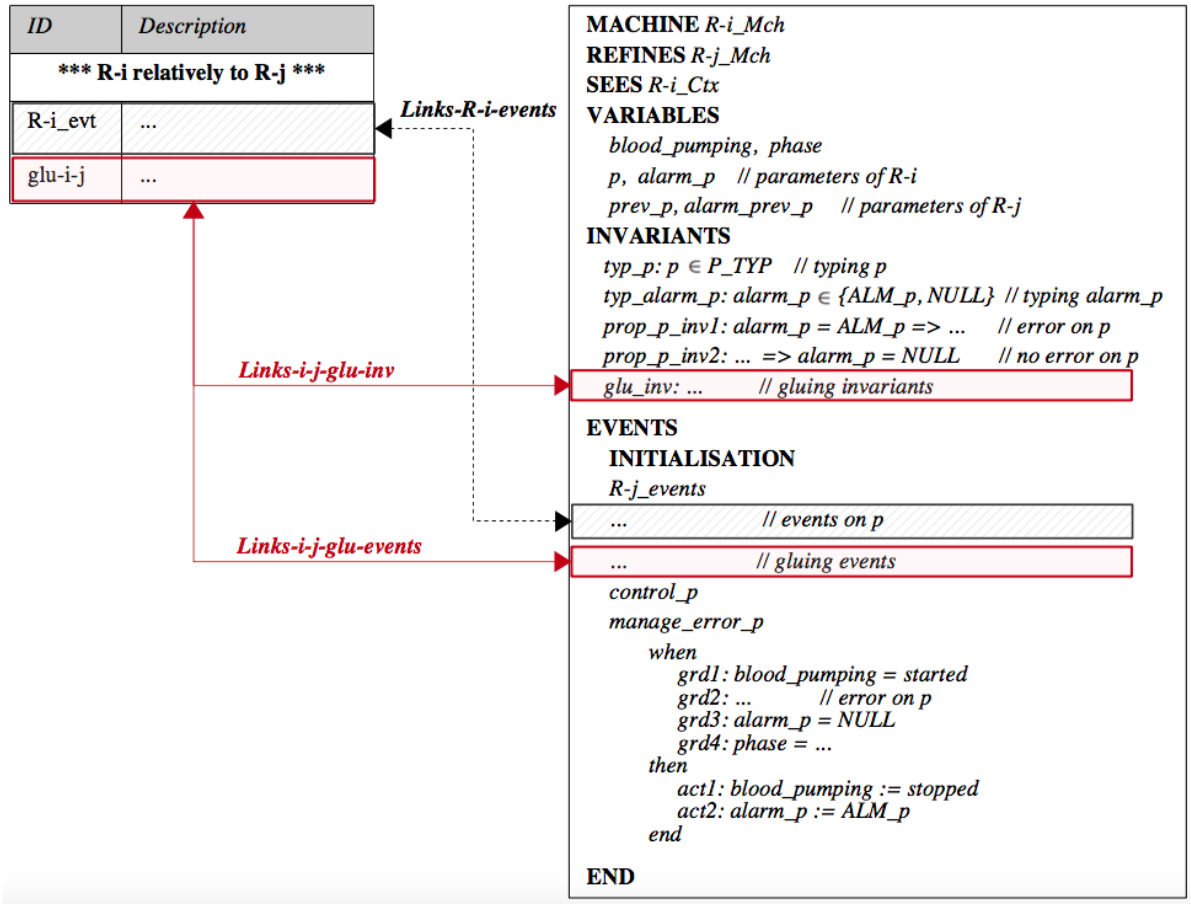


FIGURE 3 – Patron pour décrire *R-i* relativement à *R-j*

3.2.1 Réutilisation de la spécification R-j_Mch existante

La machine R-i_Mch est un raffinement de R-j_Mch. Les éléments de collage en Event-B doivent être définis.

CdC'. De nouveaux besoins *glu-i-j* sont ajoutés, voir la partie gauche de la figure 3. Ces besoins prennent en compte les différents cas en combinant les différentes valeurs de p et $prev\_p$. Ils contiennent des termes formels venant de la spécification.

Spec'. Des invariants de collage, comme glu_inv, et des événements de collage sont introduits. Ils sont définis dans la partie droite de la figure 3.

Liens'. Les besoins *glu-i-j* sont liés aux invariants et événements de collage via les liens nommés respectivement *Links-i-j-glu-inv* et *Links-i-j-glu-events*.

3.2.2 Evénements liés au paramètre p

Ces événements correspondent à l'évolution du paramètre du patron. Ils doivent être décrits par le développeur. Les descriptions formelle et informelle de ces événements sont reliées via les liens *Links-R-i-events*, voir figure 3.

4 Leçons retirées des différentes études de cas

L'utilisation des outils aide à améliorer le processus de développement, que ce soit avec ProR ou avec les outils de vérification et de validation. Le *CdC* et ses liens sont automatiquement mis à jour lors de l'évolution de la spécification. Notre propos est illustré par le développement d'un système d'hémodialyse [10], système critique, hybride et interactif contrôlant sa machine. Nous abordons les points suivants :

- rôle des alarmes,
- forme particulière des besoins présentant des anomalies,
- rôle de l'invariant de collage de la spécification relativement au triplet $<$ *CdC, Liens, Spec* $>$.

4.1 Compréhension et analyse des besoins

La prise en compte des propriétés de sécurité, de vivacité et de terminaison sont examinées lors de la compréhension du *CdC*. Certaines ambiguïtés et imprécisions sont détectées très tôt dans le développement.

Dans l'étude de cas de la machine d'hémodialyse :

- les propriétés de sécurité sont clairement exprimées dans son *CdC*,
- la notion d'alarme dans ce système est à clarifier :
  - Existe-t'il une alarme commune à toutes ses composantes ?
  - Existe-t'il une alarme spécifique à chaque composante ?
    Peut-on alors déceler plusieurs alarmes différentes à un instant donné ?

4.2 Evolution du développement à l'aide d'un patron

Nous développons le besoin *R-8* sachant que le besoin *R-6* de même forme a été développé.

R-6 During initiation, if the software detects that the pressure at the VP transducer falls below the lower limit, then the software shall stop the BP and execute an alarm signal.

La spécification formelle de *R-6* est :

MACHINE *R−6_ Mch*
SEES *R−6_ Ctx*
VARIABLES
blood_ pumping, *phase*
vp, *alarm_ vp*
INVARIANTS
typ_vp: $vp \in Z$
prop_vp_inv1: $alarm\_vp = ALM\_vp \Rightarrow vp < lower\_press\_limit$
EVENTS
INITIALISATION
increase_ vp_ initiation
decrease_ vp_ initiation
control_ vp
manage_ deficit_ vp
END

Nous utilisons le patron présenté dans la section 3 pour décrire le besoin *R-8* en termes de la spécification *R-6_ mch*. Le paramètre formel *p* de *R-8* est *ap* et le paramètre *prev_ p* de *R-6* est *vp*. Les éléments suivants doivent être complétés.

4.2.1 Réutilisation de la spécification R-6_Mch existante

CdC'. Les besoins concernant le collage entre *R-8* et *R-6* sont décrits par :

ID	*Description*
glu-8-6-1	If [ap] and [vp] fall below [lower_press_limit], then the software shall [manage_def icit_ap_def icit_vp]
glu-8-6-2	If [ap] falls below [lower_press_limit] and [vp] is normal, then the software shall [manage_def icit_ap]
glu-8-6-3	If [ap] and [vp] are normal, then the software does nothing
glu-8-6-4	If [ap] is normal and and [vp] falls below [lower_press_limit] , then the software shall [manage_def icit_vp]

Spec'. L'invariant de collage *glu_ 1* exprime la présence d'anomalies conjointes pour prendre en compte les besoins *R-8* et *R-6*. Il est défini par :

glu_1: $alarm\_ap = ALM\_ap \wedge alarm\_vp = ALM\_vp \Rightarrow ap < lower\_press\_limit \wedge vp < lower\_press\_limit$

L'événement de collage *manage_ deficit_ ap_ deficit_ vp* exprime le traitement simultané d'anomalies pour *R-8* et *R-6*. Il est défini comme suit :

EVENT manage_ deficit_ ap_ deficit_ vp
WHEN
grd1: $blood\_pumping = started$
grd2: $phase = initiation$
grd3: $alarm\_ap = NULL$
grd4: $alarm\_vp = NULL$
grd5: $ap < lower\_press\_limit$
grd6: $vp < lower\_press\_limit$
THEN
act1: $blood\_pumping := stopped$
act2: $alarm\_ap := ALM\_ap$
act3: $alarm\_vp := ALM\_vp$
END

4.2.2 Evénements liés au paramètre *ap*

Deux événements *decrease_ap_initiation* et *increase_ap_initiation* font évoluer le paramètre *ap*. L'action de décroître *ap* est définie comme suit :

EVENT decrease_ap_initiation
WHEN
grd1: $blood\_pumping = started$
grd2: $phase = initiation$
grd3: $alarm\_ap = NULL$
grd4: $ap \geq lower\_press\_limit$
THEN
act1: $ap := ap - 10$
END

4.3 Amélioration du *CdC* à partir de la spécification

Le développement de la spécification formelle permet de détecter des lacunes dans le *CdC* avec les outils de vérification et de validation. Ces lacunes concernent des oublis, des imprécisions et des incohérences. Elles n'ont pas été détectées dans l'étape de compréhension et analyse des besoins.

Exemple. Regardons le besoin *R-9*.

R-9 While connecting the patient, if the software detects that the pressure at the VP transducer exceeds + 450 mmHg for more than 3 seconds, then the software shall stop the BP and execute an alarm signal.

ID	*Description*
init1	The [blood_pumping] is [stopped]
init2	There is no pressure at the [ap]
init3	There is no pressure at the [vp]
init4	All the alarms are disabled

MACHINE *R−9_Mch*
REFINES *R−8_Mch*
VARIABLES
ap, *phase*, *alarm_ap*, *blood_pumping*, *vp*, *alarm_vp*
EVENTS
INITIALISATION
init_act1: $blood\_pumping := stopped$
init_act2:...
...
END

FIGURE 4 – Etat initial

Son modèle Event-B est défini par un raffinement de la *Machine R-8_Mch*. Nous avons détecté les anomalies suivantes :

- *Invariant de collage.* Dans le document initial, chaque besoin est isolé. Il est décrit séparément des autres besoins et ne tient pas compte de possibles interactions entre eux.
- *Etat initial.* L'activité de validation nécessite un état initial. Cet état n'a pas été mentionné explicitement dans le *CdC*. Nous avons proposé l'ajout de phrases décrivant cet état, voir figure 4.
- *Omission.* L'alarme peut être commune à toutes les composantes du système ou bien définie pour chaque composant. Les preuves aident à détecter les cas prévus dans la spécification. Par exemple, le choix d'une alarme générale signifie qu'il n'y a pas de changement de valeur de sa variable abstraite.

5 Conclusion et perspectives

Dans ce papier, nous avons abordé la gestion des besoins de la machine d'hémodialyse, ceux-ci sont décrits dans une forme identique [10]. Le document décrit les cas anormaux et présente la gestion des alarmes. L'utilisation et la ré-utilisation d'une spécification existante aborde le rôle de l'invariant de collage dans la spécification. Le patron génère automatiquement une partie de la spécification en cours de construction. La notion de collage est présentée via le *CdC*. L'effort de vérification est réduit. Les patrons utilisés dans cette

étude de cas peuvent être utilisés et adaptés à d'autres études de cas, comme celle concernant le système hybride de contrôle du train d'atterrissage d'un avion [6]. En effet, la présentation de leurs besoins est identique.

La validation en tant que processus rigoureux démarre avec la structuration des besoins, avant que la spécification associée ne soit introduite, jusqu'à l'animation des modèles Event-B. Ces modèles sont validés relativement aux besoins du client, en vue de détecter des problèmes [12].

Les outils disponibles dans la plateforme Rodin ont un rôle important tout au long du processus de développement. Nous utilisons l'outil ProR pour la ré-écriture des besoins, pour leur hiérarchisation et pour la mise à jour du *CdC* et des liens avec sa spécification. Nous utilisons les outils de vérification et de validation pour assurer la correction de la spécification avec les générateurs d'obligations de preuve, les prouveurs et l'animateur/model-checker ProB.

Pour la suite de notre travail, il est important de décrire rigoureusement les paramètres intervenant dans la prise en compte d'un besoin par rapport à un système existant. Ces paramètres servent à définir des patrons afin de réutiliser des modèles existants et corrects. L'étude de l'apport de l'étape d'analyse sert à la détection de lacunes dans le système existant.

Dans notre approche, nous étudions des systèmes hybrides. Ce sont des systèmes discrets et temporels qui fonctionnent dans un environnement continu. De tels systèmes combinent des aspects matériels et logiciels. Il serait important de prendre en compte les contraintes de l'environnement et de ses hypothèses relativement aux nouveaux besoins.

Références

[1] Rodin platform, http ://wiki.event-b.org.

[2] J-R. Abrial. B : passé, présent, futur. *Technique et Science Informatiques*, 22(1) :89–118, 2003.

[3] J.-R. Abrial. Faultless Systems : Yes We Can ! *IEEE Computer*, 42(9) :30–36, 2009.

[4] J.-R. Abrial. *Modeling in Event-B : System and Software Engineering.* Cambridge University Press, 2010.

[5] J-R. Abrial, M. J. Butler, S. Hallerstede, T. Son Hoang, F. Mehta, and L. Voisin. Rodin : an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6) :447–466, 2010.

[6] F. Boniol and V. Wiels. Landing Gear System case Study. In *ABZ Conference, Communications in Computer and Information Science, Springer*, volume 433, pages 1–18, 2014.

[7] T. S. Hoang, A. Fürst, and J.-R. Abrial. Event-B Patterns and their Tool Support. *Software and System Modeling*, 12(2) :229–244, 2013.

[8] M. Jastram. ProR, an Open Source Platform for Requirements Engineering based RIF. *SEISCONF*, 2010.

[9] M. Leuschel and M. J. Butler. ProB : A Model Checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *International Symposium of Formal Methods Europe, Pisa, Italy, Proceedings*, volume 2805 of *LNCS*, pages 855–874. Springer, 2003.

[10] A. Mashkoor. The Hemodialysis Machine Case Study. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 5th International Conference, ABZ 2016, Linz, Austria, Proceedings*, pages 329–343. Springer International Publishing, 2016.

[11] C. Ponsard, R. Darimont, and A. Michot. Combining Models, Diagrams and Tables for Efficient Requirements Engineering : Lessons Learned from the Industry. In *Actes du XXXIIIème Congrès INFORSID, Biarritz, France, May 26-29*, pages 235–250, 2015.

[12] I. Sayar and J. Souquières. La Validation dans le Processus de Développement. In *Actes du XXXIVème Congrès INFORSID, Grenoble, France, May 31 - June 3*, pages 67–82, 2016.

[13] W. Su, J.-R. Abrial, R. Huang, and H. Zhu. From Requirements to Development : Methodology and Example. In *13th International Conference on Formal Engineering Methods, Durham, UK*, pages 437–455, 2011.

Suivi de flux d'information correct sous Linux

Laurent Georget Guillaume Piolle Mathieu Jaume
Frédéric Tronel Valérie Viet Triem Tong Sorbonne Universités,
EPC CIDRE CENTRALESUPELEC/ UPMC, CNRS, LIP6
INRIA/CNRS/Université de Rennes 1 UMR 7606
laurent.georget@irisa.fr

Résumé

À l'échelle des systèmes d'exploitation, le contrôle de flux d'information porte sur la façon dont les données circulent entre des objets tels que les processus, les fichiers, les sockets réseau, etc. afin de prévenir les fuites et modifications illégales d'information. La maîtrise des flux d'information peut servir de multiples objectifs si l'on peut garantir la qualité des implémentations des moniteurs de flux d'information. Nous avons étudié Laminar [3], KBlare [6] et Weir [2], des moniteurs de flux d'information dont l'implémentation repose sur l'interface des *Linux Security Modules* (LSM). Même si LSM a été conçu à l'origine pour le contrôle d'accès [5], ces implémentations laissent à penser que cette interface offre, d'un point de vue pratique, un bon support pour l'implémentation d'un moniteur de flux d'information. Nous examinons donc ici la question suivante : *Est-il possible d'implémenter de manière fiable le contrôle de flux d'information sous Linux en se basant sur LSM ?* Nous décrivons ici deux axes de réflexion, l'un ayant conduit à vérifier une condition nécessaire, l'autre à proposer une condition suffisante pour répondre à cette question. Nous décrivons aussi Rfblare, la principale contribution pratique issue de ces travaux. Rfblare est une nouvelle version du suivi de flux d'information de KBlare résistante aux conditions de concurrence entre appels système.

1 Introduction

Dans un système d'exploitation, les processus, fichiers, sockets réseau et zones de mémoire partagée sont des conteneurs d'information. Ils participent au stockage, au traitement et à l'acheminement de l'information à l'intérieur d'un système ainsi que depuis et vers l'extérieur. Tous les flux d'information ne sont pas désirables. Par exemple, si un processus exfiltre des données confidentielles hors du système ou falsifie une information dans une base de données, le système a échoué à garantir les propriétés de confidentialité et d'intégrité des données. Pour garantir ces propriétés, le système repose traditionnellement sur le *contrôle d'accès* : en prévenant tout accès, même indirect, à des données sensibles, on empêche les flux illégaux. Cependant, il est utile dans certaines situations d'être capable de permettre l'accès d'un

processus à une donnée s'il ne la retransmet à son tour qu'à des processus autorisés à y accéder. Le contrôle de flux d'information permet cela. Il se distingue du contrôle d'accès en maintenant un historique des flux passés dans le système. Cette connaissance supplémentaire lui permet d'avoir les mêmes garanties d'absence de flux illégaux, tout en permettant plus de flux.

La connaissance des flux est généralement implémentée par un mécanisme de *propagation de teintes*. Chaque conteneur d'information est initialisé avec une méta-donnée appelée *label*, ou *teinte*, qui représente son contenu. À chaque flux d'un conteneur vers un autre, le label de la destination est mis à jour avec le label de la source afin de noter que son contenu comporte à présent des données issues de la source, qui elle-même a pu les hériter d'autres sources. Le label d'un conteneur représente donc à tout moment de la vie du système l'ensemble des sources qui ont pu contribuer à son contenu, même indirectement.

Nous avons étudié trois implémentations du contrôle de flux d'information pour Linux : Laminar [3], KBlare [6] et Weir [2]. Laminar et KBlare sont implémentés pour le noyau Linux officiel et Weir pour le noyau Android. Ils sont basés sur des modèles de labels différents mais une particularité commune marquante est qu'ils reposent tous les trois sur le *framework Linux Security Modules* (LSM). LSM est une interface présente dans le noyau depuis 2001 permettant le développement et l'intégration de modules de sécurité en complément du modèle de sécurité de base de Linux. LSM ajoute deux éléments au noyau : des champs de sécurité supplémentaires dans les structures de données internes du noyau (comme la structure *inode* représentant les fichiers) et des *crochets*, des endroits prédéterminés dans le code où les modules de sécurité peuvent enregistrer des fonctions. L'usage attendu d'un module LSM est qu'il conserve son état (comme le domaine de sécurité de chaque fichier) dans les champs de sécurité à sa disposition et qu'il utilise les crochets pour interrompre l'opération en cours (il existe par exemple un crochet pour intercepter l'opération de lecture des fichiers), consulter son état et prendre la décision d'autoriser ou non la poursuite de l'opération présentant un risque de sécurité.

LSM a été conçu pour le contrôle d'accès [5] mais son utilisation dans les implémentations existantes de moniteurs de flux d'information montre qu'il constitue une base intéressante pour ce cas d'usage également. Cependant, la question de savoir s'il constitue une base *fiable* reste ouverte. Dans cet article nous présentons deux axes de recherche nous ayant conduit à vérifier, premièrement, que chaque flux réalisé par un appel système est détecté, puis que les flux engendrés par des appels système concurrents sont bien pris en compte (section 3). Nous détaillons finalement Rfblare, une nouvelle version du suivi de flux de KBlare, présentant la particularité unique à notre connaissance, d'être capable de suivre les flux même en présence de conditions de concurrence entre appels système et de gérer les flux causés par les mémoires partagées.

2 Un crochet pour chaque flux

Dans un système d'exploitation, un flux d'information a lieu lorsqu'un processus écrit ou lit de l'information depuis un fichier ou une socket réseau,

lorsqu'il opère la copie d'un fichier vers un autre, ou encore lorsqu'il met en place une mémoire partagée avec un autre processus. Ces opérations sont effectuées par des *appels (au) système*. Les appels système sont des fonctions implémentées par le noyau et appelables depuis les processus utilisateur pour accomplir une action demandant des privilèges, comme l'accès aux périphériques matériels. Nous faisons l'hypothèse que tous les flux d'information explicites [1] requièrent un appel système. Cette hypothèse, commune à tous les moniteurs de flux implémentés dans un noyau de système d'exploitation, est justifiée par le fait que seul le système possède un accès direct au matériel, comme les disques, aux structures de données représentant les conteneurs d'information et à l'intégralité de la mémoire du système. Les processus utilisateur sont cloisonnés et limités en privilèges.

Les crochets LSM sont positionnés dans le code des appels système, après les vérifications basiques sur la validité des paramètres de l'appel et avant que l'opération réelle de l'appel système présentant un risque de sécurité (et donc le flux d'information, dans le cas des appels système générant des flux) soit effectuée, afin que le module de sécurité puisse l'empêcher. Les implémentations de moniteurs de flux d'information enregistrent via les crochets les fonctions effectuant la propagation de teintes. Une condition nécessaire au suivi correct des flux est donc la présence d'au moins un crochet LSM le long de chaque chemin d'exécution générant des flux d'information dans les appels système. Nous appelons cette propriété la *médiation complète* ; si elle n'est pas vérifiée, il est possible d'effectuer un flux d'information sans que le moniteur n'en soit averti.

Nous avons construit un modèle du code des appels système sous la forme de graphes de flot de contrôle. Nous avons ensuite conçu une analyse statique, applicable indépendamment sur chaque appel système qui nous a permis de caractériser quelques problèmes de LSM dans le placement des crochets pour le suivi de flux d'information. Pour les besoins de notre analyse statique, le code d'un appel système est un graphe orienté dans lequel chaque nœud représente une instruction et un arc matérialise le fait que l'exécution d'une instruction peut être suivie par l'exécution d'une autre. Les arcs peuvent être étiquetés par une condition de branchement. Ce modèle est construit de manière automatique par le compilateur du noyau, GCC. L'analyse statique est en effet implémentée comme un plugin de GCC. Le langage des instructions du graphe n'est pas directement du C mais une représentation intermédiaire construite par GCC.

Chaque appel système étant représenté par un graphe, les chemins d'exécution de cet appel sont donc un sous-ensemble des chemins commençant à l'entrée de l'appel (le seul nœud sans prédecesseur) et terminant au retour (un nœud sans successeur). Tous les chemins du graphe ne sont pas des chemins d'exécution valides. En effet, certains ne correspondent à aucune exécution concrète, par exemple parce qu'ils nécessitent de passer par deux arcs étiquetés par des conditions incompatibles entre elles. Comme les crochets LSM ne sont pas disposés à l'entrée de l'appel système, il peut exister dans le graphe un chemin qui contourne les crochets mais passe dans la branche où le flux est effectué. Si ce chemin est impossible, ce n'est pas un problème, en revanche, s'il est possible, c'est une violation de la propriété de médiation complète.

1. Nous excluons ici les canaux cachés et les flux implicites.

Vérifier la propriété revient donc à vérifier que chaque chemin d'exécution esquivant les crochets LSM est impossible.

L'analyse statique proposée est *path-sensitive*. Elle est fondée sur une notion de *configuration* qui associe à chaque variable un ensemble de contraintes arithmétiques sur ses valeurs possibles. Pour chaque chemin, l'analyse commence avec une configuration vide. À chaque nœud, la configuration est modifiée en fonction des propriétés prédictibles sur la sémantique concrète du programme. Certaines instructions, comme les affectations, ajoutent des contraintes tandis que d'autres, comme des appels de fonctions, dont il n'est pas toujours possible de prédire le comportement, relâchent ces contraintes. Le franchissement d'arcs étiquetés ajoute également des contraintes. Les boucles dans le code (conduisant à la génération d'une infinité de chemins) sont abstraites en calculant une surapproximation de la configuration couvrant un nombre arbitrairement grand d'itérations. Nous sacrifions donc un peu de précision de l'analyse mais garantissons sa terminaison et maintenons sa correction. Un chemin est considéré comme impossible lorsque, pour au moins une variable, l'ensemble de ses contraintes n'est pas satisfaisable. Par exemple, passer par le nœud $x = \&y$ puis $*x = 3$ puis par un arc étiqueté par $y < 2$ met d'abord à jour la configuration avec $\{x = \&y\}$ puis $\{x = \&y, y = 3\}$ puis $\{x = \&y, y = 3, y < 2\}$ qui est une configuration insatisfaisable. La satisfaisabilité est déterminée par le SMT-solveur Yices [1]. Dès qu'une configuration insatisfaisable est obtenue, le chemin analysé est déclaré impossible. Si l'analyse se poursuit jusqu'à la fin du chemin, alors il est déclaré possible. L'analyse est correcte : les chemins déclarés impossibles ne peuvent pas correspondre à une exécution concrète du programme. Bien sûr, une telle analyse ne peut pas être complète : certains chemins déclarés possibles sont en réalité impossibles. L'analyse que nous proposons est très simple, beaucoup de traits du langage ne sont pas pris en compte et donnent lieu à des surapproximations (via le relâchement de contraintes) mais nos résultats montrent qu'elle suffit pour répondre à nos objectifs.

Nous avons appliqué cette analyse à tous les appels systèmes correspondant à des communications inter-processus ou à une lecture–écriture dans un fichier ou encore à une création de processus. Nous avons placé dans le code une annotation sur les crochets LSM et les appels de fonction correspondant à la réalisation du flux d'information (par exemple, l'appel de la fonction modifiant le contenu des fichiers, après toutes les vérifications) pour notre plugin. Les résultats que nous avons obtenus identifient un petit ensemble d'exécutions ne satisfaisant pas la propriété de médiation complète. Ces résultats permettent de rajouter ou déplacer des crochets LSM dans le code pour que ces exécutions soient couvertes. Nous pouvons conclure que LSM reste approprié pour le contrôle de flux d'information car les exécutions problématiques ne représentent qu'une faible fraction de toutes celles existantes, et l'analyse permet d'étendre facilement ce framework. Cette approche démontre également que concevoir des analyses statiques via le compilateur est très bénéfique. Notre analyse porte ainsi sur le code qui sera exécuté, selon la sémantique que lui donne le compilateur, et non directement sur celui qui est écrit. Par exemple, cela permet de bénéficier des représentations internes du compilateur, et en particulier des graphes de flot de contrôle. Nous avons analysé la version 4.3 du noyau Linux mais cette analyse est reproductible

sur les autres versions du noyau. Le code et tous les résultats présentés ici sont disponibles sur `kayrebt.gforge.inria.fr`.

3 Gérer les flux concurrents

La présence d'un crochet sur chaque chemin engendrant un flux n'est pas une condition suffisante pour affirmer qu'aucun flux n'échappe au moniteur. En effet, les moniteurs de flux sont sujets à des *conditions de concurrence* car dans un appel système, la séquence d'opérations ⟨passage dans un crochet LSM, réalisation effective du flux⟩ n'est pas atomique. Imaginons par exemple deux processus, l'un effectuant l'appel système `write` pour écrire dans un fichier et l'autre effectuant l'appel système `read` pour lire depuis ce même fichier. Une exécution possible est la séquence ⟨passage du lecteur dans son crochet LSM, passage de l'écrivain dans son crochet LSM, flux de l'écrivain vers le fichier, flux du fichier vers le lecteur⟩. Dans cette exécution, il y a un flux indirect de l'écrivain vers le lecteur via le fichier. Cependant, le moniteur de flux d'information ne voit que les passages dans les crochets LSM et donc, de son point de vue, le flux fichier→lecteur est antérieur au flux écrivain→fichier. Les teintes ne sont pas propagées de l'écrivain au lecteur.

Pour démontrer que cette attaque ne touche pas que KBlare mais est en réalité inhérente à tous les mécanismes implémentés avec LSM, nous avons implémenté plusieurs attaques très simples. La première situation implique deux processus, un *émetteur* et un *récepteur*, deux fichiers et un tube (*pipe* UNIX). Le but de l'opération est pour l'émetteur de lire un fichier *source* secret et tagué, de le transmettre via le tube au récepteur, afin que ce dernier copie les informations secrètes dans le fichier *destination*. L'attaque est considérée comme réussie si le fichier destination n'est pas tagué mais contient les données de la source. L'attaque réussit dans le système protégé par KBlare pour la raison suivante. Le récepteur commence par lire le tube avant que l'émetteur n'y écrive, la propagation de teinte a alors lieu mais le tube n'étant pas tagué initialement, rien ne se passe. La lecture bloque car le tube est vide. Simultanément, l'émetteur lit le fichier source puis le copie dans le tube. À ce point, la propagation de teinte fait que le tube reçoit le tag du fichier source. Cependant, comme la propagation de teinte a déjà eu lieu du tube vers le récepteur, ce dernier ne devient pas tagué. Il peut donc écrire les informations de la source dans la destination sans qu'une alerte ne soit levée. Il faut noter ici que l'usage d'un tube facilite l'exploitation de la condition de concurrence car les tubes sont normalement bloquants (lire depuis un tube vide bloque la lecture jusqu'à ce qu'un autre processus y écrive) sauf dans les systèmes protégés par Laminar qui réclame l'usage des tubes en mode non-bloquant. Cependant, si le tube est non-bloquant, ou bien si l'on utilise un fichier standard, la condition de concurrence existe toujours car le processus récepteur peut malgré tout s'endormir et interrompre son exécution entre le crochet LSM et le flux, l'attaque réussit seulement moins souvent. Nous avons reproduit cette attaque sur Weir, en développant deux applications Android, avec les mêmes résultats.

Ce type de condition de concurrence pourrait être résolu en rendant atomique la séquence ⟨passage dans le crochet, flux⟩. C'est d'ailleurs l'approche adoptée par Laminar [3] mais cela pose au moins un problème majeur : cer-

tains flux peuvent dépendre d'autres (par exemple, lire dans un *pipe* vide bloque jusqu'à ce qu'un processus y écrive), ce qui peut conduire à des interblocages. Nous considérons que le moniteur de flux d'information peut uniquement avoir connaissance des bornes de la section de code entre lesquelles le flux peut avoir lieu. La borne ouvrante est le crochet LSM déjà présent (nous avons vérifié dans la section précédente qu'il existe bien un tel crochet dans tous les cas). Le retour de l'appel système peut être pris comme borne fermante. Entre les bornes, le flux est dit *ouvert*.

L'algorithme que nous proposons repose sur deux ensembles de flux : les flux déjà réalisés (ayant déjà eu lieu) dans le système et les flux en cours. Au début de la vie du système, ces deux ensembles sont vides. Un flux ouvert est une relation entre deux conteneurs et la fermeture réflexive et transitive de cette relation définit l'ensemble des flux en cours. C'est l'ensemble des flux, directs et indirects, s'effectuant en concurrence. Les flux réalisés dans le système sont mis à jour à chacun des événements d'ouverture et fermeture de flux en calculant la composition de ces flux avec les flux en cours. Par exemple, si l'ensemble des flux réalisés est $\{A \to B\}$ et l'ensemble des flux ouverts est $\{C \to D, B \to C\}$, alors l'ensemble des flux en cours est $\{C \to D, B \to C, B \to D\}$ et $\{A \to B, A \to C, A \to D, B \to C, B \to D\}$ correspond alors au nouvel ensemble des flux réalisés (on omet ici les flux réflexifs pour alléger la présentation). Nous avons démontré que ce résultat correspond à la plus petite surapproximation des flux que l'on puisse calculer qui prenne en compte tous les entrelacements de flux possibles. Autrement dit, chaque flux identifié peut être engendré par une exécution concrète compatible avec la séquence d'événements d'ouverture et de fermeture de flux rencontrés. Nous avons implémenté cet algorithme dans le moniteur de flux d'information Blare [6] et nous l'avons évalué. Une propriété remarquable de notre moniteur est qu'il peut gérer correctement les flux d'information causés par les mémoires partagées (ou les projections en mémoire de fichiers), qui sont des exemples typiques de flux indirects : si un processus A a une mémoire partagée avec B qui lui-même a une mémoire partagée avec C, alors il existe un flux indirect entre A et C qui ne peut pas être détecté en considérant les flux indépendamment les uns des autres. De plus, les mémoires partagées requièrent des appels système uniquement pour mettre en place et détacher la mémoire (les opérations individuelles de lecture–écriture ne nécessitent aucun appel système). Le problème causé par les mémoires partagées est connu et considéré comme difficile, comme l'attestent certains commentaires dans les codes source de Laminar et Blare. Notre algorithme permet de le résoudre en considérant toute la période d'existence d'une mémoire partagée comme un flux.

Afin de démontrer l'utilisabilité en pratique de notre algorithme, nous l'avons implémenté dans *Rfblare* (pour *Race-Free Blare*, Blare sans conditions de concurrence), une nouvelle version de la propagation de teintes de KBlare. *Rfblare* s'attache à couvrir un grand nombre de canaux ouverts, certains bien connus comme `read`, `write` et leurs dérivés, d'autres spécifiques à Linux comme `process_vm_readv`, bien que l'on ne puisse pas prétendre à l'exhaustivité.

Conformément à la description formelle de l'algorithme, nous utilisons un crochet LSM comme événement d'activation du flux et un autre comme

événement de désactivation. La projection de notre modèle sur le code du noyau n'est pas immédiate et la connaissance amassée au cours du travail présenté dans la section précédente nous a servi ici. Il est intéressant de noter que l'événement de désactivation n'est pas systématiquement nécessaire. En effet, certains appels système ne peuvent pas entrer en condition de concurrence avec d'autres car le conteneur sur lequel il agit est verrouillé par ailleurs, ou bien il est créé par l'appel système lui-même. Le cas de `shmat`, `mmap` et `mprotect`, qui servent à la manipulation des projections en mémoire de fichiers, est particulier. Contrairement aux autres flux où *Rfblare* maintient lui-même l'information d'activation et de désactivation, les flux causés par les fichiers projetés en mémoire est géré en interrogeant le noyau lui-même sur les projections actives. Plus précisément, pour chaque fichier, le noyau maintient la liste des mémoires de processus dans lequel il est projeté ; et pour chaque mémoire, on peut également connaître les fichiers qui sont projetés dedans. Le noyau maintient cette connaissance pour son propre usage, en particulier la libération correctes des ressources. Nous disposons donc en permanence d'un graphe précis et à jour, jamais désynchronisé de la réalité, en ce qui concerne les flux dus aux fichiers projetés. Nous utilisons ce graphe pour propager les teintes de manière appropriées. Nous utilisons néanmoins les crochets LSM d'activation du flux afin de propager les teintes dès l'apparition d'une projection. La projection est faire du fichier vers le processus si elle est en lecture seule et dans les deux sens si elle est en lecture–écriture. Le cœur de *Rfblare* est une simple table de flux activés, doublement indexée par la source du flux (ce qui forme donc un graphe) et par le processus ayant déclenché le flux (essentiellement afin de permettre la désactivation du flux et la libération des ressources). À chaque nouvelle apparition d'un flux, un parcours du graphe des flux activés est réalisé afin de propager les teintes à tous les conteneurs atteignables. En plus du graphe maintenu par *Rfblare*, nous utilisons aussi le graphe des fichiers projetés, comme décrit plus haut. Le parcours en largeur converge car la foction calculée est monotone. On peut arrêter le parcours d'un chemin du graphe dès la rencontre d'un nœud possédant déjà les tags de la source du flux.

Nous avons testé les attaques effectuées sur *KBlare* présentées plus haut afin de valider le bon fonctionnement de la propagation de teinte. Dans le cas de la première attaque exploitant la condition de concurrence sur les tubes. La séquence d'événements est bien sûr la même que précédemment mais *Rfblare* propage correctement les teintes. Le premier événement est l'ouverture du flux du tube vers le récepteur. Le flux est enregistré comme ouvert à ce moment puis le récepteur s'endort avant de sortir de l'appel système. Ensuite, le processus émetteur ouvre le flux du fichier source vers sa mémoire. À ce point le tag du fichier source est transféré à l'émetteur. Comme aucun flux ayant pour source l'émetteur n'est actif, la propagation s'arrête à ce point. Le flux est ensuite fermé après la lecture du fichier. L'événement suivant est l'ouverture du flux d'écriture dans le tube par l'émetteur. À ce point, le flux du tube vers le récepteur est toujours ouvert, et par conséquent la teinte de l'émetteur, contenant le tag du fichier, est transmise au tube et au récepteur. Enfin, les flux sont fermés, et le flux du récepteur au fichier destination a lieu. Le tag initial du fichier source, et celui de tous les conteneurs du chemin de flux, se retrouve dans la teinte du fichier destination. Le contenu du fichier

destination, modifié par celui de la source, est donc reflété correctement dans ses teintes, en dépit de la différence entre l'ordre des flux et l'ordre des ouvertures de ceux-ci.

Nous avons également reproduit une seconde attaque similaire utilisant des fichiers projetés en mémoire et une zone de mémoire partagée en remplacement du tube. Dans cette attaque, nous pouvons faire varier l'ordre dans lequel les projections et la mémoire partagée sont mises en place, toujours en faisant en sorte de copier le contenu du fichier source dans la mémoire partagée, puis de la mémoire partagée dans le fichier destination. Grâce aux structures de données du noyau nous permettant de connaître la correspondance entre les fichiers projetés et les mémoires de processus, *Rfblare* est capable dans tous les cas de propager les teintes correctement.

Nos preuves, vérifiées avec Coq [4], implémentations, tests et résultats sont disponibles en ligne : `blare-ids.org/rfblare`.

4 Conclusion

Nous avons présenté deux contributions majeures : une analyse permettant de vérifier le bon positionnement du crochet LSM dans un appel système donné et un mécanisme permettant au moniteur de flux d'information d'être résistant face aux conditions de concurrence. Ce mécanisme permet aussi de gérer une nouvelle catégorie de flux jusqu'alors hors de portée des moniteurs de flux d'information : les mémoires partagées et projections de fichiers en mémoire. Ces contributions visent à améliorer la confiance que l'on peut avoir dans les moniteurs de flux d'information fondées sur LSM.

Références

[1] Bruno DUTERTRE et Leonardo de MOURA. *The Yices SMT solver*. SRI International, 2006.

[2] Adwait NADKARNI et al. « Practical DIFC Enforcement on Android ». In : *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX.

[3] Donald E. PORTER et al. « Practical Fine-Grained Information Flow Control Using Laminar ». In : *ACM Transactions on Programming Languages and Systems* 37.1 (nov. 2014).

[4] THE COQ DEVELOPMENT TEAM. *The Coq Proof Assistant Reference Manual*. 14 déc. 2016.

[5] Chris WRIGHT et al. « Linux Security Modules : General Security Support for the Linux Kernel ». In : *USENIX Security Symposium*. 2002.

[6] Jacob ZIMMERMANN. « Détection d'intrusions paramétrée par la politique par contrôle de flux de références ». Thèse de doct. Université de Rennes 1, 16 déc. 2003.

HardenedGolo : pour augmenter le niveau de confiance en un code Golo

Oscar CARRILLO Nicolas STOULS Raphael LAURENT
Nikolai PLOKHOI Qifan ZHOU Julien PONGE Frédéric LE MOUËL

Univ Lyon, INSA Lyon, CITI, F-69621 Villeurbanne, France
prenom.nom@insa-lyon.fr

Résumé

Cet article décrit un travail préliminaire autour du langage de programmation Golo. Notre objectif est de fournir aux développeurs des outils permettant de renforcer leur confiance en leur code. Pour ce faire, nous avons expérimenté plusieurs approches (test dynamique, analyse de type et preuve de programme) et nous cherchons maintenant des choix pertinents pour avancer dans chacune de ces pistes.

Mots clefs : Golo – typage – test – preuve.

1 Introduction

Golo [13] est un langage et un outil développé dans le cadre du projet *Eclipse* éponyme [1]. Ce langage est dynamiquement typé et s'exécute sur une machine virtuelle Java standard. Son typage dynamique permet notamment de créer des objets dont la structure peut être mutée durant l'exécution. Cela signifie qu'il est par exemple possible d'ajouter de nouveaux attributs ou de nouvelles méthodes à un objet pendant son cycle de vie.

Initialement pensé pour exploiter certaines fonctionnalités de la JVM non exploitables dans le langage Java (instruction `invokeDynamic` du bytecode Java), le cas d'usage de prédilection de Golo est l'IoT. En effet, l'aspect dynamique de l'environnement où de nouveaux objets communicants peuvent apparaitre et disparaitre peut être plus facilement intégré pour faire de l'exécution contextuelle [12].

L'exemple de la Figure 1 est un code Golo mettant en évidence l'évolution de la définition du type d'une variable, avec les contraintes d'exécution que cela pose. On y voit notamment que la fonction `genereDiviseurPar` ajoute une méthode `divise` au paramètre `v` reçu en paramètre. Ainsi, suivant le chemin pris par la fonction `TestDivision`, la variable `v` peut, ou non, posséder la méthode appelée sur sa dernière ligne. Le `main` de cet exemple fait 3 appels successifs à cette fonction, pour mettre en évidence 3 cas de figure : la division a lieu, une méthode inexistante est appelée et une division par 0 est réalisée. Il est intéressant de noter que si aucun des deux derniers cas n'est souhaitable, seul le dernier génère une exception. Le cas 2 renverra le résultat d'un appel à `toString`.

1. `https://projects.eclipse.org/projects/technology.golo`

Dans le cadre de la présente communication, nous décrivons une réflexion préliminaire visant à fournir aux développeurs Golo des outils permettant de renforcer leur confiance en leur code [11]. Pour ce faire, nous avons commencé à expérimenter différentes approches : analyse symbolique du code pour générer de cas de tests, analyse de type et preuve de programme. Les avantages tirés et la complexité de mise en œuvre de chacune de ces solutions ne sont pas les mêmes. C'est pourquoi, nous voudrions pouvoir laisser le choix au développeur de vérifier les différentes parties de différentes manières sans qu'une politique globale d'évaluation soit formellement définie. Dans un monde idéal, notre objectif serait de pouvoir maintenir un *bilan de confiance* en le logiciel, en maintenant à jour tout ce qui a été vérifié et ce qui devrait l'être. De même, les outils de vérification étant intégrés dans le compilateur lui même, il serait envisageable que les éléments non vérifiés puissent être compilés avec l'ajout automatique d'une instrumentation vérifiant à l'exécution le respect d'une propriété. En conservant également la description de la méthode utilisée, nous espérons qu'il serait ensuite possible de rejouer les vérifications, pour faire de la non-régression.

```
module hardenedgolo.dynamic
local function genereDiviseurPar = |v,denum| {
    v:define("divise", |this,num| -> num/denum)
}

local function TestDivision = |n,d| {
    let v = DynamicObject()
    if(d!=1) { # Erreur. Ça aurait du être d!=0
        genereDiviseurPar(v,d)
    }
    return v:divise(n)
}

function main = |args| {
    # Cas 1 : Ça fonctionne
    var res = TestDivision(42,12)
    println("42 / 12 = "+res)

    # Cas 2 : appel d'une méthode qui n'existe pas
    res = TestDivision(42,1)
    println("42 / 1  = "+res)

    # Cas 3 : division par 0
    res = TestDivision(42,0)
    println("42 / 0  = "+res)
}
```

FIGURE 1 – Exemple

L'implémentation de nos expérimentations est disponible dans l'outil HardenedGolo [2], diffusé sous licence *Eclipse*. Les parties bilan de santé, rejeu et ajout de code de vérification à l'exécution n'ont pas encore été étudiées.

Dans la suite de cette communication, nous présentons nos expérimentations initiales sur les différentes approches de vérification, puis nous établissons un lien avec l'état de l'art qui nous semble pertinent pour avancer dans ce travail, avant de conclure sur les perspectives de ce travail.

2 Pistes explorées

Dans nos expérimentations visant à augmenter la confiance en un code Golo, nous avons exploré trois pistes développées ci-après : le test symbolique, vérification de typage et la preuve de programme. L'ajout de spécifications permettant d'aider les outils d'analyse, nous avons également proposé une adaptation d'un langage de spécification par annotations.

2. https://github.com/dynamid/HardenedGolo

2.1 Test Symbolique

Nous avons exploré la génération de cas de tests par exécution symbolique [9]. Cette analyse va générer des jeux de valeurs d'entrée qui permettront ensuite d'exécuter les chemins sélectionnés.

Pour pouvoir générer ces valeurs d'entrée, nous avons besoin d'un *solveur de contraintes* qui permette de définir un domaine de définition de chaque donnée. Cependant, certaines expressions ne sont pas solvables en arithmétique linéaire [2]. Ainsi, pour diminuer le nombre de conditions sans solution, nous proposons d'utiliser des tests symboliques dynamiques, et notre choix s'est porté sur le *Concolic Test* [6]. Ce type de test permet de trouver des valeurs pour les fonctions complexes à partir de l'exécution du programme avec un mix des valeurs concrètes et symboliques.

À ce jour, le compilateur HardenedGolo intègre un outil de génération de tests symboliques dynamique (`golo symtest`), qui explore les différents chemins et en même temps trouve une solution possible pour le chemin exploré. Pour résoudre les conditions des valeurs nous utilisons *Choco solver* [8] pour sa facilité d'utilisation dans Java. À ce stade, notre génération de tests contient encore de grosses limites et le non respect de ces limites arrêtera l'exploration du code :

- les variables manipulées ne peuvent être que des valeurs entières ;
- seuls des opérateurs simples peuvent apparaitre dans les conditions mathématiques envoyées au solveur (`+,-,*,/,%,=,>,<,!=`) ;
- pas de support pour l'appel de fonctions ;
- pas de support pour les boucles ;
- seul le critère de couverture *tous les chemins* a été implémenté ;
- pas d'utilisation des annotations de spécification présentes dans le code (et présentées dans la suite).

2.2 Vérification de typage

Étant donnée la nature dynamique de Golo et l'impossibilité dans le langage de pouvoir typer les paramètres d'une fonction, il est impossible de pouvoir toujours déterminer le type d'une variable et donc de savoir s'il est autorisé d'y appeler une méthode ou d'y lire un attribut. D'autant plus, qu'en plus de la mutabilité des types, il est possible de réaffecter une variable avec une donnée d'un autre type. Cependant, avoir la connaissance d'informations de type est indispensable pour certaines analyses. C'est pourquoi nous avons commencé à développer les prémisses d'une analyse de type pour vérifier leur concordance, en commençant par certains cas simples.

Pour l'instant, l'analyse implémentée se base sur les annotations, qui contiennent des informations de typage et sur la concordance des usages en se limitant aux trois familles de types : numérique, booléen et autre. Il nous est ainsi possible de lever des `warning` dans certains cas simples, tel que par exemple l'usage d'une valeur booléenne dans une expression arithmétique.

2.3 Ajout d'annotations de spécification

Pour permettre au développeur d'exprimer des propriétés sur le programme vérifié, nous proposons d'utiliser des annotations dans le programme. Cela n'est pas une technique de vérification, mais un outil permettant au concepteur d'exprimer des propriétés à vérifier sur son système. Mais cela permet également de gagner en connaissances sur le programme notamment par rapport au typage des éléments.

Au jour d'aujourd'hui, notre développement HardenedGolo supporte l'ajout d'annotations de spécifications aux fonctions. Le langage utilisé est celui de WhyML [5], encapsulé entre des balises `spec/ ... /spec`. Ces annotations sont actuellement exploitées pour faire une analyse de type simple des variables. Elles sont également propagées dans la traduction en WhyML de fonctions Golo, comme discuté dans la section suivante. L'exemple de la figure 3 montre une fonction de valeur absolue dont la pré et la post-condition (resp. `requires` et `ensures`) sont décrites avec pour objectif de vérifier son correct usage. Excluant donc de fait l'entier représentable minimum (`MinInt`) des valeurs autorisées en entrée.

Cependant, pour prendre en compte la dynamique du langage, certains cas particuliers sont à considérer. Par exemple, la figure 2 montre un exemple de quelque chose que l'on veut pouvoir faire, mais où nous n'avons pas encore décidé comment le faire ou l'exprimer. Dans cet exemple, la fonction `calculateur` reçoit trois paramètres et nous voulons exprimer que le troisième paramètre est un objet fournissant au moins la fonction `additionneur`. Cette fonction doit avoir comme pré-condition que ses deux paramètres sont des entiers 32bits et que son résultat est un entier 32bits. Cela peut ensuite exploser si, par exemple, cette fonction doit elle même renvoyer une fonction renvoyant un objet devant posséder une méthode dont le type de retour doit être compatible avec un type donné.

```
local function calculateur = |x,y,a| spec/
        requires{
            x:Int32 /\ y:Int32 /\
            hasmethod(a,                    # Nom de l'objet
                      additionneur |v1,v2|, # Méthode qu'il doit posser
                      v1:Int32 /\ v2:Int32 /\ result:Int32 # Spec de additionneur
                     )
        }
        ensures { (result :Int32) }
    /spec {
    return a:additionneur(x,y)
}
```

Fonction dont la spécification tente d'exprimer qu'un paramètre doit posséder une méthode respectant un certain cahier des charges.

FIGURE 2 – Exemple de pré-condition avec propriété sur l'objet reçu

2.4 Preuve de programme

Le langage Golo étant ce qu'il est, il ne nous semble pas raisonnable de croire que les développeurs Golo auront l'envie ou le besoin de prouver des pans entiers de programme. Cependant, l'objectif étant d'augmenter la confiance que l'on peut avoir en un code, il nous semble intéressant de permettre la preuve de parties de programme. Dans ce cas là, plusieurs

questions se posent. L'une d'entre elle est de savoir quel crédit accorder à un programme partiellement prouvé.

En nous inspirant de l'approche de Krakatoa [4], nous avons proposé la traduction d'un code Golo vers du code WhyML. En adressant ainsi l'outil Why, nous espérons pouvoir bénéficier de tout son écosystème.

Dans la version actuelle de nos développements [10], la traduction vers WhyML ne supporte que les fonctions, les types entiers et les conditionnelles simples. Les annotations de fonction sont préservées dans le WhyML produit, ce qui permet de prouver le respect de certaines propriétés. La figure 3 est un exemple de fonction pouvant bénéficier de cette vérification, où l'absence de la pré-condition ne permettrait plus de vérifier la post-condition.

```
function myAbs = |x| spec/
        requires{ x >= -2147483647 } # 2147483647 = MinInt+1
        ensures{ (result >= 0) /\ (result = x \/ result = -x) }
    /spec {
    if (x < 0){
        return (0 - x)
    } else {
        return x
    }
}
```

Fonction pouvant être prouvée et où l'oubli de la pré-condition pourra bloquer la preuve.

FIGURE 3 – Exemple de fonction Golo prouvable

3 État de l'art

Dans la littérature, le problème de la vérification de programmes décrits dans des langages dynamiques est assez largement abordée. Par exemple, les travaux Quist, et. al [1] portent sur la vérification de type par induction et la génération de cas de test pour des programmes dynamiques, décrits en Dart. Ils exploitent un mécanisme d'analyse statique pour obtenir des assurances de complétude sur le taux de couverture des tests produits.

De même, Csallner et Smaragdakis [3] ont proposé une approche de vérification de programmes Java combinant l'analyse statique et la génération de tests. Leur approche consiste à partir d'une sur-approximation des cas de test et de les élaguer par résolution de contraintes.

Ces différents résultats sont très intéressants par rapport à nos objectifs. Mais dans la plupart des cas, les langages dynamiques étudiés sont polymorphes de niveau 1 (prenex). C'est notamment le cas des langages Dart ou Java. À l'inverse, le polymorphisme de Golo, permettant l'extension dynamique, est de niveau N, comme pour Lisp, JavaScript, Python, Self ou Ruby.

Le problème d'inférence de type d'un système 2-polymorphe a été prouvé impossible dans le cas général par Schubert [14]. Mais sans information de typage, les vérifications statiques seront impossibles. Parmi les solutions proposées pour aborder ce problème, nous avons identifié la thèse de Don Stewart [15], qui propose notamment de décomposer le programme à vérifier en fragments, puis à vérifier dynamiquement la compatibilité de type entre eux.

L'analyse proposée dans leur outil est faite en plusieurs passes (certaines statiques et d'autres dynamiques), mais c'est finalement dynamiquement que l'analyse finale est faite.

Une autre approche utilisée pour vérifier statiquement le respect d'une politique de sécurité par un programme Javascript consiste à considérer un sous-ensemble vérifiable du langage [7]. Nous ne devrons pas négliger cette solution, par exemple pour permettre l'usage d'outils de preuve de programme sur des parties du système qui seraient identifiées comme critiques.

4 Conclusion et perspectives

Notre objectif n'est pas d'avoir un outil permettant de prouver intégralement un programme Golo. Non seulement cela serait impossible, mais en plus, cela ne correspondrait pas aux besoins liés à son usage. Notre objectif est de permettre aux développeurs de renforcer leur confiance en leur code. Pour ce faire, nous voulons pouvoir exploiter les avantages des différentes approches explorées.

Ainsi, un concepteur pourrait rapidement lancer des tests structurels de son code. Puis après ajout de spécifications, pourrait vérifier des éléments de typage et des tests fonctionnels. Enfin, le développeur pourra vouloir garantir différentes propriétés particulières sur certaines parties de son code par de la preuve. Afin de pouvoir cumuler les différentes approches en laissant la main au développeur, il sera important que nous ayons un moyen de produire un bilan de santé de l'application vérifiée, qui listerait l'ensemble des actions réalisées avec succès et l'ensemble des parties peu ou pas vérifiées. Par ailleurs, si ce rapport contient une mémoire des actions réalisées et des résultats obtenus, alors il serait intéressant de permettre de rejouer les actions faites, dans une démarche de non-régression.

Actuellement, presque tout reste à faire. Seuls de petits prototypes de chaque piste ont été expérimentés et nous sommes preneurs de tous les avis. Notamment, nous n'avons pas trouvé dans la littérature de manière propre de décrire des propriétés faisant référence à des éléments de type d'une donnée plutôt qu'à son type. L'objectif serait notamment d'exprimer une propriété que devrait remplir une fonction prise en paramètre, ou de spécifier une fonction ajoutée durant l'exécution à une variable.

Références

[1] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. Analyzing test completeness for dynamic languages. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 142–153. ACM, 2016.

[2] Cristian Cadar and Koushik Sen. Symbolic execution for software testing : three decades later. *Communications of the ACM*, 56(2) :82–90, 2013.

[3] Christoph Csallner and Yannis Smaragdakis. Check 'n' crash : Combining static checking and testing. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 422–431, New York, NY, USA, 2005. ACM.

[4] Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In *International Conference on Computer Aided Verification*, pages 173–177. Springer, 2007.

[5] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[6] Patrice Godefroid. DART : Directed Automated Random Testing. In *conference on Programming language design and implementation (PLDI '05)*, pages 213–223, 2005.

[7] Salvatore Guarnieri and V Benjamin Livshits. Gatekeeper : Mostly static enforcement of security and reliability policies for javascript code. In *USENIX Security Symposium*, volume 10, pages 78–85, 2009.

[8] Narendra Jussien, Guillaume Rochart, and Xavier Lorca. Choco : an Open Source Java Constraint Programming Library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Constraint Programming (OSSICP'08)*, pages 1–10, Paris, France, 2008.

[9] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7) :385–394, 1976.

[10] Raphael Laurent. Hardened Golo : Donnez de la confiance en votre code Golo. Master's thesis, INSA Lyon, June 2016.

[11] Yannick Loiseau. API for unified tests in Golo. https ://github.com/golo-lang/rigolo/blob/master/RiGolo/accepted/tests-api.adoc, 2016.

[12] Baptiste Maingret, Frédéric Le Mouël, Julien Ponge, Nicolas Stouls, Jian Cao, and Yannick Loiseau. Towards a decoupled context-oriented programming language for the internet of things. In *Proceedings of the 7th International Workshop on Context-Oriented Programming, COP 2015, Prague, Czech Republic, July 4-10, 2015*, pages 7 :1–7 :6. ACM, 2015.

[13] Julien Ponge, Frédéric Le Mouël, and Nicolas Stouls. Golo, a dynamic, light and efficient language for post-invokedynamic jvm. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform : Virtual Machines, Languages, and Tools*, pages 153–158. ACM, 2013.

[14] Aleksy Schubert. Second-order unification and type inference for church-style polymorphism. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 279–288. ACM, 1998.

[15] Don Stewart. *Dynamic extension of typed functional languages*. PhD thesis, PhD thesis, School of Computer Science and Engineering, University of New South Wales, 2010.

Sur l'assignation de buts comportementaux à des coalitions d'agents

Christophe Chareton
LORIA CNRS
54506 Vandœuvre-lès-Nancy

Julien Brunel
ONERA/DTIS
31055 Toulouse cedex 2

David Chemouil
ONERA/DTIS
31055 Toulouse cedex 2

Résumé

Dans cet article, nous présentons un cadre de modélisation formelle pour l'ingénierie du besoin qui prenne simultanément en compte les buts comportementaux et les agents. Pour ce faire, nous introduisons un langage noyau, appelé KHI, ainsi que sa sémantique dans une logique de stratégies appelée USL. Dans KHI, les agents sont décrits par leurs capacités et les buts sont définis par des formules de logique temporelle linéaire. Une « assignation » associe alors chacun des buts à un ensemble (une coalition) d'agents, qui sont responsables de sa satisfaction. Nous présentons et discutons ensuite différents critères de correction pour cette relation d'assignation. Ceux-ci permettent d'évaluer la « pertinence » d'une assignation de buts à des coalitions. Ils diffèrent selon les interactions qu'ils permettent entre les coalitions d'agents. Nous proposons alors une procédure décidable de vérification pour la satisfaction des critères de correction pour l'assignation. Elle consiste à réduire la satisfaction des critères à des instances du problème de *model-checking* pour des formules d'USL dans une structure dérivée des capacités des agents.

1 Contexte

Si, en toute rigueur, la discipline de la modélisation du besoin ne se restreint pas à elles seules [17, 14], les approches dites par buts [18] ou par agents [2, 9] ont le vent en poupe dans la communauté idoine (cf. les citations précédentes mais aussi [12, 15]).

En KAOS [18], la question première est de déterminer les besoins dont il faut tenir compte pour rendre compte d'un *système* au sein d'un *environnement,* le tout formant un *système global* à mettre au point. Celui-ci doit répondre à des *buts* et est constitué *d'agents* (entités actives).

Un *but* est défini comme un *énoncé prescriptif* sous la responsabilité d'agents du système global. Les buts peuvent être de toutes sortes (on retrouve les traditionnelles taxonomies autour des buts *non-fonctionnels* [11]). Mais on distingue en particulier les buts *comportementaux* qui caractérisent des traces et peuvent donc faire l'objet d'une formalisation dans une logique temporelle telle que LTL.

Bien que partageant superficiellement de nombreuses notions avec KAOS, TROPOS se concentre avant tout sur la notion *d'acteur,* défini comme un agent *intentionnel*. Un tel agent est muni de buts qu'il souhaite voir remplis mais dont la satisfaction, partielle comme complète, n'est pas nécessairement de sa responsabilité. Celle-ci peut être déléguée à d'autres acteurs. TROPOS [2] pousse ainsi à l'explicitation des liens de dépendance et de collaboration entre acteurs. Ceci s'explique en particulier par le fait que les systèmes visés par la méthode sont susceptibles de comprendre des acteurs « humains » ou institutionnels.

TROPOS a aussi fait l'objet d'une proposition formelle visant à étudier dans quelle mesure des acteurs peuvent contribuer à satisfaire des buts pour d'autres acteurs. L'approche en question [9, 10] introduit à cette fin les notions, dites « sociales », de *rôle, d'engagement (commitment)* et de *protocole.* Le rôle représente le comportement *attendu* des acteurs. Une *assignation* de rôles à des acteurs est alors évaluée au moyen d'un critère de correction. Celui-ci revient essentiellement à vérifier que les capacités d'un acteur entraînent les conséquents des engagements où le rôle assigné apparaît comme débiteur.

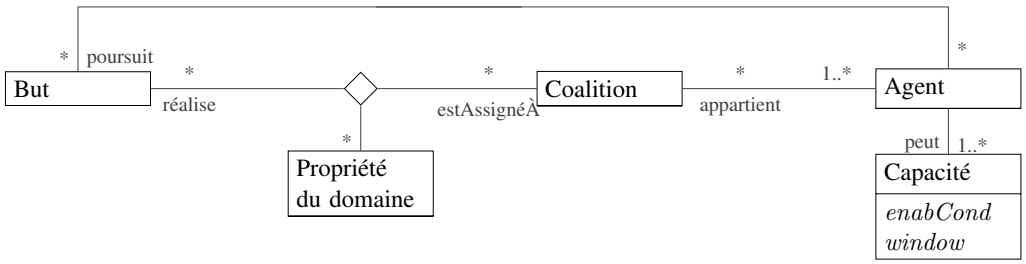


FIGURE 1 – Métamodèle du langage KHI.

Si la dimension « sociale » de cette proposition est plus forte qu'en KAOS, un certain nombre de faiblesses demeurent. Premièrement, on ne peut pas assigner de rôle à un ensemble d'acteurs qui collaboreraient pour l'assurer. Comme en KAOS, la question du partage de variables d'état entre plusieurs acteurs n'est pas traitée ; et les capacités des acteurs demeurent indépendantes du contexte. Surtout, la restriction du formalisme à la logique propositionnelle constitue une lacune importante : l'emploi d'un formalisme temporel, à l'image de ce qui est fait en KAOS, permettrait de véritablement caractériser des comportements attendus des acteurs.

Ces considérations nous ont mené à l'étude formelle ce que nous avons appelé le *problème de l'assignation.* Informellement, étant donnée une assignation de buts à des coalitions (ensembles) d'agents, la question est de déterminer dans quelle mesure ces dernières sont en mesure de satisfaire les premiers, y compris en tenant compte des interactions avec les autres coalitions. Cette question nous a amenés à définir un langage de modélisation appelé KHI et à élaborer une logique temporelle multi-agent, USL, fournissant un domaine sémantique permettant de formaliser et d'évaluer le problème de l'assignation selon différents critères.

2 KHI

Le cadre de modélisation KHI (figure 1) comprend un ensemble restreint de concepts destinés à permettre la mise en œuvre d'une modélisation du besoin *à la fois* par buts et par agents. Il peut en un sens être vu comme une union enrichie (des aspects principaux) de KAOS et des variations autour de TROPOS. Une première version de KHI a été introduite dans [4, 5] ; nous présentons ici une version simplifiée d'une seconde version, plus aboutie et décrite en détail dans [3].

Comme en TROPOS, les buts sont poursuivis par des agents, possiblement mais pas nécessairement aptes à contribuer à leur satisfaction. Les agents disposent de capacités plus fines que dans les propositions précédentes : une capacité ne peut s'exercer que dans certaines conditions $(enabCond)$ et consiste à pouvoir agir sur une variable du système mais seulement dans une certaine fenêtre $(window)$, c'est-à-dire dans un ensemble fini de valeurs possible. Originalité supplémentaire : les agents sont regroupés en coalitions (ensembles) qui se voient assigner les buts à remplir (autrement dit, l'assignation n'est pas restreinte à des agents isolés).

Par ailleurs, la formalisation des buts suit le même principe qu'en KAOS : les buts sont décrits dans la logique LTL. Ces formules de LTL apparaîtront comme sous-formules de formules de la logique USL sur laquelle s'appuie la formalisation complète de KHI. Or cette logique repose sur des modèles admettant des exécutions finies. La sémantique utilisée pour LTL pour le critère de correction du raffinement des buts par des sous-buts et de l'opérationnalisation s'appuie donc sur des traces possiblement finies (et non-vides), de manière par ailleurs standard (techniquement comme la logique LTL^n d'[13]).

L'existence d'exécutions finies résulte du choix de modélisation de l'assignation entre buts et coalitions d'agents : celle-ci est définie comme une application de l'ensemble des buts dans les coalitions non vides d'agents (donc de type But $\to \mathscr{P}_{>0}(\text{Agent})$). On remarque qu'il est tout à fait possible pour un agent de se retrouver dans plusieurs coalitions. Dès lors, la participation de cet agent à ces différentes coalitions peut induire des spécifications contradictoires pour son comportement. Cet agent pourra alors

être engagé envers la réalisation d'actions *incompatibles* entre elles, depuis un même état. La présence d'exécutions finies dans notre formalisme provient donc du fait que nous avons souhaité favoriser la description fine des capacités des agents à composer leurs comportements pour la satisfaction de différents buts. Par ailleurs, il faut noter que notre formalisme permet ensuite de spécifier des modèles où les exécutions sont forcément infinies.

3 Problème de l'assignation

L'évaluation d'une assignation ne se fait pas de manière binaire. Un ensemble de critères permet de la caractériser plus finement ; en particulier en déterminant si des coalitions peuvent interagir de sorte à voir assurée la satisfaction de leurs buts :

Correction locale Ce critère consiste à s'assurer que chaque but est assigné à une coalition capable d'assurer sa satisfaction, quoi que fassent les autres agents.

Correction globale Le critère de correction locale est insuffisant dès lors qu'un agent appartient à plusieurs coalitions, car rien ne dit qu'il est apte à participer à toutes ces coalitions (en vue de réaliser leurs buts respectifs) *à la fois* (les comportements demandés pourraient être contradictoires). Le critère de *correction globale* demande donc s'il y a *un* comportement (par agent) qui permette à chaque coalition de satisfaire les buts qui lui sont assignés. Cet unique comportement permet de s'assurer de la cohérence des choix des agents.

Collaboration Le critère précédent peut être trop fort au sens où il stipule que chaque coalition doit pouvoir satisfaire ses buts assignés, quoi que fassent les autres agents. Or on peut souhaiter que certaines coalitions *collaborent* avec d'autres pour assurer la satisfaction de leurs buts.

Contribution Les trois critères précédents reposent sur l'hypothèse qu'il est possible de contrôler (et spécifier) tous les agents dans le système. Or il se pourrait que ce ne soit pas le cas. Le critère de *contribution* pose alors la question de savoir si une coalition indépendante, en satisfaisant ses propres buts assignés, est en mesure de produire des « effets de bord » qui contribuent à la satisfaction des buts assignés à d'autres coalitions.

Ces critères sont tous traduits en problèmes de *model-checking* dans la logique USL. On aboutit donc à des problèmes de la forme $\mathfrak{K} \models c$, où c est une formule d'USL traduisant un critère et où $\mathfrak{K}$ est une CGS *(Concurrent Game structure)*, soit un système de transitions particulier.

Intuitivement, les états de la CGS $\mathfrak{K}$ sont déterminés par les valuations possibles des variables décrivant le système (et compatibles avec les propriétés du domaine). En ce qui concerne la fonction de transition, les états suivants d'un état donné dépendent de celui-ci ainsi que de l'intersection des choix des agents, choix eux-mêmes déterminés par leurs capacités (pour cette raison, la fenêtre d'action d'un agent doit être exprimée comme un ensemble fini de valeurs possibles pour une variable donnée).

Pour la formule c, il s'agit à chaque fois d'exprimer que des coalitions d'agents sont en mesure d'assurer la satisfaction de propriétés temporelles, en tenant compte du fait que les autres coalitions agissent elles-aussi et qu'un agent peut appartenir à plusieurs coalitions. Les coalitions pouvant poursuivre des objectifs variés, il faut pouvoir spécifier qu'un agent peut enrichir de *différentes* manières son comportement selon les objectifs auxquels il concourt.

4 Logique des stratégies actualisables

Ces motifs mènent naturellement vers les *logiques temporelles multi-agents*. Toutefois, aucune proposition ne présentant les caractéristiques nécessaires [1] à l'élaboration des critères, nous avons défini notre propre logique : USL. Dans les logiques telles qu'ATL [1] ou SL [8, 16], chaque agent est muni d'une *stratégie* qui, en fonction du déroulement du jeu jusqu'alors, indique quelle action il choisit, ce

1. En particulier, donc, la possibilité pour un agent de suivre diverses « stratégies » et de les raffiner de diverses façons.

qui concourt à établir une décision et donc à déterminer l'état successeur. Eu égard à KHI, cette construction contribue à matérialiser la notion selon laquelle un agent a le moyen (lire : une stratégie) d'assurer une propriété. Toutefois, ici, en conformité avec les critères d'assignation, il est nécessaire de pouvoir composer des stratégies de plusieurs manières différentes. Dans USL, un agent auquel est assigné une stratégie ς peut se voir à nouveau assigner une autre stratégie ς'. Il composera alors son comportement de manière à satisfaire à la fois ς et ς'. On dit qu'il *raffine* sa stratégie. Ceci nous a menés, pour USL, à l'utilisation de stratégies *non-déterministes :*

Définition 4.1 (Multistratégie)**.** *Une* multistratégie *est une application qui, à tout déroulement*[2] *d'un jeu, associe un ensemble non-vide d'actions.*

Nous sommes maintenant en mesure de définir une syntaxe pour USL, avec les contraintes suivantes : *(a)* pouvoir raisonner sur les multistratégies, ce qui mène à l'introduction de modalités *ad hoc* ; *(b)* pouvoir se référer à des multistratégies données, d'où une notion de *variable* associée (et donc de *quantificateur*) ; *(c)* pouvoir composer, pour un même agent, plusieurs multistratégies différentes dans des sous-formules distinctes, et ce sans *révoquer* les multistratégies déjà associées, ce qui nécessite des opérateurs liant ou révoquant explicitement une multistratégie à un agent.

Définition 4.2 (Formules d'USL)**.** *Soient Ag un ensemble d'agents, At un ensemble de propositions, et X un ensemble de variables de multistratégies. Alors, l'ensemble des* pseudoformules d'USL *sur (Ag, At, X) est engendré par la grammaire suivante :*

— *Pseudoformules d'états :* $\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \langle\langle x \rangle\rangle \varphi \mid (A \rhd x)\psi \mid (A \not\rhd x)\psi$

— *Pseudoformules de chemins :* $\psi ::= \varphi \mid \neg\psi \mid \psi \wedge \psi \mid \mathsf{X}\ \psi \mid \psi\ \mathsf{U}\ \psi$

où $p \in At$, $x \in X$ et $A \subseteq Ag$.

Une formule (bien formée) *d'*USL *est une pseudoformule dans laquelle chaque variable de multistratégie introduite par un quantificateur est fraîche par rapport à la portée dans laquelle elle apparaît.*

Naturellement, lors de la détermination de la possible satisfaction d'une formule dans une CGS, la présence de quantificateurs et d'opérateurs de liaison et révocation doit être prise en compte dans un *contexte* (d'évaluation). Un tel contexte κ est un couple comprenant : *(a)* une *assignation* α, qui a pour objet de mémoriser la multistratégie effective associée à chaque variable liée rencontrée jusqu'alors ; *(b)* et un *engagement* γ, c'est-à-dire l'inventaire, pour chaque agent, des (variables de) multistratégies selon lesquelles il joue. Ce couple doit par ailleurs être cohérent : toute variable de multistratégie listée dans l'engagement doit disposer d'une instanciation, donnée par l'assignation.

Nous renvoyons le lecteur à [6] pour une exposition plus fine, qui fait appel à un certain nombre de détails techniques, mais nous donnons ici une intuition des points saillants de la sémantique :

— L'opérateur $\langle\langle x \rangle\rangle$ est un quantificateur existentiel sur les multistratégies : une formule $\langle\langle x \rangle\rangle \varphi$ est vraie dans un état s, dans un contexte κ, ssi il existe une multistratégie ς t. q. la formule φ est vraie dans le contexte κ enrichi du fait que x est instanciée par ς. On peut aussi définir un quantificateur universel de la sorte : $[\![x]\!]\varphi \coloneqq \neg\langle\langle x \rangle\rangle\neg\varphi$.

— Une formule $(A \rhd x)\psi$ est vraie dans un état s, dans le contexte κ, ssi la formule ψ est vraie dans toute exécution *issue* de s et $\kappa[A \oplus x]$, où $\kappa[A \oplus x]$ est le contexte κ enrichi du fait que les agents dans A sont maintenant liés à la multistratégie instanciant x dans κ (en plus des multistratégies auxquelles ils étaient éventuellement déjà liés).

Les *issues* $out(\kappa, s)$ d'un état dans un certain contexte constituent les exécutions possibles à partir de cet état dans la CGS si chaque agent ne joue que des actions autorisées par toutes les multistratégies auxquelles il est lié dans ce contexte.

Il faut remarquer que parmi ces issues, certaines exécutions peuvent être finies, ce en raison de la possibilité pour un agent de jouer en même temps selon des multistratégies contradictoires (c'est-à-dire renvoyant des ensembles disjoints d'actions). Ceci implique l'usage d'une sémantique idoine pour les opérateurs temporels dans les formules de chemin.

2. C'est-à-dire un préfixe non-vide d'exécution dans la CGS.

— $(A \not\rhd x)$ délie les agents dans A de la multistratégie instanciant x dans le contexte courant.

Theorème 4.1 ([6]). *Pour finir, on remarque que :* (a) USL *est strictement plus expressive que* SL *[8, 16] ;* (b) *comme pour cette dernière, la satisfaisabilité est indécidable ;* (c) *en revanche, le* model-checking *sur une* CGS finie [3] *est décidable, quoique en temps non-élémentaire, ce qui est aussi le cas de* SL.

5 Évaluation d'une assignation

Les caractéristiques uniques d'USL (en particulier le raffinement de multistratégies) permettent de formaliser les critères de correction présentés en § 3. On ne présente pas ici la procédure (complexe) de construction d'une CGS *finie* à partir d'un modèle KHI, cf. [3, 7]. Cette finitude dépend en particulier du type de formules atomiques et de la forme des fenêtres que KHI permet de spécifier. Elle rend décidable la vérification des critères ci-dessous.

Soit un modèle KHI et soit G un ensemble de buts. On note $\mathscr{A}$ l'application qui à tout but associe la coalition qui lui est assignée. Étant donnée une coalition $A = \{a_1, \ldots, a_n\}$ et $\overrightarrow{x}$ un vecteur de variables de multistratégies ; on note $(A \rhd \overrightarrow{x})$ pour $(a_1 \rhd x^{a_1}) \ldots (a_n \rhd x^{a_n})$.

Correction locale L'assignation est localement correcte ssi pour tout but $g \in G$, il existe un vecteur de multistratégies t. q., en les jouant, les agents concernés peuvent assurer la satisfaction de g :

$$\mathrm{LC}_{\mathscr{A}}[G] \coloneqq \bigwedge_{g \in G} \left[\langle\!\langle \overrightarrow{x_g} \rangle\!\rangle (\mathscr{A}_g \rhd \overrightarrow{x_g}) g\right]$$

Correction globale Ici, on considère un unique vecteur de multistratégies, ce qui impose aux agents de jouer en cohérence ($\overrightarrow{x_g}$ représente la partie de $\overrightarrow{x}$ qui concerne les agents dans $\mathscr{A}_g$) :

$$\mathrm{GC}_{\mathscr{A}}[G] \coloneqq \langle\!\langle \overrightarrow{x} \rangle\!\rangle \Big[\bigwedge_{g \in G} (\mathscr{A}_g \rhd \overrightarrow{x_g}) g \Big]$$

Collaboration Pour qu'une coalition associée à un but h collabore à la satisfaction de l'ensemble de buts G, il faut un vecteur de multistratégies $\overrightarrow{x_h}$ t. q. si les agents dans $\mathscr{A}_h$ les jouent, alors ces multistratégies assurent *à la fois* que le but h est satisfait et que l'évolution du modèle est contrainte de sorte à ce que l'assignation devienne globalement correcte pour G. Soit donc h un but t. q. $h \notin G$. :

$$\mathrm{Coll}_{\mathscr{A}}[h, G] \coloneqq \langle\!\langle \overrightarrow{x_h} \rangle\!\rangle (\mathscr{A}_h \rhd \overrightarrow{x_h}) (h \wedge \mathrm{GC}_{\mathscr{A}_G}[G])$$

Contribution La contribution est définie comme une variante universellement quantifiée de la collaboration ; les agents concernés doivent pouvoir assurer h et tout vecteur de multistratégies qui permet aux dits agents d'assurer h doit aussi contraindre l'évolution du système de sorte à ce que l'assignation devienne globalement correcte pour G :

$$\mathrm{Contr}_{\mathscr{A}}[h, G] \coloneqq \left[\langle\!\langle y_h \rangle\!\rangle (\mathscr{A}_h \rhd \overrightarrow{y_h}) h\right] \wedge \left[[\![\overrightarrow{x_h}]\!] \Big\{ (\mathscr{A}_h \rhd \overrightarrow{x_h}) h \rightarrow \big((\mathscr{A}_h \rhd \overrightarrow{x_h})\, \mathrm{GC}_{\mathscr{A}_G}[G] \big) \Big\} \right]$$

6 Perspectives

KHI constitue à notre connaissance la première proposition de cadre de modélisation *formelle* pour l'ingénierie du besoin qui prenne simultanément en compte les buts comportementaux et les agents.

Ceci étant, le cadre doit clairement être amélioré. Tout d'abord, la sémantique de KHI ne fait pas appel à toute l'expressivité d'USL. Une question importante en termes d'applicabilité de nos propositions serait de déterminer s'il est possible de caractériser un fragment d'USL suffisant pour la traduction de KHI et pour lequel le *model-checking* disposerait d'une complexité « raisonnable » en pratique.

3. C'est-à-dire t. q. les ensembles d'états et d'actions sont finis.

De son côté, USL constitue aussi une proposition originale. Elle bénéficie par ailleurs de propriétés métathéoriques comparables aux logiques similaires et dispose d'un fort pouvoir expressif dont il s'agirait de creuser les conséquences en profondeur.

Hors de l'ingénierie du besoin proprement dite, USL pourrait bien avoir des applications intéressantes. Ainsi de l'analyse de « systèmes de systèmes », ensembles dans lesquels, en particulier, les sous-systèmes apparaissent comme des entités autonomes dont les capacités sont connues et qui disposent de leurs objectifs propres en sus de ceux du système global. Une autre application possible réside dans la sécurité : en effet, USL permet en principe de raisonner sur des agents dont les objectifs sont malveillants.

Pour finir, USL permet de raisonner sur le comportement (les multistratégies) possible des agents. Une question supplémentaire serait celle du comportement *effectif* du système. Cette question appelle certainement plusieurs directions de recherche autour de la *synthèse* de contrôleurs, de normes, de stratégies...

Références

[1] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5) :672–713, 2002.

[2] J. Castro, M. Kolp, and J. Mylopoulos. A requirements-driven development methodology. In *Proceedings of the 13th International Conference on Advanced Information Systems Engineering*, CAiSE '01, pages 108–123, London, UK, UK, 2001. Springer-Verlag.

[3] C. Chareton. *Modélisation formelle d'exigences et logiques temporelles multi-agents*. PhD thesis, Institut supérieur de l'aéronautique et de l'espace (ISAE), Université de Toulouse, 2014.

[4] C. Chareton, J. Brunel, and D. Chemouil. A Formal Treatment of Agents, Goals and Operations Using Alternating-Time Temporal Logic. In *Formal Methods, Foundations and Applications (SBMF)*, 2011.

[5] C. Chareton, J. Brunel, and D. Chemouil. Vers une sémantique des jeux pour un langage d'ingénierie des exigences par buts et agents. In *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL)*, 2012.

[6] C. Chareton, J. Brunel, and D. Chemouil. A Logic with Revocable and Refinable Strategies. *Information and Computation*, 242 :157–182, 2015.

[7] C. Chareton, J. Brunel, and D. Chemouil. Evaluating the assignment of behavioral goals to coalitions of agents. In *Proc. 18th Brazilian Symposium, SBMF 2015, Belo Horizonte, Brazil, 2015*. Springer, 2015.

[8] K. Chatterjee, T. A. Henzinger, and N. Piterman. Strategy logic. *Inf. Comput.*, 208(6) :677–693, 2010.

[9] A. K. Chopra, F. Dalpiaz, P. Giorgini, and J. Mylopoulos. Modeling and reasoning about service-oriented applications via goals and commitments. In *Proceedings of the 22nd International Conference on Advanced Information Systems Engineering (CAiSE)*, volume 6051 of *LNCS*, pages 113–128. Springer, 2010.

[10] A. K. Chopra and M. P. Singh. Multiagent commitment alignment. In *Proceedings of the Eighth International Conference on Autonomous Agents and Multiagent Systems*, pages 937–944. IFAAMAS, 2009.

[11] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 2000.

[12] E. Dubois, P. Du Bois, and M. Petit. ALBERT : An agent-oriented language for building and eliciting requirements for real-time systems. In *HICSS (4)*, pages 713–722, 1994.

[13] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. V. Campenhout. Reasoning with temporal logic on truncated paths. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV)*, pages 27–39, 2003.

[14] M. Jackson. *Problem Frames*. Springer, 2001.

[15] T. P. Kelly and R. A. Weaver. The goal structuring notation - a safety argument notation. In *Proceedings of the Dependable Systems and Networks 2004 Workshop on Assurance Cases*, 2004.

[16] F. Mogavero, A. Murano, and M. Y. Vardi. Reasoning about strategies. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2010*.

[17] B. Nuseibeh and S. M. Easterbrook. Requirements engineering : a roadmap. In *22nd International Conference on Software Engineering, Future of Software Engineering Track*, 2000.

[18] A. van Lamsweerde. *Requirements Engineering — From System Goals to UML Models to Software Specifications*. Wiley, 2009.

Un Data-Store pour la Génération de Cas de Test

Loukmen Regainia, Cédric Bouhours et Sébastien Salva
LIMOS - UMR CNRS 6158, Auvergne University, France
Email : [prenom.nom]@uca.fr

Résumé

De nos jours, un grand nombre de documents publiques de sécurité sont disponibles. Ces documents, souvent complexes, exposent les développeurs à la difficulté de choisir des solutions de sécurité appropriées d'une application tout au long de son cycle de vie. Dans cet article, nous proposons une méthodologie, basée sur l'acquisition et l'intégration des données afin de construire un data-store permettant de classifier des patrons de sécurité et des attaques et de générer des Arbres d'attaques de défenses. Nous présentons également une méthodologie qui permet l'extension de ce data-store dans le but de générer automatiquement des cas de test.
Mots clés : Patrons de sécurité ; Patrons d'attaque ; Cas de test ; Arbres d'attaque et de défense

1 Introduction

La conception et l'implémentation d'une application sécurisée est une tâche difficile. D'une part la sécurité dès la phase de conception n'est pas une compétence courante dans les équipes de développement, et d'autre part la détection des problèmes de sécurité aux dernières étapes de la vie de l'application est compliquée à la vue de la difficulté pour couvrir l'ensemble des vulnérabilités connues.

Afin d'aider les équipes de développement à gérer la sécurité des applications dès les premières phases du cycle de vie, les concepteurs ont à leur disposition la notion de patrons de sécurité [1] qui sont des solutions communautaires génériques aux problèmes récurrents de sécurité permettant de prévenir partiellement ou complètement des attaques. La nature abstraite et le nombre croissant des patrons de sécurité permet de couvrir une myriade de problématiques de sécurité indépendamment des contextes d'application. Néanmoins, cette représentation abstraite et le nombre de patrons disponibles rend le choix difficile.

Dans cet article, nous présentons premièrement une classification des patrons de sécurité pour faciliter leurs choix face aux attaques. Cette classification prend en compte plusieurs critères de qualité [2] : la Non-ambiguïté, vu que nous présentons en détail la démarche de classification . Elle prend également en compte la Navigabilité, en utilisant les relations inter-patrons. Puis, Nous présentons une méthode des arbres d'attaques et de défense (ADTree), permettant de présenter les actions d'attaque et de défense,

à partir de la classification obtenue. Nous présentons finalement une méthodologie de génération de squelettes de cas de test permettant de tester si une application est vulnérable aux attaques, et si des patrons de sécurité sont présents dans une application en inspectant la présence des éléments caractérisants la bonne intégration des patrons (section “conséquences” du descriptif des patrons).

Ce papier est structuré comme suit : dans la Section 2, nous présentons les notions de sécurité utilisées pour générer la classification. Ensuite, nous présentons la méthode associant les attaques et les patrons de sécurité ainsi que la génération des ADTree dans la Section 3. Puis nous présentons l’extension de notre démarche dans le but de produire des squelettes de cas de test en Section 4. Enfin, nous concluons et nous proposons quelque perspectives en Section 5.

2 Contexte

Dans le contexte de nos travaux, nous utilisons différentes notions de sécurité telles que les notions des patrons de sécurité, des principes de sécurité [4], des patrons d’attaque [5] et des arbres d’attaques et de défense [3]. Nous en présentons quelque unes dans cette section. Un patron de sécurité est une solution générique à un problème récurrent de sécurité, caractérisée par un ensemble de propriétés structurelles et comportementales à utiliser dès la phase de conception. Un patron de sécurité peut être représenté textuellement et/ou avec des diagrammes UML [1]. Il est également caractérisé par un ensemble de points forts qui décrivent les sous propriétés du patron. Un exemple de patron de sécurité, "Authorisation Enforcer", vise à fournir un mécanisme d’autorisation et de gestion de droits à une application. Il centralise le mécanisme d’autorisation et le découple de la logique fonctionnelle de l’application [6].

Un patron d’attaque est une description des éléments et des techniques utilisées pour attaquer une application, ainsi que les challenges auxquels les concepteurs doivent faire face afin de protéger leur application. La base CAPEC (Common Attack Pattern Enumeration and Classification) offre une documentation des patrons d’attaque. Dans la base CAPEC, un patron d’attaque est composé d’un ensemble de sections, à savoir, les impacts et les conséquences de l’attaque, les contremesures, etc [5]. Pour construire notre data-store, nous nous sommes concentré sur la section *“Attack Execution Flow”*, qui décrit les étapes de l’attaque ainsi que les techniques, les indicateurs et les mesures de sécurité associées à chaque étape. De plus, nous nous sommes intéressés à la section "Attack Prerequisites" et aux ressources nécessaires pour effectuer l’attaque (section "Resources Required").

Parmi les méthodes qui permettent la modélisation des attaques, les arbres d’attaque et de défense (Attack Defense Tree “ADTree”)[3] représentent les mesures qu’un attaquant peut prendre pour attaquer un système ainsi que les défenses qui peuvent être utilisées pour protéger le système. Ce sont des arbres étiquetés où chaque nœud permet de présenter une action d’attaque ou de défense. Chaque nœud peut être raffiné par des nœuds-enfants qui peuvent être associés par des opérateurs “AND”, “OR” et “SAND”. Ce dernier est un opérateur conjonctif qui impose une relation d’ordre entre les actions.

En plus de l'aspect visuel d'un ADTree, il est fournie avec une algèbre qui permet de décrire formellement les étapes d'une attaque ainsi que les défenses associées avec des ADTerms. Un ADTree peut être exprimé avec un ensemble d'ADTerms qui permet de présenter deux types de rôles attaquant (opponent "o") et défenseur (proponent "p") ainsi que trois type d'opérateurs Conjonction ($\wedge^{o/p}$), Disjonction ($\vee^{o/p}$) et Conjonction ordonnée "SAND" ($\overrightarrow{\wedge}^{p/o}$) d'actions de défense ou d'attaque (o/p). La relation entre un nœud père et un nœud fils peut être une relation de *raffinements* si les deux nœuds sont du même type et une relation de *contre-mesure* sinon, on dénote cette dernière par $c^o(a, b)$ entre deux nœuds a, b.

3 La classification des patrons de sécurité et la génération d'ADTree

Les documents présentant des attaques ou des patrons de sécurité sont souvent décrits de façon informelle avec des niveaux différents d'abstraction. Ceci rend la liaison directe entre un patron de sécurité et une attaque souvent ambiguë et sujette à des imprécisions. Dans le but de réduire l'ambiguïté et l'imprécision, nous proposons une méthode qui a pour but la construction d'un data-store qui rassemble les attaques, issues de la base CAPEC (*Common Attack Patterns Enumeration and Classification*)[5], les patrons de sécurité, les principes de sécurité, et les relations inter-patrons issus d'un catalogue de patrons [1]. Ce data-store permet la génération d'arbres d'attaques et de défenses (ADTree) pour chaque attaque. La méthode, illustrée sur la figure 1.A, se déroule en quatre étapes.

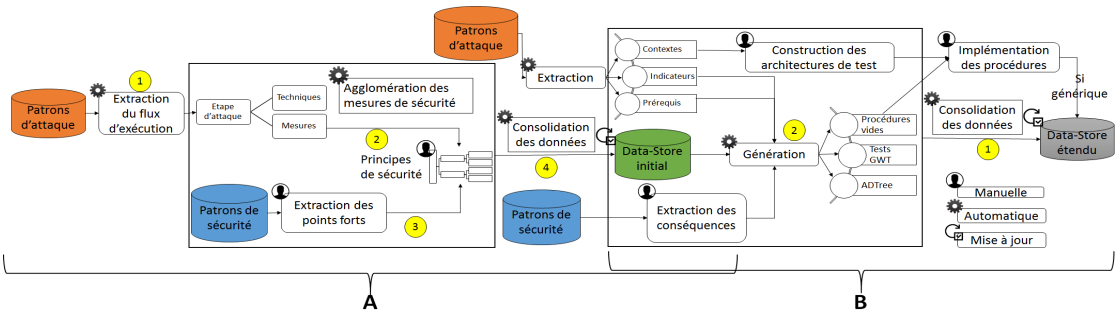


FIGURE 1 – Méthodologie

1- Extraction du flux d'exécution des attaques : depuis la base des patrons d'attaques CAPEC nous effectuons une extraction automatique de l'ensemble ordonné des étapes de chaque attaque, qui est défini dans la section *"Attack Execution Flow"* de la documentation des attaques. Cette section décrit les étapes que l'attaquant doit effectuer dans l'ordre ainsi que les différentes techniques qu'il doit exécuter pour satisfaire une étape, les indicateurs de succès de l'étape ainsi que des mesures de sécurité qui doivent protéger l'application. Nous avons extrait automatiquement 209 étapes différentes pour 215 attaques sachant que les attaques peuvent partager des étapes.

2- Extraction des principes de sécurité de chaque étape de l'attaque : chaque étape d'attaque est caractérisée par un ensemble de mesures de sécurité *"Security Controls"* (Détectives, Préventives et Correctives). Nous en avons extrait un ensemble de 217. Chaque mesure suit un principe élémentaire de sécurité (Autorisation, Défense en profondeur, Audit, etc) dont l'extraction manuelle peut être fastidieuse et rendre l'approche difficile à refaire. Nous avons utilisée une méthode issue du Data-mining avec l'outil KHCoder [1] pour agglomérer les mesures de sécurité selon leur appartenance au même ensemble de principes de sécurité. Les 217 mesures de sécurité ont été assemblées dans 21 groupes (clusters) cloisonnés. Cette méthode d'analyse de texte s'appuie sur trois notions statistiques :

— L'utilisation de "Stanford POS tagger" afin d'ordonner les mots clés *(log, input, credentials, etc.)* par rapport à leur fréquence dans le texte ainsi que leur type *(verbe, nom, adjectif)* ;

— Compte tenu des fréquences des mots clés, une matrice de distances entre les mesures de sécurité est établie moyennant la méthode "Jaccard". La distance entre deux mesures de sécurité est minimisée si les deux mesures contiennent plus de mots clés en commun ;

— Sur la base de la matrice de distances obtenue, nous avons groupé hiérarchiquement les mesures de sécurité moyennant la méthode "Ward" [7]. L'avantage de cette méthode est qu'elle privilégie la construction de niveaux de groupes étape par étape au lieu de la construction de grands groupes, qui peuvent couvrir beaucoup de principes de sécurité. La méthode "Ward" est une méthode supervisée imposant que le nombre de groupes soit choisi manuellement.

Concrètement, nous avons choisi le nombre de groupes, après une série de tests, de telle façon que les éléments d'un même groupe traitent de la même problématique de sécurité. basés sur un ensemble de travaux traitant de la notion de principes de sécurité [8, 9, 4], nous avons opté pour une organisation hiérarchique de 66 principes de sécurité afin d'améliorer leur compréhension et présenter les relations inter-principes.

Les éléments d'un groupe dans la classification obtenue sont proches dans le sens où ils traitent de la même problématique de sécurité. Cela facilite l'extraction manuelle des principes de sécurité traités par chaque groupe de mesures de sécurité. Nous avons ensuite lié chaque groupe de mesures de sécurité aux principes de sécurité qu'ils couvrent.

3- Association des principes de sécurité et des patrons de sécurité : Un patron de sécurité est caractérisé par un ensemble de propriétés appelées points forts définissant les raisons pour lesquelles un patron est une solution adaptée à un problème [10]. Les patrons de sécurité peuvent partager des points forts. Chaque point fort est caractérisé par un concept de sécurité que nous lions à un principe de sécurité.

4- Consolidation des données, Extraction de la classification et Génération des ADTree : afin de consolider les données, nous avons développé une série de flux de données sous "Talend" [2] qui alimentent le data-store suivant les associations établies dans les étapes précédentes.

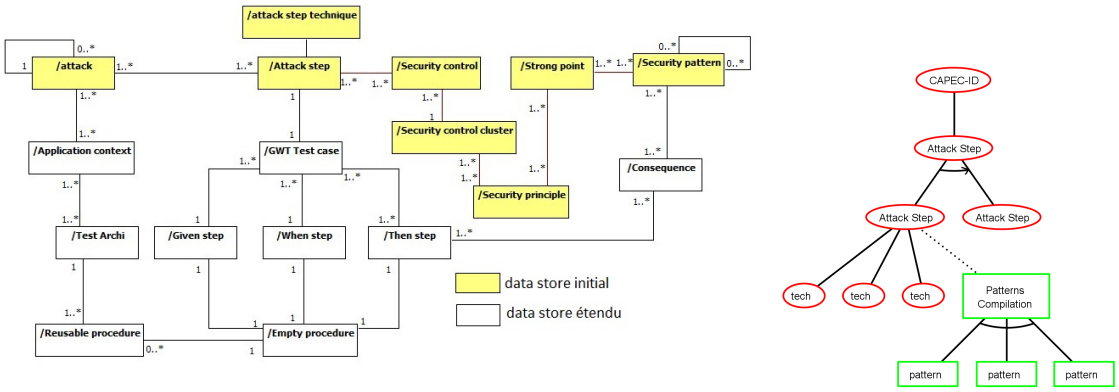
1. https ://sourceforge.net/projects/khc/
2. http ://fr.talend.com/

Le data-store obtenu contient l'ensemble des données nécessaires pour l'extraction des différents types de relations et de classifications. Le data-store exprime les liens entre les étapes des attaques et les principes de sécurité, les liens entre l'ensemble des patrons de sécurité et les principes de sécurité, ainsi que les relations inter-patrons de sécurité. Par conséquent, le data-store permet d'extraire, pour chaque attaque :

- les informations concernant l'attaque (identifiant, nom, présentation) ;
- l'ensemble ordonné des étapes et des sous-étapes ;
- les techniques qui permettent d'implémenter chaque étape ;
- les patrons de sécurité liés à chaque étape ainsi que les relations inter patrons ;

Nous avons utilisé ZOHO Reports [3] afin de rendre automatique l'extraction et la visualisation de la classification depuis le data-store sous la forme d'un tableau de bord interactif, ceci est disponible en ligne [11].

Dans le but de faciliter l'utilisation et l'analyse des ADTree, nous avons également développé un outil qui génère un ADtree pour chaque attaque présentée dans le data-store, la figure 2b illustre la forme générale des ADTree générés. Cet outil lit les informations concernant les étapes, les techniques, et les patrons de sécurité depuis le data-store et les présente sous la forme visuelle d'ADTree. Pour ce faire, ces informations sont traduites en un fichier XML que l'utilisateur peut le charger dans l'outil ADTool [4] qui permet l'affichage et l'analyse visuelle d'une attaque.



(a) data-store (b) exemple d'ADTree

FIGURE 2 – La structure des data-stores

4 L'extension du data-store pour le test

En plus de la classification des patrons de sécurité par attaques et la génération des ADTree, nous avons étendu le data-store dans le but de générer un cas de test Given, When, Then (GWT) pour chaque étape d'attaque. Ces cas de test ont pour objectif

3. https ://www.zoho.eu/reports/ ?src=zoho
4. http ://satoss.uni.lu/members/piotr/adtool/

de tester si une application est vulnérable a une attaque, et si on peut observer les conséquences de patron de sécurité depuis une application.

Un cas de test *GWT* est ainsi composé d'une section *Given*, exprimant les pré-requis et les éléments à préparer pour effectuer l'attaque, d'une section *When* présentant les actions à effectuer pour concrétiser l'attaque et d'une ou plusieurs sections *Then* présentant les conditions de succès de l'attaque (verdicts).

Nous avons associé les patrons de sécurité à des conséquences (qui peuvent être partagées entre patrons). Chaque conséquence est aussi associée à une section Then, permettant de vérifier si la conséquence est observable ou non depuis l'application sous test.

Chaque section Given, When, Then est liée à une procédure qui implémente la section du test.

Nous avons généré automatiquement l'ensemble de ces éléments en intégrant les données dans le data-store puis en générant les cas de test et les procédures commentées selon la méthode, illustrée dans la figure 1.B, décrite ci-après :

4.1 Data-store étendu

Depuis la base CAPEC nous avons extrait automatiquement pour chaque attaque, contenu dans la data-store précédant, l'ensemble d'éléments qu'il faut satisfaire préalablement à l'exécution de l'attaque. Les sections *"Attack Prerequisites"* et *"Resources Required"* du CAPEC décrivent ce qu'il faut préparer pour chaque attaque. Nous avons également, depuis la section *"Attack Step Techniques"* de chaque étape, extrait automatiquement les différentes façons dont une étape peut être implémentée. Ainsi que l'ensemble d'éléments qui permettent de décider si l'attaque est bien réussie depuis les sections *"Indicators"* et *"Outcomes"* de chaque étape. Ensuite, nous avons extrait les différents **contextes** dans lesquels une attaque est applicable depuis la partie "Environments" de chaque technique d'étape d'attaque. Chaque contexte implique une **architecture de test** différente regroupant un ensemble d'outils permettant de tester les application de ce contexte.

La présence d'un patron de sécurité dans une application peut être résumée par un ensemble de conséquences sur sa structure et son comportement. Cet ensemble d'informations, présenté dans la section *"Consequences"* de la documentation des patrons de sécurité. Nous avons extrait manuellement ces informations depuis la documentation des patrons de sécurité.

Nous avons généré un cas de test GWT pour chaque étape d'attaque ainsi qu'une procédure pour chaque section Given, When Then. Chacune de ces procédures est complété par des directives, sous forme de commentaires. Les directives fournies sont issues du data-store et sont réparties comme suit :

— pour les procédures liées aux sections Given, les directives sont extraites depuis les pré-requis et les ressources nécessaires de l'attaque ;
— pour les procédures When, les directives sont extraites depuis les techniques de chaque étape ;
— pour les procédures Then, on distingue deux cas

1. pour les assertions de succès de l'étape, les directives sont issues des indicateurs de succès de chaque étape ;
2. pour les assertions d'observations de présence des patrons de sécurité, les directives sont issues des informations concernant les conséquences des patrons de sécurité ;

Finalement, nous avons consolidé l'ensemble de ces éléments dans un data-store qui contient les informations nécessaires permettant, en plus de la classification des patrons de sécurité et des attaques, la générations des cas de test comme illustré en figure 1.

4.2 Génération des squelettes de cas test

Depuis le data-store que nous avons étendu, nous générons un ADTree pour chaque attaque donnée par un utilisateur.

De façon résumé, la génération des cas de test depuis le data-store s'effectue comme suit :

1. L'utilisateur fournit une liste d'attaques. Un outil génère un ADTree par attaque depuis le data-store ;
2. L'utilisateur choisis depuis chaque ADTree une conjonction de patrons de sécurité selon le contexte de l'application.
3. Les ADTree peuvent être exprimés formellement par des expressions appelées ADTerm. A partir des ADTree générés, les ADTerms sont des expressions sur des éléments de la forme $c^o(st, \wedge^p(sp_1, \ldots, sp_m))$ qui exprime st une étape d'attaque ainsi que $\wedge^p(sp_1, \ldots, sp_m))$ une conjonction de patrons de sécurité.
4. chaque BADSeq est automatiquement extrait depuis l'arbre et traduit en un cas de test GWT. Chaque cas de test présente une étape d'attaque avec une section Given, une section When et une section Then ainsi qu'une section Then pour chaque conséquence des patrons de sécurité lié à l'étape ;
5. Chacune des sections des cas de test GWT est implémenté par une procédure. Les squelettes de ces procédures sont générées automatiquement et complétées par des directives, sous formes de commentaires, afin d'aider le développeur dans l'implémentation. Avec le recours d'un ensemble d'outils de tests d'intrusion (*pen-testing tools*), un ensemble d'étapes *Given, When, Then* peut être pré-implémenté et rendu directement utilisable pour les concepteurs. Dans le contexte des applications web, nous en avons codés manuellement 28 cas de test ainsi que les verdicts sur la présence des conséquences de 11 patrons de sécurité. Ces cas de test peuvent être réutilisable de façon générique sur les applications web.
6. L'exécution des cas de test produit deux types de verdicts, si l'application est vulnérable aux attaques et si on peut observer des conséquences de patrons depuis l'application.

5 Conclusion

Nous avons présenté une classification associant les patrons de sécurité et les attaques. Nous avons également présenté la méthode de génération d'ADTree pour chaque attaque. Notre méthodologie a permis la construction d'un data-store contenant 215 attaques, 209 étapes, 217 mesures de sécurité et 448 techniques d'attaque, qui est disponible en ligne [11]. Nous avons également introduit une méthode de génération de squelettes de cas de test permettant de tester les attaques ainsi que les conséquences observables des patrons de sécurité et les directives permettant aux concepteurs de concrétiser ces tests. Les 209 étapes d'attaques générée automatiquement sont liée à un ensemble de 627 squelettes de procédures commentées avec des aidant le concepteur à les compléter. Parmis cet ensemble de 209 cas de test nous en avons complété 28 étapes avec 84 procédure que l'utilisateur peut directement utiliser dans le contexte des applications web.

Références

[1] K. Yskout, T. Heyman, R. Scandariato, and W. Joosen, "A system of security patterns," 2006.

[2] K. Alvi, Aleem and M. Zulkernine, "A Comparative Study of Software Security Pattern Classifications," *2012 Seventh International Conference on Availability, Reliability and Security*, pp. 582–589, 2012.

[3] B. Kordy, S. Mauw, S. Radomirović, and P. Schweitzer, "Attack–defense trees," *Journal of Logic and Computation*, p. exs029, 2012.

[4] J. Meier, "Web application security engineering," *Security & Privacy, IEEE*, vol. 4, no. 4, pp. 16–24, 2006.

[5] Mitre corporation, "Common attack pattern enumeration and classification, url :https ://capec.mitre.org/," 2015.

[6] C. Steel, *Core Security Patterns : Best Practices and Strategies for J2EE, Web Services, and Identity Management*. Prentice Hall PTR, 2005.

[7] P. Willett, "Recent trends in hierarchic document clustering : a critical review," *Information Processing & Management*, vol. 24, no. 5, pp. 577–597, 1988.

[8] J. Viega and G. McGraw, *Building Secure Software : How to Avoid Security Problems the Right Way, Portable Documents*. Pearson Education, 2001.

[9] J. Scambray and E. Olson, *Improving Web Application Security*. 2003.

[10] C. Bouhours, *Détection, Explications et Restructuration de défauts de conception : les patrons abîmés*. PhD thesis, l'Université Toulouse III – Paul Sabatier, 2010.

[11] "Security pattern classification "http ://regainia.com/research/database.html"."

CaFE : un model-checker collaboratif

Steven de Oliveira[1], Virgile Prevosto[1], Saddek Bensalem[2]

1 : CEA, List ; 2 : Université Grenoble Alpes

Résumé

La logique temporelle linéaire (LTL) est utilisée dans de nombreux travaux pour décrire formellement le comportement attendu d'un programme. Une extension particulièrement intéressante de LTL pour l'analyse de programmes est la logique *CaRet*, qui autorise des raisonnements explicites sur la pile d'appel. Ce papier présente *CaFE*, un greffon de la plate-forme *Frama-C* qui vérifie automatiquement de manière approchée qu'un programme C vérifie une formule logique *CaRet* donnée. Il décrit également la mise en œuvre de *CaFE* et ses interactions avec le reste de la plate-forme.

1 Introduction

Le bon séquencement des événements au fil du temps fait partie des nombreux sujets d'analyse de programmes, par exemple pour l'étude de protocoles d'échange d'information ou de systèmes embarqués.

En particulier, la logique temporelle linéaire (LTL) [10] permet de décrire formellement le comportement attendu d'un système, sous la forme d'une succession d'actions distinctes. Une extension particulièrement intéressante de LTL pour l'analyse de programmes est la logique *CaRet*, qui autorise des raisonnements explicites sur la pile d'appel. Au sein de la plate-forme *Frama-C* [9], le greffon *CaFE* vérifie automatiquement de manière approchée (mais correcte) qu'un programme C, qui termine ou non, vérifie une formule logique *CaRet* donnée. *CaFE* tire parti des autres greffons de *Frama-C*, notamment *EVA* et *Pilat*, afin de limiter le problème d'explosion combinatoire. Cet article présente la mise en œuvre de *CaFE* et ses interactions avec le reste de la plate-forme.

2 La logique temporelle *CaRet*

La logique *CaRet* [2] est une extension de LTL [10], dont l'intérêt principal est de fournir des modalités adaptées à l'étude de la trace d'exécution d'un programme. Celle-ci est en effet structurée par les différents appels de fonctions intervenant à partir du point d'entrée principal. Pour faire apparaitre cette structure au niveau logique, *CaRet* distingue trois types de trace, associés chacun à un jeu

d'opérateurs temporels classiques (X, désignant l'état suivant de la trace, U pour définir une propriété vraie jusqu'à un certain point, et les opérateurs dérivés G pour une propriété vraie à chaque instant et F pour une propriété qui sera vraie au moins un instant dans le futur). Ces trois types de trace sont les suivants.

- La trace d'exécution générale (X^g, U^g), i.e. instruction par instruction, correspond à des propriétés LTL pures.
- La trace abstraite (X^a, U^a) suit les instructions du module courant, mais ne descend pas dans les sous-modules. Ainsi, sur un nœud d'appel à un module M, $X^a p$ indique que la propriété p sera vraie lorsqu'on sort de M.
- la trace d'appels (X^-, U^-), remonte le long de la pile d'appel vers les états appelant le module courant. Plus précisément, $X^- p$ indique que p doit être vraie au niveau du site d'appel du module courant.

Cette logique est au moins aussi expressive que LTL, mais est plus concise et modulaire.

Exemples de spécifications. La logique *CaRet* permet d'exprimer des propriétés explicites sur la pile d'appel du programme. Elle peut représenter des propriétés de sûreté, de vivacité et contextuelles :

1. $G^a\ (p)$, exprime qu'en tout point de la trace abstraite partant du début de l'exécution (c'est-à-dire en tout point de la fonction `main` du programme, mais pas forcément dans les fonctions appelées) p doit être vérifiée ;
2. $G^g\ (X^- X^g F^a p)$ indique que toute fonction doit vérifier au moins une fois p au cours de son exécution : à tout instant, l'instant suivant sur la trace générale le dernier point d'appel, c'est-à-dire le point d'entrée de la fonction courante, doit vérifier $F^a p$, soit le fait que p doit être vraie avant que la fonction termine ;
3. $G^g\ (p \Rightarrow X^- X^a q)$, spécifie qu'à tout moment, si p est vérifiée, q doit être vérifiée à la fin de l'exécution de la fonction courante (plus précisément, l'instant suivant dans la trace abstraite de la fonction appelante le point d'appel de la fonction courante).

Si le premier exemple peut être vu comme un invariant de données faible en ACSL [3], le langage de spécifications formelles pris en charge par le noyau de Frama-C, le deuxième, qui demande qu'une propriété soit vérifiée en un point indeterminé d'une fonction, et le troisième, qui conditionne une post-condition q à un événement p interne à la fonction ne peuvent eux pas être directement représentés en ACSL.

Model Checking. Un algorithme de model-checking pour la vérification de formules *CaRet* sur des machines à états récursifs est décrit en détail dans [2]. Il s'agit de l'adaptation à la logique *CaRet* de l'algorithme classique de model-checking qui consiste à calculer l'intersection entre un automate représentant la négation d'une

formule $\mathcal{F}$ et le modèle étudié, et à vérifier si cette intersection est vide. Les modèles étudiés en *CaRet* sont les *machines à états récursifs* [1] (ou *MER*), c'est à dire un ensemble d'automates (ou modules) contenant plusieurs entrées et sorties. Certains états de ces modules permettent d'appeler d'autres modules.

L'algorithme commence par intersecter chaque module avec l'automate généré à partir de la négation de la formule $\mathcal{F}$, c'est à dire générer un automate acceptant l'intersection de l'ensemble des mots acceptés par $\mathcal{F}$ avec l'ensemble des exécutions possibles des modules. Il prend en compte les chemins infinis en marquant chaque état de l'automate par *fin* si l'exécution courante termine ou *inf* si elle ne termine pas. Ce nouvel automate contient plusieurs états acceptants correspondants aux endroits où $\mathcal{F}$ est violée par une exécution de la *MER* initiale. S'il en existe, les exécutions passant par ces états acceptants représentent des contre-exemples à la formule $\mathcal{F}$ du modèle étudié. Si au contraire l'intersection est vide, $\mathcal{F}$ est vérifiée par la *MER* analysée.

3 *CaFE* : un model checker de programmes C

CaFE (CaRet Frama-C's extension) [8] est un model-checker de programmes C développé sous la forme d'un greffon de la plate-forme *Frama-C*. Son objectif est de vérifier automatiquement des propriétés écrites dans la logique temporelle *CaRet* par l'algorithme de [2] tout en optimisant la tâche par l'appel à d'autres greffons de la plateforme. La figure 1 présente le fonctionnement de *CaFE* et ses interactions avec différents greffons de la plateforme et un prouveur externe. Nous présentons dans la suite de cette section les différents outils utilisés.

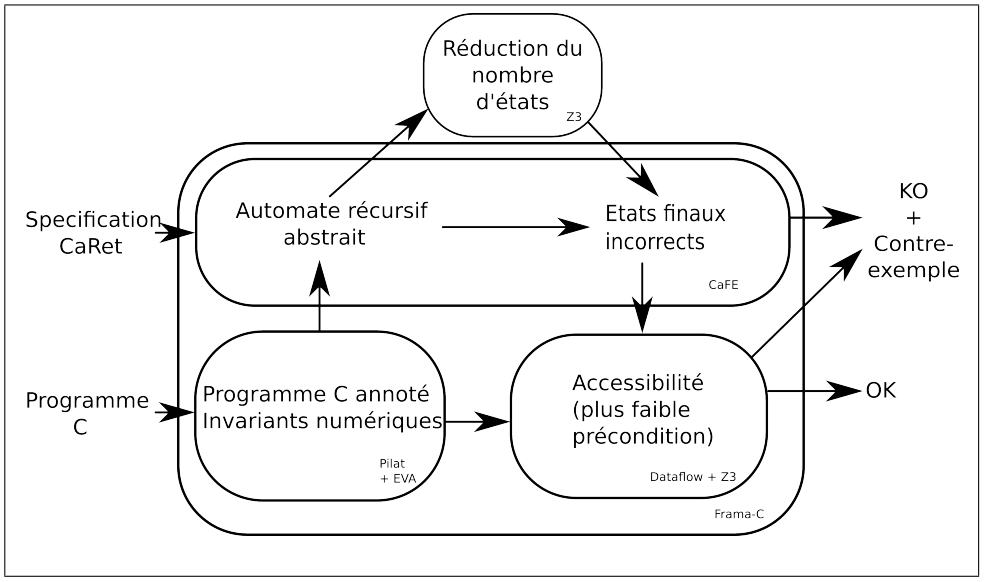


FIGURE 1: Fonctionnement de *CaFE* en interaction avec les autres greffons de *Frama-C*

Frama-C. *Frama-C* [9] est une plateforme open-source extensible et collaborative dédiée à l'analyse de programmes C et développée en OCaml. Son architecture modulaire permet la collaboration de différents outils internes et externes.

CaFE. Dans *CaFE*, les programmes sont représentés sous la forme de *MER* où les fonctions représentent les différents modules et les états sont les instructions du programme C. Afin de pallier au problème d'explosion combinatoire propre au model checking, *CaFE* collabore avec plusieurs autres greffons de *Frama-C* qui lui fournissent de nombreuses informations, réduisant drastiquement la taille de l'automate.

- *EVA* [4] : un interpreteur abstrait capable de combiner différents domaines numériques. Ses résultats sont stockés sous forme de tables, et il est possible de calculer la valeur abstraite d'une expression en un point quelconque du programme potentiellement atteignable depuis le point d'entrée initial.
- *Pilat* [7] : un générateur d'invariants inductifs de boucle. L'outil rajoute des annotations ACSL [3] aux boucles linéaires et polynomiales du programme.
- Le module Dataflow : un outil générique de parcours de programmes C en avant ou en arrière.

L'intersection entre le programme et la négation de la spécification *CaRet* utilise les résultats des deux premiers greffons afin de supprimer les états ne pouvant pas correspondre à une exécution effective du programme. Lorsque ces derniers ne suffisent pas à conclure sur la cohérence d'un état de l'automate, le SMT solver Z3 [6] est utilisé pour vérifier ou infirmer sa validité. Le nouvel automate $\mathcal{A}$ est ensuite simplifié en supprimant les exécutions non-acceptantes et non-accessibles. S'il reste des états acceptants, un calcul de plus faible pré-condition est effectué à l'aide du module Dataflow à partir des instructions liées aux états finaux des exécutions acceptantes (car chaque état de $\mathcal{A}$ correspond également à une instruction du programme original). Ce calcul utilise les invariants générés par les deux greffons précédemment cités pour extraire un contre-exemple potentiel montrant l'invalidation de la formule CaRet initiale. Un solveur SMT est utilisé pour conclure sur la validité du contre-exemple.

L'implantation de *CaFE* est correcte, au sens où la spécification *CaRet* est valide s'il ne trouve pas de contre-exemple, mais pas complet. Il peut en effet y avoir des faux positifs, c'est-à-dire des cas où l'algorithme retourne un contre-exemple potentiel alors que l'intersection est effectivement vide.

Limitations. L'outil *CaFE* se heurte malgré tout à plusieurs limitations. Premièrement, les boucles qui n'entrent pas dans le cadre de *Pilat* ne sont pas automatiquement dotées d'invariants pour aider l'analyse. Même si *EVA* arrive à surapproximer l'état de la boucle, cet état est souvent très imprécis. Lorsque cela arrive, *CaFE* renvoie de nombreux faux positifs. L'utilisateur peut néanmoins utiliser les différentes options d'*EVA* pour essayer d'obtenir un niveau de précision

satisfaisant.

Enfin, les cas d'étude de taille importante ne passent pas à l'échelle. En général, le problème est lié à la complexité de la formule *CaRet* représentant la spécification que l'on souhaite prouver. Il est nécessaire de diviser à la main la formule en plusieurs sous-formules plus petites.

4 Étude de cas : protocole MOESI

Avec l'arrivée des processeurs multi-cœurs, le concept de ressources partagées a soulevé plusieurs problèmes, notamment celui de l'accès par un cœur à des données dans le cache en cours de traitement par un autre processeur. Le protocole MOESI est un protocole de cohérence de cache répondant à ce problème. Chaque ligne du cache peut être dans l'état ***m**odified*, ***o**wned*, ***e**xclusive*, ***s**hared* ou ***i**nvalid*. Lorsqu'un cœur souhaite lire une donnée en mémoire, il effectue une requête afin de s'assurer l'exclusivité de la ligne, sauf si un autre cœur l'a déjà et effectue des calculs avec. Si le statut de chaque ligne peut être modifié, le protocole doit bien sûr par contre conserver la même quantité de lignes au total entre l'entrée et la sortie.

Pour les besoins de l'expérimentation, la représentation séquentielle du protocole en figure 2, inspirée de [5], a été utilisée. Il s'agit d'une boucle infinie simulant à chaque tour un appel à des fonctions manipulant l'état des données du cache. Le but de l'expérimentation est de comparer deux spécifications différentes testant si l'ensemble des ressources est conservé durant l'exécution du programme sous deux hypothèses :

- chaque instruction est observée séquentiellement : $G^g(m+o+e+s+i=c)$;
- les changements d'état du cache effectués par la boucle sont atomiques : $G^a(m+o+e+s+i=c)$.

En ACSL, ces spécifications peuvent être représentés par un ensemble d'assertions sur l'ensemble de la fonction *main*, chaque assertion étant à prouver séparément. Ce procédé est peu efficace pour le traitement automatique de programmes de taille conséquente. Dans *CaFE*, le programme est directement assimilé à un automate récursif où les états sont les instructions et les modules sont les différentes fonctions. L'utilisation de l'ensemble de la plateforme est ici vitale pour le traitement des spécifications.

1. *Pilat* commence à générer l'unique invariant linéaire inductif de la boucle $m+o+e+s+i=k$, où k est une constante.
2. *EVA* analyse l'ensemble du programme, prouvant en particulier que la boucle de la fonction *main* ne termine pas (le point de retour de la fonction est inatteignable) et ajoute à chaque instruction un invariant numérique dans un domaine choisi à l'avance (ici, le domaine des intervalles est utilisé).

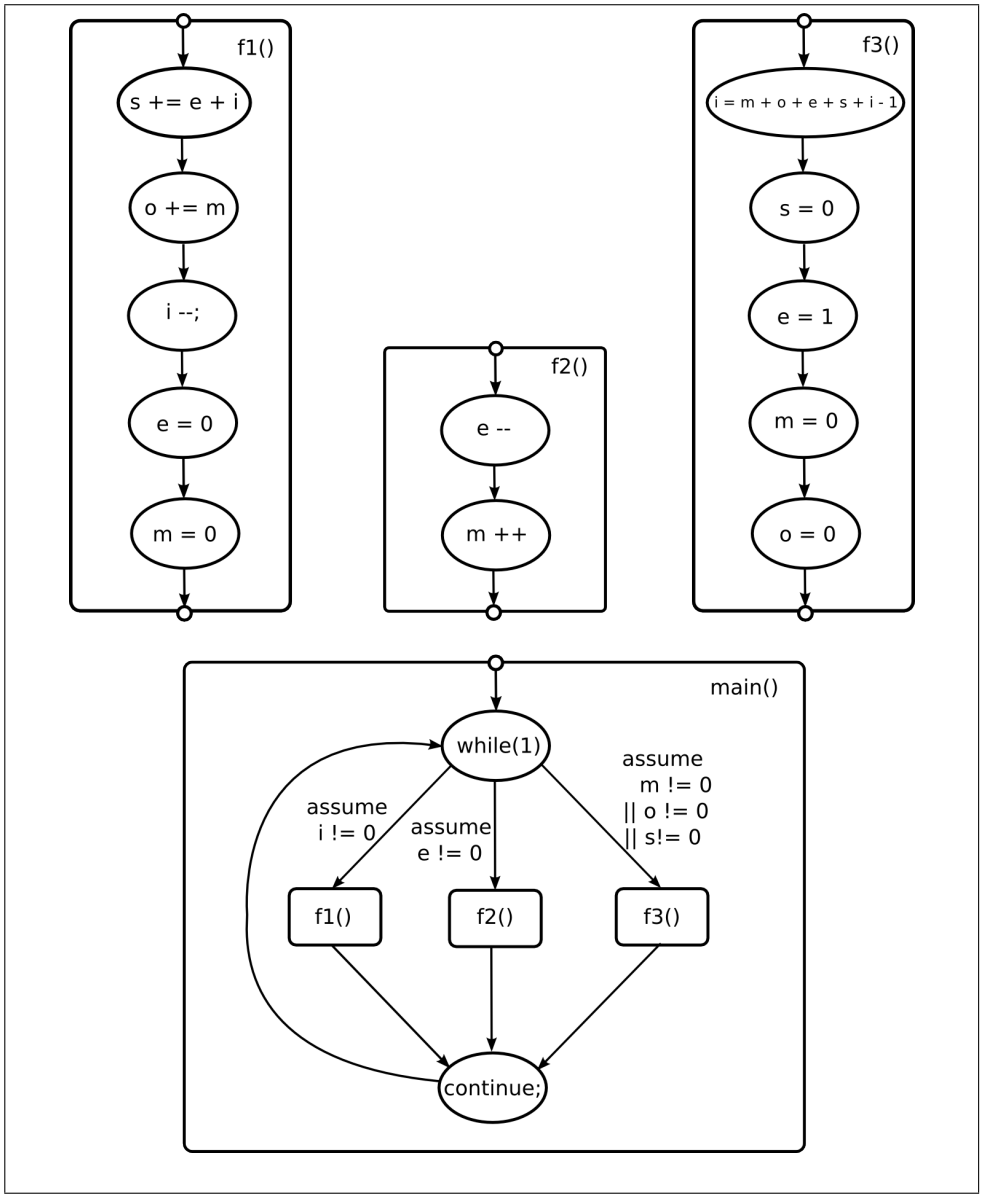


FIGURE 2: Machine à états récursifs représentant le protocole MOESI. *CaFE* effectue le produit de chaque automate par la négation de la spécification à vérifier.

3. Grâce à ces données, *CaFE* construit un automate généralisé récursif, puis supprime les états inaccessibles ou incohérents vis-à-vis des résultats des deux greffons précédents. Il utilise également Z3 afin de préciser les résultats des deux autres outils.
4. Plusieurs états finaux sont encore présents. Un parcours en arrière du programme via le module dataflow et un appel à Z3 prouve rapidement que certains de ces états ne peuvent pas être atteints. Les états correspondants

ainsi que les chemins possibles sont supprimés.

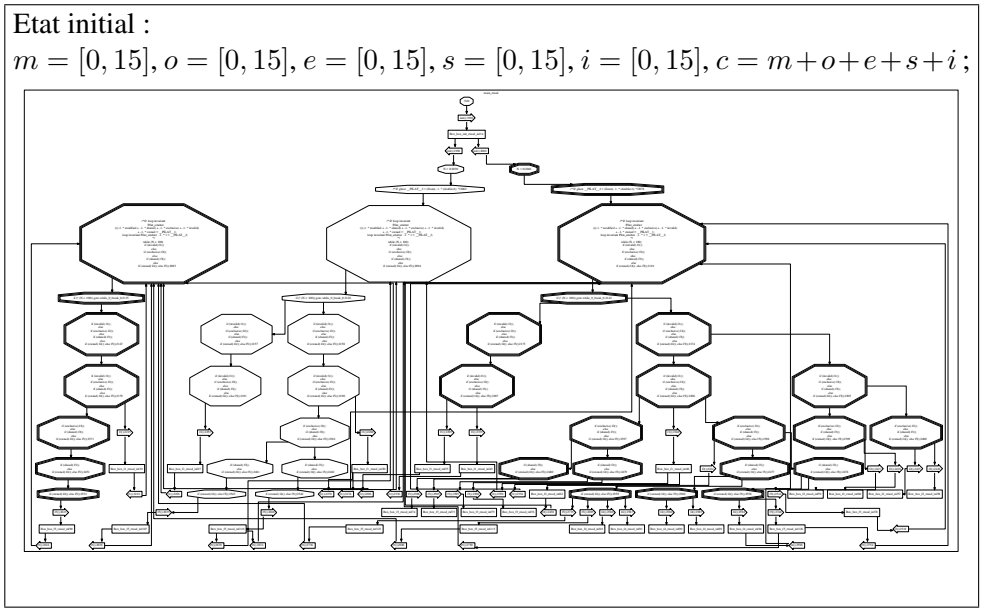


FIGURE 3: Résultat de *CaFE* : l'ensemble des contre-exemples ne respectant pas la formule $G^g(m + o + e + s + i = c)$. Il contient plusieurs états acceptant, la formule est donc infirmée. Dans le cas de la formule $G^a(m + o + e + s + i = c)$, aucun état ne reste à la fin de la simplification. Il n'existe aucun contre-exemple, et la formule est donc vérifiée.

Observations. Le résultat de la preuve de ces deux spécifications est visible en Figure 3. La spécification $G^g(m + o + e + s + i = c)$ est incorrecte puisque les fonctions n'étant pas considerées atomiques, des changements interviennent lors des mises à jour des différentes composantes de l'état du cache. *CaFE* arrive à générer plusieurs bons contre-exemples à cette spécification, qu'il retourne sous la forme d'un automate récursif contenant plusieurs états acceptants. Puisque chacun des états de cet automate correspond à une instruction, il est possible d'extraire un ensemble d'exécutions possibles menant à un état ne respectant pas la propriété. Inversement, l'invariant généré par *Pilat* prouve que l'ensemble des ressources est bel et bien préservé entre les appels et retours des fonctions modifiant l'état du cache. La spécification $G^a(m + o + e + s + i = c)$ spécifiant l'invariance du nombre de ressources entre le début et la fin de chaque étape de l'exécution du protocole est donc bien vérifiée.

5 Conclusion

Avec *CaFE*, la plate-forme *Frama-C* se dote d'un outil de model-cheking qui traite efficacement des spécifications dans une logique temporelle expressive. Il

permet également d'apprécier les interactions possibles entre différents outils qui, décorrélés, ne permettent pas de vérifier la spécification souhaitée. La suite du développement de *CaFE* est principalement axée sur l'explosion combinatoire, toujours problématique lors du traitement d'exemples industriels, et sur une recherche de contre-exemple plus efficace. Un autre objectif est l'extraction des contre-exemples sous une forme adaptée aux autres greffons de la plateforme pour permettre leur vérification ou leur infirmation.

Références

[1] R. ALUR, M. BENEDIKT, K. ETESSAMI, P. GODEFROID, T. REPS et M. YANNAKAKIS : Analysis of recursive state machines. *ACM TOPLAS*, 27(4), 2005.

[2] R. ALUR, K. ETESSAMI et P. MADHUSUDAN : A temporal logic of nested calls and returns. *In TACAS 2004*, p. 467–481. Springer, 2004.

[3] P. BAUDIN, J.-C. FILLIÂTRE, C. MARCHÉ, B. MONATE, Y. MOY et V. PREVOSTO : ACSL : ANSI C Specification Language, version 1.12, 2016.

[4] S. BLAZY, D. BÜHLER et B. YAKOBOWSKI : Structuring abstract interpreters through state and value abstractions. *In VMCAI 2017, Proceedings*, p. 112–130, 2017.

[5] E. CARBONELL : Polynomial invariant generation. http://www.cs.upc.edu/~erodri/webpage/polynomial_invariants/list.html.

[6] L. M. de MOURA et N. BJØRNER : Z3 : an efficient SMT solver. *In TACAS, Proceedings*, p. 337–340, 2008.

[7] S. de OLIVEIRA, S. BENSALEM et V. PREVOSTO : Polynomial invariants by linear algebra. *In ATVA 2016, Proceedings*, p. 479–494. Springer, 2016.

[8] S. de OLIVEIRA, V. PREVOSTO et S. BARDIN : Au temps en emporte le C. *In Actes des JFLA*, 2015.

[9] F. KIRCHNER, N. KOSMATOV, V. PREVOSTO, J. SIGNOLES et B. YAKOBOWSKI : Frama-C : A software analysis perspective. *Formal Aspects of Computing*, 27(3), 2015.

[10] A. PNUELI : The temporal logic of programs. *In 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, p. 46–57, 1977.

Intégration des (multi-)exigences tout au long du développement des systèmes complexes

Florian Galinier, Jean-Michel Bruel, Sophie Ebersold, Bertrand Meyer*

IRIT, Université de Toulouse, Toulouse, France
**Également Software Engineering Lab, Innopolis University, Russie et Politecnico di Milano, Italie*

1 Introduction

La création de systèmes complexes implique de nombreux acteurs, provenant de domaines différents. En raison de cette hétérogénéité, la spécification de ces systèmes est réalisée à l'aide d'outils différents, comme des documents textuels, des bases de données d'exigences, des diagrammes SysML (*Systems Modeling Language*)... Un des défis de l'IS (Ingénierie Système) est d'établir la cohérence et les liens entre ces différents artefacts dans l'objectif d'assurer la qualité du produit final.

En effet, il existe à ce jour un manque de cohérence entre les différentes vues de ces systèmes qui rend plus difficile l'analyse des exigences et la détection des conflits. Une approche multi-vues, avec un unique langage ou avec une abstraction commune des artefacts de spécification devrait ainsi permettre de détecter de telles incohérences en amont et faciliter les communications des parties prenantes. C'est une dimension que nous appellerons *horizontale* des multi-exigences.

Dans [1], Bertrand Meyer propose d'entremêler dans un langage de programmation la spécification et l'implémentation, afin de réduire l'écart entre les exigences et la réalisation concrète du système, dans une optique sans rupture. L'application de son concept de *multirequirements* à l'IS devrait ainsi permettre de faire le lien entre les différents niveaux d'abstraction de représentation du système et d'en faciliter la mesure d'impact du changement. C'est une dimension que nous appellerons *verticale* des multi-exigences. De plus, l'utilisation d'un langage commun pour la spécification et la conception devrait permettre la réintroduction dans le système des exigences déduites de l'implémentation.

L'objectif du travail à réaliser durant cette thèse est de définir des méthodes et outils pour permettre cette intégration sans rupture des exigences dans les deux dimensions vues précédemment, et ainsi contribuer à un problème important en ingénierie des exigences (IE) : le problème de la traçabilité entre les exigences et les artefacts développés pour y répondre, qui rend si difficile la mesure de l'impact des changements.

2 Expression des exigences dans différentes vues

L'INCOSE (*International Council on Systems Engineering*) a mis en avant le besoin de concilier les points de vues des différentes parties prenantes [2]. Notre démarche s'inscrit dans cet objectif. En effet, plutôt que d'imposer l'utilisation d'un unique langage pour exprimer les exigences, l'utilisation d'une interface commune permettrait de lier différents formalismes tout en permettant aux ingénieurs de continuer à utiliser leurs outils.

Outils en langue naturelle Il existe aujourd'hui de nombreux outils de gestion des exigences [3]. Parmi les plus connus, *IBM Rational DOORS* ou encore *Reqtify* de Dassault Systems fournissent des outils pour gérer les exigences. Ces solutions laissent les utilisateurs représenter de diverses façons les exigences et permettent de faire le lien entre elles. Ces outils permettent d'établir une traçabilité aussi bien entre les exigences et leurs réalisations, qu'entre les exigences elles-mêmes mais ils ne proposent pas de réelle sémantique pour ces liens.

Les approches GORE (*Goal-Oriented Requirements Engineering*) telles que KAOS [4] permettent également d'exprimer les exigences en langue naturelle ainsi que les relations existantes entre ces exigences. La sémantique sur les liens existants (raffinement d'une exigence, exigence composée d'autres exigences, etc.) est ici définie. Par contre, les exigences sont exprimées en langage naturel, sans syntaxe imposée, ce qui rend la déduction de liens difficile, ces derniers devant être indiqués par l'utilisateur.

Approche dirigée par les modèles L'initiative GEMOC [5] a pour objectif de définir une interface commune pour différents DSML (*Domain Specific Modeling Language*) utilisés pour exprimer les besoins spécifiques des différents acteurs impliqués dans un projet. L'utilisation des modèles comme artefacts de base est ainsi proposée, afin de combler l'écart entre différents DSML, de façon similaire à l'utilisation des artefacts comme passerelle entre spécification et implémentation proposée en ingénierie des modèles. De plus, l'acceptation de cette approche pourrait être facilitée par la croissance de l'intérêt pour l'approche MBSE (*Model-Based System Engineering*) par les industriels en IS, qui peut être utilisée afin d'exprimer les exigences comme des éléments de modèles, de la même manière que les autres artefacts de modélisation. Cette approche devrait ainsi permettre de créer des liens entre les exigences et les autres artefacts, que ce soit d'autres exigences ou des éléments de spécification fournis par les différentes parties prenantes. Ainsi, il serait possible de combiner des éléments de différents domaines dans une vue holistique, prenant en compte les liens entre les artefacts spécifiques à un domaine mais également entre ces artefacts et les exigences. Dans [6] par exemple, les auteurs proposent d'appliquer cette fédération de modèles aux exigences, considérant que chaque espace technologique est une technique d'expression des exigences. Cela permet par exemple de lier des exigences exprimées en langue

naturelle dans un document type Word avec des exigences et leurs liens exprimés grâce à l'approche KAOS.

3 Formalisation des exigences

Le standard ISO/IEC/IEEE 29148:2011 [7] définit un certain nombre de qualités nécessaires pour l'expression des exigences (la traçabilité, la vérifiabilité, la consistance et l'absence d'ambiguïtés, ...). L'utilisation d'un langage dédié aux exigences, plus formel que la langue naturelle habituellement utilisée, est un moyen possible d'assurer ces différentes qualités.

Expression des exigences L'utilisation d'un langage dédié a été étudiée à différentes occasions. Le profil SysML [8] propose ainsi un diagramme spécifique à l'expression des exigences (à la fois fonctionnelles et non-fonctionnelles), ainsi que les liens existant entre exigences ou entre les exigences et les autres éléments de modèles (blocs, cas d'utilisation). Les exigences en elles-mêmes y sont cependant représentées sous une forme textuelle (voir Fig. 1).

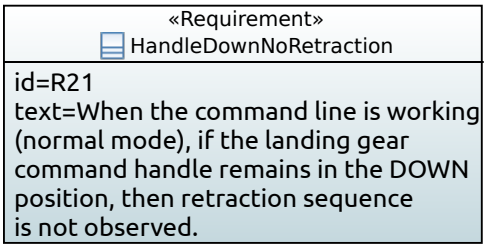


Figure 1: Représentation de l'exigence R21 de [9] en SysML

Cependant dans les approches proposées jusqu'à présent, les exigences sont toujours exprimées en langage naturel et sont, par conséquent ambiguës. Des travaux ont été proposés pour surmonter ce problème (e.g., [10], [11]). Si ces approches permettent de faire le pont entre une formalisation des exigences et une expression des exigences compréhensible par des non-informaticiens, l'utilisation de deux langages rend nécessaire de maintenir la cohérence entre les exigences ainsi exprimées, et la répercussion des changements n'est pas aussi immédiate que dans le cas de l'utilisation d'un formalisme unique.

Langages spécifiques d'expression des exigences Afin d'éviter cet écart entre langue naturelle et approches plus formelles, certains travaux mettent en avant des langages dédiés à l'expression des exigences (e.g., [12], [13]).

Cependant, à chaque fois, la syntaxe proposée n'est pas suffisamment simple ou proche des utilisateurs pour être adoptée facilement. De plus, ils s'adressent à des domaines particuliers et sont par conséquents très spécifiques.

S1: The synchronization process SHALL be initiated when Alice enters the room and at 30 minute intervals thereafter.

S2: The synchronization process SHALL distribute data to all connected devices in such a way that all devices are using the same data at all times.

Figure 2: Exemple d'exigences en RELAX extrait de [12].

Expression formelle des exigences De nombreux travaux proposent d'exprimer les exigences avec des méthodes formelles (e.g., [14], [15], [16]). Il est à noter que ces approches ne sont pas du tout destinées à des non-spécialistes et ne peuvent ainsi pas être utilisées comme interface de discussion entre acteurs de différents domaines.

Dans [1], Bertrand Meyer propose une approche pour exprimer les exigences directement dans le code source Eiffel. Il applique ainsi le *Single Model Principle* proposé dans [17]. L'exigence proposée en Fig. 1 peut ainsi être exprimée sous forme de préconditions et postconditions d'une opération exécutant la boucle d'exécution du système (donnée List. 1).

```
r21
    -- (R21) When the command line is working (normal
    -- mode), if the landing gear command handle
    -- remains in the DOWN position, then retraction
    -- sequence is not observed.
    require
      handle_status = is_handle_down
    do
      main
    ensure
      gear_status /= is_gear_retracting
end
```

Listing 1: Exemple de représentation en Eiffel de l'exigence R21 présentée Fig. 1.

L'expression des exigences à l'aide des contrats permet ainsi de fournir une interface plus accessible que les méthodes beaucoup plus formelles, tout en bénéficiant des outils formels d'Eiffel – comme AutoProof [18] un vérificateur pour Eiffel – afin de détecter les éventuelles incohérences du système. Il propose également de lier au sein d'un même formalisme les exigences, exprimées en langue naturelle, avec leur spécification (au travers de diagramme) et leur implémentation. Cette approche permet ainsi de faciliter la traçabilité.

4 Travaux futurs

L'intégration des exigences tout au long du développement d'un système complexe est un moyen de réduire les coûts engendrés par les erreurs dues à une mauvaise analyse des exigences. Les méthodes présentées ont toutes pour objectif l'ajout d'un certain formalisme afin d'améliorer le traitement des exigences. L'utilisation de la langue naturelle comme principal outil d'expression des exigences nous amène cependant à nous poser un certain nombre de questions :

- Comment réussir à exprimer les exigences de façon à ce qu'elles restent compréhensibles par un non-spécialiste tout en étant analysables de façon automatique ?
- Comment faire le lien entre les exigences exprimées par différentes parties prenantes ?
- Quelle sémantique donner aux liens existant entre les exigences ? Entre les exigences et le système ?
- Comment utiliser la formalisation des exigences pour prouver des propriétés des exigences ?

Si les travaux cités jusqu'à présent permettent de donner des pistes de réponses, il n'en existe, à notre connaissance, aucun qui permette de répondre à toutes ces questions à la fois. Un des objectifs principaux de notre travail est ainsi de définir une formalisation des exigences, via un langage qui permettra de faire le lien entre les exigences de différents domaines. Ce formalisme devra également être utilisé afin de faire le lien entre les exigences et les autres artefacts de spécification. Nous souhaitons également fournir des outils afin d'assister les ingénieurs des exigences dans le contrôle de la qualité et de la validité du système (à l'aide de traces, de couverture, ...).

Un autre objectif important de notre projet est la simplicité d'utilisation et d'appropriation de cette interface. En effet, afin de permettre à des ingénieurs non-informaticiens d'utiliser ces outils, nous souhaitons qu'ils soient aussi proches que possible de leurs outils habituels.

Afin d'expérimenter notre approche, nous utilisons l'étude de cas du système de train d'atterrissage proposée dans [9], qui définit un système et ses exigences de façon détaillée. Cet exemple, proposé durant la conférence ABZ2014 [19], est accompagné d'un ensemble d'articles proposant des formalisations des exigences identifiées, avec lesquelles nous pourrons comparer notre approche. Par la suite, nous prévoyons de valider notre approche sur un exemple industriel réel.

References

[1] B. Meyer. Multirequirements. *Modelling and Quality in Requirements Engineering (Martin Glinz Festscrhift)*, 2013.

[2] INCOSE. *SE Vision 2025*. 2014. http://www.incose.org/docs/default-source/aboutse/se-vision-2025.pdf.

[3] J. M. Carrillo de Gea, J. Nicolás, J. L. F. Alemán, A. Toval, C. Ebert, and A. Vizcaíno. Requirements Engineering Tools. *IEEE Software*, 28(4):86–91, July 2011.

[4] A. van Lamsweerde. Goal-oriented requirements engineering: a guided tour. In *Proceedings Fifth IEEE International Symposium on Requirements Engineering*, pages 249–262, 2001.

[5] B. Combemale, J. Deantoni, B. Baudry, R. B. France, J.-M. Jézéquel, and J. Gray. Globalizing Modeling Languages. *Computer*, pages 10–13, June 2014.

[6] Fahad R. Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guerin, and Christophe Guychard. Continuous Requirements Engineering using Model Federation. *RE:Next! Track at 24th IEEE International Requirements Engineering Conference 2016*, 2016.

[7] ISO/IEC/IEEE International Standard 29148:2011. *ISO/IEC/IEEE 29148:2011(E)*, pages 1–94, December 2011.

[8] Object Management Group (OMG). OMG Systems Modeling Language (OMG SysML^{TM}), V1.0, 2007. OMG Document Number: formal/2007-09-01 Standard document URL: http://www.omg.org/spec/SysML/1.0/PDF.

[9] F. Boniol and V. Wiels. The Landing Gear System Case Study. In Frédéric Boniol, Virginie Wiels, Yamine Ait Ameur, and Klaus-Dieter Schewe, editors, *ABZ 2014: The Landing Gear Case Study*, number 433 in Communications in Computer and Information Science, pages 1–18. Springer International Publishing, June 2014.

[10] W. Scott and S. C. Cook. *A Context-free Requirements Grammar to Facilitate Automatic Assessment*. PhD thesis, UniSA, 2004.

[11] R. Hähnle, K. Johannisson, and A. Ranta. An Authoring Tool for Informal and Formal Requirements Specifications. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering*, number 2306 in Lecture Notes in Computer Science, pages 233–248. Springer Berlin Heidelberg, April 2002.

[12] J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J. M. Bruel. RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. In *2009 17th IEEE International Requirements Engineering Conference*, pages 79–88, August 2009.

[13] T. Nguyen. Verification of Behavioural Requirements for Complex Systems with FORM-L, a MODELICA Extension. In *26th International Conference on Software & Systems Engineering and their Applications*, EDF R&D, 6 quai Watier, 78110 Chatou, FRANCE, 2015.

[14] F.-L. Li, J. Horkoff, A. Borgida, G. Guizzardi, L. Liu, and J. Mylopoulos. From Stakeholder Requirements to Formal Specifications Through Refinement. In Samuel A. Fricker and Kurt Schneider, editors, *Requirements Engineering: Foundation for Software Quality*, Lecture Notes in Computer Science, pages 164–180. Springer International Publishing, March 2015.

[15] A. Mammar and R. Laleau. On the Use of Domain and System Knowledge Modeling in Goal-Based Event-B Specifications. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques*, number 9952 in Lecture Notes in Computer Science, pages 325–339. Springer International Publishing, October 2016.

[16] A. Matoussi, F. Gervais, and R. Laleau. An Event-B formalization of KAOS goal refinement patterns. Technical Report Tech. Rep. TRLACL-2010-1, LACL, University of Paris-Est, 2010.

[17] R. Paige and J. Ostroff. The Single Model Principle. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, RE '01, pages 292–, Washington, DC, USA, 2001. IEEE Computer Society.

[18] J. Tschannen, C. A. Furia, M. Nordio, and N. Polikarpova. AutoProof: Auto-Active Functional Verification of Object-Oriented Programs. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, number 9035 in Lecture Notes in Computer Science, pages 566–580. Springer Berlin Heidelberg, April 2015.

[19] F. Boniol, V. Wiels, Y. Ait Ameur, K.-D. Schewe, S. D. Junqueira Barbosa, P. Chen, A. Cuzzocrea, X. Du, J. Filipe, O. Kara, I. Kotenko, K. M. Sivalingam, D. Slezak, T. Washio, and X. Yang, editors. *ABZ 2014: The Landing Gear Case Study*, volume 433 of *Communications in Computer and Information Science*. Springer International Publishing, Cham, 2014.

MODELISATION D'EXIGENCES POUR LA SYNTHESE D'ARCHITECTURE AVIONIQUE : APPLICATION A LA SURETE DE FONCTIONNEMENT

L. ZIMMER, M. LAFAYE

Dassault Aviation – Direction Générale Technique
78 quai Marcel Dassault - 92552 Saint-Cloud - France
laurent.zimmer@dassault-aviation.com
michael.lafaye@dassault-aviation.com

P.A. YVARS

Institut Supérieur de Mécanique de Paris (SupMéca)
QUARTZ
3 rue Fernand Hainaut – 93407 Saint-Ouen - France
pierre-alain.yvars@supmeca.fr

RESUME : Les plateformes informatiques embarquées doivent supporter l'exécution d'un nombre croissant de fonctions systèmes et respecter scrupuleusement de multiples contraintes fonctionnelles et non fonctionnelles. Leur architecture est donc de plus en plus complexe et par conséquent la définition préliminaire d'architectures admissibles basées sur la compétence et l'expérience d'un petit nombre de concepteurs experts est une activité de plus en plus difficile. A l'avenir, ils devront s'aider d'outils d'assistance à la génération d'architectures correctes par construction. Dans ce contexte, nous nous sommes intéressés à la modélisation formelle des exigences et à la résolution informatique de ces modèles formels pour trouver des architectures admissibles. Nous avons étudié une problématique de déploiement assisté de fonctions systèmes sur une plateforme d'avionique modulaire embarquée avec des exigences de sûreté de fonctionnement portant sur les fonctions. Nous avons développé une approche à base de modèle qui utilise le langage DEPS (DEsign Problem Specification), un langage dédié à la modélisation et à la résolution des problèmes de conception. Les résultats obtenus montrent qu'il est possible de modéliser des exigences complexes de sûreté de fonctionnement au niveau requis par l'architecte système et que leur prise en compte pendant la résolution génère des solutions correctes vis-à-vis de celles-ci.

MOTS-CLES : *modélisation, synthèse d'architecture, spécification formelle, déploiement, exigences, sûreté de fonctionnement.*

1 INTRODUCTION

Les systèmes avioniques sont de plus en en plus complexes. D'après [1], dans le domaine militaire nous sommes passés de 15 sous-systèmes et moins de 40% des fonctions systèmes à dominante logicielle pour le F16 à 135 sous-systèmes dont 90% des fonctions à dominante logicielle pour le F35. L'évolution dans le domaine civil est moins extrême mais elle suit la même tendance. En parallèle le nombre et la complexité des spécifications techniques ou réglementaires ont augmenté tout comme la complexité de l'organisation industrielle. Cet accroissement global de la complexité engendre une explosion des coûts et des délais de développement qui amène les industriels à revoir leurs méthodes et leurs outils de conception. Selon des études financées par la DARPA [1, 2, 3], le nouveau processus de conception à imaginer doit reposer sur le développement de 4 éléments clefs :

1. des outils de conception à base d'abstraction ;
2. des métriques de complexité des systèmes ;
3. des méthodes avancées de synthèse d'architecture ;
4. une gestion robuste des incertitudes.

Le travail présenté concerne exclusivement le premier [2] et le troisième point [3]. Il s'agit pour le premier point de développer ou d'utiliser un langage de description formelle (FDL) capable de représenter à la fois un système complexe ainsi que les spécifications ou exigences qui portent sur celui-ci. Le FDL doit notamment permettre de représenter le système à des niveaux d'abstraction compatibles des différentes étapes de la conception et notamment les étapes de conception préliminaire. Les langages envisagés sont AADL, UML ou SysML.

Il s'agit pour le deuxième point de disposer très tôt dans la conception de moyens « avancés de synthèse d'architecture » qui permettraient d'explorer automatiquement l'espace de conception pour rechercher des architectures admissibles c'est-à-dire compatibles avec les différentes exigences système.
L'idée étant que la complexité des systèmes à concevoir mettra à terme la tâche d'énumération et d'évaluation des architectures candidates hors de portée des experts s'ils ne bénéficient pas d'une assistance informatique.

Pour développer un outillage de conception capable de synthétiser des architectures admissibles des chaînes de traitement logicielles ont été proposées dans des travaux précurseurs [4]. Elles comportent un générateur de contraintes dédié qui prend en entrée des données système et dont la sortie est exploitée par un outil de résolution. Dans cette approche l'essentiel du développement porte sur le générateur ; l'outil de résolution est pris sur étagère et il n'y a pas de langage de modélisation en entrée. D'autres travaux s'appuient sur l'utilisation ou l'enrichissement d'un langage de modélisation de système proposant différentes descriptions (exigences, structure, comportement etc.) et l'exploite à l'aide d'outils, d'algorithmes de vérification ou de résolution [3, 5] pour énumérer ou évaluer des architectures candidates.

Plus récemment des travaux portent sur le développement d'un langage de modélisation de problème de conception en vue de leur résolution [6]. Il s'agit donc d'enrichir les langages de résolution pour aborder les problèmes de synthèse. Dans ce cadre, les travaux que nous présentons montrent l'intérêt d'utiliser le langage DEPS pour d'une part modéliser un système et des exigences qui lui sont associées et d'autre part résoudre pour trouver une solution de synthèse.

L'étude de cas, objet de cette évaluation, est une problématique de déploiement de fonctions avion sur une architecture informatique embarquée de type avionique modulaire intégrée. Nous nous sommes limités dans ce papier à la fois en types d'éléments d'architecture (les calculateurs) et en types d'exigences (la sûreté de fonctionnement) mais nous avons veillé attentivement au caractère généralisable de l'approche.

Le papier s'organise comme suit : nous positionnons d'abord nos travaux dans le contexte de l'avionique modulaire, de la sûreté de fonctionnement et de la répartition des rôles entre acteurs puis nous présentons le langage DEPS et l'outillage associé que nous utilisons. Ensuite nous décrirons successivement l'étude de cas et le problème posé. Puis nous discuterons de la modélisation qui a été faite en DEPS et des résultats que nous avons obtenus. Enfin nous présenterons des perspectives d'évolution de ce travail tant du point de vue de la généralisation de l'étude de cas que des évolutions du formalisme DEPS.

2 PROBLEMATIQUE

2.1 Cadre général

Le concept d'avionique modulaire intégrée (IMA) [7, 8], défini dans les années 1990, s'est aujourd'hui imposé comme l'une des références pour le développement de plateformes avioniques civiles. Permis par l'émergence de technologies augmentant la puissance des calculateurs, ce concept vise à regrouper plusieurs fonctions logicielles non vitales, auparavant exécutées par des calculateurs dédiés, sur un même calculateur.

Un des apports de l'IMA est de découpler le développement logiciel du matériel sous-jacent, i.e. assurer une modularité de ces logiciels qui peuvent ainsi être déployés indifféremment sur les calculateurs en fonction de leurs besoins en ressources, pourvu que ces derniers soient « compatibles » IMA.

Cette compatibilité se traduit par un ensemble d'interfaces applicatives (API) générique à tous les calculateurs et une allocation statique des ressources temporelles et d'I/O.
En IMA le concept de partition traduit cette allocation des ressources (cf. Figure 1). Un logiciel, réalisant tout ou partie d'une fonction avion, peut ainsi être projeté sur une ou plusieurs partitions suivant le besoin en ressources et les exigences associées.

En contrepartie, cette capacité de mutualisation des logiciels associée à l'accroissement de leur nombre a entrainé une augmentation de la complexité du problème d'intégration. Il s'agit de trouver un schéma d'allocation des calculateurs aux logiciels répondant aux besoins en termes de ressources tout en respectant les contraintes de latence d'exécution des chaines fonctionnelles traversantes (i.e. traitées via l'exécution de plusieurs logiciels) ou encore de sûreté de fonctionnement. Du fait de ces contraintes multiples et du nombre important de déploiements, trouver manuellement une allocation satisfaisante devient très difficile.

L'intégrateur système va utiliser sa connaissance des logiciels et des ressources calculateurs et réseau pour proposer plusieurs allocations puis évaluer à l'aide d'outils la conformité de ces allocations aux exigences jusqu'à trouver un déploiement satisfaisant. Il s'agit d'une démarche d'analyse a posteriori.

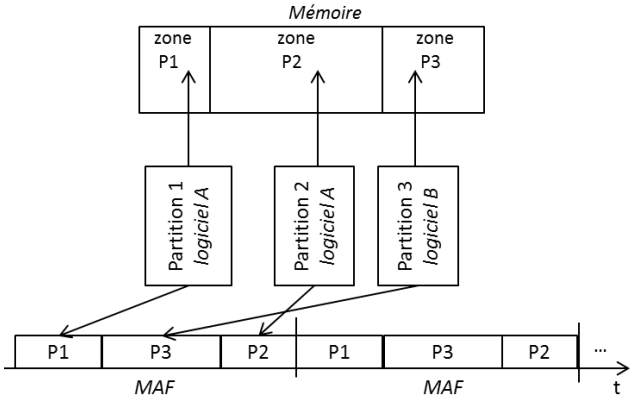


Figure 1 : principe d'allocation statique des ressources via le concept de partition

Partant de ce constat, nous avons développé une approche de déploiement des logiciels qui prend en compte a priori et non a posteriori les exigences qui portent sur les fonctions avion.

Afin d'illustrer notre démarche, nous nous placerons par la suite dans le cas d'un problème de déploiement de logiciels sur un ensemble de calculateurs IMA devant respecter uniquement des exigences de sûreté de fonctionnement. Ces exigences sont parmi les plus importantes, tout avion civil étant pensé dans une approche visant à réduire au maximum les risques d'accident et donc potentiellement de pertes humaines.

2.2 Les exigences de sûreté de fonctionnement

2.2.1 Production des exigences

La sûreté de fonctionnement vise à établir des niveaux admissibles de fiabilité, disponibilité et maintenabilité des systèmes essentiels de l'avion afin de garantir la sécurité de l'appareil en vol comme au sol. Les exigences permettant d'atteindre ces niveaux résultent de l'analyse des cas de pannes et défaillances, pouvant amener par exemple à une perte de fonctionnalité ou une corruption de données.

En terme d'architecture plateforme, ces analyses conduisent à un ensemble de « patterns ». Un « pattern » est une proposition de découpage d'une fonction avion en composants logiciels avec des exigences associées de ségrégation ou de dissimilarité. Par exemple, la robustesse à un cas de panne d'une fonction conduit à un « pattern » de type redondance du logiciel exécutant cette fonction et son déploiement sur deux calculateurs distincts. On parlera de ségrégation des ressources utilisées. Ces exigences ne sont pas uniquement applicables au niveau logiciel, mais peuvent porter par exemple sur l'ensemble de la chaine de traitement d'une fonction avion donnée, en exigeant que cette fonction soit réalisée via deux chaines ségréguées. Dans ce cas, l'ensemble des logiciels réalisant la première chaine devront impérativement être déployés sur des calculateurs distincts des logiciels réalisant la seconde chaine. Une illustration de « patterns » et d'exigences associées sera donnée dans la partie cas d'étude.

2.2.2 Utilisation à l'intégration

Afin de prendre en compte au plus tôt ces exigences et patterns pour l'identification de schémas d'intégration satisfaisants, il est nécessaire de pouvoir formaliser ces éléments de sûreté de fonctionnement à partir des sorties « manuscrites » livrées après analyse amont. Pour que cette formalisation ne soit pas seulement exploitable par des spécialistes du domaine de la sûreté de fonctionnement mais également par l'intégrateur plateforme, il est nécessaire que cette formalisation se situe au bon niveau entre pouvoir d'expression et abstraction. Notre objectif est ensuite d'utiliser ces éléments afin de déterminer au plus tôt et de manière informatisée, les possibilités de déploiements candidats (cf.
Figure 2).

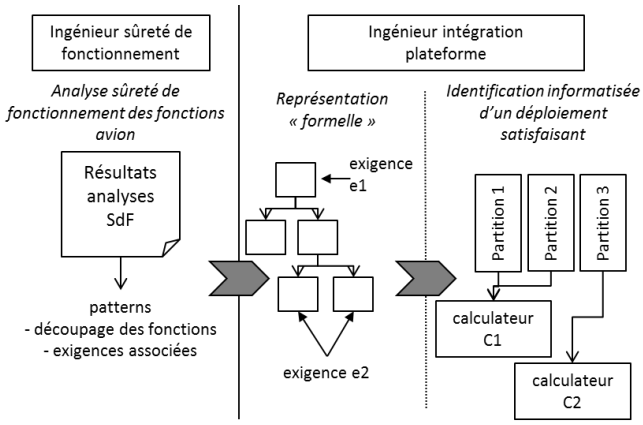


Figure 2 : principe de prise en compte au plus tôt des exigences de Sûreté de Fonctionnement

Nous sommes donc confrontés à un problème de représentation « formelle » des éléments d'entrée, ainsi qu'à un problème d'identification de solutions compatibles des exigences. Nous nous positionnons ainsi dans une démarche de modélisation et de synthèse, par opposition aux démarches actuelles centrée sur l'analyse.
Nous avons fait le choix d'utiliser un même formalisme pour traiter conjointement la question de la représentation des éléments système et celle de la recherche de solutions compatibles de l'espace des exigences. Il s'agit du langage DEPS.

3 LE LANGAGE DEPS

3.1 Paradigme

Le langage DEPS (Design Problem Specification) est ce qu'on appelle communément un langage dédié ou DSL (Domain Specific Language). Le domaine d'application visé est la spécification et la résolution de problèmes de l'ingénieur, en particulier ceux que l'on rencontre en conception de produits ou de systèmes.
DEPS est un langage dédié externe (par opposition à interne ou embarqué). Le code source est donc indépendant de tout langage généraliste hôte.

Le langage DEPS peut être vu comme une combinaison entre un langage de modélisation logiciel ou système et un langage de programmation mathématique. Aux premiers ont été empruntés les traits de structuration et d'abstraction qui permettent de représenter les éléments et le système étudié. Aux seconds ont été empruntés les concepts mathématiques nécessaires à la résolution des problèmes de l'ingénieur : inconnues, équations et inéquations.

Cette combinaison permet à la fois de représenter les problèmes de conception et poser puis résoudre ou optimiser les systèmes d'équations et d'inéquations qui les régissent [6].

3.2 Caractéristiques essentielles

3.2.1 Le Modèle

Le trait fondamental du langage est le Modèle. Tout Modèle est défini à l'aide du mot clé *Model* suivi de son nom et de sa liste d'arguments (éventuellement vide). Il comporte dans l'ordre : une zone de déclaration-définition des constantes du modèle, une zone de déclaration des variables, une zone de déclaration-création des éléments et une zone de définition des propriétés. La définition d'un Modèle DEPS se termine par le mot clé *End.*

Les propriétés d'un Modèle sont les équations et les inéquations qui portent sur les constantes et les variables de ce Modèle. Un Modèle contient donc tous les ingrédients nécessaires à la pose des contraintes qui régissent une instance de ce Modèle.

Comme dans un langage objet on dispose de l'héritage et de la composition : un Modèle peut être étendu et hériter des propriétés d'un autre Modèle et il peut être composé d'éléments, instances d'autres Modèles.

Des éléments peuvent être passés en argument d'un Modèle créant un lien d'agrégation avec celui-ci.
Des constantes peuvent aussi être passées en argument d'un Modèle ce qui permet de créer des Modèles paramétrés.

Modéliser un problème revient donc à spécifier des Modèles.

3.2.2 La Quantité

En DEPS, les constantes comme les variables sont associées à des types de grandeurs physiques ou technologiques qu'on appelle quantités (*Quantity*). Elles sont nécessaires en Ingénierie de Systèmes.

Une quantité possède :
- Un type de quantité de base (*QuantityKind*). Par exemple, réel (*Real*), entier (*Integer*), longueur (*Length*) ;
- Une borne min (resp. max) qui représente la valeur minimale (resp. maximale) pouvant être prise par toute constante ou variable ayant pour type la quantité définie ;
- Une dimension qui représente la dimension au sens de l'analyse dimensionnelle de la quantité. Par exemple [L] pour une longueur ou [U] pour une grandeur sans dimension ;
- Une unité de la quantité. Par exemple le mètre m pour une longueur.

4 L'OUTILLAGE EXISTANT

4.1 L'environnement de développement

L'environnement de modélisation et de résolution intégré associé au langage DEPS comprend :

- Des fonctions d'édition de modèle,
- Des fonctions de gestion de projet basées sur un mécanisme de packages ;
- Un compilateur,
- Un solveur.

Une approche par intégration plutôt qu'une approche par transformation de modèles a été privilégiée, En effet, dans le cas de la résolution d'un problème de synthèse de système sous-défini, il va être nécessaire en cas de résultat de calcul non satisfaisant de réaliser une mise au point des modèles dans le langage DEPS. En optant délibérément pour une approche par intégration, nous favorisons ce processus de mise au point.

4.2 La résolution

Les méthodes de calcul que nous utilisons sont tirées des travaux sur la résolution des CSP.

Un CSP (Constraint Satisfaction Problem) est défini par un triplet (*X, D, C*) tel que [9] :
- *X* est un ensemble fini de variables dites variables contraintes.
- *D* est un ensemble fini de domaines de ces variables.
- *C* est un ensemble fini de contraintes sur les variables de l'ensemble *X*.

Les domaines des variables peuvent être discrets (CSP) ou continus (NCSP pour Numerical CSP). On appelle contrainte, n'importe quel type de relation mathématique qui porte sur les valeurs d'un ensemble de variables : égalité, différence, inégalité logique et/ou algébrique (linéaire ou non linéaire).

Résoudre un CSP revient ainsi à instancier chacune des variables de X tout en satisfaisant l'ensemble C des contraintes du problème. Partant des domaines de valeurs de chacune des variables du problème, l'algorithme de résolution alterne contraction des domaines et choix d'un sous domaine pour une variable du problème jusqu'à aboutir à une solution ou bien un échec. Ce dernier est alors traité par un retour en arrière sur les points de choix précédents.

L'étape de contraction est réalisée à l'aide d'algorithmes dédiés qui mettent à profit les contraintes explicitées du problème pour réduire les domaines de chaque variable.
Dans le cas d'un problème sur-contraint, un échec peut apparaitre dès la première propagation ou bien au final après avoir exploré les parties restantes de l'arbre de recherche. Dans ce cas, la méthode garantit qu'il n'y a pas de solution au problème posé.

Le solveur implémente une méthode de propagation de type HC4 révisé [10] sur des équations et inéquations portant sur quatre types de domaines : les intervalles ouverts de réels, les intervalles d'entiers, les ensembles énumérés de valeurs flottantes et les ensembles énumérés de valeurs entières signées. Les contractions sont réalisées directement sur les domaines typés sans repasser dans les intervalles de réels. L'algorithme de recherche de solution est une méthode de branch and prune. Les stratégies round-robin et first-fail sont disponibles.

Dans le cas d'un problème sur-contraint, un échec peut apparaitre dès la première propagation ou bien au final après avoir exploré les parties restantes de l'arbre de recherche. Dans ce cas, l'échec s'interprète comme la preuve qu'il n'y a pas de solution au problème posé et non pas comme une défaillance de l'algorithme de résolution.

L'architecture orientée-objet du solveur a été pensée de manière à pouvoir être étendue à d'autres méthodes de propagation et/ou de résolution (box-consistance, méthodes locales, …).

5 APPLICATION AU CAS D'ETUDE

5.1 Description du problème

Notre cas d'étude (cf.

Figure **3**) consiste en un problème de déploiement de sept fonctions avion (notamment des applications de freinage, de remontée de pannes, de communication, etc.) sur une plateforme composée de quatre calculateurs, à déterminer en fonction des exigences de sûreté de fonctionnement associées.

Pour rappel, nous nous plaçons dans le rôle de l'intégrateur plateforme et supposons l'analyse de sûreté de fonctionnement des sept fonctions avion terminée.

Nous avons donc en entrée pour chaque fonction un document décrivant le découpage en composants logiciels puis la projection en termes de partitions, avec des exigences de ségrégation matérielle (projection des composants ségrégués sur des calculateurs distincts) ou inversement de co-localisation pour assurer la cohérence de données à traiter séquentiellement dans un temps maitrisé court. Nous allons détailler ci-après la décomposition du cas d'étude (cf.
Figure 3) sur deux fonctions avion.

La première fonction consiste en une fonction de gestion du système d'atterrissage. Cette fonction dite SAT est initialement projetée sur un seul composant logiciel.

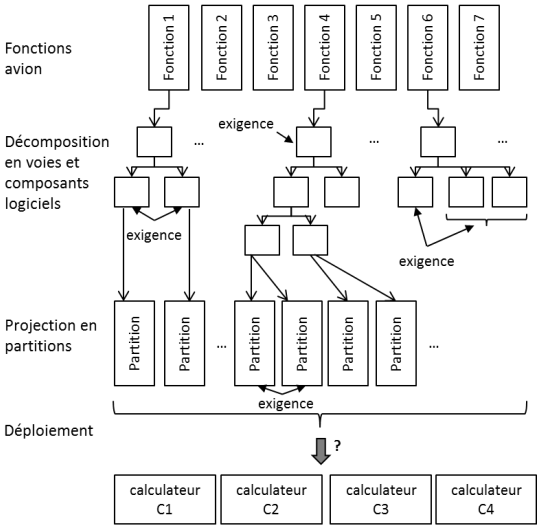


Figure 3 : description générale du cas d'étude

L'analyse de sûreté de fonctionnement nous impose deux exigences :
- Exigence 1 : un second composant logiciel dissimilaire (appelée CS pour composant « safety ») doit être prévu, afin de prendre le relais en cas de perte du composant principal (appelé CP). Ces composants doivent être ségrégués matériellement afin qu'une panne matérielle affectant CP n'affecte pas CS ;
- Exigence 2 : cette chaine de traitement (ou voie) CP + CS doit être redondée et ségréguée matériellement afin d'être robuste à la perte de la première chaine.

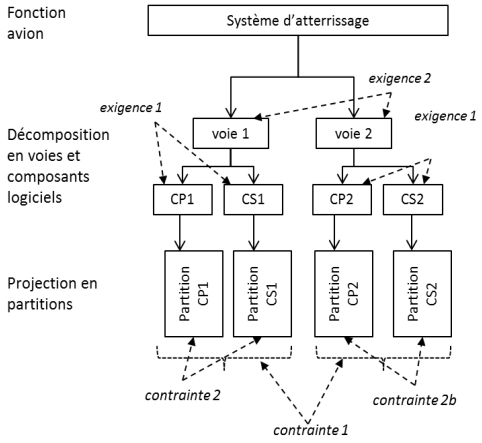


Figure 4 : décomposition de la fonction SAT

D'un point de vue traitement, le fournisseur de la fonction nous indique que chaque composant peut être projeté sur une seule partition. Nous nous retrouvons donc avec un schéma de décomposition résultant sur quatre partitions, avec les contraintes suivantes (cf. Figure 4) :
- Contrainte 1 : les partitions {CS1, CP1} doivent être matériellement ségréguées des partitions {CS2, CP2} (cf. Exigence 1);
- Contrainte 2 : les partitions CS1 et CP1 doivent être matériellement ségréguées (cf. Exigence 2) ;
- Contrainte 2bis : les partitions CS2 et CP2 doivent être matériellement ségréguées (cf. Exigence 2).
La seconde fonction que nous détaillons consiste en une fonction de gestion des remontées de pannes (SRP).
Cette fonction est initialement projetée sur un seul composant logiciel CGP (composant de gestion pannes).

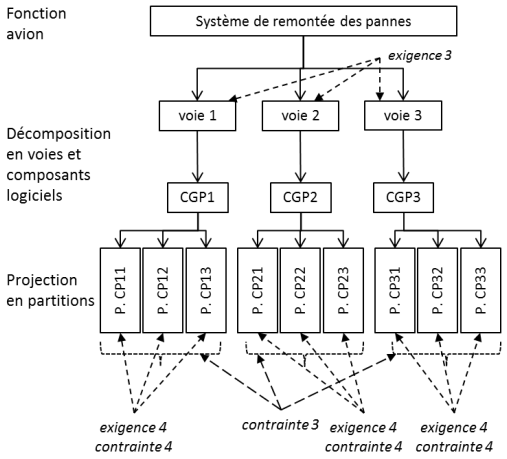


Figure 5 : décomposition de la fonction SRP

L'analyse de sûreté de fonctionnement nous impose deux exigences :
- Exigence 3 : afin d'offrir une disponibilité suffisante, le composant doit être tripliqué, chaque occurrence devant être matériellement ségréguée des autres ;

- Exigence 4 : chaque composant sera découpé suivant trois traitements projetés sur trois partitions différentes qui doivent, pour des raisons de temps, de séquencement et de rapidité des traitements, être co-localisées sur un même calculateur. Nous nous retrouvons avec un schéma de décomposition amenant à 9 partitions avec les contraintes suivantes (cf.

Figure 5):
- Contrainte 3 : les triplets de partitions {PCGP11, PCGP12, PCGP13}, {PCGP21, PCGP22, PCGP23} et {PCGP31, PCGP32, PCGP33} doivent être matériellement ségréguées;
- Contrainte 4 : toutes les partitions d'un même triplet doivent être projetées sur un même calculateur.

5.2 Formalisation en DEPS

5.2.1 Les éléments du système

Nous avons modélisé les éléments constitutifs d'une fonction avion aux différents niveaux d'abstraction nécessaires à l'expression des exigences de sûreté de fonctionnement.

```
Package Partition ;
Uses Universal;
Quantity CpuIndex
 Kind : Integer ;
 Min  : 1 ;
 Max : 4 ;
 Unit : u ;
End

Model Partition()
Constants
Variables
icpu : CpuIndex;
Elements
Properties
End
```

Figure 6 : Model DEPS d'une partition

Ce sont les modèles de : fonction avion, chaîne de traitement, composant logiciel, et partition (cf.
Figure 3).
- Une fonction avion est composée d'une ou plusieurs voies. Les voies sont des éléments du modèle de la fonction avion et sont donc des instances de chaîne de traitement.
- Une chaîne de traitement est composée d'un ou plusieurs composants logiciels ainsi que d'autres éléments (capteurs, actionneurs, réseau ...) hors du champ d'étude de cet article.
- Un composant logiciel est découpé en une ou plusieurs partitions.
- Une partition doit être projetée sur une ressource de calcul pour pouvoir s'exécuter (cf. Figure 6).

5.2.2 Les exigences

Le modèle qui suit montre comment on modélise au niveau des composants logiciels et des partitions l'exigence de redondance et de ségrégation de la chaine de traitement du système d'atterrissage (cf. Figure 7).
Les exigences sont propres à chaque fonction avion et elles portent à différents niveaux de décomposition de la fonction considérée.

```
Model NonColoc (P1, P2)
Constants
Variables
Elements
 P1 : Partition ;
 P2 : Partition ;
Properties
  P1.icpu - P2.icpu <> 1;
End

Model Segregation(C1, C2)
Constants
Variables
Elements
 C1 : Compl ; (* composant
                 logiciel 1 partition*)
 C2 : Comp1 ;
 NCL : NonColoc(C1.P, C2.P) ;
Properties
End

Model ChaineTraitementSA ()
Constants
Variables
Elements
 CP : Comp1() ;  (* composant principal *)
 CS : Comp1() ;  (* composant safety     *)
 SEG : Segregation (CP, CS) ;
Properties
End
```

Figure 7 : Modèle d'une chaîne de traitement du système d'atterrissage

En procédant ainsi on a pu modéliser l'exigence d'indépendance matérielle entre les deux voies dupliquées de la chaine de traitement du système d'atterrissage tout comme l'exigence de co-localisation sur un même calculateur des partitions des composant logiciels du système de remontée de pannes.

5.3 Résultats

Après modélisation des sept fonctions avions et de leurs exigences respectives on a obtenu les solutions de déploiement des partitions de ces fonctions sur les calculateurs. Les méthodes de résolution de DEPS retrouvent bien dans toutes les solutions que, conformément aux exigences, la fonction SAT a besoin de quatre calculateurs pour son déploiement tandis que la fonction SRP en a besoin de trois (cf. Figure 8).

.

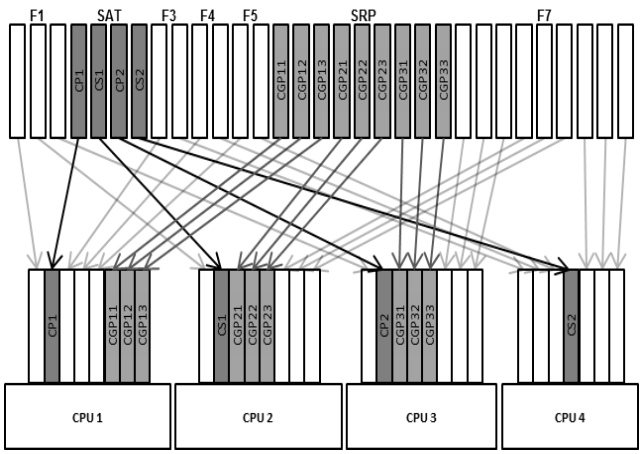


Figure 8 : Un déploiement des sept fonctions avion

6 CONCLUSIONS ET PERSPECTIVES

Dans cet article nous avons montré qu'il est possible de capturer des exigences de sûreté de fonctionnement difficiles à formaliser directement dans le formalisme (V, D, C) des CSP.

Nous avons utilisé le langage DEPS qui nous a permis de modéliser les bonnes abstractions pour d'une part décrire un modèle sous-défini d'architecture IMA et d'autre part des modèles d'exigences qui portent sur les bonnes abstractions.

Le problème de déploiement posé a été résolu en appliquant des méthodes de satisfaction de contraintes aux propriétés du problème.

Les traits de structuration offerts par le langage facilitent la réutilisation des modèles « métier » développés.

Les travaux en cours portent sur l'expression d'autres exigences d'architecture et sur leur prise en compte conjointe pour trouver des solutions de déploiement admissibles sur des problèmes à l'échelle.

REFERENCES

[1] Becz, S., Pinto, A., Zeidner, L. E., Banaszuk, A., Khire, R., and Reeve, H. M.,Design System for Managing Complexity in Aerospace Systems," 13th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, Texas, 2010

[2] Pinto A., Becz S., Reeve H.M., Correct-by-construction design of aircraft electric power systems, in AIAA Aviation Technology, Integration, and Operations Conf., 2010.

[3] Zeidner L, Reeve H, Khire R, Becz S., Architectural Enumeration and Evaluation for Identification of Low-Complexity Systems, MATIO10, Texas, 2010.

[4] Bieber P., J.P Bodeveix J.P., Castel C., Doose D., Filali M., Minot F., Pralet C., Constraint-based Design of Avionics Platform - Preliminary Design Exploration ERTS2008 Toulouse January 2008.

[5] Albarello, N., Welcomme J.B. and Reyterou C., A formal design synthesis and optimization for systems architectures, MOSIM'12, Bordeaux, France, 2012.

[6] Yvars P.A., Zimmer L., DEPS un langage pour la spécification de problèmes de conception de systèmes, MOSIM'14, Nancy, 2014.

[7] ARINC Specifications 653-1 Avionics Application Software Standard Interface, SAE ITC, 2017

[8] DO-297 Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations. RTCA, 2017.

[9] E. Tsang, Foundations of Constraint Satisfaction. London and San Diego: Academic Press, 1993.

[10] Benhamou F., Goualard F., Granvilliers L., Puget J.F., Revising Hull and Box consistency,16th International Conference on Logic Programming, 1993.

Analyse de Bytecode par Raffinement

Boubacar Demba Sall, Frédéric Peschanski, Emmanuel Chailloux
Sorbonne Universités, UPMC Univ Paris 06, CNRS, LIP6 UMR 7606
4 place Jussieu 75005 Paris, France.
{boubacar.sall | frederic.peschanski | emmanuel.chailloux}@lip6.fr

Résumé

La technique du *raffinement* permet la dérivation de programmes corrects par construction à partir des spécifications. Dans cet article nous proposons une application de cette technique formelle à l'analyse des programmes pour un langage de bas niveau de type *bytecode*. Il s'agit plus précisément de s'appuyer sur l'assistant de preuve *Coq* pour formaliser un analyseur de bytecode, et prouver sa correction.

1. Introduction

Dans cet article, nous nous intéressons à la preuve formelle de correction d'une analyse de programmes dans le système *Coq* [18]. Certains concepts manipulés en analyse de programmes (sémantique, domaine abstrait, invariant, garde, pré/post condition, ...) n'étant pas toujours présents dans le formalisme natif des assistants de preuve, un encodage de ces concepts peut s'avérer nécessaire. Dans le cas d'un *plongement profond* (*deep embedding*), l'introduction de nouvelles structures de données permet de représenter les concepts requis. On trouve dans [5] un plongement profond de la théorie de l'*interprétation abstraite* [8] qui permet de construire une analyse statique certifiée dans le système *Coq*. Dans le cas d'un *plongement léger* (*shallow embedding*), on s'efforce de se limiter au formalisme disponible pour exprimer les concepts requis. Comparé au plongement profond, un plongement léger est moins expressif mais permet de réduire les niveaux d'indirection qui complexifient le raisonnement. Le *raffinement* [19, 11, 10, 4] est une démarche d'abstraction qui a été bien étudiée, et qui semble se prêter à un plongement léger dans un formalisme issu de la théorie des types. On trouve dans [6] et [14] des exemples de plongements légers du raffinement permettant la dérivation de programmes corrects par construction dans le système *Coq*.

Nous proposons de privilégier un plongement léger, et d'appliquer la démarche de raffinement à la preuve de correction d'une analyse de programmes pour un langage de bas niveau de type *bytecode*. Dans la section 2, nous présentons le *raffinement de données* en faisant ressortir sa relation avec l'analyse de programmes. La section 3 présente les étapes d'un plongement léger dans le système *Coq* d'une analyse de bytecode, et décrit l'encodage des obligations de preuve (à satisfaire) permettant de valider la correction de l'analyseur. La section 4 présente quelques travaux connexes et la section 5 conclut cet article.

2. Raffinement de données et analyse de programmes

Soit $T = (\pi s, os_1, os_2, ..., os_n)$ une signature comprenant la signature πs d'un constructeur qui permet de créer un objet de type T, et les signatures os_i d'opérations qui agissent sur les objets de type T. Soient deux types abstraits de données $A = (\pi a, oa_1, oa_2, ..., oa_n)$ et $C = (\pi c, oc_1, oc_2, ..., oc_n)$ ayant la même signature T. Dans un souci de simplification nous nous limiterons aux opérations séquentielles et déterministes ($\pi a, oa_i, \pi c$ et oc_i sont des fonctions). On dit que C raffine A si pour tout programme $P[T]$ paramétré par leur signature commune, on a tout comportement observable de $P[C]$ est un comportement observable de $P[A]$ et $P[C]$ termine [1] à chaque fois que $P[A]$ termine. Pour prouver que C raffine A, il faut établir une correspondance entre leurs espaces d'état respectifs. Cette correspondance peut être spécifiée par un *invariant de liaison* $I(a, c)$ qui formalise le lien logique qui existe entre un état abstrait a et un état concret c [12].

Cas des opérations gardées. Dans le domaine des systèmes réactifs, les opérations sont liées à des évènements. L'applicabilité des opérations est généralement spécifiée par une garde : condition nécessaire à la survenue de l'évènement associé (ce dernier est réputé impossible dès lors que la garde correspondante est fausse, donc dans ce cas l'opération laisse l'état inchangé). On notera par Go_i la garde associée à l'opération o_i. Dans un tel contexte, pour prouver que C raffine A en s'appuyant sur l'invariant de liaison I, il suffit de réunir les conditions suivantes :

- *Etablissement de l'invariant* : $I(\ \pi a(...),\ \pi c(...)\ )$
- *Maintien de l'invariant* : $\forall\ i\ a\ c, I(a, c) \wedge Goc_i(c)\ \Rightarrow\ I(oa_i(a), oc_i(c))$
- *Renforcement des gardes* : $\forall\ i\ a\ c, I(a, c) \wedge Goc_i(c)\ \Rightarrow\ Goa_i(a)$

Il s'agit des obligations de preuve permettant de valider le raffinement de données dans les systèmes réactifs [3] en général, et entre machines *Event-B* [1] en particulier.

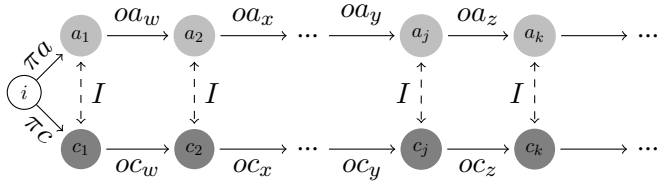
Cas des opérations partielles. Dans le cadre classique de la programmation séquentielle le domaine d'une opération est spécifié par une précondition : condition suffisante à une exécution sans erreur de l'opération (une violation de la précondition mène à un résultat indéfini). On notera par Po_i la précondition associée à l'opération o_i. Prouver que C raffine A dans ce cadre requiert les conditions suivantes :

- *Etablissement de l'invariant* : $I(\ \pi a(...),\ \pi c(...)\ )$
- *Maintien de l'invariant* : $\forall\ i\ a\ c, I(a, c) \wedge Poa_i(a)\ \Rightarrow\ I(oa_i(a), oc_i(c))$
- *Affaiblissement des préconditions* : $\forall\ i\ a\ c, I(a, c) \wedge Poa_i(a)\ \Rightarrow\ Poc_i(c)$

Il s'agit des obligations de preuve permettant de valider le raffinement entre modèles *Z*[16] ou *B classique*[2].

1. ne boucle pas indéfiniment, et s'arrête normalement (pas d'erreur à l'exécution)

Du raffinement de données à l'analyse de programmes. Comme l'illustre la figure ci-dessous, les obligations de preuve énoncées précédemment, assurent qu'à toute trace d'exécution $\tau_c = (c_1, c_2, ..., c_n)$ de $P[C]$ est associée une trace de même cardinalité $\tau_a = (a_1, a_2, ..., a_n)$ de $P[A]$ telle que $\forall\ j \cdot I(a_j, c_j)$.



Donc, lorsque C raffine A, pour tout programme P, et tout état concret c_j atteignable [2] par $P[C]$, il existe un état abstrait a_j atteignable par $P[A]$ tel que $I(a_j, c_j)$ est vrai. Cette correspondance entre les états atteignables de $P[C]$ et ceux de $P[A]$ peut être exploitée à des fins de vérification. En particulier, si tous les états atteignables a de $P[A]$ sont tels que $Poa_i(a)$ est vrai à chaque fois que oa_i est exécutée dans l'état a, alors la condition d'*affaiblissement des préconditions* nous garantit que tous les états atteignables c de $P[C]$ sont tels que $Poc_i(c)$ est vrai à chaque fois que oc_i est exécutée dans l'état c. Il s'agit d'une propriété intéressante car elle garantit qu'aucune opération ne sera jamais activée en dehors de son domaine de définition. La relation de raffinement entre A et C permet de déduire cette propriété de sûreté pour $P[C]$ à partir du moment où on a réussi à établir cette même propriété pour $P[A]$. En d'autres termes, si C raffine A, alors analyser $P[A]$ permet de conclure à propos de $P[C]$.

Pour concevoir une analyse, il peut être intéressant de s'appuyer sur des propriétés supposées établies (grâce par exemple à une autre analyse). Prenons l'exemple du langage de bytecode de la *JVM*, et de l'instruction *getfield f* qui permet de charger le champ *f* d'un objet à partir d'une référence. Durant l'exécution d'un programme bien formé, à chaque fois que l'instruction *getfield* est sur le point d'être exécutée, nous pouvons compter sur le fait que la référence de l'objet est présente en tête de la pile car la *JVM* effectue une analyse permettant de s'en assurer. Ceci peut être exprimé plus formellement en considérant que cette instruction admet la *garde* suivante : *length* (*stk c*) ≥ 1, où *c* représente un état concret et *stk* est un accesseur qui retourne la pile des opérandes. Par contre, nous ne pouvons pas toujours compter sur le fait que cette référence est différente de *null*, alors que dans ce cas l'opération n'est pas définie (elle provoque une erreur d'exécution). On considérera donc que cette instruction requiert la précondition suivante : *top* (*stk c*) $\neq$ *null*. Nous utiliserons des gardes pour représenter les propriétés supposées établies, et des préconditions pour représenter les propriétés à établir. Il nous faut donc considérer le raffinement dans le cas des opérations partielles et gardées.

Raffinement des opérations partielles et gardées. Lorsqu'on associe à une opération à la fois une garde et une précondition, nous l'interpréterons comme suit : (1) l'opération est définie si $Go_i \Rightarrow Po_i$, (2) si l'opération s'exécute c'est que Go_i

2. un état ϕ est atteignable par P ssi il existe une trace $\tau = (..., \phi, ...)$ admissible par P

est vrai au début de l'exécution, (3) l'opération est indéfinie si $Go_i \wedge \neg Po_i$. Cette interprétation est tirée de [15], où des conditions suffisantes de raffinement sont aussi proposées. Cependant le traitement est limité au raffinement algorithmique (sans changement de représentation des données).

Pour étudier la situation dans le contexte du raffinement de données, plaçons nous d'abord dans le cas des opérations gardées, et observons ensuite ce qui se passe lorsque les opérations sont rendues partielles. Supposons que C raffine A et considérons un programme P. Soit une trace $\tau_a = (..., a_j, a_k, ...)$ admissible par $P[A]$ en correspondance avec une trace $\tau_c = (..., c_j, c_k, ...)$ admissible par $P[C]$. Si maintenant on rend les oc_i partielles en supprimant c_j de leur domaine (pour ce faire il suffit d'associer les préconditions appropriées à ces opérations), la trace τ_c n'est plus admissible par $P[C]$. La relation de raffinement peut perdurer tant que d'autres traces correspondent encore à τ_a. Mais quand τ_c est la seule trace en correspondance avec τ_a, C ne raffine plus A. Cette situation est celle que permet d'éviter l'obligation d'*affaiblissement des préconditions* dans le cas des opérations partielles. Ainsi en fusionnant les obligations de preuve des deux cas précédents, et en apportant une légère modification à l'obligation de *maintien de l'invariant* (pour tenir compte de la partialité des opérations), on obtient les conditions suffisantes pour prouver que C raffine A dans un contexte où les opérations peuvent être partielles et gardées :

- *Etablissement de l'invariant* : $I(\ \pi a(...),\ \pi c(...)\ )$
- *Maintien de l'invariant* : $\forall\ i\ a\ c, I(a,c) \wedge Goc_i(c) \wedge Poa_i(a)\ \Rightarrow\ I(oa_i(a), oc_i(c))$
- *Renforcement des gardes* : $\forall\ i\ a\ c, I(a,c) \wedge Goc_i(c)\ \Rightarrow\ Goa_i(a)$
- *Affaiblissement des préconditions* : $\forall\ i\ a\ c, Poa_i(a) \wedge I(a,c)\ \Rightarrow\ Poc_i(c)$

3. Formalisation et preuve d'une analyse de bytecode en *Coq*

Soit un programme P constitué d'une liste d'instructions bytecode. Soit S_c la sémantique opérationnelle associée au langage de bytecode en question. On peut considérer que P est paramétré par le jeu d'instructions du langage (la signature de S_c). Une analyse de programme pour le langage de bytecode peut alors être spécifiée sous la forme d'une sémantique abstraite S_a ayant la même signature que S_c et décrivant de manière opérationnelle le fonctionnement de l'analyseur. Si S_c raffine S_a alors l'analyse est correcte dans le sens où, les propriétés de sûreté établies pour $P[S_a]$ sont valables pour $P[S_c]$. On peut donc en déduire la démarche suivante pour prouver la correction d'une analyse de programmes étant donnée la sémantique concrète S_c du langage de programmation : il suffit (1) de formaliser S_c et S_a, (2) de formaliser l'invariant de liaison qui relie les deux sémantiques, et (3) de satisfaire aux obligations de preuve énoncées plus haut afin d'établir que S_c raffine S_a.

Syntaxe et environnements d'exécution. Pour formaliser les sémantiques, nous commençons par spécifier la syntaxe des instructions du langage, ensuite on décrit les environnements d'exécution concret et abstrait, ainsi que les environnements

d'exécution initiaux. Par exemple, dans l'extrait ci-dessous on s'intéresse au langage de bytecode de la *JVM* et à la détection des dé-référencements de pointeurs *null*. On déclare donc le type *Instr* dont chacun des constructeurs représente une instruction du langage. On déclare ensuite les types *ConcreteState* et *AbstractState* qui représentent respectivement les environnements d'exécution concret et abstrait. On définit enfin *ConcreteInit* et *AbstractInit* qui représentent respectivement les environnements initiaux concret et abstrait.

Inductive *Instr* := *new* | *aload v* | *astore v* | *getfield f* | *setfield f* | *ifnonnull* | ...

Environnement concret	Environnement abstrait
Record *ConcreteState* := *mkCS* {	***Record*** *AbstractState* := *mkAS* {
stk : list Ref ;	*stk : list AbstractRef ;*
heap : Ref → Field → Ref	*nonnull : list AbstractRef*
}.	}.
Definition *ConcreteInit* := *mkCS [] (fun ...).*	***Definition*** *AbstractInit* := *mkAS [] [].*

AbstractState représente l'environnement d'exécution du futur analyseur. Le champ *nonnull* est une liste des références symboliques (abstraites) pour lesquelles les références concrètes correspondantes ont été identifiées comme différentes de *null*. Cette liste pourra être mise à jour à la création d'une nouvelle référence ou suite à un test de nullité, l'analyseur pourra ainsi s'assurer de la non nullité d'une référence en testant sa présence dans cette liste.

Invariant de liaison. La définition de l'invariant de liaison va consister à fournir une fonction qui construit un prédicat reliant un état abstrait a à un état concret c. Dans l'exemple ci-dessous, l'invariant de liaison stipule que dans les deux contextes d'exécution la pile des opérandes contient le même nombre d'éléments, et que si une référence a été identifiée dans l'abstraction comme n'étant pas *null*, alors son vis-à-vis dans le contexte d'exécution concret n'est effectivement pas *null*.

Definition *Invariant (a : AbstractState) (c : ConcreteState) : Prop* :=
length (stk a) = length (stk c) ∧
∀ v_a v_c, *In* (v_a,v_c) *(combine* [3]*(stk a) (stk c))* ∧ *In* v_a *(nonnull s)* → $v_c \neq$ *null*

Sémantiques des instructions. Pour spécifier le fonctionnement opérationnel de chaque instruction, on définira les fonctions *Guard*, *Pre* et *Exec* préfixées de *Concrete* ou *Abstract* selon la sémantique concernée. Dans la continuité des exemples précédents, l'extrait ci-dessous illustre une définition des sémantiques concrète et abstraite d'instructions bytecode. Les paramètres *i*, *c* et *a* sont respectivement de type *Instr*, *ConcreteState* et *AbstractState*. Les fonctions sont définies par filtrage de l'argument *i*. *ConcreteGuard* et *AbstractGuard* spécifient les gardes pour chaque instruction. *ConcretePre* et *AbstractPre* spécifient les préconditions associées à chaque instruction.

3. *combine : (list A) → (list C) → list (A×C)*
(combine $[v_{a1}; v_{a2}; ...; v_{an}]$ $[v_{c1}; v_{c2}; ...; v_{cn}]) = [(v_{a1}, v_{c1}); (v_{a2}, v_{c2}); ...; (v_{an}, v_{cn})]$

Sémantique concrète	Sémantique abstraite
Definition *ConcreteGuard i c : Prop :=* ***match*** *i* ***with*** *| getfield f ⇒ (length (stk c)) ≥ 1* *| ...* ***end***.	***Definition*** *AbstractGuard i a : Prop :=* ***match*** *i* ***with*** *| getfield f ⇒ (length (stk a)) ≥ 1* *| ...* ***end***.
Definition *ConcretePre i c : Prop :=* ***match*** *i* ***with*** *| getfield f ⇒ (top (stk c)) ≠ null* *| ...* ***end***.	***Definition*** *AbstractPre i a : Prop :=* ***match*** *i* ***with*** *| getfield f ⇒ In (top (stk a)) (nonnull a)* *| ...* ***end***.
Definition *ConcreteExec i c* *: option ConcreteState :=* ***match*** *i,(stk c)* ***with*** *| getfield f, r::s ⇒* *Some (mkCS ((heap c r f)::s) (heap c))* *| ...* ***end***.	***Definition*** *AbstractExec i a* *: option AbstractState :=* ***match*** *i,(stk a)* ***with*** *| ifnonnull, r::s ⇒* *Some (mkAS s (r::(nonnull a)))* *| ...* ***end***.

ConcreteExec et *AbstractExec* sont partielles (d'où le type *option* dans leur déclaration), elles retournent *None* lorsque la garde ou la précondition est fausse, dans le cas contraire elles calculent une mise à jour de l'environnement d'exécution correspondant.

Obligations de preuve. Afin de prouver la correction de l'analyse, nous devons maintenant énoncer les obligations de preuve nous permettant de conclure que la sémantique concrète raffine la sémantique abstraite :

Lemma *Initiation : Invariant AbstractInit ConcreteInit.*

Lemma *Invariance :* $\forall\ i\ a\ c$, *Invariant* $a\ c \wedge$ *ConcreteGuard* $i\ c \wedge$ *AbstractPre* $i\ a$ $\rightarrow \forall\ a'\ c'$, *AbstractExec* $i\ a$ = *Some* $a' \rightarrow$ *ConcreteExec* $i\ c$ = *Some* $c' \rightarrow$ *Invariant* $a'\ c'$.

Lemma *GuardStrengthening :* $\forall\ i\ a\ c$, *Invariant* $a\ c \wedge$ *ConcreteGuard* $i\ c \rightarrow$ *AbstractGuard* $i\ a$.

Lemma *PreconditionWeakening :* $\forall\ i\ a\ c$, *AbstractPre* $i\ a \wedge$ *Invariant* $a\ c \rightarrow$ *ConcretePre* $i\ c$.

Il nous reste ensuite à établir la validité des lemmes énoncés ci-dessus en nous appuyant sur les tactiques disponibles dans le système *Coq*. Une fois cette étape effectuée, notre analyse est correcte vis-à-vis du langage cible tel que spécifié par la sémantique concrète.

4. Travaux connexes

Les méthodes *Z* et *B* sont basées sur le raffinement de données mais diffèrent de l'approche présentée ici. En particulier, ces méthodes, comme les plongements en *Coq* du raffinement qu'on trouve dans [6, 7, 9], donnent une interprétation différente à la combinaison garde/précondition. Il est en principe possible de contourner le problème, mais cela peut mener à des obligations de preuve plus fortes, donc à un effort de preuve plus important. Par exemple, en notation *B classique*, on pourrait écrire ***pre*** $Go_i \Rightarrow Po_i$ ***then*** *(****if*** Go_i ***then*** *... **else** skip)* pour spécifier une opération de précondition Po_i gardée par Go_i. Néanmoins, l'obligation de *maintien de l'invariant* serait alors plus forte, en particulier dans le cas où on a $Goa_i(a) \wedge$

$\neg Goc_i(c)$, il faudrait montrer $I(a,c) \wedge Poa_i(a) \Rightarrow I(oa_i(a),c)$, alors que ce cas est trivialement vrai dans notre approche.

La démarche de raffinement est proche de l'interprétation abstraite. En particulier la condition de correction d'un interprète abstrait est une condition de raffinement vis-à-vis de l'invariant de liaison $I(a,c) \equiv c \in \gamma(a)$, où γ est une *fonction de concrétisation* [17]. Cependant, les définitions fonctionnelles sont contraintes dans le formalisme de *Coq* (notamment pour assurer la totalité). Lorsque ces contraintes sont trop fortes, il est possible de recourir aux prédicats inductifs qui sont plus expressifs, ou à une axiomatisation de γ [5], mais cela nécessite un plongement profond. En revanche, la définition d'un invariant de liaison, de nature essentiellement logique, est assez naturelle dans un formalisme basé sur la théorie des types.

5. Conclusion

Nous avons proposé une application de la démarche de raffinement à la preuve de correction d'une analyse de programmes. Une fois la correction prouvée, une implémentation exécutable de l'analyseur consiste à parcourir l'espace d'état abstrait en appliquant *AbstractExec* à l'état abstrait initial, puis à tous les états atteignables par application de cette fonction. A chaque fois qu'un nouvel état est généré, on s'assure que ce dernier est sûr en évaluant les préconditions des instructions qui peuvent être exécutées dans cet état. Si l'état n'est pas sûr la vérification échoue et l'algorithme s'arrête. Si l'algorithme termine sans rencontrer d'état ne satisfaisant pas les critères de sûreté, alors la vérification a réussi. A moins qu'il soit possible d'exhiber une mesure décroissante, la convergence de cet algorithme n'est pas garantie, par conséquent il faut se donner un nombre maximum d'itérations. L'aspect de la convergence reste donc à approfondir. Il serait également intéressant de pouvoir procéder à une extraction certifiée de code à partir de la sémantique de l'analyseur. Enfin, il est nécessaire de garder à l'esprit le risque d'explosion de l'espace d'état durant la conception de la sémantique abstraite.

L'approche présentée dans cet article est actuellement appliquée dans le cadre d'une étude de cas qui vise une analyse de bytecode capable de détecter des erreurs de programmation liées à l'*aliasing* [13] dans les programmes objet. On s'intéresse à l'échappement des références qui peut constituer une violation de l'encapsulation et ainsi aboutir à des problèmes d'intégrité de données. Pour éviter ces situations, on restreint le domaine de l'instruction *setfield* au cas où la référence cible est présente dans la pile d'appels. Actuellement, le développement [4] *Coq* compte environ 4200 lignes de script, et une centaine de preuves. Sur le plan pratique, la démarche de raffinement donne un cadre formel de réflexion. Le recours à un plongement léger dans un style opérationnel permet de simplifier les preuves. En particulier, cela permet de tirer avantage des tactiques de simplification, et les concepts manipulés sont assez proches de leurs descriptions formelles, ce qui aide à maîtriser la complexité.

4. Sources *Coq* sur `https://github.com/bsall/afadl-2017`

Références

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

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

[3] R. Back. Refinement calculus, part ii : Parallel and reactive programs. In *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness*, pages 67–93. Springer, 1990.

[4] R.-J. Back and J. Wright. *Refinement calculus : a systematic introduction*. Springer Science & Business Media, 2012.

[5] Y. Bertot. Structural Abstract Interpretation : A Formal Study Using Coq. In *Language Engineering and Rigorous Software Development*, pages 153–194. Springer, 2009.

[6] S. Boulmé. Intuitionistic refinement calculus. In *International Conference on Typed Lambda Calculi and Applications*, pages 54–69. Springer, 2007.

[7] C. Cohen, M. Dénes, and A. Mörtberg. Refinements for free ! In *International Conference on Certified Programs and Proofs*, pages 147–162. Springer, 2013.

[8] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[9] B. Delaware, C. Pit-Claudel, J. Gross, and A. Chlipala. Fiat : Deductive synthesis of abstract data types in a proof assistant. In *ACM SIGPLAN Notices*, volume 50, pages 689–700. ACM, 2015.

[10] P. Gardiner and C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87(1) :143–162, 1991.

[11] J. He, C. Hoare, and J. W. Sanders. Data refinement refined resume. In *European Symposium on Programming*, pages 187–196. Springer, 1986.

[12] C. A. R. Hoare. Proof of correctness of data representations. In *Software pioneers*, pages 385–396. Springer, 2002.

[13] J. Hogg, D. Lea, A. Wills, D. de Champeaux, and R. Holt. The Geneva Convention on the Treatment of Object Aliasing. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 7–14. Springer, 2013.

[14] A. Joao and S. Wouter. Embedding the refinement calculus in coq. *SCP (submitted)*, 2016. `http://www.staff.science.uu.nl/~swier004/publications/2017-scp-draft.pdf`.

[15] R. Miarka, E. Boiten, and J. Derrick. Guards, preconditions, and refinement in Z. In *International Conference of B and Z Users*, pages 286–303. Springer, 2000.

[16] J. M. Spivey and J. Abrial. *The Z notation*. Prentice Hall Hemel Hempstead, 1992.

[17] A. Spiwack. Abstract interpretation as anti-refinement. *arXiv preprint arXiv :1310.4283*, 2013.

[18] C. D. Team et al. The Coq proof assistant reference manual Version 8.5. *TypiCal Project (formerly LogiCal)*, 2015.

[19] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4) :221–227, 1971.

Une ligne de produits corrects par construction

Thi-Kim-Dung Pham[1], Catherine Dubois[2], Nicole Levy[1]
1. Conservatoire National des Arts et Métiers, lab. Cedric, Paris
2. ENSIIE, lab. Samovar, Évry

Résumé

L'ingénierie des lignes de produits logiciels met l'accent sur la gestion de la variabilité et la réutilisation. Dans cet article, nous complétons cette approche avec la recherche de garanties de correction sur les produits issus d'une ligne de produits. Nous proposons une méthode permettant de produire des produits corrects par construction à partir d'une ligne de produits. Cette méthode s'appuie sur un langage, FFML, inspiré de FoCaLiZe et incorporant des mécanismes pour exprimer la variabilité, et deux outils : un compilateur vers FoCaLiZe et un composeur. Le composeur permet de générer automatiquement des produits corrects par construction à partir d'une configuration valide. Les outils de FoCaLiZe permettent de vérifier les preuves de correction et de générer du code exécutable OCaml. La méthode est illustrée dans cet article sur un exemple simple de ligne de produits.

1 Introduction

D'après Clements et Northrop dans [3], une ligne de produits décrit un ensemble (fini) de systèmes qui partagent un ensemble commun de caractéristiques (*features* en anglais) répondant à des besoins spécifiques, par exemple ceux d'un segment de marché ou d'un domaine. Ces systèmes sont développés par une composition précise d'un ensemble commun d'artefacts de base (*assets* en anglais).

Ainsi, l'utilisation d'une ligne de produits a principalement deux objectifs : l'expression de la variabilité et la réutilisation. Une ligne de produits est spécifique à un domaine et exprime la variabilité du domaine et des besoins du problème décrit. Elle identifie systématiquement les points communs et les points variables. Ces points sont les caractéristiques mentionnées dans la définition de Clements et Northrop. Pour les visualiser, il est fréquent d'utiliser un diagramme de caractéristiques (*Feature diagram*) qui permet de structurer les décisions sous forme d'un arbre *FODA* (pour *Feature Oriented Domain Analysis*) [5] avec des relations entre nœuds père et fils qui peuvent être Obligatoires ou Optionnelles, ou décrire des choix exclusifs ou multiples.

Le deuxième objectif essentiel des lignes de produit est la réutilisation. Pour cela, à chaque caractéristique sont associés des artefacts à réutiliser. Ainsi, configurer un système c'est choisir des caractéristiques et réutiliser les artefacts associés.

Dans les différentes définitions des lignes de produits, la notion d'artefact varie et peut recouvrir différents sens. Il peut s'agir d'une spécification en langue naturelle ou en langage formel ou même d'un morceau de code. L'idée est que l'artefact représente la décision associée à une caractéristique et le problème est que pour le réutiliser, il faut pouvoir le composer avec d'autres artefacts.

Notre objectif est la définition d'une ligne de produits dans laquelle un artefact décrit formellement est associé à chaque caractéristique avec, pour objectif, de pouvoir configurer des systèmes corrects par construction, c'est-à-dire prouvés corrects par rapport à leur spécification. Remarquons, que si de nombreux travaux portent sur la validation d'une ligne de produits, peu portent sur la validation des produits issus de lignes de produits. Dans [1], les auteurs proposent de trouver des défauts dans les produits générés en C et Java en suivant une démarche basée sur le model-checking. Notre approche est proche de celle de Thüm [8] qui utilise la notion de contrat et les vérifie de manière déductive. Nous nous distinguons principalement par le fait que nous réutilisons et composons les preuves automatiquement.

Afin de pouvoir déployer une ligne de produits corrects par construction, nous proposons un langage adapté à la description des lignes de produits, orienté variabilité, FFML, ainsi qu'un mécanisme de composition qui permet de construire des produits corrects par construction. Le langage FFML est inspiré de FoCaLiZe, langage qui permet de spécifier un logiciel, de l'implémenter dans un langage fonctionnel et de prouver la correction du programme à l'aide du prouveur Zenon [4]. FFML, comme nous l'avons montré dans [7], lui apporte une orientation Ligne de Produits en introduisant des modules spécifiques et des constructions décrivant les décisions de développement prises. Deux outils accompagnent cette proposition : un compilateur de FFML vers FoCaLiZe ainsi qu'un outil de génération automatique de produits par composition d'artefacts définis à l'aide du langage FFML. Les deux outils sont présentés succinctement ici, ils sont formalisés dans [6].

La ligne de produits utilisée comme exemple dans cet article, inspirée de [8], décrit la gestion d'un compte bancaire. La caractéristique racine, notée BA pour BankAccount, décrit la gestion de base d'un compte : mémorisation du solde courant et transactions (ajout ou retrait d'argent) sur un compte. Un client peut retirer plus d'argent du compte que disponible dans une limite fixée. Comme le montre le diagramme de caractéristiques de la figure 1, la caractéristique BA dispose de trois caractéristiques filles optionnelles DailyLimit (DL), LowLimit (LL) et Currency (CU). DL permet à la banque de limiter le montant retiré chaque jour. LL autorise un client à débiter son compte uniquement d'un montant supérieur à une limite basse. CU permet de gérer la devise. Enfin, la caractéristique CE pour CurrencyExchange, fille optionnelle de CU,

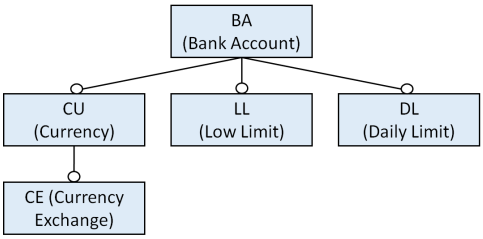


FIGURE 1 – Ligne de produits des comptes bancaires

réalise la conversion de devises. Un produit est configuré par une sélection de ces caractéristiques. Par exemple, un utilisateur qui désire un système de gestion bancaire avec une limite fixée pour les retraits journaliers et à chaque retrait, choisira les caractéristiques BA, LL et DL.

L'article est structuré de la manière suivante : le langage FFML est présenté dans la section suivante et illustré sur l'exemple de la ligne de produits des comptes bancaires. En section 3, nous montrons comment est généré un produit correct par construction à partir d'une configuration. Nous concluons en section 4.

2 Développement de la ligne de produits

2.1 Artefacts en FFML

A chaque nœud du modèle de caractéristiques est associé un artefact écrit dans un module du langage FFML. Ce dernier contient 3 parties : la spécification, l'implantation et les preuves de correction. La spécification contient les signatures des fonctions développées dans le module et les propriétés attendues sous la forme de formules du premier ordre, l'implantation des fonctions est décrite dans un style fonctionnel et les preuves de correction des fonctions sont faites par rapport aux propriétés énoncées dans la spécification. Le langage FFML est inspiré de FoCaLiZe, ainsi les preuves sont écrites dans le langage de preuve de FoCaLiZe et apparaissent dans FFML sous la forme de commentaires spécialisés. La particularité de FFML est qu'il demande de ne décrire, pour une caractéristique donnée, que ce qui change par rapport à la caractéristique dont elle est issue (ou caractéristique parent) : nouvelles fonctions, propriétés nouvelles ou modifiées, nouvelles preuves. La syntaxe concrète et la sémantique de FFML sont décrites dans [6].

Les modules associés aux nœuds BA, DL et LL du modèle de la figure 1 sont reproduits (partiellement) à la figure 2, ces modules sont répartis dans des fichiers différents et portent le nom de la caractéristique associée.

Le listing 1 montre le code associé à la caractéristique racine BA. Deux fonctions $update$ et get_bal ainsi qu'une constante $over$ sont déclarées et définies. Le type BA désigne le type des comptes bancaires, il est associé au type concret int à la ligne 11. Un compte est donc représenté par son solde. La spécification de la fonction $update$ (i.e. son contrat) est ici la propriété bal_succ_BA. Elle indique que le retrait ou le dépôt est possible si, après cette opération, une somme suffisante reste sur le compte (solde supérieur à $over$). Une propriété dite invariante, $ba\_bal\_gr\_over$, spécifie la fonction get_bal. Cette propriété devra être satisfaite par toutes les caractéristiques filles et donc tous les produits issus de la ligne de produits. Le module se termine par les scripts de preuve des propriétés précédentes : la preuve de la première propriété se fait en utilisant la définition des fonctions $update$ et get_bal, la seconde propriété est admise, elle ne peut être prouvée directement. Elle est en fait induite par les preuves que la propriété est préservée par les opérations qui ont un résultat de type BA, par exemple ici la preuve de la propriété bal_succ_BA.

```
fmodule BA
  signature update: BA -> int -> BA;
  signature get_bal: BA -> int;
  signature over: int;
  contract get_bal :: invariant property
        ba_bal_gr_over:
   all x : BA, get_bal(x) >= over;
  contract update :: property bal_succ_BA: all
        x : BA, all a : int,
   (get_bal(x) + a) >= over -> get_bal (update
        (x,a)) = get_bal(x) + a;
  representation = int;
  let get_bal(x) = x;
  let over = 0;
  let update (x, a) =
     if ((get_bal(x) + a) >= over) then
          get_bal(x) + a
     else get_bal(x);
  proof of ba_bal_gr_over = assumed;
  proof of bal_succ_BA = by definition of
        update, get_bal;
```

Listing 1 – BA

```
fmodule LL from BA
  signature limit_low: int;
  contract update :: property bal_succ_LL
   refines BA!bal_succ_BA
   extends premise ((a >= 0) || (a <=
        limit_low));
  representation = BA;
  let limit_low = -10;
  let update (x, a) =
    if ((a >= 0) || (a <= limit_low))
    then BA!update (x, a)
    else x;
  proof of bal_succ_LL = by definition of
        update, get_bal, over
     property BA!bal_succ_BA, int_ge_le_eq;
```

Listing 2 – LL

```
fmodule DL from BA
  signature limit_with: int ;
  signature get_with: DL -> int ;
  signature next_day : DL -> DL;
  contract update :: property bal_succ_DL
   refines BA!bal_succ_BA
   extends premise (a <= 0) /\ (get_with(x) +
        a >= limit_with);
  contract update :: property dl_upd_limit:
        all x : DL, all a : int,
   ((a<=0)/\(get_with(x)+a>=limit_with)) ->
     get_with(update(x,a) = get_with(x) + a;
  representation extends BA with int;
  let limit_with = -70;
  let get_with (x) = second(x);
  let next_day (x, a) = (x, limit_with);
  let update (x, a) =
     if (a <= 0) then
          if (get_with(x)+a>=limit_with) then
            (BA!update(x,a), get_with(x)+a)
          else x
     else (BA!update(x,a), get_with(x));
  proof of bal_succ_DL =
    <1>1 assume x: DL, a : int,
     hypothesis h1: (a <= 0) /\ (get_with(x) +
          a >= limit_with),
     prove (get_bal(x) + a) >= over -> (
          get_bal (update(x,a))) = get_bal(x)
          + a
     <2>1 prove first(update(x,a)) = BA!update
          (first(x),a)
          by definition of first, make,
               update hypothesis h1
     <2>e qed by step <2>1 definition of over,
          get_bal property BA!bal_succ_BA
    <1>e conclude ;
  proof of dl_upd_limit = by definition of
       update, get_with;
```

Listing 3 – DL

FIGURE 2 – Quelques artefacts de la ligne de produits des comptes bancaires

Le listing 3 définit DL à partir de BA. Le module introduit deux nouvelles fonctions get_with et $next\_day$ (les propriétés et preuves relatives à $next\_day$ ne sont pas développées ici) et une nouvelle constante $limit\_with$. Toutes les fonctions définies dans BA sont présentes dans DL mais $update$ a un comportement légèrement modifié : on ne doit pas dépasser le montant journalier de retrait autorisé. La spécification de $update$ est donc un raffinement de la spécification de $update$ dans BA (cette construction a été décortiquée dans [7]). On remarquera que le script de preuve de cette propriété réclame la preuve d'un lemme intermédiaire. Une nouvelle propriété de $update$, dl_upd_limit, est ajoutée et prouvée. Le type concret est étendu : on adjoint à la représentation du compte bancaire dans BA, un autre entier qui est la somme retirée dans le jour. La représentation devient donc un couple d'entiers. La fonction $update$ est redéfinie et ré-utilise la fonction $update$ de BA (notée $BA!update$). Le module LL (listing 3) suit un schéma similaire.

2.2 Vérification des preuves de correction

La compilation en FoCaLiZe, à ce stade, permet de faire les preuves des propriétés et de les vérifier. La démarche du développeur est la suivante : pour une caractéristique donnée, il écrit le code FFML et le compile à l'aide du compilateur FFML ; il obtient un premier code FoCaLiZe dont il complète les scripts de preuve ; les preuves sont faites par Zenon et le tout est vérifié - par Coq - lorsque la compilation du fichier FoCaLiZe est lancée. Une fois les scripts de preuve complétés et vérifiés, ils sont copiés-collés aux bons endroits dans le fichier FFML, en tant que commentaires.

3 Génération de produits corrects par construction

Une fois la ligne de produits décrite en FFML, un utilisateur peut obtenir différents produits de manière automatique (ou quasi-automatique). Pour cela, il doit fournir une configuration valide, c'est-à-dire un ensemble de caractéristiques qui respectent les contraintes exprimées dans le modèle. Par exemple {BA, DL, LL} est une configuration valide mais {BA, DL, CE} ne l'est pas car CE requiert la présence de CU. La vérification de la validité d'une configuration est orthogonale à notre travail et de nombreux outils rendant ce service existent (par exemple [2]). L'outil présenté ici se veut complémentaire des outils existants et considère les configurations étudiées comme ayant déjà été validées. Nous supposons également que les contraintes du modèle de caractéristiques font foi. Ainsi nous ne vérifions pas que les caractéristiques puissent engendrer des interactions. Si deux caractéristiques ne sont pas compatibles, une contrainte d'exclusion est attendue.

A partir d'une configuration valide et des fichiers FFML associés aux caractéristiques choisies, le composeur FFML produit un fichier FFML qui correspond au produit demandé. A l'aide du compilateur FFML vers FoCaLiZe, il est traduit en un fichier FoCaLiZe qui, une fois compilé produira un exécutable en OCaml.

Le résultat de la composition pour la configuration {BA, LL, DL} est illustré ci-dessous (listing 4).

```
1  fmodule DLLL from LL
2    signature limit_with : int;
3    signature get_with : DLLL -> int;
4    signature next_day : DLLL -> DLLL;
5    contract update ::  property bal_succ_DL
6     refines LL!bal_succ_LL
7     extends premise (a <= 0) /\ (get_with(x) + a >= limit_with);
8    representation extends LL with int;
9    ...
10   let update (x,a) =  if (a <= 0) then
11    if (get_with(x) + a >= limit_with)
12    then (LL!update(x,a),get_with(x) + a)
13    else x
14    else (LL!update(x,a),get_with(x));
15   proof of bal_succ_DL =  ... ;
```

Listing 4 – Le module FFML correspondant à la configuration {BA, LL, DL}

On peut le voir comme une extension/modification des fonctionnalités associées à LL. Le composeur agit en appliquant une opération de composition binaire,

capable de combiner les propriétés, les définitions de fonction et les scripts de preuve [6]. Cette opération génère automatiquement de nouvelles propriétés, implantations et scripts de preuve. Les modules FFML associées aux caractéristiques de la configuration sont composés deux par deux en suivant un parcours *Gauche-Droite-Racine* du modèle de caractéristiques. Notons que l'arbre FODA considéré est ordonné et que la composition suit cet ordre, donnant, par exemple pour {BA, DL, LL}, un module qui dérive de LL plutôt que de BA ou DL.

La table 1 présente un bilan quantitatif de la génération de produits pour l'exemple des comptes bancaires. Le code FFML compte 14 propriétés en FFML et 18 preuves faites par Zenon. Il y a plus de preuves en FoCaLiZe qu'en FFML car la compilation vers FoCaLiZe génère des preuves, toutes faites automatiquement [7]. A partir de cette ligne de produits, nous pouvons construire 12 produits. La colonne cP (resp. cPf) dénombre les propriétés (resp. scripts de preuve) générées par la composition. La colonne auto indique le nombre de preuves faites par Zenon à partir des scripts générés par la composition et manu le nombre de preuves dont le script a été donné manuellement [1]. La colonne Ze-Pf donne le nombre total de preuves à faire. Les autres configurations ne figurent pas dans la table car les produits correspondants sont les modules FFML associés aux nœuds du modèle de caractéristiques. Ainsi, le produit associé à {BA, CU, CE} est le module CE.

Configuration	cP	cPf	Ze-Pf	auto	manu
{BA,DL,LL}	5	6	7	7	0
{BA,DL,CU}	4	5	7	6	1
{BA,LL,CU}	3	3	6	6	0
{BA,DL,LL,CU}	5	6	8	7	1
{BA,LL,CU,CE}	3	3	8	7	1
{BA,DL,CU,CE}	3	3	9	7	2
{BA,DL,LL,CU,CE}	5	6	10	8	2

TABLE 1 – Bilan quantitatif

4 Conclusion

Dans cet article, nous avons illustré la méthode que nous proposons pour produire des produits corrects par construction à partir d'une ligne de produits. Cette méthode s'appuie sur un langage, FFML, inspiré de FoCaLiZe et incorporant des mécanismes pour exprimer la variabilité, et deux outils : un compilateur et un composeur. Le composeur permet de générer automatiquement des produits corrects par construction à partir d'une configuration valide et le compilateur permet de traduire tout module FFML en un programme FoCaLiZe. Cette compilation permet non seulement d'obtenir du code exécutable écrit en OCaml mais aussi de vérifier les preuves. Actuellement, certains scripts de preuve doivent être produits manuel-

1. Tous les scripts manuels sont dus à la même erreur de Zenon qui est en cours de correction.

lement, des travaux complémentaires sont en cours pour automatiser ces preuves. Enfin, la méthode est expérimentée actuellement sur un exemple plus conséquent : une ligne de produits concernant le poker et ses différentes règles applicables. Cette ligne de produits permet de générer automatiquement 16 produits et leurs propriétés, en combinant ses 12 caractéristiques. Les preuves à réaliser sont simplifiées et sont en partie obtenues automatiquement.

Références

[1] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for product-line verification : case studies and experiments. In *ICSE'15, San Francisco, CA, USA, 2013*, pages 482–491, 2013.

[2] D. Benavides, S. Segura, P. Trinidad, and A. R. Cortés. FAMA : tooling a framework for the automated analysis of feature models. In *VaMoS 2007, Limerick, Ireland, 2007*, pages 129–134, 2007.

[3] P. Clements and L. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, 2001.

[4] FoCaLiZe. http://focalize.inria.fr/.

[5] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, 1990.

[6] T.-K.-D. Pham. *Development of Correct-by-Construction Software using Product Lines*. Draft, thèse de doctorat, CNAM, soutenance prévue en 2017.

[7] T.-K.-D. Pham, C. Dubois, and N. Levy. Vers un développement formel non incrémental. In *AFADL'2016*, pages 106–113, Besançon, France, 2016.

[8] T. Thüm, I. Schaefer, M. Kuhlemann, and S. Apel. Proof composition for deductive verification of software product lines. In *VAST'11*, pages 270–277. IEEE Computer Society, 2011.

⋆ ⋆ ⋆ AFADL 2017 ⋆ ⋆ ⋆

Résumés longs

Domestiquer la variété des critères de test avec le langage HTOL et l'outil LTest *

Michaël Marcozzi, Sébastien Bardin,
Nikolai Kosmatov, Virgile Prevosto et Mickaël Delahaye
CEA, LIST, Laboratoire de Sûreté des Logiciels, PC 174, 91191, Gif-sur-Yvette, France

Contexte. Automatiser le test en boîte blanche est un sujet majeur en ingénierie du logiciel. Au cours des années, de nombreux outils ont ainsi été développés pour supporter les différentes parties du processus de test. Ces outils se basent implicitement ou explicitement sur un critère de couverture de code pour guider l'automatisation. Le critère spécifie formellement quels sont les objectifs de test. Ceux-ci peuvent alors être utilisés automatiquement pour mesurer la qualité d'une suite de tests et pour piloter la génération de tests pertinents. Dans les domaines régulés, comme l'aéronautique, tester le logiciel selon un critère de test précis peut être une exigence normative stricte. Dans les autres domaines, l'utilisation de critères de test performants est reconnue comme une bonne pratique de développement et comme un ingrédient clé pour le développement dirigé par le test.

Problème. Des dizaines de critères de couverture de code ont été proposés dans la littérature, de la couverture de branches au test par mutation, offrant notamment des ratios différents entre l'effort et la minutie du test. Cependant, ces critères ont toujours été vus comme des guides d'automatisation très différents, si bien que la plupart des outils de test sont limités à un petit sous-ensemble prédéfini de critères. En conséquence, la grande variété et la profonde sophistication des critères académiques de test est à peine exploitée dans l'industrie.

Objectif et défis. L'objectif des deux articles résumés ici est de combler le fossé entre le large corpus de travail académique sur les critères de couverture de code d'une part, et leur usage limité dans l'industrie d'autre part. En particulier, le premier article [1] vise à proposer un mécanisme de spécification unificateur pour ces critères, permettant une séparation claire entre déclaration précise des objectifs de test d'une part, et automatisation du test guidée par ces objectifs d'autre part. Cette approche est ambitieuse car le mécanisme proposé doit à la fois être bien défini, suffisamment expressif pour encoder les objectifs de test de la plupart des critères existants et suffisamment souple pour servir de base à l'automatisation. L'objectif du second article [2] est de monter comment ce mécanisme de spécification peut être techniquement placé au cœur d'un outil d'automatisation de test, qui ne soit plus limité à un sous-ensemble de critères, mais bien proprement générique et uni-

*Ce travail a été partiellement financé par l'ANR (grant ANR-12-INSE-0002).

versel. L'outil doit pouvoir automatiser tous les aspects du test (génération de tests, mesure de couverture, détection d'objectifs incouvrables) pour du code réel.

Proposition. Nous introduisons HTOL (Hyperlabel Test Objective Language), un langage de spécification générique pour les objectifs de test en boîte blanche. Les hyperlabels de HTOL sont une extension des labels, développés précédemment par notre équipe au sein de l'outil de test LTest. Bien que capables d'encoder une grande variété de critères, les labels sont trop limités en terme d'expressivité, et l'utilisation d'hyperlabels est nécessaire pour exprimer des critères importants, comme MCDC. De plus, les hyperlabels sont suffisants pour exprimer tous les critères de la littérature (sauf mutations fortes), mais également d'autres objectifs de test intéressants, comme vérifier d'importantes propriétés de sécurité logicielle. Enfin, en comparaison aux approches existantes, les hyperlabels proposent un point d'équilibre entre généricité, spécialisation aux critères de couverture et capacité d'automatisation. L'outil LTest est mis à jour afin de générer des tests, mesurer la couverture ou détecter l'incouvrabilité pour des objectifs de couverture spécifiés avec HTOL, et plus seulement à l'aide de labels. L'efficacité et la capacité de passage à l'échelle de l'outil sont vérifiées sur des programmes open-source standards.

Contributions. Les cinq principales contributions des articles présentés ici sont :
1. Une nouvelle taxonomie des critères de test, orthogonale aux classifications existantes, car sémantique et basée sur la nature des contraintes d'accessibilité sous-jacente aux critères classés. Une représentation visuelle de cette classification est proposée : le cube des critères de couverture.
2. Une syntaxe et une sémantique formelle pour le langage HTOL, qui introduisent des opérateurs permettant de combiner les labels, étendant ainsi leur expressivité des critères à l'origine du cube à l'entièreté des critères portés par celui-ci.
3. Une présentation à la fois pédagogique et exhaustive de comment l'entièreté des critères standards (sauf mutations fortes) peuvent être encodés à l'aide de HTOL.
4. Un rapport sur les avancées techniques réalisées dans LTest pour offrir le support de HTOL et une présentation des nouvelles APIs de l'outil (open source).
5. Des expériences montrant l'efficacité et le passage à l'échelle de LTest, avec des suites de 10000 cas de test sur des programmes comme OpenSSL et SQLite.

Impact potentiel. Les hyperlabels sont une *lingua franca* pour définir, étendre et comparer des critères de test d'une façon clairement documentée. Ils sont aussi un langage de spécification pour écrire des outils de test universels, extensibles et interopérables, comme l'outil LTest, qui rend la variété et la sophistication des critères de test facilement utilisables dans un cadre industriel.

Ce résumé court synthétise les deux articles suivants:

[1] Generic and Effective Specification of Structural Test Objectives

[2] Taming Coverage Criteria Heterogeneity with LTest

acceptés à *10th IEEE Int. Conf. on Software Testing, Verification and Validation*

Une extension probabiliste pour Event-B *

Mohamed Amine Aouadhi Benoît Delahaye Arnaud Lanoix

Université de Nantes
LS2N UMR CNRS 6004

Dans l'article *"Moving from Event-B to Probabilistic Event-B"* accepté à la conférence internationale *SAC-SVT 2017* [1] nous présentons des travaux autour d'une extension probabiliste à *Event-B*.

Afin de vérifier les systèmes de plus en plus complexes, il est nécessaire d'ajouter aux techniques de modélisation et aux outils de vérification actuels de nouvelles facettes permettant de prendre en compte tous les aspects de ces systèmes. L'une de ces facettes est le *raisonnement probabiliste* qui permet par exemple de modéliser des incertitudes ou de simuler des comportements aléatoires.

Event-B [2] est un langage et une méthode de vérification basée sur des techniques de preuves et principalement dédié à la modélisation de systèmes à événements discrets. [3] suggère que les probabilités devraient être introduites dans Event-B comme un raffinement du *non-déterminisme*. Dans Event-B, le non-déterminisme apparaît à plusieurs endroits : *i)* lors du *choix de l'événement à activer* parmi l'ensemble des événements activables, *ii)* lors du *choix des valeurs* des paramètres des événements et *iii)* lors de la résolution des *affections non-déterministes* de la forme $x :| Q(x,x')$ (forme prédicative) ou $x :\in \{E_1 \ldots E_n\}$ (forme énumérée). À notre connaissance, les travaux existants jusqu'ici [4, 5, 6, 7] ne se sont intéressés qu'au remplacement des affections non-déterministes par des affections probabilistes (quantifiées ou non).

Dans [1], nous proposons une extension probabiliste à Event-B dans laquelle toutes les sources de non-déterminisme sont remplacées par des choix probabilistes : *i)* un *poids* est ajouté à chaque événement afin de pouvoir établir la probabilité d'activation de cet événement, *ii)* le choix des valeurs des paramètres est réalisé à partir d'une distribution *uniforme* et *iii)* nous proposons deux nouvelles *affectations probabilistes quantifiées* afin de remplacer les affections non-déterministes : $x:\oplus Q_x(x,x')$, où le choix des valeurs possibles pour x est résolu à l'aide d'une distribution *uniforme*, et $x := E_1 @ p_1 \oplus \ldots \oplus E_n @ p_n$, où chaque valeur E_i est choisie avec une *probabilité* p_i. Nous donnons de nouvelles Obligations de Preuves (OPs) nécessaires pour démontrer la consistence d'un *modèle Event-B probabiliste*. Afin de démontrer la correction de notre proposition, nous montrons que la sémantique

*This work is partially supported by the ANR national research program PACS (ANR-14-CE28-0002).

opérationnelle d'un modèle Event-B probabiliste s'exprime par une chaîne de Markov à temps discrets comportant un nombre potentiellement infini d'états [8]. Finalement, nous étendons [4] afin d'établir les OPs nécessaires afin de montrer la *convergence presque-sûre* d'un ensemble d'événements dans un modèle Event-B probabiliste.

Comme illustré en Fig. 1, nous avons commencé le développement d'un plugin pour *Rodin* [9] permettant la modélisation et la vérification de modèles Event-B probabilistes.

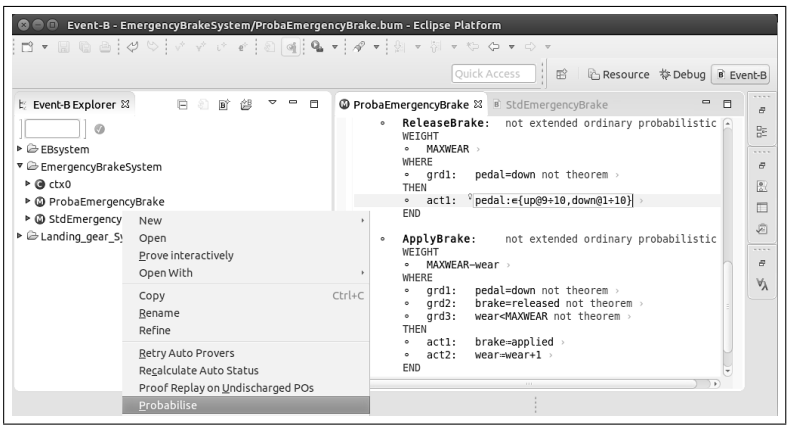


FIGURE 1 – Plugin probabiliste pour *Rodin*

Références

[1] Mohamed Amine Aouadhi, Benoît Delahaye, and Arnaud Lanoix. Moving from event-b to probabilistic event-b. In *Proceedings of the 32th Annual ACM Symposium on Applied Computing (to appear)*. ACM, 2017.

[2] Jean-Raymond Abrial. *Modeling in Event-B : system and software engineering*. Cambridge University Press, 2010.

[3] Carroll Morgan, Thai Son Hoang, and Jean-Raymond Abrial. The challenge of probabilistic event b—extended abstract—. In *ZB 2005 : Formal Specification and Development in Z and B*, pages 162–171. Springer, 2005.

[4] Stefan Hallerstede and Thai Son Hoang. Qualitative probabilistic modelling in event-b. In *Integrated Formal Methods*, pages 293–312. Springer, 2007.

[5] Emre Yilmaz. *Tool support for qualitative reasoning in Event-B*. PhD thesis, Master Thesis ETH Zürich, 2010, 2010.

[6] Anton Tarasyuk, Elena Troubitsyna, and Linas Laibinis. Integrating stochastic reasoning into event-b development. *Formal Aspects of Computing*, 27(1) :53–77, 2015.

[7] Anton Tarasyuk, Elena Troubitsyna, and Linas Laibinis. Towards probabilistic modelling in event-b. In *Integrated Formal Methods*, pages 275–289. Springer, 2010.

[8] Christel Baier, Joost-Pieter Katoen, et al. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008.

[9] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin : an open toolset for modelling and reasoning in event-b. *International journal on software tools for technology transfer*, 12(6) :447–466, 2010.

Approximer des abstractions de systèmes d'événements en couvrant leurs états et leurs transitions*

J. Julliand O. Kouchnarenko P.-A. Masson G. Voiron

FEMTO-ST, UMR 6174 CNRS and Univ. Bourgogne Franche-Comté
16, route de Gray F-25030 Besançon Cedex France
{jjulliand, okouchna, pamasson, gvoiron}@femto-st.fr

1 Introduction

Le flot de contrôle des systèmes événementiels étant implicite, leur abstraction peut engendrer des traces déconnectées et états inatteignables depuis l'état initial. Cet article présente un résumé de [2] où une méthode algorithmique permet le calcul d'une sous-approximation de l'abstraction d'un système événementiel. Cette sous-approximation est calculée avec des instances concrètes des transitions abstraites d'une abstraction par prédicats, afin d'en couvrir tous les états et toutes les transitions. La méthode intègre deux heuristiques favorisant la connectivité et l'atteignabilité de ces transitions.

2 Méthode de calcul d'une abstraction et heuristiques pour une meilleure couverture

La méthode est définie par un algorithme de calcul d'une abstraction à partir de prédicats. L'abstraction résultante est constituée de 2^n états abstraits à partir de n prédicats. L'algorithme consiste à évaluer l'existence de transitions entre chaque couple d'états abstraits pour chaque événement en évaluant une condition d'existence par un SMT solver. Le solver produit des transitions concrètes témoins de la satisfiabilité des conditions. Ces transitions concrètes constituent une sous-approximation de la sémantique du système d'événements qui couvre tous les états et toutes les transitions abstraites. Mais l'enjeu est d'obtenir cette couverture sous la forme d'un système concret atteignable et connecté. C'est l'objectif poursuivi par les deux heuristiques suivantes : faire choisir un ordre de traitement des états abstraits et des événements par l'utilisateur et utiliser une coloration des états concrets ; en vert pour indiquer les états atteints depuis l'état initial, et en bleu pour ceux dont on ne sait pas s'ils pourront être atteints.

*Cet article est la présentation de la communication [2]

La connectivité entre les transitions concrètes qui témoignent de la satisfiabilité de la condition d'existence d'une transition dépend de l'ordre de prise en compte des états et des événements. En effet, souvent, certains événements doivent en précéder d'autres pour qu'ils soient déclenchables. Ceci peut aussi dépendre de leurs états cibles. Cet ordre d'application est connu de l'utilisateur du système qui peut donc ainsi guider le calcul. Par ailleurs, la couleur permet de privilégier la connectivité des transitions en optimisant la concrétisation des transitions à partir des états verts et en ciblant des états bleus. Ainsi, des états non atteints le deviennent.

3 Résultats expérimentaux

Nous comparons les résultats expérimentaux obtenus par deux algorithmes sur quatre systèmes d'événements. Le premier est l'implantation de la méthode décrite dans la section précédente sans heuristique. Le second est le même algorithme optimisé avec les deux heuristiques. Nous observons que les heuristiques améliorent les taux de couverture des états abstraits et des transitions abstraites de tous les systèmes pour tous les ensembles de prédicats d'abstraction. Les heuristiques améliorent aussi le taux d'efficacité de la méthode qui est défini comme le nombre de transitions concrètes construites pour couvrir une transition abstraite.

4 Conclusion

Ce papier présente une méthode algorithmique permettant de construire des sous-approximations finies de systèmes de grande taille, voire de taille infinie, définis par des systèmes d'événements. L'enjeu est d'obtenir un système concret connecté et atteignable à partir des états initiaux. Les heuristiques proposées se révèlent efficaces sur les exemples. Ces approximations peuvent permettre d'engendrer des tests à partir d'objectifs de test définis par un ensemble de prédicats d'abstraction. Nous envisageons d'introduire un nouveau paramètre dans le système sous la forme d'une fonction de pertinence telle qu'elle est proposée dans [1] pour cibler certains états déterminés à partir d'un objectif d'abstraction.

Références

[1] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *ISSTA*, pages 112–122, 2002.

[2] J. Julliand, O. Kouchnarenko, P.-A. Masson, and G. Voiron. Approximating event system abstractions by covering their states and transitions. In *A.P. Ershov Informatics Conference, PSI'17*, 2017. LNCS to appear.

Spécification légère et analyse de systèmes dynamiques munis de configurations riches*

Nuno Macedo
HASLab, INESC TEC & U. do Minho

Julien Brunel
ONERA/DTIS

David Chemouil
ONERA/DTIS

Alcino Cunha
HASLab, INESC TEC & U. do Minho

Denis Kuperberg
ENS Lyon

Résumé

Cet article constitue un bref résumé d'un article accepté et présenté à la conférence *Foundations of Software Engineering 2016,* intitulé *Lightweight Specification and Analysis of Dynamic Systems with Rich Configurations* [3].

Le *model-checking* est de plus en plus populaire dans les premières phases du processus de développement logiciel. Pour valider une conception logicielle, il faut généralement vérifier des propriétés structurelles et comportementales (ou temporelles). Malheureusement, la plupart des langages de spécification et des *model-checkers* excellent uniquement dans l'analyse de l'une ou l'autre sorte. Cela limite leur capacité à vérifier les systèmes dynamiques munis de configurations riches : des systèmes dont l'espace d'états est caractérisé par des propriétés structurelles, mais dont l'évolution devrait aussi vérifier des propriétés temporelles. Pour résoudre ce problème, nous proposons d'abord Electrum, une extension du langage de spécification Alloy avec des opérateurs de logique temporelle, où des configurations riches et des propriétés temporelles peuvent aisément être définies à la fois. Deux alternatives de techniques de *model-checking* sont ensuite proposées, l'une bornée et la seconde non-bornée, pour vérifier des systèmes exprimés dans ce langage, c'est-à-dire vérifier que les propriétés temporelles sont satisfaites dans toutes les configurations possibles.

1 Problématique

La spécification et la vérification des logiciels sont essentielles dans les premières phases de développement, car elles permettent au développeur de raisonner sur le système et ses propriétés et de détecter en temps opportun les erreurs de conception. Les cadres dits « légers », en ce sens qu'ils fournissent un langage formel simple mais expressif et flexible, permettent à l'utilisateur de spécifier différentes classes de systèmes et de propriétés à différents niveaux d'abstraction, et ils sont accompagnés d'outils qui automatisent leur analyse, offrant un retour rapide sur la correction de la spécification.

Deux classes de propriétés sont particulièrement importantes à considérer : des propriétés *structurales* (ou statiques), typiquement exprimées dans une variante de la logique du premier ordre, qui traitent de la bonne formation de l'état du système ; et des propriétés *comportementales* (ou dynamiques), typiquement exprimées dans une logique temporelle, qui traitent de l'évolution de l'état du système. Bien que pas nécessairement dans la même mesure, la plupart des systèmes réalistes nécessitent la spécification et l'analyse des propriétés des deux classes. L'analyse des algorithmes distribués est une classe paradigmatique dont les propriétés comportementales devraient être vérifiées pour des topologies de réseau arbitraires, dans une plage spécifiée par des propriétés structurelles particulières. Nous désignons ces composantes de l'état du système, qui sont initialement arbitraires, mais qui restent inchangées à mesure que le système évolue, sous le nom de *configurations*. Une autre classe pertinente est celle des

*Financements : *European Regional Development Fund* (ERDF) *via Operational Programme for Competitiveness and Internationalisation* (COMPETE 2020) ; Fundação para a Ciência e a Tecnologia (FCT) projet POCI-01-0145-FEDER-016826 ; projet DGA/ANR C*x* (ref. ANR-13-ASTR-0006) ; *fondation STAE* (IFSE, BRIefcaSE).

```
open util/ordering[Id]
sig Id {}
sig Process {
        id: one Id,
        succ: one Process,
        var toSend: set Id }
var sig elected in Process {}
pred step [p: Process] {
  some pid: p.toSend {
    p.toSend' = p.toSend - pid
    p.succ.toSend' = p.succ.toSend + (pid - prevs[p.succ.id]) }}
fact { always { all p: Process | step[p] or step [succ.p] or skip[p] }}
...
check { (some Process and Progress) => sometime some elected } for 4
```

FIGURE 1 – Extrait de code source Electrum pour l'exemple *leader election* [1]

lignes de produits logiciels. Ainsi, les systèmes dynamiques avec configurations riches sont au centre de ce travail, et concrètement, cette classe de systèmes présente les exigences suivantes :

R1 Une distinction claire entre la spécification de la configuration du système et l'évolution du système ;

R2 Des configurations contraintes par des propriétés structurelles riches (comme l'héritage, des relations complexes entre des entités ou des propriétés d'accessibilité) ;

R3 Une spécification déclarative de l'évolution du système (les actions possibles affectant l'état), éventuellement au moyen d'idiomes variés ;

R4 La nécessité de vérifier les propriétés (temporelles) de sûreté et de vivacité du système spécifié.

Malheureusement, la plupart des langages de spécification formels (et les *model-checkers* associés) ne sont pas conçus ni optimisés pour analyser des problèmes de ce type. Par exemple, la plupart des *model-checkers* standard ne fonctionnent bien qu'avec des configurations fixes, tandis que les langages plus orientés vers l'analyse des propriétés structurales, habituellement sans support natif pour la logique temporelle, exigent que l'utilisateur vérifie les propriétés comportementales par des mécanismes *ad hoc*.

2 Résultats

Notre travail vise précisément à combler ce créneau, et propose un langage et un *model-checker* adaptés à l'analyse légère de systèmes dynamiques avec de riches configurations. Concrètement, nous proposons : *(a)* Electrum, un langage de spécification formel (cf. figure 1), inspiré par Alloy [1] et TLA+ [2] (deux langages de spécification très utilisés de nos jours), qui simplifie la spécification de systèmes présentant toutes les exigences définies ci-dessus ; *(b)* Deux techniques de *model-checking*, l'une bornée et l'autre non bornée, pour vérifier les systèmes exprimés dans un tel langage, à savoir vérifier que chaque propriété temporelle souhaitable est valable pour toute configuration possible.

Comparativement à Alloy et TLC (le *model-checker* associé à TLA+), les performances de nos outils se situent dans le même ordre. Sur le plan de l'expressivité pour l'utilisateur, nous pensons qu'Electrum se situe sur un créneau intéressant pour la spécification de systèmes arborant des propriétés comportementales et des configurations riches.

Références

[1] D. Jackson. *Software Abstractions : Logic, Language, and Analysis*. The MIT Press, 2006.

[2] L. Lamport. *Specifying Systems, The* TLA$^+$ *Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[3] N. Macedo, J. Brunel, D. Chemouil, A. Cunha, and D. Kuperberg. Lightweight specification and analysis of dynamic systems with rich configurations. In *Proc. of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 373–383, New York, NY, USA, 2016. ACM.

Temporary Read-Only Permissions for Separation Logic (Extended Abstract)

Arthur Charguéraud[1,2] and François Pottier[1]

[1] Inria, Paris, France*
[2] ICube – CNRS, Université de Strasbourg, France

Abstract. We present an extension of Separation Logic with a general mechanism for temporarily converting any assertion (or "permission") to a read-only form. No accounting is required: our read-only permissions can be freely duplicated and discarded. We argue that, in circumstances where mutable data structures are temporarily accessed only for reading, our read-only permissions enable more concise specifications and proofs. The metatheory of our proposal is verified in Coq.

Separation Logic [2] offers a natural and effective framework for proving the correctness of imperative programs that manipulate the heap. However, practice shows some limitations of the specification language. Consider the following example, which describes the specification of the concatenation of two arrays.

$$\{a_1 \rightsquigarrow \mathsf{Array}\, L_1 \mathbin{\star} a_2 \rightsquigarrow \mathsf{Array}\, L_2\}$$
$$(\texttt{concat}\, a_1\, a_2)$$
$$\{\lambda a_3.\ a_3 \rightsquigarrow \mathsf{Array}\, (L_1 +\!\!+ L_2) \mathbin{\star} a_1 \rightsquigarrow \mathsf{Array}\, L_1 \mathbin{\star} a_2 \rightsquigarrow \mathsf{Array}\, L_2\}$$

Above, $a \rightsquigarrow \mathsf{Array}\, L$ asserts the existence (and unique ownership) of an array at address a whose content is given by the list L. A separating conjunction $\star$ is used in the precondition to require that a_1 and a_2 be disjoint arrays. Its use in the postcondition guarantees that a_3 is disjoint with a_1 and a_2. In this specification, the fact that the arrays a_1 and a_2 are unaffected must be explicitly stated as part of the postcondition, making the specification seem verbose.

We wish to extend the specification language of Separation Logic so as to be able to specify the same function more concisely, as follows.

$$\{\mathsf{RO}(a_1 \rightsquigarrow \mathsf{Array}\, L_1) \mathbin{\star} \mathsf{RO}(a_2 \rightsquigarrow \mathsf{Array}\, L_2)\}$$
$$(\texttt{concat}\, a_1\, a_2)$$
$$\{\lambda a_3.\ a_3 \rightsquigarrow \mathsf{Array}\, (L_1 +\!\!+ L_2)\}$$

At first sight, it may seem that the notation "RO", which stands for "read-only", could be interpreted as syntactic sugar for repeating part of the precondition in the postcondition. However, we wish to assign RO a much stronger

* This research was partly supported by the French National Research Agency (ANR) under the grant ANR-15-CE25-0008.

interpretation. First, we wish an assertion of the form $\mathsf{RO}(H)$ to prevent writing the heap fragment described by H, whereas the syntactic sugar interpretation only guarantees that this heap fragment satisfies H upon entry and upon exit. Second, we wish to allow aliasing of read-only arguments, whereas traditional Separation Logic requires a separate specification to handle the case where the two arguments are aliased. Third, we wish to save the user the need the trouble of proving that H is still satisfied upon exit: since a read-only heap fragment cannot be mutated, its properties cannot be falsified.

Fractional permissions [1] offer a partial solution. Let $a \overset{\alpha}{\rightsquigarrow} \mathsf{Array}\, L$ denote a fraction α of the ownership of the array. When α is 1, this assertion is equivalent to $a \rightsquigarrow \mathsf{Array}\, L$. When α is less than 1, this assertion grants read-only access. Fractional permissions can be split or merged using the following rule:

$$a \overset{\alpha+\beta}{\rightsquigarrow} \mathsf{Array}\, L \;=\; a \overset{\alpha}{\rightsquigarrow} \mathsf{Array}\, L \;\star\; a \overset{\beta}{\rightsquigarrow} \mathsf{Array}\, L \qquad \text{with } 0 < \alpha, \beta \leqslant 1.$$

With fractional permissions, the concatenation function may be specified as:

$$\begin{array}{l} \forall \alpha\beta.\ \{a_1 \overset{\alpha}{\rightsquigarrow} \mathsf{Array}\, L_1 \;\star\; a_2 \overset{\beta}{\rightsquigarrow} \mathsf{Array}\, L_2\} \\ \quad (\texttt{concat}\, a_1\, a_2) \\ \quad \{\lambda a_3.\ a_1 \overset{\alpha}{\rightsquigarrow} \mathsf{Array}\, L_1 \;\star\; a_2 \overset{\beta}{\rightsquigarrow} \mathsf{Array}\, L_2 \star a_3 \overset{1}{\rightsquigarrow} \mathsf{Array}\, (L_1 +\!\!+ L_2)\} \end{array}$$

This specification allows a_1 and a_2 to be aliases. Nevertheless, fractional permissions suffer from several limitations. First, they require explicit quantification over the fractions α and β, and may involve arithmetic operations when splitting and merging permissions. Second, they require careful accounting: if a single nonzero fraction is lost, a full read/write permission can never be recovered. Third, this approach lacks generality because scaling $\frac{1}{2}H$ cannot be defined for an arbitrary assertion H.

We propose an extension of the logic with a modality "RO" that applies to any assertion H. The assertion $\mathsf{RO}(H)$ is duplicable, allowing read-only arguments to be aliased. Moreover, an assertion $\mathsf{RO}(H)$ appears only in preconditions, never in postconditions. Thus, compared with traditional Separation Logic, our specifications are more concise, and the number of proof obligations for establishing postconditions is reduced.

Read-only permissions are introduced by a generalization of the frame rule:

$$\frac{\{H \star \mathsf{RO}(H')\}\ t\ \{Q\} \qquad \mathsf{no\text{-}ro\text{-}in}\, H'}{\{H \star H'\}\ t\ \{Q \star H'\}}\ \textsc{frame-ro}$$

where the side condition $\mathsf{no\text{-}ro\text{-}in}\, H'$ means that H' does not contain any read-only assertions. The intuition is as follows. With the traditional frame rule, the assertion H' that is "framed out" disappears within a certain scope and reappears when that scope is exited. With the read-only frame rule (above), instead of disappearing altogether, the assertion H' is turned into $\mathsf{RO}(H')$ within the scope, and reappears when that scope is exited. The soundness of this extension of Separation Logic has been entirely formalized in Coq.

References

1. Boyland, J.: Checking interference with fractional permissions. In: Static Analysis Symposium (SAS). Lecture Notes in Computer Science, vol. 2694, pp. 55–72. Springer (2003)
2. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science (LICS). pp. 55–74 (2002)

Un outil d'assistance à la construction de tests de modèles à composants et services

André, Pascal Ardourel, Gilles Mottu, Jean-Marie
Sunyé, Gerson

LS2N UMR CNRS 6004
Université de Nantes, IMT-Atlantique, Inria
Prenom.Nom@univ-nantes.fr

Dans l'article *"COSTOTest : A Tool for Building and Running Test Harness for Service-Based Component Models"* publié dans les proceedings de la conférence internationale *ISSTA 2016* [1] et présenté en session démonstration, nous décrivons comment l'outil COSTOTest nous assiste pour tester directement les modèles de composants logiciels basés sur les services. Ces travaux concernent la vérification de systèmes logiciels à base de composants et services (*Service-based Component* ou SBC) et exploitent l'ingénierie dirigée par les modèles (IDM).

Contexte Tester le plus tôt possible permet de réduire le coût du processus de vérification et de validation [2]. En IDM, c'est encore plus vrai et la correction des modèles est essentielle car ils sont le point de départ de transformations plus ou moins directes vers les modèles opérationnels et le code. Alors que les modèles servent de spécification dans les approches *Model-Based Testing* (MBT) pour générer des tests exécutés sur le code final, nous considérons la vérification de ces modèles qui peuvent être erronés [3]. Nous contribuons donc à la vérification des modèles en les testant directement (*Model Testing* MT). Ces tests sont complémentaires d'autres vérifications appliquées sur les modèles (preuves, model checking [4]). L'effort est porté sur la détection en amont des erreurs "métiers" (*platform independent*), coûteuses à corriger lorsqu'elles sont repérées tardivement et propagées dans différentes versions et implantations du système. De plus, le test de modèles permet de s'affranchir des erreurs spécifiques à l'implantation (*platform specific*), et réduit la complexité globale du test [5]. En exploitant l'IDM, les tests sur les modèles sont évolutifs car ils sont eux-mêmes des modèles. On bénéficie ainsi des mêmes outils de transformation de modèles que le système sous test.

Objectifs Nous ciblons les tests fonctionnels (unitaire, intégration, recette fonctionnelle hors IHM) pour des modèles à composants et services, ayant un niveau de description suffisamment précis et détaillé pour pouvoir exécuter les tests. Assurer la correction de ces modèles reste un défi. La vérification formelle permet de filtrer

une partie des modèles erronés mais elle ne suffit pas parce que les outils restent limités face à la combinatoire de langages expressifs. Nous considérons le test pour améliorer le niveau de correction. Notre objectif est de tester ces modèles à composants c'est-à-dire de concevoir des cas de tests, de les appliquer sur les modèles mis dans un contexte adéquat pour être exécutés et obtenir un verdict.

Processus Le processus part d'une intention de test (variables et oracles) et d'un modèle sous test. Nous considérons la construction de *harnais de test* unitaire ou d'intégration pour des systèmes à composants et services, qui permettent de passer des données de test, d'exécuter les tests, et de récupérer le verdict (succès ou échec du test). Pour réduire l'effort de construction du harnais de test, nous proposons une méthode qui guide le testeur dans le processus de conception des tests. Sachant que le testeur découvre l'application à tester et que les intentions de test restent informelles, il a besoin d'itérer jusqu'à ce que le niveau de précision soit atteint pour que le système soit testable. Il a besoin d'être assisté pour définir la forme véritable du test et déterminer la partie du système nécessaire au test. Définir concrètement l'oracle et comment trouver ou fournir les données de test est difficile : par exemple, pour atteindre une variable, il faut trouver les services qui la manipulent ou qui en dépendent. L'assistance à la construction est basée sur la détection d'incohérences et d'incomplétude entre le harnais et le modèle de test ainsi que sur des propositions générant les éléments manquants. Le programme de test est alors transformé vers une plateforme technique dédiée à l'exécution des tests.

Mise en œuvre La mise en œuvre est réalisée avec un outillage qui permet d'expérimenter l'approche. Il est mis en œuvre dans COSTO (*COmponent Study Toolbox*), la plate-forme Eclipse (`http://costo.univ-nantes.fr/`) dédiée aux modèles à composants écrits avec le DSL Kmelia. Nous avons développé un *framework* en Java qui donne un cadre d'exécution et d'animation pour les modèles de test (plateforme spécifique). L'approche est illustrée sur un cas d'étude simple, un système de *platoon* de véhicules.

Références

[1] Pascal André, Jean-Marie Mottu, and Gerson Sunyé. Costotest : a tool for building and running test harness for service-based component models (demo). In Andreas Zeller and Abhik Roychoudhury, editors, *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 437–440. ACM, 2016.

[2] Graeme Shanks, Elizabeth Tansley, and Ron Weber. Using ontology to validate conceptual models. *Commun. ACM*, 46(10) :85–89, October 2003.

[3] Martin Gogolla, Jørn Bohling, and Mark Richters. Validating uml and ocl models in use by automatic snapshot generation. *Software and Systems Modeling*, 4(4) :386–398, 2005.

[4] Pascal André, , Christian Attiogbé, and Jean-Marie Mottu. Combining techniques to verify service-based components. In *Proceedings of the International Workshop on domAin specific Model-based AppRoaches to vErificaTion and validaTiOn, AMARETTO@MODELSWARD 2017, Porto, Portugal, February 19-21, 2017.*, pages 645–656, 2017.

[5] Marc Born, Ina Schieferdecker, Hans-gerhard Gross, and Pedro Santos. Model-driven development and testing - a case study. In *First European Workshop on MDA with Emphasis on Industrial Application*, pages 97–104. Twente Univ., 2004.

Preuve formelle du théorème de Lax–Milgram

Sylvie Boldo [1] François Clément [2] Florian Faissole[1]
Vincent Martin [3] Micaela Mayero [4]

1 Introduction

La méthode des éléments finis est fréquemment utilisée pour résoudre numériquement des équations aux dérivées partielles qui interviennent par exemple en physique ou en biologie [4]. Pour augmenter la confiance dans les programmes qui l'implémentent, on doit commencer par la formalisation des notions mathématiques nécessaires à sa correction, à l'aide par exemple de preuves formelles. Le théorème de Lax–Milgram est l'un de ces résultats fondamentaux. Il permet, sous certaines hypothèses de coercivité et de complétude, de montrer l'existence et l'unicité de la solution de certains problèmes aux limites et de leur version discrète. Cet article présente une formalisation complète du théorème de Lax–Milgram dans l'assistant de preuves Coq, et utilise plus particulièrement la bibliothèque Coquelicot pour l'analyse réelle [2]. Des résultats mathématiques variés sont requis, provenant de l'algèbre linéaire, la géométrie et l'analyse fonctionnelle des espaces de Hilbert. Ce travail repose sur une preuve papier détaillée [3], est plus précisément décrit dans [1] et est disponible : `https://www.lri.fr/~sboldo/elfic/`.

2 Espaces de Hilbert

On étend, à l'aide du mécanisme de structures canoniques [5], la hiérarchie de la bibliothèque Coquelicot pour définir les espaces préhilbertiens et hilbertiens. Un espace préhilbertien est un module équipé d'un produit scalaire (avec les propriétés usuelles). On prouve qu'un espace préhilbertien est un module normé, dont la norme se dérive du produit scalaire. Un espace de Hilbert est un espace préhilbertien complet. Dans ces espaces, on définit des notions géométriques comme le projeté orthogonal d'un vecteur sur un sous-espace ou le complémentaire orthogonal d'un sous-espace et on prouve formellement les résultats associés (lemmes d'existence ou de caractérisation).

On formalise les espaces d'applications linéaires, puis on vérifie l'équivalence entre plusieurs définitions de la continuité de telles fonctions. L'une d'elle est la finitude de la norme d'opérateur $|||f|||$ d'une fonction f de E dans F, modules normés. On définit ainsi $\texttt{clm}(E,F)$ l'espace des applications linéaires continues et on prouve que la norme d'opérateur lui confère la structure de module normé (le cas $F=\mathbb{R}$ est utilisé dans la preuve du théorème de Lax–Milgram et correspond au dual topologique E' de E) :

```
Record clm := Clm {                    (* type dépendant *)
  m :> E -> F ;      (* la fonction, avec une coercion pour usage direct *)
  Lf : is_linear_mapping m;            (* preuve de linéarité *)
  Cf : is_finite (operator_norm m) }.  (* preuve de continuité *)
```

1. Inria, Université Paris-Saclay, F-91120 Palaiseau - LRI, CNRS & Univ. Paris-Sud, F-91405 Orsay
2. Inria, 2 rue Simone Iff, CS 42112, FR-75589 Paris cedex 12, France
3. LMAC, UTC, BP 20529, FR-60205 Compiègne, France
4. LIPN, Université Paris 13, CNRS UMR 703, Villetaneuse, F-93430, France

3 Preuves des théorèmes de Riesz–Fréchet et Lax–Milgram

Le théorème de Riesz–Fréchet est un résultat intermédiaire pour la preuve du théorème de Lax–Milgram. Il s'agit d'un résultat de représentation, qui identifie, à l'aide du produit scalaire, une fonction f de E' à un unique vecteur de l'espace hilbertien E. Ce théorème se démontre avec les lemmes de caractérisations du projeté orthogonal, le complémentaire orthogonal de $ker(f)$ et la complétude de E. Il est nécessaire de distinguer le cas où f est identiquement nulle, ce qui n'est pas décidable. Ainsi, on ajoute des hypothèses de décidabilité.

Le théorème de Lax–Milgram s'énonce comme suit :

Théorème (Lax–Milgram). *Soit* $E : Hilbert$, $f \in E'$, $0 < \alpha$, $\varphi : E \to \texttt{Prop}$ *un sous-module complet. Soit* a *une forme bilinéaire sur* E*, bornée et* α*-coercive.*
On suppose $\forall f \in E',\ decidable(\exists u \in E, u \in \ker(f)\ \wedge\ \neg\neg\varphi(u)) \wedge$
$(\forall u \in E, \forall \varepsilon \in \mathbb{R}_+, decidable(\exists w \in E, \varphi(w) \wedge \|u - w\| < \varepsilon))$.
Alors : $\exists! u \in E, \neg\neg\varphi(u) \wedge \forall v \in E, \neg\neg\varphi(v) \Longrightarrow f(v) = a(u,v) \wedge \|u\|_E \leq \frac{1}{\alpha} \cdot |||f|||_\varphi$.

Une simplification calculatoire et deux applications du théorème de Riesz–Fréchet permettent de se ramener à chercher l'unique point fixe d'une fonction contractante bien choisie. On applique alors un théorème de point fixe de Banach. On utilise les axiomes de $ProofIrrelevance$ et de $FunctionalExtensionality$ dans les preuves d'analyse fonctionnelle. Ceci étant, on ne fait pas appel à la logique classique et afin d'identifier certains points critiques, on préfère ajouter des hypothèses de décidabilité lorsque nécessaire ainsi que quelques doubles négations.

4 Conclusion

On obtient une preuve formelle complète du théorème de Lax–Milgram en Coq. Le développement Coq comporte environ 7 000 lignes de code, tout comme la preuve papier LaTeX d'environ 50 pages. L'usage de Coq et de Coquelicot pose des écueils non présents dans les preuves papier, comme la représentation des sous-espaces ou l'usage de filtres dans les raisonnements topologiques. Cette formalisation est la première étape d'un travail de certification autour de la méthode des éléments finis, qui vise à prouver des programmes l'implémentant.

Références

[1] S. Boldo, F. Clément, F. Faissole, V. Martin, and M. Mayero. A Coq Formal Proof of the Lax–Milgram theorem. In *6th Conference on Certified Programs and Proofs*, Paris, 2017.

[2] S. Boldo, C. Lelay, and G. Melquiond. Coquelicot : A user-friendly library of real analysis for Coq. *Mathematics in Computer Science*, 9(1) :41–62, 2015.

[3] F. Clément and V. Martin. The Lax-Milgram Theorem. A detailed proof to be formalized in Coq. Research Report RR-8934, Inria Paris, July 2016.

[4] A. Ern and J-L. Guermond. *Theory and practice of finite elements*, volume 159 of *Applied Mathematical Sciences*. Springer-Verlag, New York, 2004.

[5] A. Mahboubi and E. Tassi. Canonical structures for the working coq user. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, pages 19–34, 2013.

ELIOM : Un langage ML pour la programmation Web sans tiers

Gabriel RADANNE [a], Jérôme VOUILLON [a,b,c], et Vincent BALAT [a,b]

[a]IRIF UMR 8243 CNRS, Univ Paris Diderot, Sorbonne Paris Cité
[b]BeSport
[c]CNRS

Résumé

ELIOM est un dialecte d'OCAML pour la programmation Web qui permet, à l'aide d'annotations syntaxiques, de déclarer code client et code serveur dans un même fichier. Ceci permet de construire une application complète comme un unique programme distribué dans lequel il est possible de définir des widgets faciles à composer avec des comportements à la fois client et serveur. Notre langage expose également un modèle de communication simple et sûre via un typage statique fort. ELIOM correspond aux spécificités de la programmation Web en permettant de mélanger code client et serveur tout en maintenant une communication unidirectionnelle et efficace. Le langage ELIOM est suffisamment minimaliste pour être implémenté comme extension d'un langage existant, et suffisamment expressif pour implémenter la plupart des idiomatismes du Web.

Les applications Web traditionnelles sont composées de différents tiers : les pages Web sont écrites en HTML et CSS, ces pages peuvent être produites par un langage quelconque : PHP, Ruby, C++, ... Le comportement dynamique est contrôlé par un langage client tel que JAVASCRIPT. La manière usuelle de composer ces différents tiers est d'écrire un programme client et un programme serveur distincts. Le programmeur doit alors respecter une interface commune entre les deux programmes. Cette contrainte n'est généralement pas vérifiée automatiquement et doit être respectée par le programmeur. Ceci est évidement sujet à erreurs et est la cause de nombreux bugs dans les applications Web.

L'un des buts des frameworks modernes de programmation Web client-serveur est d'offrir la possibilité de créer des pages Web dynamiques d'une manière *composable*. Le programmeur devrait pouvoir définir *sur le serveur* une fonction qui créer un fragment de page, ainsi que le comportement associé *du coté client*. Les langages de programmation sans tiers ont pour but de résoudre ces problèmes de modularité en permettant au programmeur de mêler librement code client et serveur. Pour la plupart de ces langages, deux parts sont extraites d'un programme : une part est exécutée sur le serveur tandis que l'autre est compilée vers JAVASCRIPT et exécutée sur le client. Ce paradigme de programmation permet de mélan-

ger code client et serveur librement tout en fournissant de forte garanties sur les communications client-serveur ainsi qu'une notion très fine de composition.

ELIOM est un dialecte d'OCAML pour la programmation Web sans tiers qui supporte des interactions client-server statiquement typées et composables. ELIOM fait parti du projet OCSIGEN qui inclus également le compilateur JS_OF_OCAML, un serveur Web et diverses librairies pour la programmation Web. Les librairies OCSIGEN sont implémentées en utilisant un langage minimal qui permet d'exprimer toutes les fonctionnalités nécessaires pour la programmation Web sans tiers. Ce langage est bâti sur un certain nombre de concepts.

Composition ELIOM permet de construire des composants indépendants et réutilisables qui peuvent être assemblés facilement. Il permet de définir et de manipuler *sur le serveur*, comme valeurs de première classe, des fragments de code qui seront exécutés *sur le client*. Ceci permet de construire des widgets réutilisables qui capturent à la fois un comportement serveur et un comportement client.

Typage statique ELIOM introduit un nouveau système de type qui permet une construction modulaire des programmes client-serveur tout en préservant les garanties du typage statique et la capacité d'abstraction. Ceci garantit, via le système de type, que le code client n'est pas utilisé dans le code serveur (et inversement) et assure que les communications sont correctes.

Communication explicites Les communications entre le serveur et le client sont explicites dans ELIOM. Ceci permet au programmeur de raisonner sur l'exécution du programme et le comportement résultant. Le programmeur peut, par exemple, s'assurer que certaines données ne quittent pas le serveur ou le client, ou choisir la quantité de communication ainsi que l'endroit ou les calculs sont effectués.

Modèle d'exécution efficace ELIOM repose sur un nouveau modèle d'exécution efficace pour les communications client-serveur qui évite un aller-retour répété. Ce modèle est simple et prévisible, ce qui est particulièrement important dans le cadre d'un langage impur tel que ML.

Dans cette présentation, nous exposerons notre travail sur la définition, formalisation et implémentation du langage ELIOM. La formalisation du langage d'expression à été présentée dans Radanne et al. [2016]. Un système de module est présenté dans Radanne and Vouillon [2017], en cours de soumission.

Références

G. Radanne and J. Vouillon. Tierless Modules. Submitted to ICFP 2017, Mar. 2017. URL https://hal.archives-ouvertes.fr/hal-01485362.

G. Radanne, J. Vouillon, and V. Balat. Eliom : A core ML language for tierless web programming. In A. Igarashi, editor, *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, volume 10017 of *Lecture Notes in Computer Science*, pages 377–397, 2016. doi : 10.1007/978-3-319-47958-3_20. URL http://dx.doi.org/10.1007/978-3-319-47958-3_20.

CONC2SEQ : un outil pour la vérification des compositions parallèles de programmes C* (résumé étendu)

Allan Blanchard[†1], Nikolai Kosmatov[‡2], Matthieu Lemerre[§2], and Frédéric Loulergue[¶3]

[1]Univ Orléans, INSA Centre Val de Loire, LIFO EA 4022, Orléans, France
[2]CEA, LIST, Software Reliability Lab, 91191 Gif-suf-Yvette, France
[3]School of Informatics, Computing, and Cyber Systems, Northern Arizona University, Flagstaff, USA

FRAMA-C est une plateforme de vérification de programmes C qui offre différentes analyses statiques et dynamiques sous la forme de greffons capables de collaborer. Cela inclut de l'interprétation abstraite, de la vérification déductive, de la génération de tests, de la vérification à l'exécution et bien d'autres. FRAMA-C propose un langage de spécification, ACSL, permettant d'annoter les programmes C avec des contrats. Actuellement, la plupart des analyseurs ne supportent pas les programmes concurrents, et ne permettent notamment pas la preuve de propriétés fonctionnelles sur les programmes concurrents.

Pour la classe des programmes *séquentiellement consistants* (les programmes dont les exécutions peuvent être vues comme l'ensemble des entrelacements de leurs instructions), l'analyse d'un programme concurrent peut être réalisée en analysant un programme séquentiel qui simule les entrelacements des fils d'exécution du programme. Cette méthodologie, basée sur la simulation, a été appliquée à une étude de cas dans [1]. Le programme séquentiel correspondant est généralement plus long et difficile à produire manuellement. Nous présentons CONC2SEQ [2], un nouveau greffon pour FRAMA-C ayant pour but de l'étendre à l'analyse de programmes concurrents. Contrairement à certains analyseurs dédiés aux programmes concurrents, l'idée est ici de permettre à des analyseurs de programmes séquentiels de traiter des programmes concurrents en automatisant la méthode de transformation de code et de spécification utilisée dans [1].

*Cette soumission est un résume étendu de l'article [2] paru à SCAM 2016.

†allan.blanchard@univ-orleans.fr
‡nikolai.kosmatov@cea.fr
§matthieu.lemerre@cea.fr
¶frederic.loulergue@nau.edu

À partir d'une API concurrente écrite en langage C, FRAMA-C génère un AST normalisé grâce à CIL. CONC2SEQ peut alors extraire les lectures en mémoire globale des instructions afin de les charger indépendamment dans des variables locales assurant que chaque instruction du programme ne contient qu'un seul accès à la mémoire globale. Ensuite, la transformation vers le programme simulant est réalisée. Chaque variable locale est transformée en un tableau qui va garder pour chaque indice de fil d'exécution (le nombre de fils est supposé borné) la valeur de la variable correspondante. Il en est de même pour les variables *thread-local* (variables globales dont chaque fil a son propre exemplaire) et pour le compteur de programme qui va indiquer pour chaque fil sa position dans l'exécution du code. Les variables globales sont conservées telles quelles.

Ensuite, chaque instruction atomique est transformée en une fonction qui va réaliser l'opération voulue pour le fil d'exécution dont l'identifiant est reçu en paramètre. Dans cette opération, chaque occurrence d'une variable locale est remplacée par un accès au tableau qui la simule. Une fois l'opération réalisée, le compteur de point de programme est mis à jour vers la prochaine instruction à exécuter (conformément au CFG dans le cas des conditionnelles et des boucles). Nous générons enfin une dernière fonction consistant en une boucle qui va, à chaque tour, sélectionner un fil d'exécution au hasard et lui faire exécuter le prochain pas d'exécution qu'il doit réaliser, produisant ainsi les entrelacements des instructions.

À la manière de la méthode d'Owicki-Gries, nous supportons la notion d'invariant global en imposant qu'un tel invariant doit être vrai avant et après chaque instruction atomique, ces spécifications sont donc transformées conformément à la simulation des variables locales et les invariants sont indiqués comme une pré et post condition de chaque fonction de simulation (et comme invariant de la boucle d'entrelacement).

Les perspectives futures d'extension du greffon vont du support du raisonnement *Rely-Guarantee*, au support des modèles mémoire faibles par transformation de code. Les travaux en cours sont la réalisation de la preuve de correction de la transformation à l'aide de l'assistant de preuve COQ sur un mini-langage impératif.

Remerciements. Ce travail a été partiellement financé par la DGE et BPIFrance (projet S3P).

Références

[1] A. Blanchard, N. Kosmatov, M. Lemerre, and F. Loulergue. A case study on formal verification of the Anaxagoros hypervisor paging system with FRAMA-C. In *International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, LNCS, pages 15–30, Oslo, Norway, June 2015. Springer.

[2] A. Blanchard, N. Kosmatov, M. Lemerre, and F. Loulergue. CONC2SEQ : A FRAMA-C plugin for verification of parallel compositions of C programs. In *16th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 67–72, Raleigh, NC, USA, 2016. IEEE.

Model Based Testing : maximiser la couverture structurelle de tests fonctionnels

Yanjun SUN, Gérard MEMMI and Sylvie VIGNES
LTCI, Télécom ParisTech, Université Paris-Saclay, 75013, Paris, France

Le problème que nous avons traité dans l'article de QRS MVV [1] concerne le renforcement de la génération de suites de tests fonctionnels à exécuter sur un modèle de façon à assurer une très haute couverture structurelle du modèle.

Le domaine d'application concerne le prototypage virtuel d'un système de contrôle commande de centrale nucléaire. Le contrôle commande est constitué de plusieurs centaines de systèmes élémentaires (SE) en charge de la protection et de la supervision du procédé physique. Le gain espéré de la démarche est de raccourcir les délais de production des études en garantissant le haut niveau de qualité et en bénéficiant de façon complémentaire des approches formelles, des tests et des simulations. Le contexte du système existant initialement développé par des automaticiens a conduit assez naturellement au choix de modéliser en Esterel Scade.

Un modèle Scade est formellement représenté par un ensemble hiérarchisé de blocs connectés offrant chacun et donc globalement des interfaces bien définies en termes de ports d'entrée/sortie. Chaque bloc décrit un comportement par une Machine à états finis étendue (EFSM) ; les interfaces sont des paires de vecteurs d'entrée et de sortie contraintes par les connections entre blocs à chaque pas de temps. La relation de transition d'un bloc s'exprime par des expressions booléennes et arithmétiques sur des vecteurs d'entrée/sortie aux valeurs booléennes ou numériques. Le processus de génération de suites de tests présenté dans l'article part d'un ensemble de tests exécutés sur le modèle Scade dont on mesure le critère de couverture par exemple MC/DC. Ce modèle étant transformé en Lustre, le processus permet d'augmenter la couverture en utilisant le model checker GATeL pour générer des tests pour des branches non initialement atteintes.
Dans cet article, nous présentons les résultats de validation fonctionnelle d'un SE du Contrôle Commande SRI, Système de Refroississement Intermédiaire.

[1] Y. Sun, G. Memmi and S. Vignes, "Model-Based Testing Directed by Structural Coverage and Functional Requirements," 2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), Vienna, 2016, pp. 284-291.

⋆ ⋆ ⋆ AFADL 2017 ⋆ ⋆ ⋆

Section doctorants

Critères de couverture pour combiner analyses statiques et dynamiques

VIET HOANG LE [1,2],
SOUS DIRECTION DE VIRGINIE WIELS[2], JULIEN SIGNOLES[1] et LOÏC CORRENSON[1]

[1]CEA LIST, Laboratoire de Sûreté des Logiciels
[2]ONERA/DTIS

May 5, 2017

Contexte et Problématique

La vérification sert à examiner le logiciel avant son utilisation pour prévenir des risques potentiels. Les systèmes informatiques étant néanmoins de plus en plus gros et complexes, les risques potentiels sont de plus en plus nombreux et difficiles à trouver. Plusieurs méthodes de vérification sont développées pour répondre à ce problème. Ces méthodes peuvent être classées en trois familles : les analyses dynamiques (représentant : test), les analyses statiques (représentant : preuve formelle) et les analyses combinées. Il est également nécessaire de s'assurer que les vérifications menées sur le logiciel sont suffisantes. Une méthode couramment utilisée est l'analyse de couverture. Il existe deux types de couverture :

- La couverture fonctionnelle, pour s'assurer que chaque exigence est couverte par les vérifications.
- La couverture structurelle, pour s'assurer que la structure du code est couverte par les vérifications.

La norme DO-178C[1, 2] définit ainsi des objectifs concernant l'analyse de couverture fonctionnelle et structurelle, déclinés selon la méthode de vérification utilisée (test pour la DO-178C, vérification formelle pour le supplément technique sur les méthodes formelles, la DO-333). Cependant, aucune déclinaison n'est proposée dans le cas d'analyses combinées mêlant à la fois test et vérification formelle statique alors même qu'elles sont aujourd'hui de plus en plus utilisées. L'objectif principal de mon travail est ainsi de définir des critères de couverture adaptés aux analyses combinées, critères qui n'existent aujourd'hui pas dans la littérature. Afin de pouvoir être adaptée à plusieurs types de combinaisons, la solution recherchée est une notion de couverture commune aux analyses dynamiques et statiques.

Présentation de l'approche

Couverture combinée

Un code contient deux parties : les points de programme et les instructions entre ces points. On s'inspire d'un système de label existant [3] pour définir un critère de couverture structurelle comme un ensemble de labels. Un label est une propriété associée à un point de programme. Il sert à vérifier une condition à ce point lorsqu'il est atteignable.

Une spécification est une condition entre deux points de programme différents. Elle sert à garantir les propriétés du code. La validation des spécifications est exigée : si une exécution du code ne valide pas une spécification alors le code contient une erreur. Le critère de couverture fonctionnelle correspond à un ensemble de spécifications à satisfaire. La spécification S peut être définie sous une forme $H \implies G$, où H désigne la précondition (ou l'hypothèse) et G la postcondition (ou le but). La spécification $\overline{S} = H \implies \neg G$ est l'opposition de S.

Chaque couple (label, spécification) forme une **connexion** qui est la base de la **couverture combinée** proposée ici. Un critère de couverture combinée est une collection de connexions. Au lieu de "couvrir" une spécification et un label, la vérification "couvre" une connexion. Autrement dit, la spécification de cette connexion couvre son label.

Témoin

Un témoin est le résultat d'une vérification pour une connexion d'un label L et d'une spécification S du programme P. Dans ce travail, on suppose que tous les témoins sont corrects. Selon la méthode de vérification considérée, on distingue deux témoins différents. Le **témoin de preuve** $P = \forall x.P(x) \vdash S$ indique que toutes les exécutions couvrent la spécification alors que **le témoin de test** "$T = \exists x.P(x) \vdash L, S$" indique qu'au moins une exécution couvre une connexion. On utilise le témoin pour une connexion (L, S) afin de noter l'existence d'une vérification qui montre que sa spécification S est satisfaite.

Mutation et couverture d'une connexion

En s'inspirant des techniques de *test par mutation* [4], on ajoute 4 autres témoins. Une mutation est un changement du code source. Dans ce travail, chaque mutation est guidée par un label : un mutant est créé par effacement de l'instruction au point du programme du label, de telle façon que, si une exécution ne passe pas le label, elle reste inchangée, tandis que si cette exécution passe le label, elle saute directement au point de programme suivant le label.

On peut définir 4 témoins supplémentaires en vérifiant la validité du mutant M_L (guidé par le label L) par rapport aux spécifications S et $\overline{S}$:

Témoins montrent $\overline{S}$	par preuve :	$OP = \forall x.M_L(x) \vdash \overline{S}$
	par test :	$OT = \exists x.M_L(x) \vdash L, \overline{S}$
Témoins montrent S	par preuve :	$SP = \forall x.M_L(x) \vdash S$
	par test :	$ST = \exists x.M_L(x) \vdash L, S$

Les témoins T, OT et ST représentent un test du code source ou du mutant. Ils doivent être accompagnés des cas de test qu'ils représentent. Une connexion C de label L et de spécification S est **couvert** si et seulement s'il existe d'une part un témoin P ou T et, d'autre part, un témoin OP ou OT pour C. Lorsque T et OT sont utilisés conjointement, ils doivent porter sur un même cas de test.

Exemple

```
int Fcorrecte(int x, int m, int n){
  L0: int res = n;
  L1: if(x >= n) { res = x; }
  L2: if(x >= m) { res = m; }
  L3: printf("%d",  res);
  return res;
}
```

```
int Ferreur(int x, int m, int n){
  L0: int res = n;
  L2: if(x >= m) { res = m; }
  L1: if(x >= n) { res = x; }
  L3: printf("%d",  res);
  return res;
}
```

On considère 2 versions d'une même fonction F qui sont présentées ci-contre, ainsi que les 3 spécifications suivantes de F :

- $S1 : n < x < m \implies \backslash result \equiv x;$
- $S2 : x \geq m \geq n \implies \backslash result \equiv m;$
- $S3 : x \leq n \leq m \implies \backslash result \equiv n.$

La première version (en haut) de F est correcte. La seconde version (en bas) est erronnée à cause d'une inversion de 2 instructions "if".

De manière similaire à [3], on définit les labels en fonction du type de couverture structurelle qui nous intéresse. Ici, on choisit la couverture des instructions. Les labels correspondants sont ainsi : $(L0, true)$, $(L1, true)$, $(L2, true)$ et $(L3, true)$.

```
int Fcorrect(int x, int m, int n){
  L0: int res = n;
  // L1:if(x >= n) { res = x; }
  L2: if(x >= m) { res = m; }
  L3: printf("%d",  res);
  return res;
}
```

Dans cet exemple, on a par conséquent 3 spécifications et 4 labels, et donc 12 connexions au total pour la couverture combinée. On crée ensuite des mutants. Le code ci-contre présente ainsi un mutant dans lequel l'instruction $L1$ a été effacée. On applique ensuite des vérifications sur le mutant. S'il existe un témoin OT ou OP pour une certaine spécification S, on sait que le mutant est tué par rapport à la vérification de S.

On essaie de prouver les spécifications $S1$, $S2$, $S3$ et $\overline{S1}$ sur les 2 codes $Fcorrecte$ et $Ferreur$. On teste aussi le mutant de la première version avec les jeux d'entrée $E_1 = (3, 9, 6)$ et $E_2 = (10, 9, 6)$, puis le code source initial et le mutant de la deuxième version avec les jeux d'entrée $E_1 = (3, 9, 6)$ et $E_3 = (10, 10, 6)$. Ces vérifications sont réalisées à l'aide de Frama-C [5].

Les résultats sont collectés sous formes de témoins et présentés dans les tableaux ci-après. Ils permettent de définir les connexions couvertes. Par exemple, on peut prouver $S1$ dans les deux versions, donc il existe un témoin P pour toutes ses connexions. On peut aussi prouver $\overline{S1}$ dans le mutant M_{L1} des deux versions, donc il

existe un témoin OP pour la connexion entre $S1$ et $L1$. Par conséquent, la connexion de $S1$ et $L1$ est couverte (autrement dit, $S1$ couvre $L1$). Par ailleurs, le label $L3$ des deux versions et le label $L2$ de la seconde version ne sont couverts par aucune spécification, tandis que $S2$ de la seconde version ne couvre aucun label.

		$S1$		$S2$		$S3$	
		$n < x < m \implies \backslash result \equiv x$		$x \geq m \geq n \implies \backslash result \equiv m$		$x \leq n \leq m \implies \backslash result \equiv n$	
$L0$	$(L0, true)$	P	SP	P	SP	P	$OT(E_1)$
$L1$	$(L1, true)$	P	OP	P	SP	P	SP
$L2$	$(L2, true)$	P	SP	P	$OT(E_2)$	P	SP
$L3$	$(L3, true)$	P	SP	P	SP	P	SP

		$S1$		$S2$		$S3$	
		$n < x < m \implies \backslash result \equiv x$		$x \geq m \geq n \implies \backslash result \equiv m$		$x \leq n \leq m \implies \backslash result \equiv n$	
$L0$	$(L0, true)$	P	SP	$T(E_3)$	$ST(E_3)$	P	$OT(E_1)$
$L2$	$(L2, true)$	P	SP	$T(E_3)$	$ST(E_3)$	P	SP
$L1$	$(L1, true)$	P	OP	$T(E_3)$	SP	P	SP
$L3$	$(L3, true)$	P	SP	$T(E_3)$	$ST(E_3)$	P	SP

Conclusion

Nous avons présenté une première approche définissant une couverture adaptée aux analyses combinant test et preuve. Elle permet d'obtenir un tableau de couverture entre label (représentant de code) et relation (représentant de spécification) dans lequel des témoins indiquent le type des vérifications effectuées sur le programme.

References

[1] Y. Moy, E. Ledinot, H. Delseny, V. Wiels et B. Monate : Testing or Formal Verification: DO-178C Alternatives and Industrial Experience. *IEEE Software*, pages 50–57, 2013.

[2] D. Brown, H. Delseny, K. Hayhurst et V. Wiels : Guidance for Using Formal Methods in a Certification Context. *ERTS 2010*, pages 19–21, 2010.

[3] S. Bardin, N. Kosmatov et F. Cheynier : Efficient Leveraging of Symbolic Execution to Advanced Coverage Criteria. *ICST 2014*.

[4] R. A. DeMillo, R. J. Lipton et F. G. Sayward : Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer*, 11:34–41, 1978.

[5] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles et B. Yakobowski : Frama-C: A software analysis perspective. *Formal Aspects of Computing*, page 573–609, 2015.

⋆ ⋆ ⋆ AFADL 2017 ⋆ ⋆ ⋆

Outils

OntoEventB : Un outil pour la modélisation des ontologies dans B Événementiel

Linda Mohand-Oussaïd and Idir Aït-Sadoune

LRI, CentraleSupelec, Université Paris Saclay
Plateau du Moulon, Gif-sur-Yvette, France
{linda.mohandoussaid, idir.aitsadoune}@centralesupelec.fr

1 Introduction

Cet article présente un plug-in intégré à la plateforme Rodin [2] implémentant deux approches [3][4] de formalisation des ontologies décrites par des langages de description des ontologiques (OWL, Plib, ...) en utilisant la théorie des ensembles et la logique du premier ordre supportées par B Événementiel [1]. L'intérêt de cette formalisation est d'enrichir le processus de spécification et de vérification utilisant la méthode B Événementiel, en intégrant des modèles de données et de connaissances décrits dans des ontologies. L'utilisation des ontologies dans un développement B Événementiel va servir à annoter et/ou à typer les données manipulées par le système, ce qui permet de vérifier, en plus des propriétés caractéristiques du système, des propriétés liées à la cohérence des données manipulées et induites par les contraintes du domaine exprimées dans les ontologies [3].

Cet article est organisé de la manière suivante : la section 2 présente l'architecture interne du plug-in, et la section 3 est consacrée à la description de la procédure d'installation et d'utilisation du plug-in. Enfin, nous concluons l'article avec des perspectives d'évolution relatives à ce plug-in.

2 L'architecture interne de OntoEventB

Le plug-in *OntoEventB* est développé selon une architecture interne articulée autour de trois composants (voir Figure 1) :

— **Le composant Modèles d'Entrée.** Ce composant est dédié au traitement des fichiers contenant les descriptions des ontologiques. Ce composant parcourt le contenu des fichiers en entrée, extrait les concepts ontologiques de bases (classes, propriétés, relations, ...) et les envoie en entrée du composant Modèle Pivot.

— **Le composant Modèle Pivot.** Ce composant contient un modèle intermédiaire qui regroupe les concepts communs et spécifiques utilisés par les langages de description ontologiques comme OWL, Plib ou le diagramme de classe d'UML (classes, propriétés, relations entre classes, relations entre propriétés). Le composant Modèle Pivot récupère l'ensemble des concepts reçus depuis le composant Modèles d'Entrée, les fait correspondre aux concepts de son modèle pivot et les insère dans ce modèle. Il prépare ces concepts au traitement (formalisation B Événementiel) qui est effectué par le composant Modèles de Sortie.

— **Le composant Modèles de Sortie.** Ce composant implémente les deux approches de formalisation des ontologies, Shallow [3] et Deep [3][4]. L'approche Shallow modélise les concepts définis dans une ontologie en s'appuyant sur la sémantique de B Événementiel (les classes et les propriétés de l'ontologie sont formalisées par des ensembles et des relations dans un contexte B Événementiel) tandis que l'approche Deep utilise une modélisation en profondeur qui consiste, dans un premier temps, à formaliser les concepts ontologiques génériques dans un contexte B Événementiel générique (les notions de classes, de propriétés, d'héritage, ..., sont définies dans un contexte B Événementiel), et à modéliser les concepts définis dans une ontologie dans un second contexte B Événementiel (qui étend le contexte générique) par un ensemble d'instances reliées aux concepts génériques du contexte générique. Le composant Modèles de Sortie parcourt le modèle pivot et génère un projet Rodin contenant des contextes B Événementiel selon l'approche choisie par l'utilisateur.

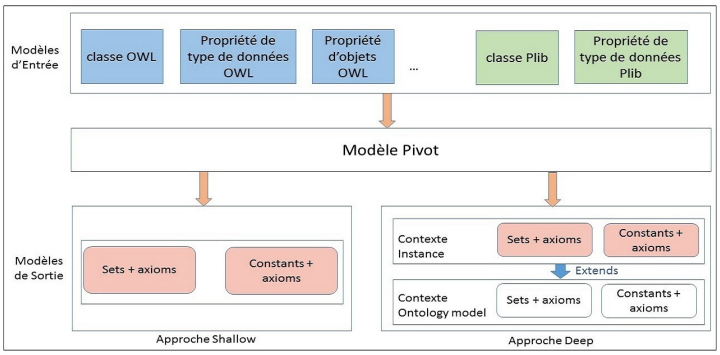


FIGURE 1. L'architecture interne de OntoEventB.

L'utilisation de cette architecture permet d'étendre le plug-in *OntoEventB* en intégrant de nouveaux langages de description des ontologies en entrée sans redéfinir les règles de formalisation en B Événementiel entre les composants Modèle Pivot et Modèles de Sortie. Une fois la correspondance entre les concepts du nouveau langage et les concepts du Modèle Pivot établie, la formalisation en B Événementiel est directement effectuée sans modifier les règles de formalisation définies par les approches Shallow et Deep.

3 Installation et utilisation de OntoEventB

Dans le soucis d'intégrer l'outil *OntoEventB* à la plateforme Rodin, nous avons fait le choix de le développer sous la forme d'un plug-in Eclipse. Ainsi, pour l'utiliser, il

faut d'abord l'intégrer à la plateforme Rodin en passant par l'assistant d'installation des nouveaux logiciels de la plateforme, et en utilisant l'adresse du update site du plug-in [1].

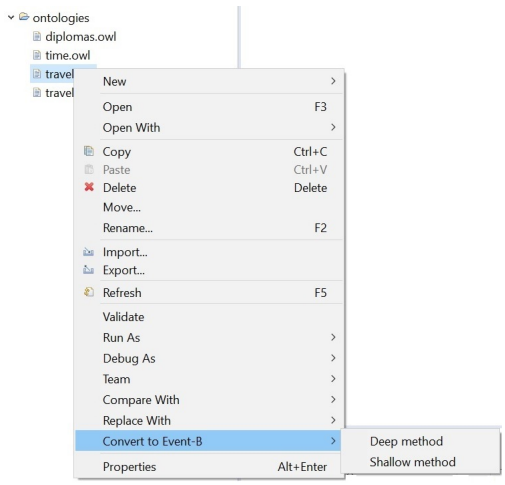


FIGURE 2. Le sous-menu OntoEventB.

Une fois l'installation du plug-in effectuée, un sous-menu portant l'étiquette *Convert to Event-B* devient accessible par clic droit sur un fichier possédant l'extension *.owl* [2] dans l'explorateur de projet comme indiqué dans la figure 2. Ce menu propose deux fonctions : la fonction Deep Method et la fonction Shallow Method. Nous illustrons le fonctionnement de OntoEventB sur une ontologie OWL relative au domaine des voyages. Un extrait de cette ontologie est présenté ci-dessous. Il contient deux classes *Accommodation* et *Destination* et une propriété d'objet *hasAccommodation*.

travel.owl

```
<owl:Class rdf:ID="Accommodation"></owl:Class>
<owl:Class rdf:ID="Destination"></owl:Class>
<owl:ObjectProperty rdf:ID="hasAccommodation">
        <rdfs:range rdf:resource="#Accommodation"/>
        <rdfs:domain rdf:resource="#Destination"/>
</owl:ObjectProperty>
```

1. **La fonction Shallow method.** L'invocation de cette fonction sur un fichier OWL conduit à la création d'un nouveau projet B Événementiel composé d'un unique contexte contenant la formalisation de l'ontologie reçue en entrée en utilisant des ensembles, des constantes et des axiomes, résultat de l'application de

1. `http://downloads.sourceforge.net/project/ontoeventb/update-site`
2. La version actuelle du plug-in supporte uniquement les ontologies OWL en entrée.

l'approche Shallow. La Figure 3 présente le contexte *Travel* généré relativement à l'extrait présenté ci-dessus. Les classes *Accommodation* et *Destination* sont formalisées par les ensembles *Accommodation* et *Destination* sous-ensembles de *Thing* (ensemble abstrait correspondant à la classe racine Thing de OWL). La relation d'héritage est formalisée par l'opérateur $\subseteq$. La propriété *hasAccommodation* est formalisée par une relation possédant comme domaine l'ensemble *Accommodation* et comme co-domaine l'ensemble *Destination*.

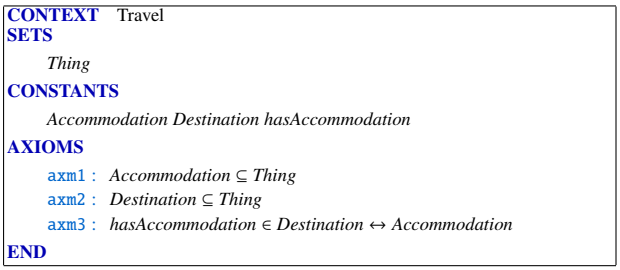


FIGURE 3. La fonction OntoEventB Shallow

2. **La fonction Deep method.** L'invocation de cette fonction sur un fichier OWL conduit à la création d'un nouveau projet B Événementiel contenant deux contextes : un contexte générique *ontologyModel* définissant les concepts ontologiques génériques, généré pour toutes les ontologies et un contexte instance *Travel* étendant le contexte *ontologyModel* et décrivant l'ontologie en appliquant l'approche Deep. La Figure 4 présente les contextes générés relativement à l'extrait présenté ci-dessus. Dans le contexte *ontologyModel*, sont définis les ensembles de classes *CLASS* et de propriétés *PROPERTY*. Les relations *HAS_PROPERTY* et *IS_A* sont définies pour modéliser respectivement la relation entre les classes et les propriétés qui les caractérisent et la relation de subsomption entre classes. Dans le contexte *Travel*, les classes *Accommodation* et *Destination* sont formalisées par des instances de l'ensemble *CLASS*, elle sont reliées à l'ensemble Thing par des instances de *IS_A*. La propriété *hasAccommodation* est formalisée par une instance de l'ensemble *HAS_PROPERTY*.

4 Conclusion

Cet article présente l'implémentation d'une approche de formalisation des ontologies dans une méthode formelle comme B Événementiel à travers le plug-in *OntoEventB*. Cet outil permet de générer automatiquement des contextes B Événementiel formalisant des ontologies de domaine à intégrer dans un processus de développement formel d'un système utilisant B Événementiel. Ces contextes ainsi générés permettent de spécifier et de vérifier des contraintes spécifiques au domaine. D'autre part, l'utilisation d'un formalisme tel que B Événementiel offre également la possibilité de valider

CONTEXT ontologyModel
SETS

$CLASS\ PROPERTY$

CONSTANTS

$HAS\_PROPERTY\ IS\_A$

AXIOMS

axm1 : $HAS\_PROPERTY = CLASS \leftrightarrow PROPERTY$

axm2 : $IS\_A = \{IsA \mid IsA \in CLASS \leftrightarrow CLASS \land (\forall x, y \cdot (x \in CLASS \land y \in CLASS \land x \mapsto y \in IsA \Leftrightarrow ...)\}$

END

CONTEXT Travel
EXTENDS OntologyModel
CONSTANTS

$Thing\ Accommodation\ Destination\ hasAccommodation\ isA\ hasProperties$

AXIOMS

axm1 : $partition(CLASS, \{Thing\}, \{Destination\}, \{Accommodation\})$

axm2 : $partition(PROPERTY, \{hasAccommodation\})$

axm3 : $hasProperties = \{Destination \mapsto hasAccommodation\}$

axm4 : $isA = \{Destination \mapsto Thing, Accommodation \mapsto Thing\}$

axm5 : $isA \in IS\_A$

axm6 : $hasProperties \in HAS\_PROPERTIES$

END

FIGURE 4. La fonction OntoEventB Deep

les règles d'inférence utilisés par les raisonneurs ontologiques, qui peuvent être formalisées par des théorèmes à prouver.

La version actuelle du plug-in se base sur une architecture qui facilite l'extension du plug-in à d'autres langages de description des ontologies en entrée. En effet, le Modèle Pivot intègre l'ensemble des constructeurs utilisés par des langages comme OWL, Plib ou UML. La formalisation de l'ensemble des éléments composant le modèle Pivot en B Événementiel a été effectuée. Comme l'outil ne supporter que des ontologies OWL en entrée, l'intégration de nouvelles ontologies en entrée (comme les ontologies Plib par exemple) se fera au niveau du composant Modèles d'Entrée qui doit définir la correspondance des éléments du nouveau langage avec les éléments du modèle Pivot.

Remerciement. Ce travail est supporté par le projet ANR IMPEX.

Références

1. J-R Abrial. *Modeling in Event-B : system and software engineering*, Cambridge University Press, 2010.
2. J-R Abrial, M J. Butler, S. Hallerstede, T. Son Hoang, F. Mehta et L. Voisin. *Rodin : an open toolset for modelling and reasoning in Event-B*, STTT, vol 12(6), 2010.
3. IMPEX Consortium. Formal models for ontologies, June 2016.
4. Kahina Hacid, Yamine Aït Ameur. *Strengthening MDE and Formal Design Models by References to Domain Ontologies. A Model Annotation Based Approach*. ISoLA (1) 2016 : 340-357.