INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Projet LANDE

Logiciel : analyse et développement

Rennes

THÈME 2A

Table des matières

1 Composition de l'équipe

Responsable scientifique

Thomas Jensen [CR CNRS]

Assistante de projet

Catherine Godest [TR CNRS]

Personnel Inria

Pascal Fradet [CR]

Florimond Ployette [IR, Atelier]

Olivier Ridoux [maître de conférences ; CR en détachement à l'université de Rennes 1 depuis février 1999]

Personnel Université

Thomas Genet [maître de conférences]

Michaël Périn [Ater jusqu'en septembre 2000]

Valérie-Anne Nicolas [Ater jusqu'en septembre 2000]

Erwan Jahier [allocataire MENRT jusqu'en septembre 2000, Ater depuis octobre 2000]

Siegfried Rouvrais [allocataire MENRT jusqu'en septembre 2000, Ater depuis octobre 2000]

Personnel Insa

Mireille Ducassé [professeur]

Julien Mallet [Ater Insa jusqu'en septembre 2000]

Chercheurs doctorants

Frédéric Besson [allocataire MENRT]

Thomas Colcombet [allocataire ENS]

Marc Éluard [boursier Inria]

Sébastien Ferré [boursier CNRS/RÉGION]

Lakshminarayanan Renganarayanan [boursier Inria]

Yoann Padioleau [allocataire MENRT, depuis octobre 2000]

Jean-Philippe Pouzol [boursier Inria, depuis octobre 2000]

Valérie Viet Triem Tong [allocataire MENRT, depuis octobre 2000]

Chercheurs post-doctorants

Fausto Spoto

Visiteurs

Romain Guider [de mai à juin 2000]

Bjarke Ebert [d'octobre à décembre 2000]

2 Présentation et objectifs généraux

Le thème de recherche central du projet Lande est la conception de méthodes et d'outils d'aide au développement et à la validation de logiciels. Ces méthodes possèdent deux caractéristiques majeures :

- Elles reposent sur des bases formelles (sémantique de langage, modèle de sécurité, etc.) permettant d'apporter des garanties quant à la correction des outils.
- Elles conduisent, autant que faire se peut, à des outils automatiques. Les utilisateurs visés sont en effet des programmeurs ou des valideurs qui ne possèdent pas forcément d'expertise particulière en matière de méthodes formelles ou de techniques de preuves.

Pour atteindre ces objectifs, nous distinguons deux types d'interventions :

- Les actions de type «amont » qui apportent des solutions de nature linguistique: leur résultat a donc un impact sur le processus de développement lui-même.
- Les actions de type « aval », comme l'analyse de programmes ou le test, qui s'appliquent à des logiciels dont on ne maîtrise pas forcément le développement.

Nous illustrons ces deux types d'activités par quelques exemples de travaux récents ou en cours dans le projet.

Actions « amont »

Les actions «amont» sont utilisables dans le cas, idéal, où il est possible d'agir dans la phase de développement du logiciel. Le but visé est alors de fournir des moyens de construction qui faciliteront la phase ultérieure de validation du logiciel. Ces actions se traduisent notamment par des méthodes et des langages qui induisent une *discipline de développement ou de programmation*. Ces langages restreints fournissent un pouvoir d'expression appréciable tout en offrant un niveau de description qui facilite le raisonnement formel. On constate d'ailleurs actuellement l'émergence d'un domaine de recherche sur les «langages dédiés» qui repose essentiellement sur ces constatations.

Une action importante dans la catégorie « amont » concerne les architectures de logiciels. L'objectif en la matière consiste à spécifier l'organisation globale de logiciels afin d'améliorer la maîtrise des gros systèmes (développement, analyse, test, maintenance, etc.). Une ambition majeure dans ce domaine est le passage à l'échelle de techniques comme l'analyse, le raffinement ou la vérification de programmes. Nous avons proposé une manière de spécifier des architectures en terme de graphes. Nos travaux actuels sur ce thème concernent la possibilité de traiter des vues multiples d'une architecture tout en assurant une forme de cohérence globale de la spécification. L'application de ces travaux est effectuée dans le cadre d'une collaboration industrielle concernant la spécification de propriétés de sécurité sur les architectures (projet Castor financé par le Celar).

Nous avons également entamé des travaux sur une nouvelle technique de programmation appelée «programmation par aspects». Elle consiste à décrire un logiciel comme un ensemble formé d'un composant principal et d'une collection d'aspects décrivant des tâches comme la gestion mémoire, la synchronisation, les optimisations, etc. Un outil, appelé tisseur, est chargé de produire automatiquement un programme intégrant les différents aspects au composant principal. L'intérêt de cette approche est de localiser (dans les aspects) des choix de mise en œuvre qui seraient sinon dispersés dans le code source. Nous avons proposé un cadre générique de programmation par aspects et son application à la production de logiciels robustes. Nous avons ensuite appliqué ce paradigme à la sécurité où un aspect décrit une politique de sécurité.

Actions « aval »

Les traitements « en aval » ou *a posteriori* concernent la validation de codes existants (vérification, test, débogage). Ce type d'actions s'applique à des logiciels dont on ne maîtrise pas forcément le développement. Nous concevons dans ce cadre des techniques d'aide à la validation de programmes qui reposent essentiellement sur des *analyses de programmes* (statiques ou dynamiques). Nous avons proposé notamment un cadre pour la vérification de propriétés de sécurité de programmes. Des analyses de flot de contrôle et de données sont utilisées pour construire un modèle abstrait du programme ; la vérification se fait ensuite sur ce modèle abstrait. Ces travaux vont se poursuivre dans le cadre du projet européen Secsafe. Nous nous intéressons également à l'analyse dynamique qui sert de base pour des outils de maintenance dans le cadre de la programmation impérative (Coca), de la programmation logique (Opium, Morphine) et avec contraintes.

Le projet consacre également une part de ses travaux au test de logiciels, qui représente une activité complémentaire à l'analyse. Nous avons proposé une méthode de génération de

suites de test qui a conduit à l'outil Casting développé en collaboration avec la société AQL. Les suites de test engendrées dépendent de stratégies spécifiées par l'utilisateur, permettant ainsi d'atteindre la souplesse d'utilisation exigée pour un usage industriel. La première version de Casting prend en entrée des spécification dans la syntaxe AMN de la méthode B. Nous travaillons actuellement à la transposition de ces idées au test structurel (de programmes C, et C++) dans le cadre du projet européen Two.

3 Fondements scientifiques

Une caractéristique importante des méthodes proposées dans le projet Lande est de reposer sur des bases formelles. S'agissant des langages de programmation, ces bases peuvent être fournies de différentes manières par ce qu'on appelle des *sémantiques*. Ces sémantiques sont ensuite utilisées pour définir des *analyses de programmes* qui permettent d'extraire des informations à partir du code des programmes (analyse statique) ou d'une trace d'exécution (analyse dynamique). Les analyses peuvent avoir différentes applications et celles qui intéressent au premier chef le projet Lande sont l'aide à la mise au point ou *débogage* de programmes et le *test de logiciels*. Ces applications ne concernent pas un langage de programmation spécifique mais la validation des programmes peut être notoirement simplifiée si on peut imposer une discipline de programmation *a priori*. Les langages de haut niveau, en particulier les *langages déclaratifs* peuvent être vus comme un moyen d'introduire une telle discipline.

3.1 Sémantique des langages de programmation

Mots clés : sémantique, sémantique dénotationnelle, sémantique opérationnelle.

Résumé : *La sémantique d'un langage de programmation s'attache à donner un sens mathématique aux programmes. Il existe différentes méthodes formelles de définition de sémantique comme les méthodes opérationnelle et dénotationnelle. Une sémantique dénotationnelle attribue un sens aux programmes d'un langage en associant à chaque construction syntaxique du langage une valeur dans un domaine de définition. Une sémantique opérationnelle donne un sens aux programmes en terme d'étapes de calcul (ou réécritures). Quel que soit son mode de définition, une sémantique permet d'ôter toute ambiguïté dans la définition d'un langage de programmation. Elle peut aussi fournir une base pour des techniques de manipulation formelle de programmes : preuves de propriétés de correction, analyse, transformation. C'est dans cette optique que les sémantiques de langages sont utilisées dans le projet Lande.*

La sémantique d'un langage de programmation s'attache à donner un sens mathématique aux programmes. Il existe différentes méthodes formelles de définition de sémantique comme les méthodes axiomatique, algébrique, opérationnelle ou dénotationnelle. Nous présentons ici les méthodes dénotationnelle et opérationnelle sur un langage très simple d'expressions arithmétiques :

$$E ::= N \mid E_1 + E_2 \qquad \text{où } N \text{ représente un entier}$$

Sémantiques dénotationnelles

Une sémantique dénotationnelle [Sch86] attribue un sens aux programmes d'un langage à l'aide d'une fonction qui associe à chaque construction syntaxique du langage une valeur dans un domaine de définition. La sémantique d'une expression est construite à partir de celle de ses sous-expressions ; on dit que la sémantique est compositionnelle. La technique de preuve classique quand on travaille avec de telles sémantiques est la récurrence sur la structure (*structural induction*).

En prenant les entiers naturels Nat comme domaine sémantique et la fonction $Plus : Nat \times Nat \rightarrow Nat$, la sémantique dénotationnelle de notre langage se décrit comme suit :

$$\begin{array}{lcl} \mathcal{E} & : & \text{Expression} \rightarrow Nat \\ \mathcal{E}[\![N]\!] & = & Val(N) \\ \mathcal{E}[\![E_1 \ + \ E_2]\!] & = & Plus\ (\mathcal{E}[\![E_1]\!],\ \mathcal{E}[\![E_2]\!]) \end{array}$$

Dans la deuxième ligne de cet exemple, il est important de noter la distinction entre le symbole N, qui dénote un élément de syntaxe du langage, et $Val(N)$ qui représente la valeur correspondant à N dans l'ensemble Nat. Sur ce langage élémentaire, la sémantique dénotationnelle apparaît presque comme une paraphrase de la syntaxe. Ce n'est plus le cas pour des langages plus réalistes. Par exemple, la sémantique d'un langage impératif classique encode à l'aide de fonctions un environnement, une mémoire et le flot de contrôle ; la sémantique d'un programme récursif est la plus petite solution de l'équation qui le définit (*plus petit point fixe*).

Sémantiques opérationnelles

Les sémantiques opérationnelles donnent un sens aux programmes en terme d'étapes de calcul (ou réécritures). Nous présentons ici deux styles de sémantiques opérationnelles : les sémantiques opérationnelles structurelles et les sémantiques naturelles.

Une *sémantique opérationnelle structurelle* (SOS) [NN92] est un système composé d'axiomes et de règles d'inférence qui décrit le comportement du programme en terme d'étapes élémentaires de calcul (on parle de sémantique à petits pas). La technique de preuve classique associée à ce type de sémantique est la récurrence sur le nombre d'étapes de calcul.

La SOS de notre langage se décrit à l'aide d'un axiome et de deux règles d'inférence :

$$N_1 + N_2 \Rightarrow N \text{ où } N \text{ est la somme de } N_1 \text{ et } N_2$$

$$\frac{E_1 \Rightarrow E_1'}{E_1 \ + \ E_2 \Rightarrow E_1' \ + \ E_2} \qquad \frac{E_2 \Rightarrow E_2'}{N \ + \ E_2 \Rightarrow N \ + \ E_2'}$$

Une règle d'inférence est constituée d'hypothèses (partie haute) et de conclusions (partie basse). Dans cet exemple, N dénote une expression complètement réduite (c'est à dire un entier) et E_i des expressions quelconques. La seconde règle ne peut donc s'appliquer que si l'expression à gauche du symbole + a déjà été calculée, ce qui impose un ordre d'évaluation des arguments « gauche-droite ».

[Sch86] D. Schmidt, *Denotational Semantics*, Allyn & Bacon, 1986.

[NN92] H. R. Nielson, F. Nielson, *Semantics with applications*, John Wiley & Sons, INC., 1992.

Une *sémantique naturelle* [Kah87] décrit le comportement du programme par un arbre de dérivation décrivant le calcul de ses composants. Elle ne fait apparaître que les réductions des expressions en leur résultat final (leur forme normale). On parle de sémantique à grands pas et la technique de preuve associée est la récurrence sur les arbres de dérivation.

La sémantique naturelle de notre langage se décrit comme suit.

$$N \Rightarrow N \qquad \frac{E_1 \Rightarrow N_1 \quad E_2 \Rightarrow N_2}{E_1 \ + \ E_2 \Rightarrow N} \quad \text{où } N \text{ est la somme de } N_1 \text{ et } N_2$$

Contrairement à la SOS précédente, cette sémantique n'impose pas d'ordre d'évaluation particulier entre E_1 et E_2. Les sémantiques naturelles permettent de cumuler certains avantages des SOS et des sémantiques dénotationnelles : comme les premières, elles fournissent des informations sur les étapes de calcul, ce qui facilite la définition d'un certain nombre d'analyses ; comme les secondes, elles déterminent le sens d'une expression en fonction de ceux de ses sous-expressions. Cette forme de compositionnalité facilite les raisonnements sur les programmes.

Quel que soit son mode de définition, une sémantique permet d'ôter toute ambiguïté dans la définition d'un langage de programmation. Elle peut aussi fournir une base pour des techniques de manipulation formelle de programmes : preuves de propriétés de correction, analyse, transformation. C'est dans cette optique que les sémantiques de langages sont utilisées dans le projet Lande.

3.2 Analyse de programmes

Mots clés : analyse dynamique, analyse statique, sémantique, interprétation abstraite, compilation optimisante.

Glossaire :

Interprétation abstraite L'interprétation abstraite est un cadre permettant de relier différentes interprétations sémantiques d'un programme. Souvent, l'interprétation abstraite sert à montrer la correction d'une analyse, présentée comme une définition de la sémantique d'un langage sur un ensemble de propriétés «abstraites» (par rapport à la sémantique standard du langage).

Itération de points fixes Le résultat d'une analyse est souvent donné comme la solution d'une équation $x = f(x)$ où f est une fonction monotone sur un ordre partiel. Le théorème de Knaster-Tarski indique un algorithme pour trouver un tel point fixe en calculant la limite de la suite itérative $f^n(\bot)$ où $\bot$ désigne l'élément le plus petit dans l'ordre partiel.

Résumé : *L'analyse de programmes désigne l'ensemble des techniques qui permettent de déduire mécaniquement des propriétés des programmes. Ses principaux domaines d'application sont la compilation et l'aide à la mise au point de programmes. Comme exemples d'analyses existantes, nous pouvons citer l'analyse d'alias, le* slicing*, les analyses de dépendances. Une analyse peut être dynamique,*

[Kah87] G. KAHN, « Natural semantics », *in : Proceedings of STACS'87*, LNCS *247*, Springer Verlag, p. 22–39, 1987.

> *elle porte alors sur une trace d'exécution particulière, ou statique, et valable pour toute exécution de programme. Dans les deux cas, sa correction doit être assurée, ce qui signifie que les informations qu'elle procure doivent être cohérentes avec la sémantique du programme analysé.*

L'analyse de programmes désigne l'ensemble des techniques qui permettent de déduire mécaniquement des propriétés des programmes. Ses principaux domaines d'application sont la compilation et l'aide à la mise au point de programmes.

- Les analyses de flots de données qui permettent de détecter notamment les variables inutiles ou les expressions calculées à un point de programme donné.
- Les analyses d'alias qui produisent des informations sur le partage entre variables dans les langages à manipulation explicite de pointeurs.
- Les analyses de nécessité qui identifient les arguments qui sont indispensables à l'évaluation d'une fonction.
- Les analyses de mode qui déterminent le degré d'instanciation des variables dans les prédicats logiques.
- Le filtrage de programmes (*slicing*) qui consiste à identifier les instructions d'un programme nécessaires au calcul de variables données.

On distingue deux classes d'analyses : les analyses dynamiques et les analyses statiques. L'analyse dynamique déduit des propriétés d'un programme à partir d'une trace d'exécution particulière [BGL93]. En revanche, l'analyse statique [NNH99] permet d'établir des propriétés satisfaites par un programme pour toutes ses exécutions. L'information recherchée est en général incalculable ou d'une complexité importante. Une analyse statique ne peut donc calculer qu'une approximation de la solution idéale. En conséquence, les résultats de l'analyse statique sont moins précis mais plus généraux que ceux fournis par une analyse dynamique.

La conception d'une analyse comprend deux phases : la spécification et l'implantation. L'analyse doit être spécifiée d'une manière qui permet de prouver sa correction ; celle-ci garantit la cohérence du résultat de l'analyse par rapport à la sémantique du langage (cf module 3.1). La correction et la précision des analyses ont été étudiées de manière extensive dans le cadre de l'interprétation abstraite [Cou97]. Le résultat de cette première phase de conception d'analyse est souvent un système d'équations récursives dont la solution décrit les propriétés recherchées. On dispose d'algorithmes itératifs pour résoudre ce système d'équations (« itérateurs de points fixes»). On peut également s'appuyer sur des calculs formels sur l'algèbre des propriétés étudiées (calcul symbolique) afin d'améliorer l'efficacité de la résolution.

[BGL93] B. Bruegge, T. Gottschalk, B. Luo, « A framework for dynamic program analyzers », *in : Proc. of the OOPSLA'93 Conference*, p. 65–82, 1993.

[NNH99] F. Nielson, H. Nielson, C. Hankin, *Principles of Program Analysis*, Springer, 1999.

[Cou97] P. Cousot, « Abstract Interpretation Based Static Analysis Parameterized by Semantics », *in : Proc. of 4th Static Analysis Symposium*, P. Van Hentenryck (éditeur), Springer Verlag, LNCS vol. 1302, p. 388–394, 1997.

3.3 Débogage

Mots clés : environnement de programmation, analyse de programme, sémantique.

Glossaire :

Erreur Une erreur est une action humaine qui fait qu'un résultat incorrect est produit par un programme. Par exemple, une erreur peut être d'intervertir deux variables A et B.

Faute Une faute est une étape, un processus ou une définition de données erronés dans un programme. Une erreur peut générer une ou plusieurs fautes. Par exemple, une faute induite par l'erreur citée plus haut peut être qu'un test d'arrêt d'une boucle se fait sur A qui n'est pas mise à jour.

Panne Une panne est l'incapacité d'un programme à effectuer ses fonctionnalités requises. Une faute peut générer une ou plusieurs pannes [ANS]. Un exemple de panne résultant de la faute citée plus haut est que le programme ne termine pas.

Résumé : *Le débogage consiste à localiser et corriger les fautes qui sont responsables des pannes logicielles. Le débogage est une activité cognitive complexe qui nécessite, en général, de remonter jusqu'à l'erreur humaine pour comprendre les raisons des fautes qui ont engendré les pannes.*

Il existe des outils, communément appelés débogueurs*, qui aident le programmeur à identifier les comportements non-attendus du programme. Ces outils donnent une image (appelée* trace*) des détails de l'exécution des programmes. On peut identifier trois tâches principales pour la réalisation d'un véritable débogueur. La première tâche consiste à déterminer les informations qui doivent apparaître dans la trace. La deuxième tâche est la mise en œuvre des traceurs. La troisième tâche consiste à automatiser le filtrage et l'analyse des traces d'exécution afin de donner des informations pertinentes au programmeur qui peut ainsi se concentrer sur le processus cognitif.*

Le débogage consiste à localiser et corriger les fautes qui sont responsables des pannes logicielles. Une panne peut être détectée après une exécution, par exemple à la suite de phases de test (cf module 3.4) ou lors d'une phase de vérification formelle. La première situation, la plus fréquente dans la pratique actuelle, correspond à ce qu'on appelle le *débogage dynamique* ; la seconde sera qualifiée de *débogage statique*. Dans les deux cas, l'objectif visé est de faire cesser les pannes identifiées.

Le débogage est une activité cognitive complexe qui nécessite, en général, de remonter jusqu'à l'erreur humaine pour comprendre les raisons des fautes qui ont engendré les pannes. Une panne est un symptôme de faute qui se manifeste en un comportement erroné du programme. Bien souvent le programmeur ne maîtrise pas toutes les facettes du comportement d'un programme. Par exemple, des points de sémantique opérationnelle du langage peuvent lui échapper, le programme peut être trop complexe, ou les librairies utilisées peuvent avoir une documentation obscure. Nous présentons dans un premier temps la problématique du débogage dynamique avant de résumer les particularités introduites par le débogage statique.

[ANS] «ANSI/IEEE Standard 729-1983», Glossary of Software Engineering Terminology.

Pour ce qui est du débogage dynamique, il existe des outils, communément appelés *débogueurs*, qui aident le programmeur à identifier les comportements du programme qui ne correspondent pas à l'idée qu'il s'en faisait. Ces outils, qui devraient plutôt s'appeler *traceurs*, donnent une image (appelée *trace*) des détails de l'exécution des programmes. Une trace est composée d'*événements* remarquables.

On peut identifier trois tâches principales pour la réalisation d'un véritable débogueur dynamique :

1. La première tâche consiste à déterminer les informations qui doivent apparaître dans la trace. La trace est calquée sur la sémantique opérationnelle du langage sans forcément en donner tous les détails. Elle fournit une abstraction des étapes de calcul dont l'objectif est la compréhension par l'utilisateur du comportement des programmes. Elle dépend donc du langage et du type d'utilisateur potentiel.

2. La deuxième tâche est la mise en œuvre des traceurs qui nécessite l'insertion d'instructions de trace dans les mécanismes d'exécution des programmes (appelée *instrumentation* dans la suite). Cette instrumentation peut se faire à différents niveaux : dans le code source, dans le compilateur ou dans l'émulateur quand il en existe un. La pratique courante consiste à instrumenter à un niveau bas [Ros96] mais plus l'instrumentation est faite à un niveau haut, plus elle est portable.

3. Quand le programmeur dispose d'un traceur, il lui reste à analyser les traces pour comprendre les comportements des programmes et localiser les fautes. Cependant ces traces donnent souvent trop de détails par rapport à la panne analysée. La troisième tâche consiste à automatiser le filtrage et l'analyse des traces d'exécution afin de donner des informations plus pertinentes au programmeur qui peut ainsi se concentrer sur son processus cognitif.

La dichotomie débogueur statique / débogueur dynamique reflète tout à fait la distinction introduite plus haut (module 3.2) entre analyse statique et analyse dynamique. De fait, un débogueur statique peut être vu comme un analyseur statique dédié à la vérification de certaines classes de propriétés et intégré dans un outil interactif. L'interaction doit permettre à l'utilisateur de vérifier certaines hypothèses sur le comportement du programme et d'identifier d'éventuelles causes de dysfonctionnement sans exécuter le programme. Les trois tâches identifiées plus haut pour le débogage dynamique se retrouvent *mutatis mutandis* dans le contexte du débogage statique : les traces sont alors des abstractions de la sémantique opérationnelle du langage (cf module 3.1) et le filtrage réalise une approximation permettant de rendre décidable la propriété recherchée.

3.4 Test de logiciels

Mots clés : critère de test, hypothèse de test, test unitaire, test d'intégration, test

[Ros96] J. Rosenberg, *How debuggers work, Wiley Computer Publishing*, John Wiley & Sons, INC., 1996, ISBN 0-471-14966-7.

fonctionnel, test système, test en boîte noire, test en boîte blanche, test structurel.

Glossaire :

Jeu de test Un jeu de test est un ensemble de données de test.

Critère de test Un critère permet de spécifier formellement un objectif (informel) de test. Un critère de test peut, par exemple, indiquer le parcours de toutes les branches d'un programme, ou l'examen de certains sous-domaines d'une opération.

Validité Un critère de test est dit valide si pour tout programme incorrect, il existe un jeu de test non réussi satisfaisant le critère.

Fiabilité Un critère est dit fiable s'il produit uniquement des jeux de test réussis ou des jeux de test non réussis. Les jeux de test satisfaisant un critère fiable sont donc équivalents du point de vue du test.

Complétude Un critère est dit complet pour un programme s'il produit uniquement des jeux de test qui suffisent à déterminer la correction du programme (pour lequel tout programme passant le jeu de test avec succès est correct) [XMd+94]. Tout critère valide et fiable est complet.

Hypothèse de test La complétude étant hors d'atteinte en général, on peut qualifier un jeu de test par des hypothèses de test qui caractérisent les propriétés qu'un programme doit satisfaire pour que la réussite du test entraîne sa correction

Résumé :

Le test comporte une grande variété de tâches qui comprend notamment la conception des jeux de test, leur instrumentation, leur exécution, le dépouillement des résultats et la sélection des tests de non-régression (en cas de modification des programmes). La plupart de ces étapes repose sur l'empirisme et l'aide fournie par les outils actuels reste insuffisante. Cependant, certaines de ces tâches peuvent être systématisées et même, dans une certaine mesure, mécanisées. La génération de jeux de test en fait partie et sa systématisation constitue l'objectif majeur des activités du projet Lande sur ce thème. Les principales difficultés à résoudre concernent la formalisation des critères de test et l'analyse des documents d'entrée (spécification ou code source) pour engendrer des données constituant un jeu de test satisfaisant. Le bénéfice d'une telle systématisation est double : d'une part les jeux de test ainsi produits sont de meilleure qualité que ceux que peut inventer un testeur (et justifiés par rapport à un critère précis) ; d'autre part, la possibilité de mécaniser le procédé (au moins partiellement) apporte des gains significatifs en terme de productivité.

On distingue généralement quatre types de tests, chacun étant lié à l'une des phases de conception des logiciels. Les premiers tests soumis au logiciel ont pour cible les composants élémentaires de l'application à tester. Pour cette raison, ils sont appelés *tests unitaires* (on trouve aussi le terme *test de composant*). La seconde phase de test, les *tests d'intégration*, correspond à la phase d'intégration progressive des différents composants élémentaires qui ont déjà passé avec succès l'épreuve des tests unitaires. L'objectif est de mettre en évidence les dysfonctionnements engendrés par leur assemblage. Les *tests fonctionnels* sont ensuite exécutés

[XMd+94] S. Xantakis, M. Maurice, A. de Amescua, O. Houri, L. Griffet, *Test et contrôle des logiciels. Méthodes techniques et outils*, EC2, 1994.

sur l'application dont tous les composants ont été assemblés et intégrés. Le dernier type de test s'applique à la version complète de l'application déployée dans son environnement d'exécution. Ces tests, que l'on nomme *tests système*, consistent à détecter des fautes ou des comportements incorrects de l'ensemble du système en situation réelle. Les *tests de recette* sont des tests système.

Pour concevoir ces différents types de test, il existe un ensemble de techniques qui se décompose en deux familles [XMd+94]. La première famille réunit les techniques de test dites en *boîte noire* qui reposent sur une spécification (informelle, semi-formelle ou formelle) du programme. Le code du programme est considéré inaccessible et n'est pas utilisé pour sélectionner les données de test. Les tests produits sont dits *fonctionnels*.

La seconde famille est constituée des techniques de test dites en *boîte blanche* qui s'appuient exclusivement sur des analyses du code de l'application [Bei90]. Ces techniques reposent sur l'examen de la structure du programme et le calcul de flots de contrôle ou de données. Les tests produits sont dits *structurels*.

Le test comporte une grande variété de tâches qui comprend notamment la conception des jeux de test, leur instrumentation, leur exécution, le dépouillement des résultats et la sélection des tests de non-régression (en cas de modification des programmes). La plupart de ces étapes repose sur l'empirisme et l'aide fournie par les outils actuels reste insuffisante. Cependant, certaines de ces tâches peuvent être systématisées et même, dans une certaine mesure, mécanisées. La génération de jeux de test en fait partie et sa systématisation constitue l'objectif majeur des activités du projet Lande sur ce thème. Les principales difficultés à résoudre concernent la formalisation des critères de test et l'analyse des documents d'entrée (spécification ou code source) pour engendrer des données constituant un jeu de test satisfaisant. Le bénéfice d'une telle systématisation est double : d'une part les jeux de test ainsi produits sont de meilleure qualité que ceux que peut inventer un testeur (et justifiés par rapport à un critère précis) ; d'autre part, la possibilité de mécaniser le procédé (au moins partiellement) apporte des gains significatifs en terme de productivité.

3.5 Langages déclaratifs

Mots clés : langage fonctionnel, langage de programmation logique, correction, efficacité, évolutivité, maintenance.

Résumé :

Les langages de programmation déclaratifs sont fondés sur la déclaration du résultat à atteindre plutôt que du moyen de l'atteindre. Leur mise en œuvre exige un effort spécifique pour passer automatiquement d'une définition de nature déclarative à une version opérationnelle efficace. En contrepartie, ces langages sont adaptés à l'usage de méthodes formelles (analyse de programmes, vérification). Les langages déclaratifs étudiés dans le projet Lande appartiennent soit à la famille de la programmation fonctionnelle, soit à celle de la programmation logique.

[Bei90] B. Beizer, *Software testing techniques, 2nd ed.*, International Thomson Computer Press, 1990.

Les langages de programmation forment des familles qui incarnent des disciplines de programmation. La famille des langages de programmation déclaratifs comprend les langages qui sont fondés sur la déclaration du résultat à atteindre plutôt que du moyen de l'atteindre. La discipline mise en oeuvre dans ces langages consiste à s'engager le moins possible dans des détails opérationnels afin de diminuer le fossé entre ce que souhaite le programmeur et ce que le langage de programmation permet d'exprimer.

Le projet Lande s'intéresse à deux espèces de langages de programmation déclaratifs qui sont les langages fonctionnels (Lisp, ML, Haskell, etc.) et les langages logiques (Prolog, λProlog, Mercury, etc.). Une remarque importante à faire à leur sujet est que ces langages utilisent des formalismes qui ont présidé à la formalisation de la notion de calcul : le λ-calcul [Ros84] et le calcul des prédicats. Dans les deux cas, les programmes sont des *formules* mais elles sont interprétées différemment. L'opération essentielle des langages fonctionnels est la *réduction* qui permet de remplacer une formule par une autre formule équivalente, mais plus «simple», jusqu'à obtenir une formule qui n'est plus réductible, et que l'on appelle une *forme normale*. On convient que cette forme normale est le résultat du calcul. L'opération essentielle des langages logiques est la *déduction*. On l'emploie pour construire des preuves, et on convient que le résultat du calcul est extrait de ces preuves. Il s'agit le plus souvent des valeurs données dans les preuves à certaines variables. Pour autant que la correspondance de Curry-Howard s'applique (langages fonctionnels typés), la preuve est l'objet commun à ces deux familles de langages de programmation ; les langages fonctionnels les normalisent, et les langages logiques les construisent.

L'intérêt premier des langages de programmation déclaratifs est qu'ils se prêtent aux manipulations formelles. La raison majeure est l'absence d'*effets de bord* dans ces langages : les entités de base (fonctions ou prédicats) peuvent ainsi être manipulées directement comme des objets mathématiques.

Les enjeux des langages fonctionnels et logiques sont assez similaires. D'une part, il faut réussir à mettre en œuvre efficacement les calculs décrits dans ces langages. D'autre part, il faut concevoir les outils de programmation qui accompagnent ces langages.

Un autre formalisme déclaratif traité dans le projet Lande est celui des *bases de données déductives*. Il partage les mêmes fondements que la programmation logique mais à des fins différentes. Ici, l'enjeu est la description de grands volumes de données, des lois qui structurent ces données et des requêtes des utilisateurs. La complétude calculatoire n'est plus recherchée. Au contraire, on veut que le problème de répondre à une requête soit décidable.

4 Domaines d'applications

Mots clés : sécurité, sûreté, confidentialité, intégrité, logiciel critique, carte à puce, commerce électronique, génération de jeux de test, outil de débogage, architecture sécurisée.

Résumé : *Les deux cibles privilégiées du projet Lande sont:*

1. *Les applications qui exigent un degré de confiance très important justifiant*

[Ros84] J. Rosser, « Highlights of the History of the Lambda-Calculus », *Annals of the History of Computing 6*, 4, 1984.

l'emploi de méthodes formelles. L'accent est mis en particulier sur la sécurité des informations (confidentialité, intégrité).

2. *Les logiciels complexes ou qui nécessitent des modifications fréquentes : il s'agit du domaine d'excellence des langages de programmation logique.*

De par sa nature même, le projet Lande est orienté « technologie » plutôt que « domaine d'application». La plupart des domaines cités ici sont donc des illustrations de travaux passés ou en cours, plutôt que des centres d'intérêt propres du projet. Un exception peut être faite toutefois pour ce qui concerne la sécurité des systèmes d'information (au sens de confidentialité et d'intégrité notamment). Il s'agit d'un domaine d'application où l'exigence de méthodes formelles se fait fortement sentir et qui comporte des implications industrielles majeures, en particulier pour le développement du commerce électronique. Le projet Lande a investi depuis plusieurs années dans ce domaine qui prend une importance croissante dans l'ensemble de ses activités.

De manière générale, on peut identifier deux cibles privilégiées pour les outils formels et les langages de haut niveau qui sont étudiés dans le projet :

– La première concerne les applications complexes ou exigeant un degré de confiance très important justifiant l'emploi de méthodes formelles. Il peut s'agir de logiciels critiques pour la confidentialité ou l'intégrité des informations, la sécurité des personnes. Sur ce thème, nous travaillons (dans le cadre de Dyade et du projet européen secsafe) à l'analyse et la vérification de transformateurs de bytecodes Java Card, avec comme domaine d'application la sécurité des cartes à puce (cf modules 6.2.3, 7.3 et 7.4).

  Les langages d'architectures de logiciels constituent une démarche plus récente pour le développement de logiciels complexes. Sur ce thème, nous sommes impliqués dans une action industrielle (l'action CASTOR cf 7.5) qui porte sur la conception d'architectures sécurisées. Elle s'effectue en collaboration avec Matra SI, AQL, TNI et le projet EP-ATR de l'Irisa.

  On peut citer également dans cette catégorie nos travaux sur la génération automatique de jeux de test qui se poursuivent en collaboration avec l'éditeur d'outils de tests Attol Testware et des industriels utilisateurs comme Siemens et Spacebell (cf module 7.2).

– La seconde cible de nos travaux concerne les logiciels qui nécessitent des modifications fréquentes : on peut citer par exemple les systèmes qui mettent en œuvre des ensembles de règles (de facturation, de réservation, etc.) devant suivre les évolutions du marché ou de la législation. Il s'agit du domaine d'excellence des langages de programmation logique. La mise en œuvre du langage λProlog réalisée dans le projet a notamment été utilisée pour la programmation d'un module de recherche de composants logiciels, pour la reconnaissance de partitions musicales (cf module 5).

5 Logiciels

Résumé : *Le projet Lande réalise un effort de développement important dans les deux catégories citées dans la présentation générale de nos objectifs (« amont» et*

« aval »). Nous détaillons le prototype de générateur de suites de test Casting développé en collaboration avec AQL*, Coca et Morphine les débogueurs automatisés pour les langages* C *et Mercury, le solveur de points fixes Reqs ainsi que le compilateur de λProlog.*

Le projet Lande réalise un effort de développement important dans les deux catégories citées dans la présentation générale de nos objectifs (« amont » et « aval »). Nous détaillons ici le prototype de générateur de suites de test Casting développé en collaboration avec AQL, Coca et Morphine les débogueurs automatisés pour les langages C et Mercury ainsi que le compilateur de λProlog.

Générateur de suites de test Casting

Correspondant: Olivier Ridoux

La méthode de génération de suites de test définie dans la thèse de Lionel Van Aertryck a conduit à la réalisation d'un prototype en collaboration avec la société AQL. La méthode elle-même est décrite dans le module 6.2.9 ; nous nous focalisons ici sur sa mise en œuvre et les fonctionnalités du prototype.

Nous avons utilisé deux outils principaux pour l'implantation de Casting qui sont Precc[1] (un compilateur de compilateur permettant de calculer des attributs hérités et synthétisés) et le solveur de contraintes Ilog Solver[2]. La version actuelle comporte un seul frontal dédié au langage de spécification de la méthode B (cf. module 6.2.9) ; une version pour les langages de programmation C et C++ est en cours de développement (cf module 7.2). Le prototype est constitué d'une partie générique qui regroupe tous les traitements internes (élimination des spécifications de cas de test insatisfiables, génération du graphe d'états symboliques, génération des hypothèses de test et des suites de test, etc.) et une partie liée à la spécification pour laquelle le testeur souhaite engendrer des suites de test (frontal).

Avec ce prototype l'utilisateur a accès à un environnement de génération de suites de test qui lui permet de définir une stratégie de test et de l'appliquer à des spécifications écrites dans un sous-ensemble de B. L'utilisateur dispose de différents moyens de contraindre la recherche de solution (temps alloué au solveur, taux de couverture, etc.) ainsi que la possibilité d'interagir avec le solveur de contraintes de manière à l'orienter directement vers des solutions.

Le développement a été réalisé sous Solaris 2 et il intègre des codes C, C++ (algorithmes de recherche de chemins et de génération de données), λProlog (mise sous forme normale disjonctive) et tcl/tk (interface graphique). L'interface réalisée permet d'assister le testeur dans toutes les phases de la génération des suites de test (choix d'une stratégie, paramétrage et aide à la résolution, visualisation de la couverture obtenue, etc.).

Pour plus de renseignements concernant cet outil et son état d'avancement, on peut se reporter à l'adresse : `http://www.irisa.fr/lande/vanaertr/bcasting.html` ou contacter Olivier Ridoux (ridoux@irisa.fr).

Débogueurs Coca et Morphine

1. P.T Breuer et J.P. Bowen. *A PREttier Compiler-Compiler: Generating higher order parsers in C*, Technical Report PRG-TR-20-92, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, UK, novembre 1992.

2. Ilog Solver est une marque déposée de Ilog S.A.

Correspondant : Mireille Ducassé

Coca [Duc99] est un débogueur automatisé pour C où le mécanisme des points d'arrêt est basé sur des événements relatifs à des constructions du langage. Ces événements ont une sémantique, alors que les lignes du code source utilisées par la plupart des débogueurs n'en ont pas. Morphine [JD99] est un débogueur automatisé pour le langage de programmation en Logique Mercury[SHC96].

Une trace est une séquence d'événements. Elle peut être vue comme une relation d'ordre sur une base de données. Les utilisateurs de nos systèmes peuvent spécifier exactement les événements qu'ils veulent voir en précisant des valeurs pour les attributs des événements. À chaque événement, les variables visibles peuvent être examinées. Le langage d'interrogation de trace est Prolog augmenté de quelques primitives. Le mécanisme d'interrogation de trace cherche dans la trace d'exécution en utilisant à la fois des informations sur le flot de contrôle et sur les données alors que les débogueurs effectuent habituellement leur recherche de manière exclusive en fonction du contrôle ou en fonction des données. Contrairement aux débogueurs totalement relationnels qui utilisent effectivement une base de données, le mécanisme d'interrogation de trace de Coca et de Morphine repose sur une analyse à la volée : il ne nécessite donc aucun stockage. Coca et Morphine sont donc plus puissants que les débogueurs dont les points d'arrêt sont des lignes du code source et plus efficaces que les débogueurs relationnels.

Un prototype de Coca est opérationnel sous sparc/Solaris2.[5,6]. Il nécessite le compilateur GCC ainsi que le système Prolog Eclipse 4.1. Il intègre des codes C, C++, Prolog et Bison.

Un prototype de Morphine a été développé dans le cadre du projet industriel européen ARGO (*Ruggedized and High performance logic Programming for the Real World*, Industrial RTD Project no 25503, ref. Inria : 1 97 C 843), novembre 1997-juin 1999. Morphine est actuellement maintenu et opérationnel sous sparc/Solaris2.[5,6,7] et i686/Linux2.0, il nécessite le système Eclipse 4.1, ainsi que le compilateur Mercury (versions postérieures au 1999-03-12). Morphine a été déposé à l'agence pour la protection des logiciels sous la référence IDDN.FR.001.090030.00.S.P.2000.000.10600. Il fait partie de la distribution standard de Mercury (http://ww.cs.mu.oz.au/research/mercury). Le système Morphine, ainsi que sa documentation, sont également disponibles sur le web `http://www.irisa.fr/lande/jahier/download.html`.

Pour plus de renseignements concernant Coca et Morphine on peut contacter Mireille Ducassé (`mailto:ducasse@irisa.fr`).

Compilateur λProlog

Correspondant: Olivier Ridoux

Le compilateur de λProlog développé à l'Irisa représente un investissement de plusieurs an-

[Duc99] M. Ducassé, « Coca: An automated Debugger for C », *in : Proceedings of the 21st International Conference on Software Engineering*, ACM Press, p. 504–513, mai 1999. (également RR-3489 INRIA).

[JD99] E. Jahier, M. Ducassé, *An automated debugger for Mercury - Morphine 0.1 User and reference manuals*, mai 1999, RT-231 INRIA (également PI-1234 IRISA).

[SHC96] Z. Somogyi, F. Henderson, T. Conway, « The execution algorithm of Mercury, an efficient purely declarative logic programming language », *Journal of logic Programming 29*, October-December 1996, p. 17–64, `http://www.cs.mu.oz.au/research/mercury/papers.html`.

nées (à l'origine dans le projet Mali). Son schéma est fondé sur un modèle à continuations [BR93] et sur la mémoire Mali [Rid91]. Ce système, appelé Prolog/Mali, implémente le langage λProlog complet, plus des facilités comme l'ordonnancement dynamique des buts (*freeze*), les captures de continuations (d'échec et de succès), et l'appel de procédures C depuis λProlog (et vice-versa). Il constitue un système flexible qui permet la coopération de modules écrits en λProlog et en d'autres langages. Ce trait est couramment employé dans les applications un tant soit peu complexes. Le système comporte aussi un traceur symbolique et un profileur. Ce système a été développé sous Solaris 1 (SunOs 4) puis porté sous Solaris 2 (SunOs 5). Il est disponible sous FTP (ftp://ftp.irisa.fr/local/pm). Les logiciels Mali et Prolog/Mali ont été déposés à l'APP (numéros 87–12–005–01 et 92–27–012–00) et sont munis d'une documentation.

Le compilateur λProlog est employé en enseignement, dans des applications de projets de l'Irisa, et dans d'autres laboratoires plus ou moins distants. Parmi les applications les plus notables, on peut citer: la reconnaissance de partitions d'orchestre (Irisa, projet Imadoc), la coopération entre agents intelligents (SEPT, Caen), la recherche de composants systèmes (Irisa, projet Solidor), la transformation de grammaires attribuées (Irisa, projet Lande) et le compilateur Prolog/Mali, lui-même écrit en Prolog/Mali et en C et C/Motif. Une équipe de l'université d'Édimbourg l'utilise pour le développement d'un démonstrateur automatique pour une logique d'ordre supérieur.

Pour tout renseignement concernant Mali ou Prolog/Mali, le point de contact à l'Irisa est Olivier Ridoux (ridoux@irisa.fr, voir aussi [Rid98,BBR]).

Solveur de point fixes Reqs.

Correspondant : Florimond Ployette.

Reqs est un outil de résolution de système d'équations récursives produites dans le cadre d'analyses statiques de programmes (cf module 6.2.1). L'objectif de ce travail est de factoriser le travail de résolution de point fixe pour des analyses de programmes impératifs, logiques ou fonctionnels. L'outil propose un ensemble de stratégies permettant d'adapter au mieux la résolution à la nature du système (dense ou non, point fixe local, etc...). Pour résoudre un système d'équations sur un domaine donné, l'utilisateur doit, soit fournir une implémentation des opérations utilisées dans les équations, soit utiliser les domaines et opérations prédéfinis dans REQS.

Nous avons réalisé plusieurs expériences avec REQS pour implémenter des analyses provenant de différents types de langage. Dans le domaine de langages à objets, reqs a été utilisé pour implémenter une analyse de flot de contrôle (pour la vérification de propriétés de sécurité) ainsi qu'une analyse de forme pour Java. Il a également servi pour réaliser un analyseur du langage SIGNAL (cf module 6.2.1).

[BR93] P. Brisset, O. Ridoux, « Continuations in λProlog », *in : 10th Int. Conf. Logic Programming*, D. Warren (éditeur), MIT Press, p. 27–43, 1993.

[Rid91] O. Ridoux, «MALIv06: Tutorial and Reference Manual», *Publication Interne n° 611*, Irisa, 1991, ftp://ftp.irisa.fr/local/lande/or-tr-irisa611-91.ps.Z.

[Rid98] O. Ridoux, *λProlog de A à Z, ... ou presque*, document d'habilitation à diriger des recherches, Université de Rennes 1, avril 1998.

[BBR] C. Belleannée, P. Brisset, O. Ridoux, « A pragmatic reconstruction of λProlog », *Journal of Logic Programming, 41(1), 1999.*, Version française dans TSI 14(9):1131–1164:1995.

Pour des renseignements supplémentaires concernant l'outil REQS on peut contacter Florimond Ployette (`ployette@irisa.fr`). ou consulter le web (`http://www.irisa.fr/lande/REQS/presentation_reqs.html`).

6 Résultats nouveaux

Nous utilisons la dichotomie amont/aval pour présenter nos résultats récents : la partie « amont » traite des architectures logicielles, de la programmation par aspects et des langages dédiés ; la partie « aval » regroupe nos travaux sur l'analyse (dynamique ou statique) de programmes et le test de logiciels.

6.1 Actions « amont »

Nous décrivons dans cette partie les contributions du projet qui sont de nature linguistique. Dans chaque cas, il s'agit de proposer un formalisme ou un langage spécialisé adapté à un type de problème et conduisant à des traitements automatiques (vérification, analyse, transformation, etc.). Nous abordons successivement nos travaux sur les architectures de logiciels (module 6.1.1), les agents mobiles (module 6.1.2), la programmation par aspects (module 6.1.3), un langage dédié pour le parallélisme (module 6.1.4), et un cadre pour la définition de systèmes d'information logiques (module 6.1.5).

6.1.1 Architectures de logiciels

Participants : Pascal Fradet, Michaël Périn, Lakshminarayanan Renganarayanan.

Mots clés : architectures de logiciels, vues, relations inter-vues, cohérence, vérification, UML, graphe.

Résumé : *Nous avons proposé un cadre permettant de décrire des familles d'architectures de logiciels sous forme d'ensemble de vues. Ces vues sont définies formellement comme des classes de graphes. Des algorithmes de vérification ont été proposés pour détecter les incohérences de spécification (aussi bien intra-vues qu'inter-vues).*

L'intérêt d'une définition précise de l'architecture d'un logiciel est reconnu de longue date mais les architectures de logiciels étaient jusqu'à présent utilisées de manière informelle par les développeurs (souvent sous la forme de dessins décrivant des visions d'ensemble du logiciel). Une définition formelle est un premier pas vers un traitement rigoureux et systématique des architectures ; elle peut aussi servir à améliorer la communication entre les différents acteurs impliqués dans un développement (en fournissant des définitions précises et sans ambiguïté).

Par ailleurs, il s'avère que les propriétés que l'on souhaite vérifier dans ce contexte sont de natures très diverses et que chacune d'elle peut demander une présentation différente de l'organisation du logiciel. La définition d'une architecture comme un ensemble de *vues* complémentaires s'impose donc mais il est alors nécessaire de traiter les délicats problèmes de mise en relation des vues et de la cohérence de ces vues multiples.

Nous avons abordé cette question en décrivant les vues par des diagrammes avec multiplicités sur les arcs inspirés des notations UML [32, 15]. Les diagrammes répondent à un besoin de généricité et d'abstraction que l'on retrouve dans les descriptions informelles utilisées en pratique par les concepteurs. En particulier, un diagramme permet de spécifier une famille potentiellement infinie d'architectures. Ce formalisme permet de représenter des objets sémantiques très variés et les différents types de vues que nous avons considérés jusqu'à présent se décrivent naturellement de cette manière.

Il est rare toutefois que les vues soient complètement indépendantes ; il est même possible que deux vues répondent à des exigences conflictuelles. Dans notre formalisme, les relations entre vues sont représentées par des arcs supplémentaires (dits de correspondance) entre les nœuds des différentes vues. Les conditions de cohérence entre vues sont définies dans un langage de contraintes à la OCL (*Object Constraint Language* de UML) qui permet d'exprimer des propriétés de chemins dans les graphes. Un algorithme de vérification permet d'assurer les conditions de cohérence sur les familles de graphes décrites par les diagrammes des différentes vues [32, 15].

Le fait de travailler sur des diagrammes spécifiant des familles d'architectures complexifie la vérification de cohérence. Pour obtenir un algorithme complet, nous avons dû nous restreindre à un sous-ensemble de propriétés de cohérence structurelles (typiquement, l'existence de chemins entre composants). Nous étudions maintenant les problèmes de relations inter-vues et de cohérence dans le cas où chaque vue est spécifiée par un unique graphe. Les descriptions deviennent certes moins génériques mais devraient nous permettre de décrire des relations inter-vues plus complexes et de vérifier des propriétés de cohérence sémantique.

Nos travaux sur les architectures de logiciels se sont appuyés sur des études de cas (système de contrôle de trains proposé par la société néerlandaise Signaal, système de contrôle d'accès). Ils se concrétisent dans le cadre d'une action industrielle conduite en collaboration avec le projet EP-ATR, Sycomore Aérospatiale Matra, AQL et TNI. Commandité par le Celar, ce projet porte sur la conception d'un outil d'aide à la mise au point d'architectures sécurisées.

6.1.2 Analyse de propriétés d'agents mobiles

Participants : Pascal Fradet, Siegfried Rouvrais.

Mots clés : architectures de logiciel, interactions, agent mobile, propriétés non fonctionnelles, sécurité, performance.

Résumé : *Nous étudions l'analyse de propriétés d'interactions exprimées comme des compositions d'agents mobiles, de* RPC *(Remote Procedure Calls), ou d'évaluations à distance. Les propriétés analysées sont la performance (en terme de volume de données échangées) et la sécurité (confidentialité et intégrité). Ce type d'information permet de guider le concepteur de systèmes distribués dans ses choix d'implantation.*

Les agents mobiles ont été récemment proposés comme une nouvelle forme d'interaction des systèmes distribués. Une des raisons qui rendent les agents difficiles à utiliser est que leurs

avantages ou inconvénients, comparés aux interactions classiques comme les RPC, portent sur des aspects non fonctionnels (comme les performances). Les propriétés non fonctionnelles sont souvent difficiles à appréhender et choisir la bonne combinaison d'interactions pour implanter un service complexe est une tâche délicate.

Nous avons travaillé à cette question en analysant et en comparant les différents styles d'interaction selon deux types de propriétés : les performances et la sécurité. Nous avons proposé un cadre linguistique pour spécifier et implanter les services complexes. Les agents mobiles, les RPC, l'évaluation à distance ou toute combinaison de ces protocoles sont représentés comme des expressions fonctionnelles. Ces expressions peuvent alors être analysées et comparées simplement. Pour les performances, nous avons pris comme critère le volume de données échangées sur le réseau. Pour la sécurité, nous nous sommes focalisés sur les propriétés de confidentialité et d'intégrité. Ces analyses permettent de guider le concepteur de systèmes distribués dans ses choix de protocoles [25].

Nous poursuivons actuellement ce travail dans deux directions :

- la prise en compte de propriétés de tolérance aux fautes,
- l'intégration de ce cadre à un environnement de développement basé sur un langage de description d'architectures de logiciels.

Ce travail est effectué en collaboration avec Valérie Issarny du projet Solidor.

6.1.3 Programmation par aspects

Participants : Thomas Colcombet, Pascal Fradet.

Mots clés : aspect, analyse et transformation de programme, politique de sécurité, construction de programme.

Résumé : *La programmation par aspects propose de décrire un logiciel comme un ensemble formé d'un composant principal et d'une collection d'aspects. Un outil, appelé tisseur, est chargé de produire automatiquement un programme intégrant les différents aspects au composant principal. Nous avons appliqué ce paradigme à la sécurité. La politique de sécurité (l'aspect) est décrite séparément du programme par une propriété temporelle sur les traces d'exécution. Le tisseur impose automatiquement cette propriété au composant par transformation de programme. Une application particulièrement intéressante de cette technique est la sécurisation d'applets au moment de leur réception.*

La notion d'«aspect» présente des similarités avec celle de «vue» présentée dans le module précédent dans le contexte des architectures logicielles. En programmation par aspects, un logiciel est formé d'un composant principal et d'une collection d'aspects décrivant des tâches comme la gestion mémoire, la synchronisation, les optimisations, etc. Un outil, appelé tisseur, est chargé de produire automatiquement un programme intégrant les différents aspects au composant principal. L'intérêt de cette approche est de localiser (dans les aspects) des choix de mise en œuvre qui seraient sinon dispersés dans le code source.

Nous avons proposé une méthode automatique, inspirée de la programmation par aspects, pour imposer des propriétés exprimées sur les traces d'exécution des programmes [20, 12]. Le programmeur spécifie la propriété séparément de son programme comme un aspect et le tisseur produit un programme « équivalent » respectant la propriété, c'est-à-dire un programme se comportant comme le programme original mais s'arrêtant en produisant une erreur avant toute tentative de violation de la propriété. Le défi est d'imposer la propriété de la façon la moins coûteuse possible. On minimise à l'aide d'analyses statiques le nombre de tests dynamiques nécessaires pour prévenir les violations.

Le domaine d'application évident est celui de la sécurité. En effet, de nombreuses politiques de sécurité peuvent s'exprimer comme des propriétés sur les traces d'exécution. En particulier, une application potentielle est la sécurisation d'applets au moment de leur réception. Cette méthode a l'avantage de la flexibilité puisque chaque site peut imposer à la volée sa propre politique de sécurité aux applets. La séparation du programme et de la propriété conduit également à des programmes plus faciles à développer et à maintenir. Ceci est particulièrement important dans le domaine de la sécurité. Il est difficile de prévoir toutes les attaques possibles et les programmes doivent pouvoir être changés rapidement pour faire face aux nouvelles menaces identifiées. De plus, considérer les aspects comme des propriétés permet de décrire et de contrôler précisément l'impact sémantique du tissage. C'est une autre caractéristique importante dans le contexte de la sécurité.

Le cœur de l'approche a été mis en œuvre en O'Caml. Des heuristiques simples ont été utilisées pour les analyses potentiellement coûteuses et la complexité globale du processus est linéaire en fonction de la taille du programme.

La programmation par aspects a clairement d'autres applications pour la construction de programmes. La séparation des problèmes offerte par les aspects permet d'espérer pouvoir concilier programmes de haut niveau et efficacité. Nous étudions actuellement dans cet esprit un aspect de représentation de données et, en collaboration avec Mario Südolht et Rémi Douence de l'école de Mines de Nantes, un aspect d'assemblage de composants.

6.1.4 Langage dédié pour machines parallèles

Participants : Pascal Fradet, Julien Mallet.

Mots clés : langage dédié, parallélisme, schéma de programme, compilation, analyse de coût, distribution de données.

Résumé : *Nous avons proposé un langage spécialisé pour machines parallèles. Les restrictions du langage source permettent de choisir automatiquement la meilleure distribution de données parmi un ensemble de distributions standards. Ce choix repose sur une analyse de coût exacte et symbolique prenant en compte les temps de calcul et les temps de communication.*

Il n'existe pas de langage idéal pour programmer les machines parallèles : on trouve d'une part des langages à parallélisme explicite efficaces mais non portables et très complexes à utiliser ; d'autre part des langages simples et portables ont été proposés mais leur compilation

est complexe et relativement inefficace. La démarche que nous avons adoptée est celle d'un langage spécialisé permettant une estimation *exacte* et *symbolique* du coût de l'implantation parallèle. Notre langage source est un langage fonctionnel du premier ordre où la récursivité générale et les conditionnelles sont remplacées par des collections de schémas de programmes (patrons). Le résultat de l'analyse de coût, qui prend en compte les temps de calcul et les temps de communication, permet de choisir la meilleure distribution de données parmi un ensemble de distributions standards [17, 12, 35].

6.1.5 Systèmes d'information logiques

Participants : Sébastien Ferré, Yoann Padioleau, Olivier Ridoux.

Mots clés : système d'information, analyse de concept, logique.

Résumé : *Nous étudions la conception de systèmes d'information non-hiérarchiques offrant à la fois des possibilités de navigation et d'interrogation. Nous avons développé pour cela un modèle d'analyse de concepts logiques qui généralise l'analyse de concepts formels de Wille et Ganter* [GW99].

L'analyse de concepts logiques (ACL [33, 24]) part d'un *contexte logique* dans lequel des objets sont étiquetés par des formules logiques et produit un treillis de concepts dits logiques. La logique employée à cet effet peut être presque quelconque mais dans la pratique on exigera qu'elle soit décidable. Des fragments de logique classique (ou intuitionniste), les logiques de descriptions, ou même une logique d'ensembles d'attributs où la relation de contenance sert de relation de déduction, constituent des exemples de logiques qu'on peut employer pour étiqueter les objets. L'instance de l'ACL qui utilise la logique des attributs est tout simplement l'analyse de concepts formels (ACF). Les concepts sont des paires (E, I) (pour extension-intention) dont les composantes sont mutuellement maximales. En d'autres termes, E est le plus grand ensemble d'objets dont les étiquettes (qui sont des formules) impliquent I, et I est la formule la plus générale qui implique les étiquettes de E. E et I sont liés par une *connection de Galois*.

Ce modèle est très général et peut être instancié par le choix d'une logique et d'une mise en œuvre. Traditionnellement, l'ACF est mise en œuvre dans des systèmes d'analyse a posteriori de faits bruts constituant le contexte formel. On peut mettre en œuvre l'ACL de cette façon, mais nous préférons étudier son emploi dans des systèmes plus dynamiques où l'ACL est considérée comme un modèle d'organisation. Par exemple, munie d'une logique permettant de décrire les différentes vues d'une architecture logicielle, l'ACL devient un environnement de développement de logiciels qui intègre la navigation dans l'architecture. D'où l'idée de la mettre en œuvre dans un système de fichiers qu'on appellera «système de fichiers logique» [34, 23].

Nous étudions actuellement l'extension de ce modèle pour la prise en compte de relations entre objets, une métaphore de navigation dans le treillis de concepts logiques, et les algorithmes de consultation, navigation et mise à jour. Sur ce dernier point, l'enjeu principal est de ne jamais construire entièrement le treillis de concepts logiques. La métaphore de navigation envisagée possède les caractéristiques suivantes : une formule désigne un endroit où l'on

[GW99] B. Ganter, R. Wille, *Formal Concept Analysis: Mathematical Foundations*, Springer, 1999.

souhaite lire *ou* écrire, l'interrogation du contenu d'un endroit retourne entre autres la désignation d'endroits proches par des formules (combinant ainsi *interrogation* et *navigation*), et la manipulation des formules se fait aussi bien en prenant en compte leur intension que leur extension. Ce dernier point permet de faire des déductions qui sont correctes dans le contexte courant, mais pas en général.

Comme exemple de mise en œuvre, nous avons réalisé un système de fichiers (sous Linux) qui offre les fonctions de l'analyse de concepts formels sous l'interface standard d'un système de fichiers virtuel. Dans cette réalisation, seul le système de fichiers est spécifique, et les couches supérieures comme le *shell* restent inchangées. Les commandes et les applications changent de sémantique de la façon suivante : *cd d* revient à se focaliser sur l'attribut *d*, *ls* liste les fichiers qui possèdent les attributs courants avec, le cas échéant, leurs attributs supplémentaires, *ls -r* liste les fichiers qui possèdent au moins les attributs courants, *mv d1 d2* remplace l'attribut *d1* par l'attribut *d2* dans tous les fichiers accessibles, etc. Nous recherchons maintenant comment mettre en oeuvre un système de fichiers basé sur l'analyse de concepts logiques de façon la plus générique possible.

6.1.6 Transformations certifiées de programmes Java Card

Participants : Thomas Jensen, Bjarke Ebert.

Mots clés : transformation, optimisation, code intermédiaire, carte à puce, Java Card, preuve formelle, Coq.

Résumé : *Nous avons conçu et breveté une méthode qui permet de prouver la correction des transformations effectuées sur le code intermédiaire Java Card pour permettre son installation sur une carte à puce. Cette preuve a été vérifiée avec l'assistant de preuve Coq. La formalisation de ces transformations forme la base de travaux qui visent à construire un transformateur Java Card certifié en Coq.*

Notre effort de formalisation des différents aspects de Java et de sa mise en œuvre a entre autre porté sur la vérification d'optimisations pour Java Card. La définition du langage Java Card comprend un langage intermédiaire (le «Java Card byte code») et un format (le format CAP pour «Converted APplet format») utilisé pour stocker des applications sur une carte. Le format Cap joue un rôle semblable à celui du format des fichiers de classes («class files») de Java mais regroupe le code des classes de tout un package de Java contrairement aux fichiers de classes de Java (qui ne représentent qu'une seule classe par fichier). Ceci permet de remplacer des références symboliques entre classes à travers le «constant pool» par des adresses de mémoire, ce qui est plus rapide et permet de supprimer des entrées dans le «constant pool». Une autre optimisation, nécessaire puisqu'il n'y a pas de chaînes de caractères en Java Card, s'appelle la «tokenization» : elle consiste à remplacer les noms de méthodes, champs, *etc.* par un numéro (un «token»). Ceci permet entre autres d'utiliser le nom (c'est à dire le «token») pour indexer une table de méthodes au lieu d'effectuer une recherche à travers le «constant pool» et la hiérarchie de classes pour implémenter l'appel de méthodes virtuelles.

Pour prouver la correction de ces optimisations, nous avons développé un cadre qui permet de décrire la sémantique des deux formats [21]. La différence entre les deux formats est

encapsulée dans des fonctions auxiliaires, ce qui permet d'utiliser le même système d'inférence pour spécifier les deux sémantiques. En utilisant la notion de relation logique, il a été possible de définir une relation liant une entité dans un format à l'entité lui correspondant dans l'autre format. La tâche restant à accomplir pour établir l'équivalence consiste alors à montrer que lesdites fonctions auxiliaires respectent ces relations ce qui représente une simplification majeure de la preuve. Cette technique a permis d'effectuer la preuve de correction avec l'assistance de l'outil Coq.

Nous travaillons maintenant sur la construction d'un transformateur Java Card certifié en Coq. La formalisation de ces transformations forme la base d'une preuve constructive que pour chaque programme Java Card il existe un programme équivalent en format CAP. Notre approche utilise les techniques d'extraction de programmes de l'outil Coq pour extraire un transformateur à partir de cette preuve constructive.

6.2 Actions « aval »

Nous présentons d'abord des résultats généraux sur l'analyse de programmes (module 6.2.1) avant de détailler un certain nombre de recherches portant sur des analyses particulières : la vérification de politiques de sécurité (module 6.2.3), la preuve de protocoles cryptographiques à l'aide de l'interprétation abstraite (module 6.2.2) et l'analyse de programmes λProlog (module 6.2.4). Nous décrivons ensuite nos travaux récents sur le débogage dynamique (module 6.2.5), la sémantique et la génération de traces (modules 6.2.6 et 6.2.7) et leur application au problème de la détection d'intrusion. Nous concluons avec la présentation de nos activités en matière de génération de jeux de test (module 6.2.9).

6.2.1 Construction systématique d'analyses statiques

Participants : Thomas Jensen, Florimond Ployette, Frédéric Besson.

Mots clés : analyse de programme, sémantique, interprétation abstraite, calcul de point fixe.

Résumé : *Une analyse statique se divise souvent en deux phases. La première phase produit à partir d'un programme un système de contraintes dont la solution représente l'information recherchée. Une deuxième phase calcule la solution. La phase de résolution est indépendante du programme analysé. Une méthode de résolution de systèmes d'équations peut donc s'appliquer à des systèmes provenant de différentes analyses ce qui suggère la possibilité de réaliser un « moteur d'analyse ».*

De nombreux travaux ont été effectués sur les fondements de l'analyse sémantique, la correction et la précision des analyseurs. On peut regretter cependant le peu d'attention accordé jusqu'à présent à la conception d'outils d'analyse génériques. Nous travaillons à la mise au point d'un solveur de point fixe générique qui peut servir de base pour le développement de nouveaux analyseurs (cf module 5). Une analyse produit souvent comme résultat un système d'(in)équations sur un domaine de propriétés dont la solution représente l'information recherchée. La phase de résolution est indépendante du programme analysé ; une méthode de

résolution de systèmes d'équations peut donc s'appliquer à des systèmes provenant de différentes analyses. Cette observation suggère la possibilité de réaliser un « moteur d'analyse », c'est à dire un outil générique de résolution de systèmes d'(in)équations qui peut servir de base pour implanter des analyseurs. La généricité évoquée ici est relative aux propriétés et aux domaines abstraits considérés. La solution d'un tel système peut être calculée de façon itérative comme le plus petit point fixe d'un opérateur défini par le système d'équations à résoudre.

Nous avons appliqué cette technique à la vérification de propriétés de sûreté de programmes synchrones écrits dans le langage SIGNAL. En collaboration avec J.-P. Talpin du projet EPATR, nous avons conçu une analyse de flot de données de programmes Signal pour déterminer des relations entre les variables d'un programme [BJT99]. L'analyse prend en entrée un programme SIGNAL et rend un système de contraintes dont les solutions décrivent l'exécution du programme. La résolution de ce système de contraintes se fait par itération de point fixe dans le domaine des polyèdres convexes. Ce domaine contenant des chaînes infinies, nous avons été amenés à définir des opérateurs d'élargissement pour garantir la convergence du calcul de point fixe. Ces opérateurs se basent sur une représentation de polyèdres différente de celle utilisée dans la définition des opérateurs existants et permettent de faire des approximations plus fines dans certains cas ; ils complètent donc la collection d'élargissements dont on dispose pour faire converger un calcul de point fixe. Pour étudier la faisabilité de ce modèle biphasé de la conception d'analyseurs, nous avons réalisé l'analyseur SIGNAL en utilisant l'environnement de programmation SmartTool (cf module 8.1) pour générer le système de contraintes et le solveur Reqs (cf module 5) pour le résoudre.

6.2.2 Preuve et interprétation abstraite pour la vérification de protocoles cryptographiques

Participants : Valérie Viet Triem Tong, Thomas Genet.

Mots clés : preuve assistée, interprétation abstraite, vérification, protocoles cryptographiques.

Résumé : *Nous nous intéressons à l'utilisation des interprétations abstraites pour la simplification de certaines preuves réalisées à l'aide de démonstrateurs ou d'assistants de preuve. En simplifiant ces preuves, nous souhaitons améliorer l'automatisation de ces outils et ainsi faciliter leur utilisation pour la vérification de systèmes ou de logiciels ayant un niveau de complexité intermédiaire. Parmi les cibles privilégiées de ce type de vérification on trouve les protocoles cryptographiques, dont les propriétés essentielles peuvent être partiellement vérifiées par interprétation abstraite, mais dont la vérification complète nécessite des mécanismes de preuves plus puissants comme, par exemple, l'induction.*

Qu'il s'agisse de connexion à une machine distante, d'envoi d'informations confidentielles ou de transaction bancaire, toutes ces situations nécessitent l'utilisation de *protocoles crypto-*

[BJT99] F. Besson, T. Jensen, J.-P. Talpin, « Polyhedral analysis for synchronous languages », *in: Proc. of 7th Int. Symp. on Static Analysis*, G. Filé (éditeur), Springer LNCS vol. 1694, 1999.

graphiques. Par des techniques de cryptage, ces protocoles sont chargés d'assurer la sécurité des informations échangées : essentiellement la confidentialité et l'authenticité.

Il existe de nombreux protocoles cryptographiques mais la fiabilité de beaucoup d'entre-eux a été invalidée par la découverte de failles de sécurité importantes. De telles failles peuvent être exploitées par un acteur malveillant pour *attaquer* le protocole et, par exemple, obtenir une information qui ne lui était pas destinée (annule la confidentialité) ou se faire passer pour l'auteur d'un message qu'il n'a pas écrit (annule l'authenticité). L'attaque consiste, en général, en une séquence de messages contrefaits ou réutilisant des messages capturés.

La vérification des protocoles cryptographiques nécessite des techniques d'analyse particulièrement fines et sophistiquées. En effet, à cause des enjeux notamment dans le secteur bancaire, la sécurité de ces protocoles doit être garantie contre tout type d'attaque et ceci dans un contexte d'utilisation très général : pas de limites sur le nombre de sessions ou sur le nombre d'acteurs exécutant le protocole en même temps. Actuellement, la principale technique de vérification des protocoles cryptographiques – le model-checking – permet de découvrir certaines attaques, mais ne permet pas de garantir un protocole contre leur existence. D'autres techniques, faisant appel à la *démonstration automatique*, donnent une garantie beaucoup plus forte en démontrant formellement les propriétés de sécurité d'un protocole dans un contexte d'utilisation général. Cependant, les preuves demeurent fastidieuses et deviennent impraticables sur des protocoles complexes.

Nous nous proposons de concilier la démonstration automatique, qui permet une vérification profonde et précise des protocoles cryptographiques, avec des techniques d'*interprétation abstraite* afin de simplifier certaines preuves. En effet, alors que certaines propriétés nécessitent une technique de démonstration puissante, d'autres peuvent être démontrées en approchant, par interprétation abstraite, le problème et la propriété à prouver. En ne considérant qu'une approximation du problème et de la propriété, la démonstration peut, bien souvent, être effectuée à l'aide de mécanismes de déduction basiques. De plus, il est important d'intégrer la technique d'interprétation abstraite au démonstrateur car il doit être en mesure de s'appuyer sur les approximations pour mener à bien ses preuves. En particulier, dans une même preuve, le démonstrateur doit pouvoir combiner un raisonnement par approximation avec un raisonnement logique ou inductif.

En utilisant des techniques de réécriture et d'automates d'arbres, nous avons proposé une méthode de vérification de protocoles cryptographiques [26] utilisant une forme d'interprétation abstraite et un mécanisme de déduction simple sur des contraintes ensemblistes. A partir d'une description d'un protocole sous la forme d'un système de réécriture et d'un ensemble de requêtes possibles décrit par un automate d'arbre, nous calculons automatiquement une sur-approximation du comportement du protocole sur les requêtes possibles. Cette approximation est obtenue sous la forme d'un automate d'arbre reconnaissant un sur-ensemble des messages pouvant être échangés entre un nombre quelconque d'acteurs, pendant un nombre quelconque de sessions. A partir de ce sur-ensemble, pour montrer que le protocole a les propriétés souhaitées, il est suffisant de montrer que l'intersection entre l'ensemble approximation et un ensemble contenant tous les cas d'erreurs est vide. Ce travail a été réalisé en collaboration avec Francis Klay de France Telecom R&D.

Actuellement, nous étudions la combinaison de certaines formes d'interprétation abstraite avec des mécanismes de déduction plus sophistiqués afin de les intégrer dans des démonstrateurs

à base de réécriture comme CiME ou dans des assistants de preuve comme Coq, Isabelle/HOL ou PVS. A terme, ceci doit permettre de vérifier des propriétés plus fines sur des protocoles cryptographiques plus complexes comme le protocole SET de VISA et MASTERCARD.

6.2.3 Analyse de propriétés de sécurité

Participants : Frédéric Besson, Marc Eluard, Thomas Jensen, Fausto Spoto.

Mots clés : téléchargement, sécurité, chargement dynamique, Java.

Résumé : *Nous avons proposé un cadre de définition de politiques de sécurité reposant sur une logique temporelle linéaire à deux niveaux. Nous avons montré que ce modèle est suffisamment expressif pour décrire une variété de politiques de sécurité communes et nous avons proposé une méthode de « model checking » automatique pour la vérification de propriétés de cette logique. Ensuite nous avons montré comment la politique de sécurité du dernier environnement de développement de Java (JDK 1.2) peut s'exprimer dans notre logique.*

La sécurité d'un système informatique dépend en général d'un ensemble de contrôles de nature très variée (vérification formelle, analyse statique, analyse dynamique, gestionnaire de sécurité, protocole d'authentification, contrôles d'accès, etc.). Une difficulté majeure dans ce domaine est de pouvoir déterminer la politique globale de sécurité qui est assurée par cette association de contrôles spécifiques. L'évolution récente vers des langages de programmation qui offrent des fonctions de sécurité propres (comme Java ou Telescript) permet d'espérer des progrès en matière de vérification formelle de politiques de sécurité. Beaucoup reste cependant à accomplir comme le montrent les discussions récurrentes sur la sécurité des différentes versions d'environnements de Java. Dans ce contexte, on peut décomposer le problème en trois tâches complémentaires :

- La définition formelle de la sémantique du langage de programmation $\mathcal{L}$ (avec ses fonctions de sécurité).
- La spécification formelle de la politique de sécurité $\mathcal{P}$ qui doit être assurée.
- La vérification que la politique $\mathcal{P}$ est effectivement assurée par un système donné (décrit dans le langage $\mathcal{L}$).

Nous avons étudié ces trois aspects en fournissant un cadre général de spécification de politiques de sécurité et en décrivant son instanciation au langage Java et à son environnement de développement JDK 1.2 [7, 14].

Notre cadre de définition de politiques de sécurité repose sur un modèle abstrait de programmes comportant des opérations génériques de contrôle de sécurité. Leur sémantique opérationnelle est définie comme un système de transitions sur des états constitués de piles de contrôle. Les politiques de sécurité sont exprimées dans une logique temporelle linéaire à deux niveaux (les objets étant définis par des suites de piles). Nous avons montré que ce formalisme est suffisamment expressif pour décrire une variété de politiques de sécurité communes

(séparation des devoirs ou « segregation of duty », protection de ressources, bac à sable) avec une attention particulière à la politique de sécurité du dernier environnement de développement de Java (JDK 1.2) et des propriétés de sécurité d'applications pour les cartes à puce multi-applicatives programmées en Java Card.

Dans une communication précédente, nous avons proposé une méthode automatique (et complète) de vérification de propriétés pour cette logique [7]. Cette méthode de vérification était soumise à certaines restrictions (concernant notamment la récursivité mutuelle). Ces restrictions ont pu être levées grâce à une nouvelle technique de « model checking » basée sur une représentation de formules temporelles par des automates finis. Nous étudions maintenant comment modulariser cette méthode de vérification afin de mieux analyser des applications utilisant le chargement dynamique de classes en Java.

Le langage Java Card définit un modèle de sécurité légèrement différent du modèle de Java. Par défaut, les applettes installées sur une carte Java Card sont séparées par un pare-feu et ne peuvent communiquer entre elles. La communication entre applettes se fait par la création et les échanges des objets partagés. Nous nous sommes attaqués au problème de vérifier que les accès aux données rendus possibles par des objets partagés ne divulguent pas des informations confidentielles. Pour calculer une approximation sûre des accès effectués pendant l'exécution, nous nous appuyons sur des analyses statiques de flot de données et de contrôle. Ces analyses sont des analyses de byte code Java, adaptées pour prendre en compte les spécificités du pare-feu Java Card. Ces travaux se sont déroulés dans le cadre de l'action VIP du GIE Bull-Inria Dyade et vont continuer dans le projet européen IST SECSAFE (cf module 7.4).

6.2.4 Analyse statique de programmes λProlog

Participant : Olivier Ridoux.

Mots clés : analyse statique, lambda-Prolog, type.

Résumé : *Nous étudions l'analyse statique de λProlog par la technique de la* compilation abstraite *où un programme source est traduit en un autre programme dont les résultats sont interprétés comme le résultat de l'analyse du programme source. L'extension à λProlog des techniques éprouvées dans le cas de Prolog nécessite entre autres le traitement des λ-termes simplement typés.*

Nous avons choisi pour l'analyse statique de λProlog la technique de la *compilation abstraite* où un programme source est traduit en un autre programme dont les résultats sont interprétés comme le résultat de l'analyse du programme source. Cette méthode a déjà été employée pour Prolog et l'appliquer à λProlog nécessite des extensions dans trois directions :

1. La prise en compte de la structure des formules de λProlog (formules de Harrop au lieu de formules de Horn).

2. Le traitement de la structure des termes de λProlog (λ-termes simplement typés au lieu de termes de premier ordre).

3. L'application à l'analyse de propriétés nouvelles dont l'utilisation permettrait d'optimiser l'implantation du langage (état de β-normalisation, combinateurs, etc.).

Le domaine de calcul choisi pour les programmes abstraits est celui des fonctions booléennes. Dans ce cadre, nous avons développé une généralisation du domaine des fonctions booléennes positives à des fonctions sur les types, elles-mêmes positives en un sens qu'il a fallu redéfinir. Ce domaine a la caractéristique d'être infini même quand on fixe le nombre de variables (au contraire du domaine des fonctions booléennes positives). L'argument de terminaison de l'analyse est donc plus subtil que pour l'abstraction par les fonctions booléennes. Il repose sur l'hypothèse que le programme analysé est bien typé [18]. Ainsi, si la propriété analysée est l'absence de variable existentielle libre (la clôture, ou *groundness* en anglais), l'expression *(list (pair vrai faux))* sera vraie d'une liste dont le squelette ne comporte pas de variable, dont aucun élément n'est une variable, et dont tous les éléments comportent un «champ gauche» sans variable libre et un «champ droit» qui peut en comporter.

Ce travail a été réalisé en collaboration avec Patrice Boizumault de l'École des Mines de Nantes.

6.2.5 Analyse de trace automatisée

Participants : Mireille Ducassé, Erwan Jahier.

Mots clés : débogage, analyse dynamique, traceur abstrait, Prolog, Mercury, langage C.

Résumé : *Nous avons développé des outils qui analysent les traces d'exécution de programmes générées par un traceur bas niveau. Les programmeurs peuvent spécifier de manière précise ce qu'ils veulent observer du comportement du programme à l'aide de requêtes exprimées en Prolog. À partir du langage de requêtes, nous avons bâti des analyses qui fournissent des vues abstraites des exécutions selon certains critères. Il a également été possible de mettre en œuvre à faible coût des traceurs abstraits pour des langages de haut niveau. Les techniques de base exploitent une trace dont le seul pré-requis est d'être séquentielle. Nous avons montré cette généralité en appliquant ces principes aux langages Prolog, Mercury et C.*

Nous avons développé des outils (cf. module 5) qui analysent les traces d'exécution de programmes générées par un traceur bas niveau (cf. module 3.3). Le programmeur peut poser des questions sur les exécutions à l'aide de Prolog et de quelques primitives dans une forme concise en s'appuyant sur la logique et les mécanismes de recherche du langage. Les programmeurs peuvent donc, à l'aide de requêtes de haut niveau, spécifier de manière précise ce qu'ils veulent observer du comportement du programme.

Le prototype initial, Opium [5], était essentiellement dédié au débogage. Nous avons montré avec Morphine comment il est possible d'utiliser un traceur doté d'un module d'analyse de traces pour faire davantage que du débogage. Par exemple, nous pouvons calculer un taux de couverture de jeu de test afin d'en évaluer la qualité. Des «moniteurs» qui surveillent le comportement des programmes peuvent également être facilement mis en œuvre. Ainsi, au lieu de fabriquer des instrumentations *ad hoc* comme c'est le cas actuellement pour de tels outils,

on peut utiliser un environnement uniforme, ce qui permet une synergie entre les outils. De plus, les instrumentations *ad hoc* nécessitent de bien connaître le système à instrumenter, que l'instrumentation se fasse au niveau bas ou par transformation de source à source. Même si elle n'est pas techniquement très difficile, cette tâche demande un effort de programmation non négligeable. Dans notre cas, l'instrumentation étant générique, elle est faite une fois pour toutes et les analyses spécifiques peuvent être relativement simples [13].

À partir du langage de requêtes, nous avons également bâti des analyses qui fournissent des vues abstraites graphiques d'exécutions Mercury (par exemple, arbres de preuves, graphes de flot de contrôle). Grâce à la flexibilité de nos mécanismes, chacune de ces vues ne nécessite que quelques lignes de Mercury. Les temps d'exécution sont acceptables [29].

Nous démarrons un projet RNTL sur l'analyse de trace d'exécutions de programmes sous contraintes (cf. module 7.1).

6.2.6 Sémantique des traces

Participants : Mireille Ducassé, Erwan Jahier, Olivier Ridoux.

Mots clés : débogage, traceur, modèle de traces, sémantique, continuation, Prolog.

Résumé : *Nous proposons une architecture formelle pour spécifier, prototyper et valider des modèles de traces d'exécutions Prolog. Cette architecture est basée sur une sémantique opérationnelle par continuations. Les spécifications formelles peuvent être exécutées par une transcription directe en λProlog. Cela permet d'expérimenter différents modèles de traces et de les valider.*

Le modèle de base pour tracer des exécutions Prolog est le modèle de Byrd [Byr80]. De nombreux systèmes l'utilisent, mais avec différentes interprétations ce qui fait qu'ils génèrent des traces différentes. Ces divergences ne sont certainement pas toutes intentionnelles, certaines étant dues au fait que la spécification originale est informelle.

Pour remédier à ce problème, nous proposons une architecture formelle permettant de spécifier, de prototyper et de valider des modèles de traces d'exécutions de Prolog. Cette architecture est basée sur une sémantique opérationnelle par continuation. Nous donnons une spécification formelle du modèle de traces de Byrd et nous montrons comment cette spécification peut être étendue pour spécifier des modèles de traces plus riches. Nous montrons également comment ces spécifications peuvent être exécutées par une transcription directe en λProlog ; nous obtenons ainsi un méta-interprète Prolog qui calcule des traces d'exécutions. Ce méta-interprète peut être utilisé pour expérimenter différents modèles de traces et pour les valider [27, 28].

6.2.7 Génération de traces

Participante : Mireille Ducassé.

[Byr80] L. Byrd, « Understanding the Control Flow of Prolog Programs », *in : Logic Programming Workshop*, S.-A. Tärnlund (éditeur), Debrecen, Hungary, 1980.

Mots clés : débogage, traceur, transformation de programme, mesure de performance, Prolog.

Résumé : *Les traces d'exécution ne sont pas disponibles directement. Il faut des outils pour les générer, que l'on appelle des traceurs. Les traceurs actuels modifient en général en profondeur le compilateur, ce qui rend leur portage problématique. La génération de traces d'exécution par transformation source à source de programmes permet d'éviter cet écueil. Un prototype a été réalisé pour Prolog. Des mesures montrent que cette technique, bien que donnant des performances moindres que des techniques de plus bas niveau, peut être acceptable pour construire des traceurs.*

Les traces d'exécution ne sont pas disponibles directement. Il faut des outils pour les générer, que l'on appelle des traceurs. Les techniques d'implémentation de ces traceurs sont dans l'ensemble mal étudiées. Cela nuit, bien sûr, à leur implémentation, mais aussi à leur portage et à leur maintenance.

Typiquement, les traceurs modifient en profondeur le compilateur, les structures de données et les schémas d'exécution. Les inconvénients de ce type de traceurs sont de plusieurs ordres : tout d'abord, ils ne permettent généralement pas de conserver les optimisations qui donnent tout leur intérêt aux compilateurs ; ensuite, l'information fournie à l'utilisateur est de trop bas niveau, elle peut même se trouver significativement dégradée ; enfin, ces traceurs ne sont pas portables d'un compilateur à l'autre pour un même langage.

Nous étudions actuellement la génération de traces d'exécution pour Prolog par transformation source à source de programmes. Cette technique permet d'éviter les inconvénients mentionnés ci-dessus mais il faut établir son efficacité. Nous avons implémenté un prototype à cet effet. Les mesures montrent que les performances des programmes tracés par transformation de programmes sont au pire deux fois moins bonnes que celles des programmes tracés par deux traceurs opérationnels « bas niveau » [16]. Cette réduction de performance est acceptable pour un traceur « standard » surtout lorsque l'on considère le gain significatif en termes de portabilité.

Ce travail est mené en collaboration avec Jacques Noyé, de l'École des Mines de Nantes.

6.2.8 Détection d'intrusion

Participants : Mireille Ducassé, Jean-Philippe Pouzol.

Mots clés : sécurité, analyse d'audit, intrusion, attaque.

Résumé : *Dans la continuité des travaux sur l'analyse de traces, Lande démarre une activité autour de l'analyse d'audits dans le cadre de la détection d'intrusion. La détection d'intrusion est une analyse de l'activité d'un système visant à rapporter à l'officier de sécurité toute utilisation suspecte. D'une part, nous mettons en place une plate-forme de mesure de performances d'un système de détection d'intrusion existant, afin d'évaluer les problèmes techniques et humains. D'autre part, nous cherchons à concevoir un langage de haut niveau de spécification de signatures d'attaques.*

Dans la continuité des travaux sur l'analyse de traces, Lande démarre une activité autour de l'analyse d'audits dans le cadre de la détection d'intrusion.

Une stratégie classique pour sécuriser un système informatique consiste à construire un bouclier protecteur autour de lui. Les utilisateurs désirant accéder aux données ou ressources du système doivent franchir des barrières de sécurité que constituent, par exemple, des mécanismes d'identification ou de cryptographie. Toutefois, le développement des systèmes ouverts, ainsi que la prolifération de machines hétérogènes connectées à des réseaux, rendent complexe et peu praticable la mise en oeuvre de mécanismes de sécurité totalement fiables.

Une défense possible face aux utilisations abusives d'un système est la *détection d'intrusion*. La détection d'intrusion est une analyse de l'activité du système visant à rapporter à l'officier de sécurité toute utilisation suspecte. Afin de procéder à cette analyse il convient d'effectuer un audit, dit de sécurité, qui alimentera le processus d'analyse. Cet audit fournit une séquence d'événements qui s'apparente à une trace d'exécution de programme.

Comme dans le débogage, les performances des analyses effectuées par un IDS sont cruciales. Nous mettons en place une plate-forme de mesure de performances, en utilisation réelle, sur le réseau enseignement du département informatique de l'INSA de Rennes. Nous utilisons un IDS qui nous semble relativement efficace, ASAX [HCMM93], développé à l'université de Namur. Les résultats préliminaires sont encourageants, la performance moyenne de chaque machine n'est pas trop affectée [19]. Ce travail se fait en collaboration avec Véronique Abily, administrateur système du département Informatique de l'INSA.

Nous cherchons par ailleurs à concevoir un langage de haut niveau de spécification de signatures d'attaques. Les signatures d'attaques sont les manifestations des attaques dans les systèmes attaqués. S'il existe quelques bases informelles d'attaques, nous n'avons pas connaissance d'un corpus formel de signatures. Un prototype préliminaire est en cours de réalisation [31].

6.2.9 Génération systématique de jeux de test

Participants : Thomas Jensen, Valérie-Anne Nicolas, Olivier Ridoux.

Mots clés : test en boîte noire, test en boîte blanche, test structurel, contrainte, Casting, jeu de test, suite de test.

Résumé : *Nous avons proposé une méthode de génération de suites de test qui forme le noyau de l'outil Casting développé en collaboration avec la société* AQL. *La méthode est indépendante du format d'entrée, ce qui la rend utilisable aussi bien dans le cas du test structurel que fonctionnel. Les suites de test engendrées dépendent de stratégies spécifiées par l'utilisateur, permettant ainsi d'atteindre la souplesse d'utilisation exigée pour un usage industriel.*

Nous avons abordé le problème de la systématisation de la génération de jeux de test en tentant d'abolir la dichotomie « boîte noire/boîte blanche» (cf. module 3.4). Pour ce faire, nous

[HCMM93] N. Habra, B. L. Charlier, A. Mounji, I. Mathieu, «ASAX: Software architecture and rule-based language for universal audit trail analysis», *in : Proc of European Symposium on Research in Computer Security*, Toulouse, 1993.

décomposons le processus de production des données de test en trois étapes :

1. L'acquisition des critères de test et la production des hypothèses de test associées, à partir de différents supports d'entrée.
2. La décomposition des opérations en classes d'opérations et la génération d'un graphe d'accessibilité symbolique.
3. La génération des jeux de test par parcours du graphe d'accessibilité en assurant un critère de couverture donné.

Les supports d'entrée peuvent être constitués de spécifications formelles ou informelles, de programmes sources ou de propriétés fournies directement par l'utilisateur. Les opérations peuvent être des machines abstraites dans le cas du langage B, des schémas pour le langage Z, des programmes dans le cas d'un langage de programmation, etc. Dans tous les cas, les critères de test sont implantés par des *stratégies de test* et se traduisent in fine par des *hypothèses d'uniformité et hypothèses de régularité*. Ces hypothèses permettent de préciser le sens (et les limites) des jeux de test qui seront engendrés (cf. module 3.4). D'un point de vue pratique, une stratégie de test correspond à un mode d'extraction de contraintes à partir du texte source. Ces contraintes caractérisent les jeux de données qui devront être engendrés pour chaque opération du système. Le graphe d'accessibilité symbolique indique l'ordre dans lequel les opérations peuvent être appliquées pour satisfaire toutes les contraintes. Dans le cas général en effet on ne peut faire l'hypothèse qu'une opération soit toujours applicable : selon l'état du système, il peut être nécessaire d'effectuer plusieurs opérations intermédiaires avant de pouvoir appliquer une opération donnée. La dernière phase consiste à explorer ce graphe en résolvant les contraintes associées pour générer les données de test effectives.

Ces travaux ont conduit au développement de l'outil d'aide à la génération de jeux de test Casting[3] (cf. module 5). La version actuelle de Casting prend en entrée des spécifications dans la notation AMN de la méthode B [Abr96] et fait appel à Ilog Solver[4] pour résoudre les contraintes engendrées. Diverses démonstrations de ce prototype ont été réalisées (notamment à la conférence B). Nous travaillons maintenant à la transposition de cette technique pour le test d'applications C et C++ (test structurel) dans le cadre du projet européen Two et pour le test d'applications Cobol dans une collaboration industrielle sous l'égide du programme régional ITR (cf. module 7.2).

Nous nous intéressons conjointement à la manière d'assurer que des programmes sont conformes à des hypothèses de test. Nous considérons dans ce but des schémas de programmes qui peuvent être vus comme une manière de définir des classes de fautes : identifier un programme de manière non ambiguë dans une classe revient à assurer que toutes les fautes ayant pour effet de produire une version de programme dans cette classe seront détectées. À tout schéma de programme est associé un jeu de test fini et robuste [30] (c'est à dire permettant de distinguer deux fonctions quelconques du schéma). Ce résultat permet de montrer automatiquement qu'un programme satisfait une propriété donnée (à condition que l'un et l'autre

3. *Computer Assisted Software Testing*
4. Ilog Solver est une marque déposée par Ilog.

[Abr96] R. Abrial, *The B-Book: Assigning programs to meanings*, Cambridge University Press, 1996.

puissent être situés dans notre hiérarchie de schémas). La propriété attendue est définie comme une fonction et la propriété effective est dérivée du programme par interprétation abstraite. Il s'agit ensuite de déterminer le plus petit schéma de la hiérarchie qui les contient toutes les deux. On sait alors que le jeu de test associé à ce schéma est suffisant pour établir l'égalité des deux fonctions. Ce jeu de test peut être concrétisé (opération inverse de l'interprétation abstraite) pour être soumis au programme [30].

7 Contrats industriels (nationaux, européens et internationaux)

7.1 Action OADYMPPAC

Participants : Mireille Ducassé, Erwan Jahier.

Mots clés : environnement de programmation, programmation logique avec contraintes, débogage, visualisation.

Résumé : *Un des objectifs du projet est de mettre en forme et d'expérimenter des techniques de trace génériques pour des programmes logiques avec contraintes. Cela facilitera la définition d'outils d'observation et de mise au point. Un autre aspect est d'étudier l'apport, à la conception de tels outils, des progrès réalisés en matière de visualisation d'information sur de grands ensembles de données.*

Le projet OADYMPPAC démarre à l'automne 2000. Un des objectifs du projet est de mettre en forme et d'expérimenter des techniques de trace génériques pour des programmes logiques avec contraintes. Cela facilitera la définition d'outils d'observation et de mise au point. Un autre aspect est d'étudier l'apport, à la conception de tels outils, des progrès réalisés en matière de visualisation d'information sur de grands ensembles de données. Cela permettra, en parallèle, la définition et l'expérimentation de nouveaux outils de mise au point. Il y a deux champs d'application : d'une part, la programmation avec contraintes et, en particulier, une meilleure compréhension du comportement des solveurs ; d'autre part le développement de techniques génériques de visualisation d'information adaptées à l'analyse visuelle des traces produites par les phénomènes dynamiques.

Domaine de recherche assez récent, la visualisation interactive d'information a pour objectif d'aider la compréhension de données ou de processus abstraits au moyen de la production de représentations visuelles interactives de ces données. La visualisation d'information se présente comme un sous-domaine des sciences cognitives appliquées, dont le but est d'amplifier les mécanismes de cognition en tirant profit au mieux des particularités connues de notre système perceptif.

Cette action bénéficie d'un soutien du ministère de la recherche, sous la forme d'un contrat RNTL. Les partenaires industriels du consortium sont Ilog et Cosytec. Nos partenaires universitaires sont l'INRIA Rocquencourt, l'université d'Orléans et l'école des Mines de Nantes.

7.2 Action Two

Participants : Thomas Jensen, Valérie-Anne Nicolas, Olivier Ridoux, Lionel Van Aertryck.

Mots clés : test en boîte noire, test en boîte blanche, test structurel, objectif de test, contrainte, jeu de test, suite de test.

Résumé : *Nous avons proposé une méthode de génération de suites de test qui forme le noyau de l'outil Casting développé en collaboration avec la société* AQL. *La première version du prototype prend en entrée des spécifications B. Nous travaillons maintenant, en collaboration avec différents partenaires universitaires et industriels, à l'exploitation de ces techniques pour la génération de jeux de test structurels (pour C, C++ et Cobol).*

Nos travaux sur la génération automatique de jeux de test ont été initiés dans le cadre d'une collaboration avec la société rennaise AQL à travers une bourse Cifre puis le stage de post-doc industriel de Lionel Van Aertryck. Ils ont conduit à la réalisation du prototype Casting, un outil qui permet de générer des jeux de test à partir de spécifications B. Cette génération peut être automatique ou interactive en fonction des besoins de l'utilisateur. Le prototype en question est maintenant robuste et il en a été fait plusieurs démonstrations, notamment lors de la conférence B.

Ces travaux sur la génération automatique de jeux de test ont pris une nouvelle dimension dans le cadre de deux collaborations industrielles : le projet européen Two (*Test and Warning Office*, Industrial RTD Project no 25503, ref. Inria : 1 98 C 344) et l'action effectuée dans le cadre du programme régional ITR.

Le projet Two a commencé en octobre 1998 et s'étendra sur deux années et demie. Il implique le centre de recherche du CEA, le Politecnico di Milano (Italie), les sociétés Elsag Bailey (Italie), Siemens (Allemagne), Spacebel Informatique (Belgique) et l'éditeur d'outils de tests Attol Testware (France) qui pilote le projet.

L'objectif du projet Two est de concevoir un outil de génération de jeux de test structurels pour C et C++. Cet outil se décomposera en trois phases principales : l'analyse des codes sources, la génération de contraintes correspondant à un objectif de test donné et la résolution de ces contraintes pour engendrer les jeux de test effectifs. Notre intervention se situe essentiellement dans la troisième phase. L'outil mis au point dans le cadre du projet Two sera intégré à l'environnement de test actuellement commercialisé par Attol Testware : il facilitera la tâche du testeur tout en permettant les mesures de taux de couverture offertes par l'outil Attol Coverage. Les services ajoutés qui seront fournis par le projet Two correspondent à une demande forte de la part des utilisateurs actuels de l'environnement d'Attol Testware.

7.3 Action Java-Sécurité

Participants : Marc Eluard, Thomas Jensen.

Mots clés : téléchargement, vérification, analyse, sécurité, sûreté, chargement dynamique, visibilité, typage, Java.

Résumé : *Dans le cadre de l'action* VIP *du* GIE *Dyade, nous avons proposé une formalisation de la sémantique de certains aspects du langage Java et Java Card. Nous nous sommes focalisés en ce qui concerne Java sur les règles de visibilité (des classes et de leurs membres) et leur évolution lors du chargement dynamique de classes et en ce qui concerne Java Card sur le pare-feu et les objets partageables qui servent à sécuriser la communication entre applettes installées sur une carte à puce multi-applicative.*

Nous participons à l'action VIP du GIE Bull-Inria Dyade. Le thème de cette collaboration est la formalisation de certains aspects de la sémantique de Java et la preuve de propriétés de sécurité de programmes (cf. module 6.2.3). L'étude des problèmes de sécurité (au sens de confidentialité et d'intégrité notamment) dans le contexte du langage Java représente un défi de première importance pour plusieurs raisons :

- La sûreté (au sens du typage) et la sécurité sont présentées comme des arguments pour la promotion d'un langage qui a vocation à être utilisé dans des contextes mettant en jeu des coopérations entre des codes issus de sites différents.

- Le langage inclut des caractéristiques complexes (comme le chargement dynamique ou des règles de visibilité inhabituelles) qui justifient le besoin de définition formelle. Une telle définition permettrait de clarifier certains aspects du langage et servirait de base à un raisonnement rigoureux sur des propriétés cruciales comme la sûreté du typage ou la garantie de politiques de sécurité.

- Le développement de Java Card, la version de Java dédiée aux cartes à puces, augmente encore l'importance des défis cités plus haut et permet de les aborder dans un cadre restreint, permettant l'application de techniques (analyse, preuve) plus sophistiquées.

Nous nous sommes attaqués à cette formalisation en nous focalisant sur les règles de visibilité (des classes et de leurs membres) et leur évolution lors du chargement dynamique de classes. Il s'agit en effet de caractéristiques particulières de Java qui ont un impact direct sur la sécurité et dont les définitions informelles ne sont pas exemptes d'ambiguïtés ou d'insuffisances. Cette formalisation en terme de systèmes d'inférence nous a permis de décrire de manière rigoureuse l'origine d'une erreur de sécurité qui avait été découverte empiriquement par des chercheurs d'ATT.

Nous avons ensuite étudié comment décrire formellement le pare-feu de Java Card. La spécification du langage Java Card comprend la description d'un pare-feu entre les applettes installées sur une carte. Ce pare-feu garantit par défaut une séparation totale entre différentes

applettes mais il peut être contourné en s'appuyant sur la notion d'*objets partagées* qui permettent d'échanger des objets entre applettes. Afin d'analyser les conséquences en matière de sécurité d'un partage des objets il est nécessaire de mettre la description informelle de la spécification Java Card sous une forme qui permet d'effectuer des vérifications formelles. Nous avons décrit une sémantique opérationnelle du pare-feu Java Card et nous étudions actuellement comment extraire de cette sémantique des analyses statiques permettant de construire des modèles abstraits afin d'y appliquer nos techniques (décrites en module 6.2.3 de vérification de propriétés de sécurité.

7.4 Action Secsafe

Participants : Thomas Jensen, Fausto Spoto, Marc Éluard, Frédéric Besson, Florimond Ployette.

Mots clés : Sécurité, analyse statique, cartes à puce, code mobile, Java, Java Card..

Résumé : *Le projet européen Secsafe porte sur la sécurité de logiciels et l'analyse statique avec une focalisation sur les langage Java et Java Card. Deux domaines d'application sont visés : le code mobile et les cartes à puce. Le projet a deux partenaires académiques (Imperial College de l'université de Londres et l'université d'Aarhus du Danemark) et la PME Trusted Logic, spécialisée en sécurité et cartes à puce.*

Le projet européen Secsafe qui a commencé en 2000 porte sur la sécurité de logiciels et l'analyse statique avec une focalisation sur les langage Java et Java Card. Le choix de Java comme langage à étudier permet d'appliquer nos analyses à deux domaines différents : le code mobile et les cartes à puce. Les analyses envisagées seront fondées sur nos travaux de formalisation de Java et Java Card (voir modules 6.2.3 et 7.3). Le projet a deux partenaires académiques (Imperial College de l'université de Londres et l'université d'Aarhus du Danemark) et la PME Trusted Logic, spécialisée en sécurité et cartes à puce.

7.5 Action Castor

Participants : Pascal Fradet, Lakshminarayanan Renganarayanan.

Mots clés : architectures de logiciel, vues, cohérence, sécurité, analyse, vérification.

Résumé : *L'objectif du projet Castor est de fournir un environnement permettant de décrire l'architecture d'un système d'informations et d'étudier ses propriétés de sécurité. L'analyse d'une telle architecture permettra de fournir les conditions qui doivent être vérifiées par les composants pour assurer des contraintes de sécurité globales.*

Le projet Castor, financé par le Celar, regroupe les sociétés Sycomore Aérospatiale Matra, AQL et TNI et les projets EP-ATR et Lande de l'Irisa. Son objectif est de fournir un environnement permettant de décrire l'organisation globale d'un système d'informations et d'étudier ses

propriétés de sécurité. Il est donc nécessaire de formaliser l'architecture du logiciel à l'aide des vues nécessaires à l'expression des propriétés de sécurité considérées. L'analyse d'une telle architecture doit fournir les conditions qui doivent être vérifiées par les composants pour assurer des contraintes de sécurité globales. Elle doit également permettre d'identifier l'impact d'un changement de configuration sur la sécurité du système. Nous étudions dans ce cadre les problèmes liés aux descriptions multi-vues (relations, cohérence) et à l'agrégation de composants (c'est-à-dire aux vues hiérarchiques). Cette action se situe dans le prolongement des travaux en cours sur la vérification de cohérence d'architectures à vues multiples.

8 Actions régionales, nationales et internationales

8.1 Actions nationales

Le projet Lande participe à l'action ASP «Unification des méthodes de test». Les autres partenaires sont le LaMI (Evry), le LRI (Orsay) et le LSR-Imag (Grenoble).

Thomas Jensen est responsable de l'Action de Recherche Collaborative Java Card qui associe les projets Cristal, Oasis et Lande à deux partenaires extérieurs : le groupe Sécurité du Cert (Toulouse) et l'action VIP de Dyade. Son objectif est de fédérer les activités en cours en France sur la sémantique, l'optimisation et la sécurité de Java Card, la version de Java dédiée aux cartes à puces.

Le projet Lande prend part également à l'Action de Recherche Coopérative S-Java qui associe les projets Oasis, Coq et Lande et une équipe de l'ENS à trois partenaires industriels : France Telecom R&D, Gemplus et Trusted Logic. L'objectif de l'action est de concevoir des méthodes de conception et d'analyse de programmes Java certifiés.

Le projet Lande contribue au projet national RNTL OADYMPPAC sur la visualisation de programmes logiques avec contraintes (cf. module 7.1). Les autres partenaires sont Ilog, Cosytec, l'INRIA Rocquencourt, l'université d'Orléans et l'école des Mines de Nantes.

Le projet Lande participe à l'action SmartTool dans le cadre du GIE Bull-Inria Dyade sur le développement d'un environnement de programmation. Les autres partenaires sont les projets Oasis, Coq et l'action Vasy ainsi que Bull et Microsoft.

8.2 Actions financées par la Commission Européenne

Le projet Lande participe aux projets européens Two, Secsafe et Coordina. Les deux premiers sont décrits dans la section Actions industrielles (modules 7.2 et 7.4 respectivement). Coordina est un *working group* (Esprit Working Group 24512, *From Coordination Models to Applications*, ref Inria : 1 97 C 905) qui a été lancé en août 1997 pour une période de trois années. Son but est l'étude des langages et des modèles de coordination et d'architectures de logiciels. Les activités du groupe s'articulent autour d'études de cas fournies par les sociétés Signaal (Pays-Bas) et Xerox (France). Le projet inclut comme partenaire académiques le CWI, les universités de Berlin, Berne, Bologne, Chalmers, Genève,

8.3 Réseaux et groupes de travail internationaux

Mireille Ducassé est membre de l'ACM (Association for Computing Machinery). Mireille Ducassé, Erwan Jahier et Olivier Ridoux sont membres de l'ALP (Association for Logic Programming).

8.4 Relations bilatérales internationales

Le projet Lande est partenaire d'une collaboration franco-britannique dans le cadre du programme « Alliance ». L'autre partenaire est l'Imperial College de l'université de Londres. Cette collaboration bilatérale vise à améliorer les bases théoriques nécessaires pour vérifier des propriétés de sécurité et de sûreté des applications exprimées dans des formalismes à objets. Un objectif est d'étudier comment tirer profit des avancées réalisées dans le domaine de la sémantique à base de théorie des jeux pour obtenir des analyses pour les langages à objets.

8.5 Accueil de chercheurs étrangers

Le projet a reçu Bjarke Ebert, de l'université d'Aarhus du Danemark, qui a travaillé sur la construction d'un transformateur pour le byte code Java Card. La visite a duré trois mois.

Le projet a par ailleurs accueilli un certain nombre de visiteurs étrangers pour des visites ponctuelles. Nous ne les passons pas en revue ici.

9 Diffusion de résultats

9.1 Animation de la communauté scientifique

Mireille Ducassé a été responsable du comité de programme du 4ème workshop international sur le débogage automatisé (AADEBUG 2000) dont elle a édité les actes [11]. Un numéro spécial du «Journal of Automated Software Engineering» est en préparation sous sa responsabilité. Elle a également été membre du comité de programme de LOPSTR 2000 (LOgic-based Program Synthesis and TRansformation). Elle est membre élu du conseil scientifique de l'Insa de Rennes depuis septembre 1994.

Olivier Ridoux est membre du conseil d'administration de l'AFPLC (Association Française de Programmation Logique et par Contraintes, correspondant en France de l'ALP). Il a fait partie des comités de programme de JFPLC2000 (Journées Francophones de Programmation Logique et par Contraintes) et WMMLP99 (*International Workshop on Memory Management of Logic Programming Languages*). Il fait partie du comité de programme de FLOPS2001.

Thomas Jensen a fait partie des comités de programme de SAS'00 (Static Analysis Symposium 2000) et de ITRS'00 (International Workshop on Intersection Types and Related Systems). Il a présidé le comité de programme de JavaCard 2000 (International workshop on Java Card organisé par le Java Card Forum et l'Inria) [10]. Par ailleurs, il a été rapporteur sur la thèse de Romain Guider (préparée sous la responsabilité d'Isabelle Attali et Denis Caromel).

9.2 Enseignement universitaire

Pascal Fradet et Thomas Jensen assurent un module de DEA sur la sémantique et l'analyse de programmes.

Olivier Ridoux est le responsable de la maîtrise d'informatique de l'université de Rennes 1. Il enseigne la programmation et le génie logiciel en DEUG, et les systèmes d'exploitation et la compilation en maîtrise d'informatique.

Mireille Ducassé est responsable de l'option industrielle de la dernière année de la formation d'ingénieur en informatique de l'INSA de Rennes. Elle enseigne la compilation et la méthode formelle «B» au niveau bac+4, ainsi que la qualité du logiciel au niveau bac+5. Elle encadre un projet annuel de 8 étudiants, niveau bac+4, qui travaillent sur le logiciel Coca (cf module 5).

Thomas Genet enseigne la programmation fonctionnelle et le génie logiciel en DEUG.

Le projet a encadré cinq étudiants de DEA : Arnaud Le Claire (*Extraction d'un converteur Java Card en Coq*), Yoann Padioleau (*Formules de Harrop et contraintes*), Jean-Philippe Pouzol (*Détection d'intrusion*), Jean-Philippe Pouzol (*Détection d'intrusion dans les systèmes informatiques*), Agnès Simon (*Sécurisation d'applets Java*) et Romain Thomas (*Isomorphismes de types et systèmes d'information logique*).

Par ailleurs, le projet a reçu des étudiants de l'université de Rennes 1 et des Facultés Universitaires Notre Dame de Namur, Belgique, (Nicolas Brieuc, David Baudoin, Vincent Vesvart, Christelle Lecomte, Xavier Martin, Nicolas Vanderavero) pour des stages d'été et de fin d'étude.

9.3 Participation à des colloques, séminaires, invitations

Mireille Ducassé a donné un tutoriel invité sur la méthode formelle «B» à LOPSTR-99 qui a donné lieu à publication a posteriori [22].

Sébastien Ferré et Olivier Ridoux ont présenté leurs travaux sur les systèmes d'information logique au séminaire AFPLC «Data-mining et programmation par contraintes».

10 Bibliographie

Ouvrages et articles de référence de l'équipe

[1] J.-P. BANÂTRE, D. LE MÉTAYER, «Programming by multiset transformation», *Communications of the ACM 36*, 1, 1993, p. 98–111.

[2] C. BELLEANNÉE, P. BRISSET, O. RIDOUX, « A pragmatic reconstruction of λProlog », *Journal of Logic Programming, 41(1), 1999.*, Version française dans TSI 14(9):1131–1164:1995.

[3] R. DOUENCE, P. FRADET, « A systematic study of functional language implementations », *ACM Transactions on Programming Languages and Systems 20*, 2, 1998, p. 344–387.

[4] M. DUCASSÉ, J. NOYÉ, « Logic programming environments: dynamic program analysis and debugging », *Elsevier Journal of Logic Programming 19/20*, mai/juillet 1994, p. 351–384, `http://www.irisa.fr/EXTERNE/bibli/pi/pi910.html`.

[5] M. Ducassé, « Opium: An extendable trace analyser for Prolog », *Elsevier Journal of Logic programming 39*, 1999, p. 177–223, Special issue on Synthesis, Transformation and Analysis of Logic Programs, A. Bossi and Y. Deville (eds).

[6] P. Fradet, *Approches langages pour la conception et la mise en œuvre de programmes*, document d'habilitation à diriger des recherches, Université de Rennes 1, novembre 2000.

[7] T. Jensen, D. Le Métayer, T. Thorn, «Verification of control flow based security properties», *in : Proc. of the 20th IEEE Symp. on Security and Privacy*, New York: IEEE Computer Society, p. 89–103, mai 1999.

[8] T. Jensen, « Disjunctive Program Analysis for Algebraic Data Types », *ACM Transactions on Programming Languages and Systems 19*, 5, 1997, p. 752–804.

[9] O. Ridoux, *λProlog de A à Z, ... ou presque*, document d'habilitation à diriger des recherches, Université de Rennes 1, avril 1998.

Livres et monographies

[10] I. Attali, T. Jensen (éditeurs), *Proceedings of the International Workshop on Java Card (Java Card 2000)*, Cannes, France, Inria, Septembre 2000.

[11] M. Ducassé (éditeur), *Proceedings of the 4th International Workshop on Automated Debugging (AADEBUG2000)*, Munich, COrr, August 2000. Refereed proceedings to appear in the COmputer Research Repository (CORR), `http://www.irisa.fr/lande/ducasse/aadebug2000/proceedings.html`.

Thèses et habilitations à diriger des recherches

[12] P. Fradet, *Approches langages pour la conception et la mise en œuvre de programmes*, document d'habilitation à diriger des recherches, Université de Rennes 1, novembre 2000.

[13] E. Jahier, *Analyse dynamique d'exécutions : construction automatisée d'analyseurs performants et spécifications de modèles d'exécution*, thèse de doctorat, INSA de Rennes, décembre 2000.

[14] T. Jensen, *Analyse statiques de programmes : fondements et applications*, document d'habilitation à diriger des recherches, Université de Rennes 1, décembre 1999.

[15] M. Périn, *Spécifications graphiques multi-vues : formalisation et vérification de cohérence*, thèse de doctorat, IFSIC, octobre 2000.

Articles et chapitres de livre

[16] M. Ducassé, J. Noyé, «Tracing Prolog programs by source instrumentation is efficient enough», *Elsevier Journal of Logic Programming, 43*, 2, May 2000, p. 157–172.

[17] P. Fradet, J. Mallet, « Compilation of a Specialized Functional Language for Massively Parallel Computers », *Journal of Functional Programming, Cambridge University Press*, 2000.

[18] O. Ridoux, P. Boizumault, « Typed static analysis: application to the groundness analysis of Typed Prolog », *ACM Journal of Functional and Logic Programming*, 2000.

Communications à des congrès, colloques, etc.

[19] V. Abily, M. Ducassé, « Benchmarking a distributed intrusion detection system based on ASAX: Preliminary results », *in : RAID 2000 (Recent Advances on Intrusion Detection)*, H. Debar (éditeur), 2000. Refereed extended abtract, `http://www.raid-symposium.org/raid2000/program.html`.

[20] T. Colcombet, P. Fradet, « Enforcing trace properties by program transformation », *in : Proc. of Principles of Programming Languages*, ACM Press, p. 54–66, Boston, janvier 2000.

[21] E. Denney, T. Jensen, « Correctness of Java Card method lookup via logical relations », *in : Proc. of European Symp. on Programming (ESOP 2000)*, *Lecture Notes in Computer Science*, Springer, p. 104–118, 2000.

[22] M. Ducassé, L. Rozé, « Proof obligations of the B formal method: Local proofs ensure global consistency », *in : LOgic-based Program Synthesis and TRansformation*, A. Bossi (éditeur), Springer-Verlag, Lecture Notes in Computer Science, 1817, p. 11–30, 2000.

[23] S. Ferré, O. Ridoux, « A File System Based on Concept Analysis », *in :* dood*2000, 1st Int. Conf. Computational Logic,* lnai *1861*, Y. Sagiv (éditeur), 2000.

[24] S. Ferré, O. Ridoux, « A logical Generalization of Formal Concept Analysis », *in : 8th Int. Conf. Conceptual Structures,* lnai *1867*, B. Ganter, G. Mineau (éditeurs), 2000.

[25] P. Fradet, V. Issarny, S. Rouvrais, «Analyzing non-functional properties of mobile agents», *in : Proc. of Fundamental Approaches to Software Engineering, FASE'00*, *Lecture Notes in Computer Science, Vol.1783*, Springer-Verlag, p. 319–333, march 2000.

[26] T. Genet, F. Klay, «Rewriting for Cryptographic Protocol Verification», *in : Proceedings 17th International Conference on Automated Deduction*, *Lecture Notes in Artificial Intelligence*, *1831*, Springer-Verlag, 2000, `ftp://ftp.irisa.fr/local/lande/tg-fk-cade00.ps.gz`.

[27] E. Jahier, M. Ducassé, O. Ridoux, « Spécification de modèles de traces de programmes Prolog à l'aide d'une sémantique par continuation », *in : Actes des Journées francophones de Programmation Logique et par Contraintes*, Touraivane (éditeur), Hermès, Marseille, 2000.

[28] E. Jahier, M. Ducassé, O. Ridoux, « Specifying Prolog Trace Models with a Continuation Semantics », *in : Proc. of LOgic-based Program Synthesis and TRansformation*, K.-K. Lau (éditeur), London, July 2000. Technical Report Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-00-6-1., `http://www.cs.man.ac.uk/cstechrep/titles00.html`.

[29] E. Jahier, «Collecting graphical abstract views of Mercury program executions», *in : Proceedings of the International Workshop on Automated Debugging (AADEBUG2000)*, M. Ducassé (éditeur), Munich, August 2000. Refereed proceedings to appear in the COmputer Research Repository (CORR), `http://www.irisa.fr/lande/ducasse/aadebug2000/proceedings.html`.

[30] D. L. Metayer, V.-A. Nicolas, O. Ridoux, « Verification by testing for recursive program schemes», *in : LOgic-based Program Synthesis and TRansformation*, A. Bossi (éditeur), Springer-Verlag, Lecture Notes in Computer Science, 1817, 2000.

[31] J.-P. Pouzol, M. Ducassé, «Handling Generic Intrusion Signatures is not Trivial», *in : RAID 2000 (Recent Advances on Intrusion Detection)*, H. Debar (éditeur), 2000. Refereed extended abtract, `http://www.raid-symposium.org/raid2000/program.html`.

[32] M. PÉRIN, « Cohérence de spécifications multi-vues », *in : les actes de l'atelier Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'2000)*, Grenoble, France, janvier 2000.

Rapports de recherche et publications internes

[33] S. FERRÉ, O. RIDOUX, « Une généralisation logique de l'analyse de concepts formels », *Technical Report n° RR-3820*, Inria, Institut National de Recherche en Informatique et en Automatique, décembre 1999, `http://www.inria.fr/rrrt/rr-3820.html`.

[34] S. FERRÉ, O. RIDOUX, « A File System Based on Concept Analysis », *Technical Report n° RR-3942*, Inria, Institut National de Recherche en Informatique et en Automatique, avril 2000, `http://www.inria.fr/rrrt/rr-3942.html`.

[35] P. FRADET, J. MALLET, « Compilation of a Specialized Functional Language for Massively Parallel Computers », *rapport de recherche n° 3894*, INRIA, mars 2000.