INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Projet LANDE

Conception et validation de logiciels

Rennes

THÈME 2A

Table des matières

1 Composition de l'équipe

Responsable scientifique

Daniel Le Métayer [DR Inria]

Assistantes de projet

Laëtitia Brenugat [vacataire Inria, jusqu'en mai 1999]

Catherine Godest [TR CNRS, depuis juin 1999]

Personnel Inria

Pascal Fradet [CR]

Florimond Ployette [IR, Atelier]

Olivier Ridoux [CR, maître de conférences en détachement à l'Ifsic depuis février 1999]

Personnel CNRS

Thomas Jensen [CR]

Personnel Université

Thomas Genet [maître de conférences, depuis octobre 1999]

Valérie Gouranton [Ater jusqu'en janvier 1999]

Personnel Insa

Mireille Ducassé [professeur]

Julien Mallet [Ater Insa]

Chercheurs doctorants

Frédéric Besson [allocataire MENRT]

Thomas Colcombet [allocataire ENS, depuis septembre 1999]

Marc Éluard [boursier Inria]

Sébastien Ferré [boursier CNRS, depuis octobre 1999]

Erwan Jahier [allocataire MENRT]

Sarah Mallet [allocataire MENRT]

Michaël Périn [boursier Inria-Région jusqu'en septembre 1999, puis Ater Ifsic]

Siegfried Rouvrais [allocataire MENRT, en co-encadrement avec le projet Solidor]

Gaëlle Ségouat [allocataire ENS, depuis septembre 1999]

Tommy Thorn [boursier Inria, jusqu'en février 1999, ingénieur expert de mars à août 1999]

Chercheurs post-doctorants

Ewen Denney [jusqu'en novembre 1999]

Valérie-Anne Nicolas [post-doc industriel Sodifrance-Inria]

Lionel Van Aertryck [post-doc industriel AQL-Inria jusqu'en janvier 1999]

Visiteurs

Peter Bertelsen [de septembre à novembre 1999]

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 sera effectuée dans le cadre de deux collaborations industrielles entamées fin 1999 : la première concerne la spécification de propriétés de sécurité sur les architectures (projet Castor financé par le Celar) et la seconde la spécification de contraintes de cohérence de diagrammes UML décrivant des familles d'architectures (projet européen ESAPS dans le cadre de ITEA).

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.

Dans le cadre d'une collaboration avec le projet Caps de l'Irisa, nous avons également proposé un langage multi-ensembliste dédié à la conception de protocoles de cohérence de mémoires virtuelles partagées. Ce langage restreint permet la vérification automatique de propriétés de correction des protocoles. Une autre forme de langage dédié a été étudiée dans le projet: il s'agit d'un langage fonctionnel à base de modèles (ou «patrons») spécialisé pour des 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. Ces deux travaux illustrent les avantages d'une démarche «langage»: des restrictions adéquates sur le langage de programmation permettent des traitements automatiques plus poussés tout en offrant un pouvoir d'expression adapté à une classe donnée d'applications.

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. 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, Opium-M), et des bases de données déductives.

Nous considérons aussi l'analyse sémantique dans une perspective plus large en étudiant la construction systématique d'analyseurs reposant sur des sémantiques opérationnelles de langages. Nous avons proposé un *format* de sémantique naturelle et nous avons appliqué notre méthode à l'analyse de *slicing*. Il a été ainsi possible de dériver, par simple instanciation d'une définition générique, des analyses dynamiques et statiques pour un langage impératif, un langage fonctionnel et un langage de programmation logique.

En nous appuyant sur des résultats récents concernant l'inférence de types d'intersection et de types polymorphes, nous avons également conçu un algorithme qui permet de déduire, pour chaque programme, une propriété principale à partir de laquelle toute autre propriété démontrable par l'analyse peut être déterminée. Un avantage de cet algorithme est qu'il traite des fragments de programmes aussi bien que des programmes entiers. Il s'agit donc d'une démarche qui, à plus long terme, peut mener à un cadre général pour l'analyse modulaire.

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, C++ et Cobol) dans deux collaborations industrielles: au niveau européen dans le cadre du projet Two et au niveau régional sous l'égide du programme ITR.

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 $Plus : Nat \times Nat \to Nat$, la sémantique dénotationnelle de notre langage se décrit comme suit :

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

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.

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

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

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 ».

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

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

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, 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).

[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.

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.

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* ;

[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.

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

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.

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

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

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 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 lesquels 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.

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

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 *test 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 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

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

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.

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 et 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.

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

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, base de données déductive, architecture sécurisée.

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

1. *Les applications qui exigent un degré de confiance très important justifiant 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 d'une collaboration avec Bull-CP8) à la vérification de transformateurs de bytecodes Java Card, avec comme domaine d'application la sécurité des cartes à puce (cf modules 6.2.4 et 7.3).

  Les langages d'architectures de logiciels constituent une démarche plus récente pour le développement de logiciels complexes. Sur ce thème, nous sommes impliqués dans deux actions industrielles en phase d'initialisation. La première 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. La seconde concerne la conception et la vérification de propriétés de familles d'architectures décrivant des lignes de produits. Elle regroupe de nombreux partenaires au sein du projet européen ESAPS (notamment Thomson et Alcatel) et implique également le projet Compose 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). Nos travaux sur les explications dans les bases de données déductives (cf module 6.2.6) sont en cours d'application dans le domaine de la publicité ciblée pour la télévision à la demande (avec Next Century Media). Nous avons également entamé une collaboration industrielle (cf module 7.1) qui nous permet d'appliquer nos techniques de débogage à des logiciels de facturation hospitalière (avec Mission Critical) et de diagnostic de matériels défectueux (avec Dassault Electronique).

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 Opium-*M *les débogueurs automatisés pour les langages* C *et Mercury 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 Opium-M les débogueurs automatisés pour les langages C et Mercury ainsi que le compilateur de λProlog.

Générateur de suites de test Casting
Correspondant: Lionel Van Aertryck

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 6.2.10; 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 6.2.10). 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).

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.

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 Lionel Van Aertryck (vanaertr@irisa.fr).

Débogueurs Coca et Opium-M

Correspondant : Mireille Ducassé

Coca 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. OPIUM-M est un débogueur automatisé pour le language 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 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 investiguées. 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 d'Opium-M repose sur une analyse à la volée : il ne nécessite donc aucun stockage.

Coca et Opium-M 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 SunOS 5.6 Solaris, avec GCC et GDB ainsi que le système Prolog Eclipse 3.5.2. Il intègre des codes C, ML, Prolog et Bison.

Un prototype de Opium-M, développé dans le cadre du projet Esprit Argo (cf. module 7.1), est opérationnel sous sparc/Solaris2.[5,6,7] et i686/Linux2.0 ainsi que le système Eclipse. Le système, ainsi qu'un manuel utilisateur [33], sont disponible sur le web http://www.irisa.fr/EXTERNE/projet/lande/jahier/download.html.

[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.

Pour plus de renseignements concernant Coca et Opium-M 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 années (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 [8, 2]).

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,

[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.

transformation, etc.). Nous abordons successivement nos travaux sur les architectures de logiciels (module 6.1.1), la programmation par aspects (module 6.1.2), les agents mobiles (module 6.1.3), un langage dédié pour la programmation des mémoires virtuelles partagées (module 6.1.4), un langage dédié pour le parallélisme (module 6.1.5), et un cadre pour la définition de systèmes d'information logiques (module 6.1.6).

6.1.1 Architectures de logiciels

Participants : Pascal Fradet, Daniel Le Métayer, Michaël Périn.

Mots clés : architecture, logiciel, cohérence, vérification, vue, 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) et assurer la correction des évolutions dynamiques de l'architecture.*

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 peut être 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 le délicat problème de la cohérence de ces vues multiples.

Nous avons abordé cette question en proposant un cadre reposant sur la notion de graphe étiqueté [22]. 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. Afin d'offrir aux concepteurs des notations graphiques adaptées à leur pratique, nous avons proposé de décrire les vues par des diagrammes, c'est-à-dire des graphes avec multiplicités sur les arcs – inspirés des notations UML. 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.

Il est rare toutefois que les vues soient complètement indépendantes. Par exemple, la vue «répartition des données» a un impact sur les vues «sécurité» et «tolérance aux fautes». Il est même possible que deux vues répondent à des exigences conflictuelles : par exemple, l'ajout d'un lien de communication améliore la tolérance aux fautes, mais peut s'avérer néfaste pour la sécurité en introduisant un flux d'information illicite. 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.

Les aspects dynamiques du système sont spécifiés par des réécritures multi-ensemblistes qui peuvent décrire aussi bien des protocoles de communication que des évolutions dynamiques d'architectures de logiciels. Des algorithmes de vérification nous permettent d'assurer la cohérence des multiplicités, des vues et des aspects statiques et dynamiques [22]. Nous avons montré que les correspondances entre vues permettent de détecter des incohérences dès les premières phases de développement d'un système.

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, sécurité du système d'informations de l'Irisa). Ils se concrétisent maintenant dans le cadre de deux actions industrielles en phase d'initialisation: la première sera conduite en collaboration avec le projet EP-ATR et impliquera Matra Systèmes Informatiques 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. La seconde action a pour but la spécification de familles d'architectures correspondant à des lignes de produits et l'étude de leur cohérence. Elle se déroule dans de cadre du projet ESAPS qui regroupe de nombreux industriels européens et qui implique également le projet Compose de l'Irisa. Nos partenaires privilégiés seront Thomson et Alcatel.

6.1.2 Programmation par aspects

Participants : Thomas Colcombet, Pascal Fradet.

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

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 commencé à nous intéresser à cette approche en proposant 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, un aspect décrivant alors une politique de sécurité.*

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 commencé à nous intéresser à cette approche en proposant un cadre générique de programmation par aspects. Puis nous nous sommes focalisés sur deux applications: la robustesse [23] et la sécurité [19].

Un programme robuste possède un domaine standard d'entrées sur lequel il termine (et satisfait ses spécifications) et un domaine exceptionnel sur lequel il termine en produisant

une erreur ou une exception. L'écriture de programmes robustes implique le plus souvent de nombreux tests répartis dans le code. Ces tests rendent le programme source difficile à lire et à maintenir. De fait, peu de programmes sont robustes. Dans le cadre de la programmation par aspects, un programme robuste est décomposé en un programme (par exemple codé en C) qui respecte sa spécification pour le domaine standard mais pas forcément pour le domaine exceptionnel et un aspect décrivant le domaine exceptionnel. L'aspect prend la forme d'un ensemble de propriétés (e.g., $1 \leq X \leq 100$ qui spécifie que la variable X doit être comprise entre 1 et 100 durant toute l'exécution). Le programme résultant du tissage est un programme robuste, équivalent au composant pour le domaine standard mais qui termine par un message d'erreur pour le domaine exceptionnel. Pour cette application, le tisseur se doit d'intégrer une analyse de programmes afin d'éviter l'insertion de tests superflus [23]. Ce travail est effectué en collaboration avec Mario Südholt de l'École des Mines de Nantes.

Nous avons également proposé une méthode automatique, inspirée de la programmation par aspects, pour imposer des propriétés exprimées sur les traces d'exécution des programmes [19]. 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 serait la sécurisation d'applets au moment de leur réception. Cette méthode aurait l'avantage de la flexibilité puisque chaque site pourrait imposer à la volée sa propre politique de sécurité aux applets.

La plupart des travaux connexes assurent les politiques de sécurité, soit statiquement (c'est à dire en effectuant une vérification par analyse statique de programme), soit dynamiquement (par un moniteur qui observe et vérifie l'exécution du programme). Les aspects apportent une réponse originale à ce problème de sécurité en intégrant les techniques statiques et dynamiques dans une approche langage.

6.1.3 Analyse de propriétés d'agents mobiles

Participants : Pascal Fradet, Siegfried Rouvrais.

Mots clés : analyse, 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 travaillons 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. 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. Nous étudions actuellement l'intégration de ce cadre à un environnement de développement basé sur les architectures de logiciels et son utilisation pour le raffinement d'architectures. Ce travail est effectué en collaboration avec le projet Solidor.

6.1.4 Langage dédié pour la conception de mémoires virtuelles partagées

Participant : Daniel Le Métayer.

Mots clés : protocole, cohérence, cache, mémoire virtuelle, mémoire partagée, vérification, aspect, langage dédié, réécriture, ensemble.

Résumé : *Nous avons proposé un langage dédié pour le développement de protocoles de cohérence pour les mémoires virtuelles partagées. L'état du système est représenté par un ensemble de relations et les protocoles sont exprimés par des règles de réécriture conditionnelle de l'ensemble. Un langage restreint de propriétés a été défini permettant la vérification automatique d'invariants de l'algorithme. La mise en oeuvre effective d'un protocole est ensuite réalisée grâce à l'ajout d'aspects décrivant notamment l'ordre de traitement des opérations, leur localisation et les communications induites.*

Une «Mémoire Virtuelle Partagée» (MVP) est une infrastructure logicielle permettant de donner à l'utilisateur d'un système distribué l'illusion d'une mémoire commune. Ce concept a été proposé pour faciliter la programmation des systèmes distribués. Comme la performance d'un tel mécanisme repose sur l'utilisation de caches, un protocole de cohérence est nécessaire afin d'assurer que les processeurs accèdent toujours à la dernière version des données. Un certain nombre de protocoles ont été proposés, chacun étant adapté à un type de partage ou de configuration donné. Mais intégrer les protocoles de manière fixe dans les systèmes ne permet pas de les adapter à de nouvelles technologies. On souhaite donc fournir à un expert un langage lui permettant de développer de nouveaux protocoles, de les vérifier et de les intégrer dans une MVP. Nous avons proposé un langage dédié répondant à cet objectif et qui repose sur la notion de réécriture ensembliste [31, 17]. Il s'agit d'une variante spécialisée du langage Gamma étudié

depuis plusieurs années à l'Irisa. Les éléments de l'ensemble sont des relations qui décrivent l'état global du système (par exemple, le fait qu'une page p est utilisée en lecture par un noeud n peut être représenté par une relation $Read(p,n)$). Des règles de réécriture conditionnelle sont utilisées pour exprimer le protocole. Un langage restreint de propriétés a été défini permettant la vérification automatique d'invariants de l'algorithme (du style «deux noeuds différents ne peuvent pas utiliser simultanément la même page en écriture»). L'algorithme, qui fonctionne comme une analyse «arrière», rend la plus petite contrainte sur l'état initial pour garantir l'invariant recherché. La mise en oeuvre effective d'un protocole est ensuite réalisée grâce à l'ajout d'aspects (cf module 6.1.2) décrivant notamment l'ordre de traitement des opérations, leur localisation et les communications induites.

Ce travail fait l'objet de la thèse de David Mentré, co-encadrée par Thierry Priol (du projet Caps de l'Irisa) et Daniel Le Métayer.

6.1.5 Langage dédié pour machines parallèles

Participants : Pascal Fradet, Julien Mallet.

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

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 du coût de l'implantation parallèle. Le résultat de cette analyse de coût, qui prend en compte les temps de calcul et les temps de communication, conditionne des choix de compilation comme la distribution de données, l'ordonnancement, et certaines optimisations.

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). Récemment, nous avons travaillé sur une analyse de mise à jour en place, pré-requis indispensable à la mise en œuvre efficace des tableaux dans un langage fonctionnel. Nous nous sommes également attachés à enrichir la collection de patrons prise en compte.

6.1.6 Systèmes d'information logiques

Participants : Olivier Ridoux, Sébastien Ferré.

Mots clés : systèmes 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) 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».

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 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

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

mettre en oeuvre un système de fichiers basé sur l'analyse de concepts logiques de façon la plus générique possible.

6.2 Actions « aval »

Nous présentons d'abord des résultats généraux sur l'analyse de programmes (modules 6.2.1 et 6.2.2) avant de détailler un certain nombre de recherches portant sur des analyses particulières : l'analyse de programmes synchrones SIGNAL (module 6.2.3), la vérification de politiques de sécurité (module 6.2.4) et l'analyse de programmes λProlog (module 6.2.5). Nous décrivons ensuite nos travaux récents sur le le débogage dynamique (modules 6.2.6 et 6.2.7), la sémantique et la génération de traces (modules 6.2.8 et 6.2.9). Nous concluons avec la présentation de nos activités en matière de génération de jeux de test (module 6.2.10).

6.2.1 Construction systématique d'analyses génériques

Participants : Valérie Gouranton, Thomas Jensen, Daniel Le Métayer, Florimond Ployette, Gaëlle Segouat.

Mots clés : analyse de programme, sémantique naturelle, slicing, calcul de point fixe.

Résumé : *Certaines analyses de programmes ne sont pas restreintes à un langage de programmation particulier. Plutôt que d'en fournir une nouvelle définition pour chaque langage, nous avons montré qu'il est possible de définir une telle analyse de manière générique et de l'instancier pour obtenir des analyses particulières. Nous avons proposé pour ce faire un* format *de sémantique naturelle et nous avons appliqué cette technique à l'analyse de* slicing. *Nous avons pu ainsi dériver, par simple instanciation d'une définition générique, des analyses dynamiques et statiques pour un langage impératif, un langage fonctionnel et un langage de programmation logique.*

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. Il est ainsi très difficile de factoriser les efforts en matière de réalisation d'analyseurs. Nous avons abordé ce problème en considérant la construction d'analyseurs sous l'angle de la dérivation de programmes à partir de spécifications. Le programme en l'occurrence est l'analyseur lui-même et sa spécification est composée de deux parties :

1. La sémantique opérationnelle du langage de programmation décrite sous forme de sémantique naturelle.
2. La propriété recherchée exprimée sous forme de récurrences sur les arbres de preuves de la sémantique naturelle.

Les deux composants peuvent en fait s'exprimer sous forme de fonctions et la dérivation consiste en une série de transformations (pliage/dépliage) permettant d'obtenir un programme récursif autonome qui constitue l'analyseur.

L'intérêt de cette démarche est qu'elle permet d'exprimer dans un même cadre des analyses variées sur différents langages. Nous en avons fait la démonstration pour des analyses de programmes impératifs (durée de vie des variables), logiques (clôture des termes) et fonctionnels (nécessité, globalisation). Certaines de ces analyses sont spécifiques à un langage de programmation : on peut citer dans cette catégorie l'analyse de clôture pour la programmation logique. D'autres, comme le *slicing* (cf. module 3.2) ont un intérêt plus général. Plutôt que d'en fournir une nouvelle définition pour chaque langage, nous avons montré qu'il est possible de définir une telle analyse de manière générique et de l'instancier pour obtenir des analyses particulières. Nous avons proposé pour ce faire un *format* de sémantique naturelle et nous avons appliqué cette technique à l'analyse de *slicing*. Nous avons pu ainsi dériver, par simple instanciation d'une définition générique, des analyses dynamiques et statiques pour un langage impératif, un langage fonctionnel et un langage de programmation logique [15].

D'un point de vue plus pratique, nous travaillons maintenant à la mise au point d'un solveur de point fixe générique qui pourra servir de base pour le développement de nouveaux analyseurs. 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 différents 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. Cependant un algorithme naïf réalise des calculs inutiles car il ne tient pas compte des dépendances entre les variables du système. Dans un premier temps nous avons étudié le problème du choix de l'ordre d'évaluation des expressions en utilisant des informations de dépendance pour éviter au maximum les calculs inutiles.

6.2.2 Spécification et implantation d'analyses à partir de règles d'inférence

Participants : Frédéric Besson, Thomas Jensen.

Mots clés : analyse, typage, logique, modularité.

Résumé : *En nous appuyant sur des résultats récents concernant l'inférence de types d'intersection et de types polymorphes, nous avons conçu un algorithme qui permet de déduire pour chaque programme une propriété principale à partir de laquelle toute autre propriété démontrable par l'analyse peut être déterminée. Un avantage de cet algorithme est qu'il traite des fragments de programmes aussi bien que des programmes entiers. Il s'agit donc d'une démarche qui, à plus long terme, peut mener à un cadre général pour l'analyse modulaire.*

Exprimer une analyse par des règles d'inférence comme un système de typage non-standard présente l'avantage de la rendre plus compréhensible mais permet également de s'appuyer sur des algorithmes de typage connus. Dans des travaux antérieurs, nous avons conçu une méthode

efficace pour *vérifier* qu'un programme satisfait une propriété donnée. Se pose alors la question de savoir s'il est possible, étant donné seulement le programme, de trouver les propriétés satisfaites par ce programme, c'est-à-dire d'*inférer* les propriétés. En nous appuyant sur des résultats récents concernant l'inférence de types d'intersection et de types polymorphes, nous avons conçu un algorithme qui permet de déduire pour chaque programme une propriété principale à partir de laquelle toute autre propriété démontrable par l'analyse peut être déterminée. Un avantage de cet algorithme est qu'il traite des fragments de programmes aussi bien que des programmes entiers. Il s'agit donc d'une démarche qui, à plus long terme, peut mener à un cadre général pour l'analyse modulaire. Nous travaillons actuellement à étendre la démarche à d'autres analyses comme l'analyse de flot de contrôle et à d'autres langages notamment les langages à objets. Cette extension permettrait par exemple d'appliquer les techniques de vérification de propriétés de sécurité (décrites en section 6.2.4) aux programmes qui utilisent le mécanisme de chargement dynamique du code.

6.2.3 Analyse de programmes synchrones SIGNAL

Participants : Thomas Jensen, Frédéric Besson.

Mots clés : analyse de flot de données, polyèdres convexes, élargissement, interprétation abstraite.

Résumé : *Les langages synchrones comme Lustre et Signal ont été conçus pour décrire des systèmes réactifs manipulant des flots de données. La vérification de propriétés de sûreté de programmes synchrones se résume souvent à garantir que les valeurs d'une variable ne dépassent pas un certain seuil critique. Une analyse de flots de données qui déduit des relations entre variables d'un programme Signal a été conçue et prouvée. Son implantation se fait par calcul de point fixe dans le domaine des polyèdres convexes.*

Les langages synchrones comme Lustre et SIGNAL ont été conçus pour décrire des systèmes réactifs manipulant des flots de données. En collaboration avec J.-P. Talpin du projet EPATR, nous nous sommes intéressés à l'analyse de flots de données de programmes Signal [18]. La vérification de propriétés de sûreté de programmes synchrones se résume souvent à garantir que les valeurs d'une variable ne dépassent pas un certain seuil critique. Or, pour ce faire, il est en général nécessaire de prouver des propriétés plus complexes sur les relations qui existent entre plusieurs variables. L'analyse développée vise à trouver de telles relations de façon automatique. Afin de prouver la correction de l'analyse, une sémantique opérationnelle a été définie pour un noyau du langage SIGNAL. Celle-ci diffère des sémantiques antérieures par sa façon de modéliser l'état d'un programme avec une mémoire comme dans le cas des langages impératifs. Un système d'inférence définit l'ensemble des contraintes dont les solutions correspondent à un comportement possible du programme analysé. La résolution des 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.

6.2.4 Vérification de politiques de sécurité

Participants : Marc Eluard, Thomas Jensen, Daniel Le Métayer, Tommy Thorn.

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 automatique (et complète) de vérification de propriétés pour cette logique. L'idée de base de cette méthode est d'explorer un sous-ensemble fini de suites d'exécution dont la taille est déterminée par la complexité de la propriété à vérifier. Afin d'appliquer ce cadre à Java nous avons formalisé certains aspects de sa sémantique et 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 [27, 11].

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, inspection de pile, etc.). Nous avons par ailleurs proposé une méthode automatique (et complète) de vérification de propriétés pour cette logique [27]. L'idée de base de cette méthode est d'explorer un sous-ensemble fini de suites d'exécution dont la taille est déterminée par la complexité de la propriété à vérifier.

L'application de ce cadre à Java est conditionnée par la formalisation du langage (tout du moins des aspects qui ont un impact sur la sécurité). L'une des particularités de Java est la possibilité de télécharger du code et de l'exécuter de manière transparente. Cette pratique soulève de sérieux problèmes en matière de sécurité des informations (confidentialité, intégrité notamment). Nous nous sommes donc attaqués à la formalisation de cet aspect en nous focalisant sur les règles de visibilité (des classes et de leurs membres) et leur évolution lors du chargement dynamique de classes [11]. 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 montré comment la politique de sécurité du dernier environnement de développement de Java (JDK 1.2) peut se formaliser dans notre logique. La technique citée plus haut permet donc de vérifier automatiquement qu'une application Java satisfait la politique affichée. Nous travaillons actuellement à lever certaines restrictions de la méthode (concernant notamment la récursivité mutuelle) et à en dériver un analyseur modulaire (mieux adapté à un environnement ouvert comme celui de Java).

6.2.5 Analyse statique de programmes λProlog

Participant : Olivier Ridoux.

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

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 booléens. Dans ce cadre, nous avons proposé une solution au deuxième problème publiée en 1998 [MRB98]. Un développement complémentaire et qui concerne aussi le domaine de calcul de λProlog consiste à prendre en compte le type des termes pour la constitution du domaine abstrait dans lequel s'exprime la propriété analysée [32]. 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 domaine d'abstraction, sa combinaison avec les fonctions de transfert et l'analyse de propriétés spécifiques à λProlog, comme la *beta*-normalité, sont complètement développés dans la thèse de Frédéric Malésieux [Mal99] (co-encadrée par Olivier Ridoux et Patrice Boizumault de l'École des Mines de Nantes), dont la soutenance a eu lieu au début de l'année 1999.

6.2.6 Explications pour les bases de données déductives

Participants : Mireille Ducassé, Sarah Mallet.

Mots clés : débogage, explication, génération de trace, analyse de trace, base de données déductive.

Résumé : *Une base de données déductive est composée de faits (de base) et de règles de déduction déclaratives permettant d'augmenter la connaissance par des faits déduits. Ces règles permettent aux concepteurs de la base de données de se concentrer sur sa logique. Toutefois, les utilisateurs des bases de données ont besoin d'explications pour leur restituer cette logique qui peut être masquée par les détails opérationnels de la mise en œuvre. Les systèmes d'explication existants rendent à l'utilisateur des arbres de preuve qui ne correspondent pas toujours au bon profil d'exécution. Nous proposons une technique de traçage qui consiste à intégrer une trace « relationnelle » avec un méta-interprète instrumenté utilisant des ensembles de substitutions. Les aspects coûteux de la méta-interprétation sont réduits par l'utilisation de la trace qui permet d'éviter de nombreux calculs. La flexibilité de la méta-interprétation est conservée. Elle permet de produire facilement des traces de profils différents.*

Une base de données déductive est composée de deux types de données : des faits de base stockés dans une base de données relationnelle et des données, dites virtuelles, déduites des données de la base à l'aide de règles.

La forme déclarative du langage de règles permet aux concepteurs de la base de données de se concentrer sur sa logique. Toutefois, la mise en œuvre des bases de données déductives

[MRB98] F. Malésieux, O. Ridoux, P. Boizumault, « Abstract compilation of λProlog », *in: Joint Int. Conf. and Symp. Logic Programming*, J. Jaffar (éditeur), MIT Press, 1998.

[Mal99] F. Malésieux, *"Contribution à l'analyse statique de programmes LambdaProlog" , Université de Nantes*, thèse de doctorat, 1999.

fait intervenir de nombreuses optimisations afin d'obtenir des manipulations de données efficaces. De ce fait, les utilisateurs des bases de données ont besoin de facilités de débogage et d'explication pour restituer la logique des programmes qui peut être masquée par les détails opérationnels (cf. module 3.3).

Ces explications sont particulièrement nécessaires dans le contexte des bases de données déductives où les utilisateurs du logiciel doivent comprendre le déroulement des déductions pour en accepter les résultats ou mettre à jour les données, alors qu'ils ne sont pas forcément des informaticiens.

Les systèmes d'explication existants pour les bases de données déductives rendent à l'utilisateur des arbres de preuve. Ces arbres donnent une vision très détaillée de la manipulation des données. Cette présentation éclatée, qui peut être utile dans certains cas, provoque, dans le cas général, une explosion du nombre d'arbres de preuve produits. Il s'agit donc de concevoir un outil capable de présenter différents points de vue aux utilisateurs.

Nous proposons une technique de traçage qui consiste à intégrer une trace « relationnelle » à un méta-interprète instrumenté utilisant des ensembles de substitutions. La trace relationnelle fournit, de manière efficace, une information précise sur l'extraction de données de la base relationnelle. Le méta-interprète ensembliste gère des ensembles de substitutions et donne des explications sur la déduction. Les aspects coûteux de la méta-interprétation sont réduits par l'utilisation de la trace qui permet d'éviter de nombreux calculs. La flexibilité de la méta-interprétation est conservée. Elle permet de produire facilement des traces de profils différents [34, 30, 29, 10].

Ce travail fait l'objet d'une coopération avec Next Century Media. Cette société fournit Validity, un produit reposant sur la technologie des bases de données déductives, et vise les applications dans la publicité ciblée. Un prototype de recherche, développé dans Lande, met en œuvre la technique de traçage mentionnée ci-dessus en utilisant des traces relationnelles produites par le traceur de Validity.

6.2.7 Analyse de trace automatisée

Participants : Mireille Ducassé, Erwan Jahier.

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

Résumé : *Nous avons développé un outil, Opium, qui analyse les traces d'exécution de programmes générées par un traceur Prolog existant. 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. Si le contenu de la trace utilisée et les analyses proposées sont dédiés à Prolog, les techniques de base exploitent une trace dont le seul pré-requis est d'être séquentielle. Nous avons mis à profit cette généralité pour concevoir deux nouveaux prototypes de recherche qui appliquent ces idées aux langages Mercury et C.*

Nous avons développé un outil, Opium [14], qui analyse les traces d'exécution de programmes générées par un traceur Prolog existant (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 spécifier de manière précise ce qu'ils veulent observer du comportement du programme.

Ces requêtes peuvent être traitées à la volée ou *a posteriori*. Dans le cas d'un traitement à la volée, les performances du système permettent d'analyser plusieurs millions d'événements avec des temps de réponse supportables. L'analyse *a posteriori* permet de traiter un ensemble plus riche de requêtes mais nécessite de stocker les événements dans une base de données interne, ce qui prend un temps non négligeable. L'analyse proprement dite, par contre, ne prend pas plus de temps que l'analyse à la volée. Du point de vue de l'utilisateur, les requêtes se posent de la même manière dans les deux cas.

À partir du langage de requêtes, nous avons bâti des analyses qui fournissent des vues abstraites des exécutions selon certains critères (par exemple, flot de contrôle ou flot de données). Il a également été possible de mettre en œuvre à faible coût des traceurs abstraits pour des langages de haut niveau, ce qui a permis de mettre au point facilement des démonstrations d'applications sophistiquées [13].

Si le contenu de la trace utilisée et les analyses proposées sont dédiés à Prolog, les techniques de base pour mettre en œuvre le langage de requêtes exploitent une trace qui peut être produite pour n'importe quel langage séquentiel. Nous avons mis à profit cette généralité pour concevoir des débogueurs pour les langages Mercury [33] (cf. module 7.1) et C [20].

A partir du prototype d'Opium pour Mercury nous avons montré 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 au contraire, l'instrumentation étant générique, elle est faite une fois pour toutes et les analyses spécifiques peuvent être relativement simples. Chacun des exemples développés nécessite moins d'une dizaine de lignes de Prolog [26, 25].

6.2.8 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 soit 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 continuations. 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 [24].

6.2.9 Génération de traces

Participante : Mireille Ducassé.

Mots clés : débogage, traceur, transformation de programmes, 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. Ceci 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.

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

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 » [12]. Cette réduction de performance est acceptable pour un traceur « standard » surtout lorsque l'on considére le gain significatif en termes de portabilité.

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

6.2.10 Génération systématique de jeux de test

Participants : Daniel Le Métayer, Valérie-Anne Nicolas, Olivier Ridoux, Lionel Van Aertryck.

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 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 est 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 [28] (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 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 [28].

7 Contrats industriels (nationaux, européens et internationaux)

7.1 Action Argo

Participants : Mireille Ducassé, Erwan Jahier.

Mots clés : environnement de programmation, Mercury, programmation logique, débogage.

Résumé : *L'objectif du projet Argo était de mettre en œuvre un environnement industriel de développement de programmes logiques Mercury. Deux applications industrielles existantes ont été portées sur le nouvel environnement afin de le valider.*

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.

Ces applications concernent la facturation hospitalière et le diagnostic de matériel défectueux. Lande est plus particulièrement concerné par les problèmes de débogage.

Le projet Lande a participé au 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) qui a été lancé en novembre 1997 et qui s'est terminé avec succés fin juin 1999.

Les déclarations de type, de mode et de déterminisme font de Mercury un langage produisant un code à la fois plus efficace et plus sûr que les langages de programmation logique actuels. Les parties déterministes des programmes sont aussi rapides que leurs équivalents en C. De plus, beaucoup de fautes sont détectées dès la compilation. L'expérience de nos partenaires industriels montre, cependant, que les fautes résiduelles sont d'autant plus difficiles à localiser et comprendre qu'elles sont peu nombreuses. Un outil de débogage de haut niveau était donc nécessaire.

L'objectif du projet ARGo était de mettre en œuvre un environnement industriel de développement de programmes logiques Mercury. Une version de Mercury a été portée sur Windows NT par nos partenaires industriels. Deux applications industrielles existantes ont été portées sur cette nouvelle version afin de la valider. Ces applications concernent la facturation hospitalière et le diagnostic de matériel défectueux. Lande est plus particulièrement concerné par les problèmes de débogage.

Nous avons mis en oeuvre un prototype d'analyse de trace (cf. module 6.2.7) qui est actuellement distribué (cf. module 5).

Les partenaires industriels du consortium sont Mission Critical (Belgique) et Dassault Electronique (France). Mission Critical est le coordinateur du projet. Nos partenaires universitaires sont Katholieke Universiteit Leuven (Belgique), University of Melbourne (Australie), Facultés Universitaires Notre Dame de la Paix à Namur (Belgique).

7.2 Actions Two et ITR

Participants : Daniel Le Métayer, 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.

Notre action dans le cadre du programme ITR est une collaboration entre les sociétés Soft-Maint (du groupe Sodifrance), AQL et l'Irisa. Elle porte sur la génération automatique de jeux de tests pour des applications Cobol. Les principales applications de Soft-Maint se trouvent en effet dans le secteur des banques et des assurances où de forts besoins se font sentir à l'heure actuelle pour la génération de jeux de tests dans un cadre de rétro-ingénierie ou de rénovation de logiciels (passage à l'an 2000, Euro, migrations, ...). Soft-Maint dispose déjà de frontaux et d'outils d'analyse pour Cobol. Sa tâche consiste à les adapter et à les étendre pour fournir des graphes d'états symboliques qui se prêtent à une génération de jeux de tests. Cette génération est ensuite effectuée selon des stratégies propres aux applications Cobol et aux objectifs des testeurs.

7.3 Action Java-Sécurité

Participants : Marc Eluard, Pascal Fradet, Thomas Jensen, Daniel Le Métayer, Tommy Thorn.

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. Nous nous sommes focalisés sur les règles de visibilité (des classes et de leurs membres) et leur évolution lors du chargement dynamique de classes. Nous avons ensuite proposé un cadre pour spécifier des politiques de sécurité basé sur une logique temporelle linéaire*

à deux niveaux. Nous avons conçu une méthode pour vérifier qu'un programme Java, représenté comme un système de transition, satisfait une politique exprimée dans ce cadre. La méthode repose sur un résultat qui permet de limiter la taille du système en fonction de la propriété à vérifier.

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 6.2.4). 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 proposé un cadre pour spécifier des politiques de sécurité basé sur une logique temporelle linéaire à deux niveaux. Nous avons conçu une méthode pour vérifier qu'un programme Java, réprésenté comme un système de transition, satisfait une politique exprimée dans ce cadre. La méthode repose sur un résultat qui permet de limiter la taille du système en fonction de la propriété à vérifier. Notre objectif actuel est d'étudier l'effet d'une politique (c'est à dire les exigences qu'elle impose) sur les composants d'une application Java (comme le chargeur de classes).
d'optimisations pour Java Card

7.4 Action Vérification d'optimisations pour Java Card

Participants : Ewen Denney, Pascal Fradet, Thomas Jensen, Daniel Le Métayer, Gaëlle Segouat, Tommy Thorn.

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

Résumé : *Dans le cadre d'une collaboration industrielle avec le fabricant de cartes à puces Bull-*CP8*, nous avons conçu et breveté une méthode qui permet de prouver la correction des transformation effectuées sur le code intermédiaire Java Card pour permettre son installation sur une carte à puce.*

Notre effort de formalisation des différents aspects de Java et de sa mise en œuvre se concrétise également à travers une collaboration avec le fabricant de cartes à puces Bull-CP8 sur la vérification d'optimisations pour Java Card (ref. Inria : 1 98 C 144).

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» 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. 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.

Le procédé de validation a fait l'objet d'un brevet «Procédé de vérification de transformateurs de codes pour un système embarqué, notamment sur une carte à puce», déposée en juillet 1999 sous le numéro 99 08460 [35].

7.5 Action Castor

Participants : Pascal Fradet, Thomas Jensen.

Mots clés : sécurité, architecture, logiciel, composant, politique, propriété, 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 AQL et TNI et les projets EPATR 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 permettra de fournir les conditions qui doivent être vérifiées par les composants pour assurer des contraintes de sécurité globales. Elle permettra également d'identifier l'impact d'un changement de configuration sur la sécurité du système. Cette action se situe dans le prolongement des travaux en cours sur la définition de politiques de sécurité et sur la vérification de cohérence d'architectures à vues multiples.

7.6 Action Esaps

Participant : Pascal Fradet.

Mots clés : architecture, logiciel, cohérence, UML, aspect, famille, ligne de produit, spécialisation.

Résumé : *L'objectif du projet* ESAPS *est de développer une infrastructure permettant de définir et de gérer des familles d'architectures de logiciels correspondant à des lignes de produits. Ces familles d'architectures doivent exprimer les caractéristiques communes à un ensemble de logiciels et les moyens de les spécialiser pour des plates-formes ou des applications particulières.*

Lancée en juillet 1999, l'action ESAPS se déroule dans le cadre du projet Eureka ITEA. Elle regroupe une nombre important d'industriels européens (notamment Philips, Siemens, Thomson, Alcatel, ESI, Bosch Gmbh, etc.) soucieux de développer une infrastructure permettant de définir et de gérer des familles d'architectures de logiciels correspondant à des lignes de produits. Ces familles d'architectures doivent exprimer les caractéristiques communes à un ensemble de logiciels et les moyens de les spécialiser pour des plates-formes ou des applications particulières. Une telle description apporterait de nombreux bénéfices aussi bien en terme de réactivité (délais de développements réduits) que de maîtrise de gros logiciels comportant de multiples variantes. Les domaines d'application privilégiés de ESAPS sont les transports (systèmes de contrôle notamment) et le secteur médical. L'intervention du projet Lande dans ESAPS porte sur deux points: l'application de nos techniques de vérification de cohérence à des familles d'architectures décrites en UML et l'étude de l'intérêt de la notion d'aspects pour le développement de telles applications.

8 Actions régionales, nationales et internationales

8.1 Actions régionales

Le projet Lande participe avec AQL et Sodifrance à un projet soutenu par le programme ITR de la Région Bretagne (cf module 7.2).

Mireille Ducassé fait partie depuis quatre ans du jury régional du prix de la vocation féminine. Ce prix récompense des jeunes filles qui s'orientent vers des études scientifiques après le baccalauréat.

8.2 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, Croap, 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.

8.3 Actions financées par la Commission Européenne

Le projet Lande participe aux projets européens Argo, Two, ESAPS et Coordina. Les trois premiers sont décrits dans la section Actions industrielles (modules 7.1, 7.2 et 7.6 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, Leiden, Lisbonne, Londres (Imperial College) et Pise.

8.4 Réseaux et groupes de travail internationaux

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

8.5 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.

Le projet a également entamé une collaboration avec Bernhard Steffen à l'université de Dortmund à travers le co-encadrement d'une thèse sur les méthodes de résolution de systèmes d'équations et la recherche de points fixes (voir section 6.2.1).

8.6 Accueil de chercheurs étrangers

Le projet a reçu Peter Bertelsen, de l'école royale vétérinaire du Danemark, qui a travaillé sur l'analyse et la spécialisation du byte code Java. 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 est membre du chapitre français du BUG (B User Group) et de l'Affi (Association Française des Femmes Ingénieurs). Elle a été membre du comité de programme de LOPSTR-99 (LOgic-based Program Synthesis and TRansformation). Elle a organisé WLPE-99 (10th Workshop on Logic Programming Environments) dont elle a également édité les actes [9]. Elle est membre élu du conseil scientifique de l'Insa de Rennes depuis septembre 1994.

Daniel Le Métayer a été membre des comités de programme des conférences IEEE ACM ICSE'99 et IEEE ACM ICSE'2000 (International Conference on Software Engineering, Los Angeles et Limerick), AFADL'2000 (Approches Formelles dans l'Assistance au Développement de Logiciel, Grenoble). Il est également éditeur d'un numéro spécial de la revue *Theoretical Computer Science* consacré à la coordination et la sécurité [16]. Par ailleurs, il a été rapporteur sur les thèses de François Pessaux (préparée sous la responsabilité de Xavier Leroy) et de Frédéric Prost (préparée sous la responsabilité de Pierre Lescanne).

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 fait également partie du comité de rédaction de la revue TSI. Il a fait partie des comités de programme de JFPLC99 (Journées Francophones de Programmation Logique et par Contraintes) et IDL99 (*International Workshop on Implementation of Declarative Languages*). Il a par ailleurs été rapporteur sur la thèse de Mathieu Jaume (*Contributions à la sémantique de la programmation logique*, Cermics, École National des Ponts et Chaussées) préparée sous la direction de René Lalement.

9.2 Enseignement universitaire

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

Olivier Ridoux a fait partie du comité de réflexion pour le renouvellement de l'habilitation du DEA d'informatique de l'Ifsic. Il est devenu responsable de la maîtrise d'informatique de l'université de Rennes 1.

Le projet a encadré quatre étudiants de DEA : Thierry Chapelet, Thomas Colcombet, Sébastien Ferré et Gaëlle Segouat. Par ailleurs, le projet a reçu

- Gautam Gupta, de l'Indian Institute of Technology, pour un stage d'été concernant l'analyse et la vérification de sécurité en Java.
- Mitali Singh, de l'Indian Institute of Technology, et Cristelle Lecomte, de l'université de Rennes 1, pour un stage d'été concernant l'analyse modulaire de programmes fonctionnels

9.3 Participation à des colloques, séminaires, invitations

Mireille Ducassé a présenté ses recherches sur l'analyse de trace automatisée au séminaire de DEA des Facultés Universitaires de Namur, Belgique. Elle a donné un tutoriel invité sur la méthode formelle B à LOPSTR-99 [21] ainsi qu'aux journées pédagogiques de l'Ifsic, université de Rennes 1.

Mireille Ducassé, Erwan jahier et Sarah Mallet ont participé à un séminaire prospectif sur le débogage organisé par P. Deransart, Inria Rocquencourt.

Thomas Jensen a participé à la réunion du Working Group IFIP 2.8 sur la programmation fonctionnelle (St-Malo).

Daniel Le Métayer a participé aux workshops WICSAW (Working Ifip Conference on Software Architecture, San Antonio) et CSFW (Computer Security Foundation Workshop, Mordano). Par ailleurs il a présenté les travaux du projet sur les politiques de sécurité au SRI (Palo Alto) et sur les architectures de logiciels à l'université de L'Aquila.

Julien Mallet a présenté ses travaux sur la compilation de langages spécialisés pour le parallélisme aux séminaires "High Level Parallel Programming: Applicability, Analysis and Performance" (Dagstuhl, Allemagne) et "Compilation et Parallélisation Automatique" sponsorisé par le thème iHPerf du GDR ARP (Saint Nabor, Bas-Rhin).

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] S. Coupet-Grimal, O. Ridoux, « On the use of advanced logic programming languages in computational linguistics », *Journal of Logic Programming 24*, 1&2, 1995, p. 121–159.

[4] M. Ducassé, J. Noyé, « Logic programming environments: dynamic program analysis and debugging », *Journal of Logic Programming 19/20*, mai/juillet 1994, p. 351–384.

[5] P. Fradet, D. Le Métayer, « Compilation of functional languages by program transformation », *ACM Transactions on Programming Languages and Systems 13*, 1, 1991, p. 21–51.

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

[7] D. Le Métayer, « Describing software architecture styles using graph grammars », *IEEE Transactions on Software Engineering 24*, 7, juillet 1998, p. 521–533.

[8] 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

[9] M. Ducassé, A. Kusalik, G. Puebla (éditeurs), *Proceedings of the 10th Workshop on Logic Programming Environments*, http://www.cs.usask.ca/projects/envlop/WLPE/10WLPE/, novembre 1999.

Thèses et habilitations à diriger des recherches

[10] S. Mallet, *Explications dans les bases de données déductives : Associer trace et sémantique*, thèse de doctorat, Insa de Rennes, Irisa, novembre 1999.

[11] T. Thorn, *Vérification de politiques de sécurité par analyse de programmes*, thèse de doctorat, Université de Rennes I, Ifsic, Irisa, février 1999.

Articles et chapitres de livre

[12] M. Ducassé, J. Noyé, « Tracing Prolog programs by source instrumentation is efficient enough », *Journal of Logic Programming*, 2000, à paraître.

[13] M. Ducassé, « Abstract views of Prolog executions with Opium », *in : Learning to Build and Comprehend Complex Information Structures: Prolog as a Case Study*, P. Brna, B. du Boulay, et H. Pain (éditeurs), *Cognitive Science and Technology*, Ablex, 1999, ch. 10, p. 223–243, (aussi RR-3531 INRIA).

[14] M. Ducassé, « Opium: An extendable trace analyser for Prolog », *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).

[15] V. Gouranton, D. Le Métayer, « Dynamic slicing: a generic analysis based on a natural semantics format », *Journal of Logic and Computation 9*, 6, décembre 1999.

[16] D. Le Métayer, « Foreword to the special issue on coordination and mobility », *Theoretical Computer Science 240*, 1, 1999.

[17] D. Mentré, D. Le Métayer, T. Priol, « Conception de protocoles de cohérence de MVP par traduction d'une spécification Gamma à l'aide d'aspects », *Calculateurs Parallèles 11*, 2, 1999.

Communications à des congrès, colloques, etc.

[18] F. Besson, T. Jensen, J.-P. Talpin, « Polyhedral analysis for synchronous languages », *in : Proc. of 7th Int. Symp. on Static Analysis*, G. Filé (éditeur), *Lecture Notes in Computer Science*, Springer-Verlag, septembre 1999.

[19] T. Colcombet, P. Fradet, « Enforcing trace properties by program transformation », *in : Proc. of Principles of Programming Languages*, Boston, janvier 2000.

[20] 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. (aussi RR-3489 INRIA).

[21] M. Ducassé, « An introduction to the B formal method », *in : Proceedings of the 9th International Workshop on LOgic-based Program Synthesis and TRansformation*, A.-L. Bossi (éditeur), Universita' Ca' Foscari di Venezia, p. 23–30, septembre 1999. Technical report CS-99-16.

[22] P. Fradet, D. Le Métayer, M. Périn, « Consistency checking for multiple view software architectures », *in : 7th ACM SIGSOFT Symp. on the Foundations of Software Engineering*, *Lecture Notes in Computer Science*, Springer-Verlag, p. 410,428, septembre 1999.

[23] P. Fradet, M. Südholt, « An aspect language for robust programming », *in : Workshop on Aspect-Oriented Programming, ECOOP 1999*, juillet 1999.

[24] E. Jahier, M. Ducassé, O. Ridoux, « Specifying trace models with a continuation semantics », *in : Proc. of ICLP'99 Workshop on Logic Programming Environments*, M. Ducassé, A. Kusalik, G. Puebla (éditeurs), novembre 1999.

[25] E. Jahier, M. Ducassé, « A generic approach to monitor program executions », *in : Proceedings of the International Conference on Logic Programming*, D. D. Schreye (éditeur), MIT Press, novembre 1999.

[26] E. Jahier, M. Ducassé, « Un traceur d'exécution ne sert pas qu'au débogage », *in : Actes des journées francophones de programmation logique et programmation par contraintes*, F. Fages (éditeur), Hermes, p. 297–311, juin 1999.

[27] 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.

[28] D. Le Métayer, V.-A. Nicolas, O. Ridoux, « Verification by testing for recursive programs », *in : Preliminary Proceedings of the 9th International Workshop on LOgic-based Program Synthesis and TRansformation*, septembre 1999.

[29] S. Mallet, M. Ducassé, « Generating deductive database explanations », *in : Proceedings of the International Conference on Logic Programming*, D. D. Schreye (éditeur), MIT Press, novembre 1999.

[30] S. Mallet, M. Ducassé, « Myrtle: A set-oriented meta-interpreter driven by a "relational" trace for deductive database debugging », *in : LOgic-based Program Synthesis and TRansformation*, P. Flener (éditeur), Springer-Verlag, LNCS 1559, p. 328–330, 1999. Résumé, version complète en RR-3598 INRIA.

[31] D. Mentré, D. Le Métayer, T. Priol, « Towards designing SVM coherence protocols using high-level specifications and aspect-oriented translations », *in : Proceedings of ICS'99 Workshop on Software Distributed Shared Memory*, juin 1999.

[32] O. Ridoux, P. Boizumault, F. Malésieux, « Typed static analysis: application to groundness analysis of LambdaProlog and Prolog », *in : Proc. of the Int. Symp. on Functional and Logic Programming*, A. Middeldorp, T. Sato (éditeurs), Springer, 1999.

Rapports de recherche et publications internes

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

[34] S. Mallet, M. Ducassé, « Myrtle: A set-oriented meta-interpreter driven by a "relational" trace for deductive database debugging », *Research Report n° RR-3598*, INRIA, http://www.inria.fr/RRRT/RR-3598.html, janvier 1999.

Divers

[35] E. Denney, P. Fradet, C. Goire, T. Jensen, D. Le Métayer, « Procédé de vérification de transformateurs de codes pour un système embarqué, notamment sur une carte à puce », juillet 1999, Brevet d'invention.