INRIA

Rapport d'activité scientifique 1997

Projet LANDE

Conception et validation de logiciels

Thème Inria 2
Génie logiciel et calcul symbolique
(Version du 10 février 1998)

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Table des matières

1 Composition de l'équipe 4
2 Présentation générale et objectifs 5
2.1 Actions « aval » 5
2.2 Actions « amont » 6
3 Fondements scientifiques 7
3.1 Sémantique des langages de programmation 7
3.2 Analyse de programmes 9
3.3 Débogage 11
3.4 Test de logiciels 13
3.5 Langages déclaratifs 15
4 Domaines d'applications 16
5 Logiciels 17
6 Résultats nouveaux 18
6.1 Actions « aval » 18
6.1.1 Construction systématique d'analyses génériques . . 18
6.1.2 Spécification et implantation d'analyses à partir de règles d'inférence 19
6.1.3 Vérification de propriétés de sécurité de programmes Java 20
6.1.4 Analyse statique de programmes λProlog 21
6.1.5 Débogage statique 22
6.1.6 Explications pour les bases de données déductives . 23
6.1.7 Analyse de trace automatisée 24
6.1.8 Génération systématique de jeux de test 25
6.2 Actions « amont » 26
6.2.1 Implantation des langages fonctionnels 26
6.2.2 Les types graphes 28
7 Actions industrielles 31
7.1 Action ARGo 31
7.2 Action CASTING 32
7.3 Action Java-Sécurité 33
8 Actions régionales, nationales et internationales 34
8.1 Actions régionales 34
8.2 Actions nationales 34
8.3 Actions financées par la Commission Européenne 34
8.4 Réseaux et groupes de travail internationaux 34
8.5 Relations bilatérales internationales 35

8.6 Accueil de chercheurs étrangers 35
9 Diffusion de résultats 35
9.1 Animation de la communauté scientifique 35
9.2 Enseignement universitaire 36
9.3 Participation à des colloques, séminaires, invitations 36
10 Bibliographie 37

Projet LANDE

Conception et validation de logiciels

Localisation : *Rennes*

Mots-clés : Logiciel, Environnement de programmation, Langage déclaratif, Base de donnée déductive, Architecture de logiciel, Sémantique, Analyse de programmes, Typage, Correction, Sécurité, Java, Vérification, Aide à la mise au point, Débogage, Test.

1 Composition de l'équipe

Responsable scientifique

Daniel Le Métayer, DR Inria

Assistant(e) de projet

Isabelle Ballon, vacataire Inria, jusqu'en septembre 1997
Claire Street, vacataire Inria, depuis octobre 1997

Personnel Inria

Pascal Fradet, CR
Florimond Ployette, IR, Atelier
Olivier Ridoux, CR

Personnel CNRS

Thomas Jensen, CR

Personnel Insa

Mireille Ducassé, Professeur
Anne Alexandra Holzbacher, ATER, jusqu'en septembre 1997

Chercheurs doctorant

Jeanne Berthélemy, PRAG, Université de Bretagne Sud
Ronan Gaugne, boursier Inria-Région, jusqu'en septembre 1997
Valérie Gouranton, allocataire MRT, puis ATER depuis septembre 1997
Erwan Jahier, allocataire MRT, depuis octobre 1997
Julien Mallet, allocataire MRT, service national de février à novembre 1997
Sarah Mallet, allocataire MRT

Valérie-Anne Nicolas, allocataire MRT
Michaël Périn, boursier Inria-Région
Siegfried Rouvrais, allocataire MRT, depuis novembre 1997, en co-encadrement avec le projet SOLIDOR
Tommy Thorn, boursier Inria
Lionel Van Aertryck, boursier Cifre, jusqu'en octobre 1997

Chercheurs post-doctorant

Mario Südholt, boursier MAE, jusqu'en septembre 1997

2 Présentation générale et objectifs

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 :

1. Elles reposent sur des bases formelles (sémantique de langage, modèle de sécurité, etc.) permettant d'apporter des garanties quant à la correction des outils.
2. 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 traitements postérieurs à la phase de programmation (« actions aval ») et ceux qui la concernent ou la précèdent (« actions amont »).

2.1 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 mise au point de programmes qui reposent essentiellement sur des *analyses de programmes* (statiques ou dynamiques). Par exemple, nous avons proposé une technique permettant la détection d'accès incorrects à la mémoire (déréférences de pointeurs invalides). Cette analyse est extrêmement précise et permet un certain nombre d'interactions avec l'utilisateur. Nous nous intéressons également à l'analyse dynamique, qui sert de base pour l'aide à la mise au point dans le cadre de la programmation logique et des bases de données déductives.

Nous considérons aussi l'analyse sémantique dans une perspective plus large, notamment en étudiant la construction systématique d'analyseurs reposant sur des sémantiques opérationnelles de langages. En particulier, nous avons proposé un *format* de sémantique naturelle et nous avons appliqué notre méthode 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.

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 test de logiciels représente un autre centre d'intérêt important du projet. Nous avons proposé une méthode de génération de suites de tests qui a conduit à 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 (ou « boîte blanche ») que fonctionnel (ou « boîte noire »). Les suites de tests engendrées dépendent de stratégies spécifiées par l'utilisateur, permettant ainsi d'atteindre la souplesse d'utilisation exigée pour un usage industriel. Nous nous intéressons conjointement à la manière d'assurer que des programmes sont conformes à des hypothèses de test, ce qui permet de faire le lien avec la problématique « vérification de programmes ».

2.2 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. Les actions de cette catégorie se traduisent notamment par des méthodes et des langages qui induisent une *discipline de programmation*. Nous étudions dans ce cadre des langages de haut niveau comme les langages déclaratifs (logiques, fonctionnels) ou le langage Gamma conçu à l'Irisa. Ces langages fournissent un pouvoir d'expression appréciable tout en offrant un niveau de description des programmes qui facilite le raisonnement formel (cf. module 3.5). S'agissant des langages déclaratifs, les enjeux les plus importants concernent leur mise en œuvre efficace et la conception d'environnements de programmation adéquats. Ces deux problèmes sont abordés dans le projet LANDE. En particulier, nous avons proposé un cadre formel pour décrire et comparer les techniques de compilation des langages fonctionnels. Cela nous a permis d'établir une taxonomie des mises en œuvre de langages fonctionnels. Nous avons également étudié l'expression d'optimisations classiques et la conception de machines hybrides (i.e. intégrant plusieurs choix de compilation). Pour ce qui concerne les environnements de programmation, nous avons développé un outil, OPIUM, qui analyse les traces d'exécution de programmes générées par un traceur Prolog existant ; nous concevons actuellement un traceur pour Mercury, un nouveau langage de programmation logique développé à l'université de Melbourne.

Les langages déclaratifs possèdent des atouts majeurs qui devraient leur permettre de jouer un rôle plus important dans l'avenir, tout du moins dans certains secteurs d'application. Cependant, ils ne sont cependant pas encore très répandus en milieu industriel. Un autre moyen d'améliorer les pratiques en matière de programmation consiste à apporter des améliorations à des langages largement utilisés. On peut par exemple les augmenter pour y introduire des traits favorisant une programmation plus robuste. Nous avons exploré cette voie en proposant Shape-C, une extension de

C intégrant la notion de type graphe. On peut y décrire précisément des structures classiques comme les listes circulaires ou les listes doublement chaînées. La vérification de type permet de détecter de nombreuses erreurs de programmation et la manipulation de pointeurs devient plus sûre.

Une autre action importante dans la catégorie « amont » concerne les architectures logicielles. 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. La description d'une application est séparée en deux niveaux bien identifiés : d'une part, l'ensemble de ses entités de base qui représentent des calculs autonomes ; d'autre part la coordination de ces entités. Les entités individuelles peuvent être décrites dans des langages traditionnels (par exemple séquentiels) et la coordination est assurée par un composant défini séparément : le coordinateur.

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 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 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

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

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

technique de preuve classique associée à ce type de sémantique est la récurrence sur le nombre d'étapes de calcul.

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

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

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

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

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.

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

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.

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, pages 22–39. Springer Verlag, 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 toujours utilisés par 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 [AH87] 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.

[BGL93] B. Bruegge, T. Gottschalk, and B. Luo. A framework for dynamic program analyzers. In *Proc. of the OOPSLA'93 Conference*, pages 65–82, 1993.

[AH87] S. Abramsky and C. Hankin. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

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

3.3 Débogage

Mots-clés : Environnement de programmation, Analyse de programme, Sémantique.

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

[Cou97] P. Cousot. Abstract interpretation based static analysis parameterized by semantics. In P. Van Hentenryck, editor, *Proc. of 4th Static Analysis Symposium*, pages 388–394. Springer Verlag, LNCS vol. 1302, 1997.

[ANS] Ansi/ieee standard 729-1983. Glossary of Software Engineering Terminology.

de phases de test (cf module 3.4) ou lors d'une phase de vérification formelle. La première situation, la plus fréquente dans la pratique actuelle, correspond à ce qu'on appelle le *débogage dynamique* ; la seconde sera qualifiée de *débogage statique*. Dans les deux cas, l'objectif visé est de faire cesser les pannes identifiées.

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

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

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

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

3.4 Test de logiciels

Mots-clés : Critère de test, Hypothèse de test, Test unitaire, Test d'intégration, Test fonctionnel, Test système, Test en boîte noire, Test en boîte blanche, Test structurel.

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*

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

> *constituant un jeu de test satisfaisant. Le bénéfice d'une telle systématisation est double : d'une part les jeux de test ainsi produits sont de meilleure qualité que ceux que peut inventer un testeur (et justifiés par rapport à un critère précis) ; d'autre part, la possibilité de mécaniser le procédé (au moins partiellement) apporte des gains significatifs en terme de productivité.*

On distingue généralement quatre types de tests, chacun étant lié à l'une des phases de conception des logiciels. Les premiers tests soumis au logiciel ont pour cible les composants élémentaires de l'application à tester. Pour cette raison, ils sont appelés *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

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

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

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 : Langages fonctionnels, Langages de programmation logique, Correction, Efficacité, Évolutivité, Maintenance.

Résumé :

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

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

[Ros84] J.B. Rosser. Highlights of the history of the lambda-calculus. *Annals of the History of Computing*, 6(4), 1984.

langages : les entités de base (fonctions ou prédicats) peuvent ainsi être manipulées directement comme des objets mathématiques.

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

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

4 Domaines d'applications

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

1. *Les applications qui exigent un degré de confiance très important justifiant l'emploi de techniques reposant sur des méthodes formelles. Il peut s'agir de logiciels critiques pour la confidentialité ou l'intégrité des informations, la sécurité des personnes.*
2. *Les logiciels complexes ou qui nécessitent des modifications fréquentes (aide à la démonstration, reconnaissance de la parole, systèmes qui mettent en œuvre des ensembles de règles devant suivre les évolutions du marché ou de la législation): 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 ». Les domaines cités ici sont donc des illustrations de travaux passés ou en cours, plutôt que des centres d'intérêt du projet.

On peut identifier deux cibles privilégiées pour les outils formels et les langages de haut niveau qui sont au cœur de nos activités :

- La première concerne les applications exigeant un degré de confiance très important qui justifie l'emploi de techniques reposant sur des 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 à l'application de techniques d'analyse de programmes à la vérification de programmes Java, avec comme domaine d'application les cartes à puce, notamment pour le commerce électronique (cf module 6.1.3).
- La seconde cible de nos travaux concerne les logiciels complexes ou qui nécessitent des modifications fréquentes : on peut citer par exemple l'aide à la démonstration, la reconnaissance de la parole, les systèmes qui mettent en œuvre des ensembles de règles (de fonctionnement d'une organisation, 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 6.1.6) sont en cours d'application dans le domaine de la publicité ciblée pour la télévision à la demande (avec *Next Century Media*). Nous entamons également une collaboration industrielle (cf module 7.1) qui nous permettra d'appliquer nos techniques de débogage à des logiciels de facturation hospitalière (avec *Mission Critical*) et de diagnostic de matériels défectueux (avec *Dassault Electronique*). Les langages d'architectures de logiciels constituent une démarche plus récente pour le développement de logiciels complexes. Nous nous sommes engagés depuis cette année dans une collaboration avec la société *Signaal* (Pays-Bas) qui nous permet d'appliquer nos travaux sur ce thème à un système de contrôle de trafic ferroviaire (cf module 6.2.2).

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 ici uniquement le compilateur de λProlog qui est disponible à l'extérieur du projet (*FTP*). Notre outil de génération de jeux de test* CASTING *(module 6.1.8) sera disponible en 1998 et le moteur d'analyse (solveur de points fixes) cité dans le module 6.1.5 est en cours de développement. L'objectif est aussi de le mettre à disposition d'utilisateurs extérieurs (développeurs d'analyseurs).*

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 uniquement le compilateur de λProlog qui est disponible à l'extérieur du projet (FTP). Notre outil de génération de jeux de test CASTING (module 6.1.8) sera disponible en 1998 et le moteur d'analyse (solveur de points fixes) cité dans le module 6.1.5 est en cours de développement. L'objectif est aussi de le mettre à disposition d'utilisateurs extérieurs (développeurs d'analyseurs).

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 traçeur symbolique et un profileur. Ce système

[BR93] P. Brisset and O. Ridoux. Continuations in λProlog. In D.S. Warren, editor, *10th Int. Conf. Logic Programming*, pages 27–43. MIT Press, 1993.

[Rid91] O. Ridoux. MALIv06: Tutorial and reference manual. Publication Interne 611, Irisa, 1991. ftp://ftp.irisa.fr/local/lande/or-tr-irisa611-91.ps.Z.

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.

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

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 (13 000 lignes) et en C et C/Motif (20 000 lignes).

6 Résultats nouveaux

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

6.1 Actions « aval »

Nous présentons d'abord des résultats généraux sur l'analyse de programmes (modules 6.1.1 et 6.1.2) avant de détailler un certain nombre de recherches portant sur des analyses particulières : la vérification de propriétés de sécurité (module 6.1.3) et la compilation de λProlog (module 6.1.4). Nous décrivons ensuite nos travaux récents sur le débogage statique (module 6.1.5), le débogage dynamique (modules 6.1.6 et 6.1.7) et la génération de jeux de test (module 6.1.8).

6.1.1 Construction systématique d'analyses génériques

Participants : Valérie Gouranton, Daniel Le Métayer

Mots-clés : Analyse de programmes, Sémantique naturelle, *Slicing*.

Résumé : *Certaines analyses, comme le* slicing *(cf. module 3.2) 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 [11, 21]. 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) [11]. 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 [11]. Nous nous intéressons maintenant à l'application de cette méthode à d'autres analyses d'intérêt général comme l'analyse de partage et à sa mise en œuvre dans un outil d'aide à la conception d'analyseurs.

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

Participant : Thomas Jensen

Mots-clés : Typage non-standard, Logique de programmes, Modularité.

Résumé : *Nous avons proposé une méthode de spécification d'analyses, où une analyse est définie comme une logique de programmes par des règles d'inférence. Le cadre peut accommoder plusieurs analyses d'une précision variable en ajoutant des connecteurs logiques (conjonction, disjonction, etc.). 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 avons proposé une méthode de spécification d'analyses, où une analyse est définie comme une logique de programmes par des règles d'inférence. Le cadre peut accommoder plusieurs analyses d'une précision variable en ajoutant des connecteurs logiques (conjonction, disjonction, etc.). Nous avons focalisé nos recherches sur une logique avec propriétés disjonctives pour un langage avec procédures d'ordre supérieur et types récursifs (listes, arbres, etc.). Cette activité a abouti à un cadre qui réunit et étend plusieurs techniques d'analyse existantes [16].

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 *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 [24].

6.1.3 Vérification de propriétés de sécurité de programmes Java

Participants : Thomas Jensen, Daniel Le Métayer, Tommy Thorn

Mots-clés : Téléchargement, Sécurité, Chargement dynamique, Java.

Résumé : *Nous nous sommes attaqués à la formalisation de la sémantique du langage Java 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.

L'un des intérêts majeurs du langage 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) et Java y répond, entre autres, par une plus grande sûreté de programmation (typage, vérification de code importé, héritage simple, etc.), un gestionnaire de sécurité et des recommandations en matière de politique de sécurité [18]. Le langage lui-même est

supposé sûr mais cette affirmation est toujours sujette à débats. Il ressort clairement des déclarations contradictoires à ce sujet que Java intègre un certain nombre de dispositions complexes qui rendent indispensable la formalisation du langage (pour le moins de certains de ses aspects critiques). 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 [34]. 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. Notre objectif actuel est de définir de la même manière des politiques de sécurité afin de pouvoir étudier leur effet (c'est à dire les exigences qu'elles imposent) sur les composants d'une application Java (comme le chargeur de classes).

6.1.4 Analyse statique de programmes λProlog

Participant : Olivier Ridoux

Mots-clés : Analyse statique, λProlog.

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 étudions actuellement une solution au deuxième problème qui consiste à coder par un vecteur booléen une fonction de transfert de la propriété à analyser et à calculer pour chaque terme d'ordre supérieur sa contribution à la propriété et sa fonction de transfert. Ce travail fait l'objet d'une thèse à l'École des Mines de Nantes (Frédéric Malésieux) co-encadrée par Olivier Ridoux et Patrice Boizumault.

6.1.5 Débogage statique

Participants : Pascal Fradet, Ronan Gaugne, Daniel Le Métayer, Florimond Ployette

Mots-clés : Pointeur, Déréférence, Structure de données, Mise au point, Débogage statique.

Résumé : *Nous avons proposé une analyse statique de programmes pour la détection d'accès incorrects à la mémoire (déréférences de pointeurs invalides). Cet analyseur est très précis : il est capable par exemple de décrire exactement le partage induit par des structures complexes comme les listes circulaires. Dédié à l'aide à la mise au point de programmes, il permet un certain nombre d'interactions avec l'utilisateur. Celles-ci se font à l'aide d'un langage d'annotations qui s'intègre naturellement au formalisme de l'analyseur tout en restant très proche du langage de programmation afin de faciliter sa compréhension par le programmeur.*

L'utilisation incorrecte de pointeurs est une des sources d'erreurs les plus répandues dans les langages impératifs. Par conséquent, tout vérificateur de code capable de détecter ce type d'erreurs à la compilation est le bienvenu. Nous avons proposé une analyse statique de programmes pour la détection d'accès incorrects à la mémoire (déréférences de pointeurs invalides) [10, 32]. Un pointeur peut être invalide parce qu'il n'a pas été initialisé ou parce qu'il désigne une cellule de la mémoire qui a été désallouée. D'un point de vue formel, l'analyseur repose sur une sémantique opérationnelle structurelle du langage, dont nous dérivons une axiomatisation des propriétés d'accessibilité. Cette axiomatisation est ensuite raffinée en un analyseur qui est également décrit sous forme de système d'inférence. La correction de l'analyseur a été démontrée et sa complexité établie : elle est polynomiale en fonction du nombre de variables du programme et exponentielle en fonction de la profondeur d'imbrication des boucles. D'un point de vue pratique, notre analyseur se distingue de la plupart des tentatives précédentes par son traitement précis des structures de données récursives et par son intégration dans un système interactif. La précision de l'analyse et la possibilité d'interaction conditionnent en effet l'utilisation d'un tel analyseur dans un environnement d'aide à la mise au point de programmes. Notre analyseur est capable en particulier de décrire exactement le partage induit par des structures complexes comme les listes circulaires. Les interactions avec l'utilisateur se font à l'aide d'un langage d'annotations qui s'intègre naturellement au formalisme de l'analyseur tout en restant très proche du langage de programmation afin de faciliter sa compréhension par le programmeur. Les annotations correspondent soit à des hypothèses ajoutées par l'utilisateur, soit à des propriétés à vérifier par l'analyseur. Les hypothèses peuvent être utilisées pour représenter des connaissances sur le contexte d'exécution du code (ou d'appel d'une procédure) ou pour guider l'analyseur afin de se focaliser sur certaines parties de programmes ou certaines valeurs de données. Ces annotations permettent ainsi d'intégrer de manière naturelle débogage statique et dynamique (cf. module 3.3).

Une première version de prototype a été réalisée. La partie haute (analyse syntaxique) est écrite à partir d'une version augmentée de Suif empruntée à l'évaluateur

partiel Tempo (cf. avant-projet COMPOSE). La génération d'équations récursives est réalisée en SML et la résolution de ces équations repose sur un solveur de points fixes générique développé par ailleurs dans le projet.

6.1.6 Explications pour les bases de données déductives

Participants : Mireille Ducassé, Sarah Mallet

Mots-clés : Débogage, Explication, Génération et 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 donnent une vue éclatée des évaluations. Pour rester plus proche de la notion ensembliste sous-jacente aux bases de données, nous proposons une structure de données, l'arbre* DDB*, reposant sur des ensembles de substitutions.*

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 (cf. module 3.5). Toutefois, la mise en œuvre des bases de données déductives 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 provoque une explosion du nombre d'arbres de preuve produits.

Pour rester plus proche de la notion ensembliste sous-jacente aux bases de données, nous proposons une structure de données, l'arbre DDB, reposant sur des ensembles de substitutions. Elle permet de rassembler les informations éparpillées dans les arbres de preuve et de les présenter de manière plus concise. Les arbres DDB peuvent être montrés aux utilisateurs ou bien analysés par un système d'explications [25, 27, 26].

Ce travail fait l'objet d'une coopération, pour l'instant informelle, avec une société nouvellement créée, Next Century Media. Cette société fournit un produit reposant sur la technologie des bases de données déductives et vise les applications dans la publicité ciblée. Des discussions sont en cours pour définir les informations qui doivent apparaître dans les traces d'exécution afin de bâtir des explications de plus haut niveau, par exemple l'arbre DDB mentionné précédemment.

6.1.7 Analyse de trace automatisée

Participants : Mireille Ducassé, Erwan Jahier

Mots-clés : Débogage, Analyse dynamique, Traceur abstrait, Prolog, Mercury.

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 en Prolog. À partir du langage de requêtes, nous avons bâti des analyses qui donnent 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 développé un outil, OPIUM, 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 de Prolog. Les programmeurs peuvent donc spécifier de manière précise ce qu'ils veulent voir 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* 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 donnent 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.

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.

Un article récapitule les techniques et les applications, il décrit en particulier de manière détaillée l'interface entre l'analyseur de traces et le traceur [13].

Dans le cadre du projet européen ARGo (cf. 8.3), nous étudions le débogage de programmes logiques Mercury, un nouveau langage de programmation logique développé à l'université de Melbourne, Australie [SHC96]. Nous concevons actuellement un traceur pour Mercury. L'objectif à terme est d'appliquer des techniques proches de celles d'Opium pour analyser de manière automatique les traces générées.

6.1.8 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, Jeu de test, Suite de test.

Résumé : *Nous avons proposé une méthode de génération de suites de tests 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 uilisable aussi bien dans le cas du test structurel que fonctionnel. Les suites de tests 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 [30] :

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

[SHC96] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of logic Programming*, 29:17–64, October-December 1996.

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[1] dont un prototype sera disponible en janvier 1998 [30]. La première version de CASTING prend en entrée des spécifications dans la notation AMN de la méthode B [Abr96] et fait appel à Ilog Solver[2] pour résoudre les contraintes engendrées.

Nous nous intéressons conjointement à la manière d'assurer que des programmes sont conformes à des hypothèses de test : cette assurance peut être obtenue par construction (contrôle *a priori*), ou par vérification (contrôle *a posteriori*) quand il s'agit de code existant. 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. Notre objectif actuel est d'associer à tout schéma de programme un type (ou contrainte) caractérisant les jeux de test discriminants pour cette classe.

6.2 Actions « amont »

Nous décrivons dans cette partie les travaux à plus long terme sur les langages fonctionnels (module 6.2.1), Gamma Structuré, les types graphes et les architectures de logiciels (module 6.2.2).

6.2.1 Implantation des langages fonctionnels

Taxonomie des implantations séquentielles

Participant : Pascal Fradet

Mots-clés : Compilation, Langage fonctionnel, Transformation de programme.

Résumé :

De nombreuses techniques ont été proposées pour implanter les langages fonctionnels et il est difficile d'établir des comparaisons rigoureuses entre les différentes options existantes. Nous avons proposé un cadre formel pour décrire et comparer les techniques de compilation des langages fonctionnels. Cela nous a permis d'établir une taxonomie des mises en œuvre

1. *Computer Assisted Software Testing*

2. Ilog Solver est une marque déposée par Ilog.

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

> *de langages fonctionnels et d'y faire figurer de nombreuses implantations classiques comme la* SECD*, la Cam, la G-machine, Tim, etc. Nous avons également étudié l'expression d'optimisations standards et la conception de machines hybrides (i.e. intégrant plusieurs choix de compilation).*

De nombreuses techniques ont été proposées pour implanter les langages fonctionnels et il est difficile d'établir des comparaisons rigoureuses entre les différentes options existantes. Afin d'apporter des éléments de solution à ce problème difficile, nous avons proposé un cadre formel pour décrire et comparer les techniques de compilation des langages fonctionnels [12]. Nous avons repris l'idée centrale de nos précédents travaux [6] qui consiste à définir le processus de compilation comme une suite de transformations de programmes dans le cadre fonctionnel. Nous avons couvert une grande partie du domaine en considérant les mises en œuvre de l'appel par valeur et l'appel par nécessité, qu'elles soient basées sur les environnements ou la réduction de graphe. Les principales tâches d'un compilateur sont la compilation du schéma d'évaluation, de la β-réduction et des transferts de contrôle (appels et retours de fonctions). Nous avons identifié pour chaque étape plusieurs choix fondamentaux et nous les avons décrits comme des transformations de programmes. La description dans un cadre unique a permis de mettre clairement en évidence les différences et les similitudes entre la réduction de graphe et les implantations à environnement.

Un des avantages de cette démarche est de décomposer et de structurer le processus de compilation. On peut, par exemple, montrer que des implantations apparemment très différentes partagent certaines options de mise en œuvre. Nous avons établi une taxonomie des mises en œuvre de langages fonctionnels et nous avons classifié de nombreuses implantations classiques comme la SECD, la Cam, la G-machine, Tim, etc. Nous avons également étudié l'expression d'optimisations classiques, la conception de machines hybrides (i.e. intégrant plusieurs choix de compilation) et proposé des comparaisons formelles sous la forme d'étude de complexité des transformations [12]. Ce travail a fait l'objet de la thèse de Rémi Douence, actuellement en séjour post-doctoral à l'université de Carnegie Mellon.

Langages restreints pour machines parallèles

Participants : Pascal Fradet, Julien Mallet, Mario Südholt

Mots-clés : Schéma de programme, Compilation, Parallélisme.

> **Résumé :** *Nous étudions l'utilisation de langages fonctionnels restreints pour la programmation des machines massivement parallèles. La démarche que nous avons adoptée est celle des langages composés d'une collection de schémas de programmes (patrons) intrinsèquement parallèles (*map, reduce, scan*, etc). Nos travaux dans ce domaine concernent la description abstraite des distributions de données et le calcul précis des coûts de communication.*

Le but des langages restreints pour la programmation des machines massivement parallèles est de concilier efficacité, portabilité et abstraction. La démarche que nous avons adoptée est celle des langages composés d'une collection de schémas de programmes (patrons) intrinsèquement parallèles (*map, reduce, scan*, etc). Les

opérateurs parallèles de HPF ou des langages comme NESL peuvent être vus comme des exemples de ce style de programmation. Nous poursuivons deux directions de recherche :

- Les programmes à base de patrons sont d'habitude utilisés dans un cadre SPMD et opèrent sur un espace de données partitionné. Nous avons examiné une généralisation consistant en la paramétrisation des patrons par une description abstraite des distributions de données. Ces descriptions sont utilisées afin de spécifier (par les recouvrements des données) les communications potentielles. En prenant plusieurs algorithmes numériques, nous avons démontré que cette information permet de guider la compilation des langages à base de patrons [29, 28].
- Une autre solution consiste à compiler automatiquement toutes les tâches spécifiques au parallélisme sans demander plus d'information de la part du programmeur. Nous avons défini un langage restreint comportant une collection de patrons de communication. Il devient possible de calculer précisément le coût de communication et de choisir automatiquement la distribution des données. La compilation se décrit comme une succession de transformations de programmes. Après l'étape de distribution, le programme SPMD obtenu peut être directement traduit dans le langage parallèle spécifique à une machine cible. Un prototype de compilateur pour la machine Intel Paragon est en cours de développement.

6.2.2 Les types graphes

Notre réflexion sur l'introduction d'un système de types pour le langage Gamma nous a conduit à proposer une notion de type graphe correspondant à une classe de graphes d'une même forme. Ces graphes sont manipulés par des *réactions* qui extraient un sous-graphe selon des conditions locales et le remplacent par un nouveau sous-graphe. Nous avons conçu un algorithme de vérification qui permet d'assurer que le type est préservé par une réaction. Ces types peuvent alors être vus comme des invariants sur la forme des données.

Nous avons proposé une nouvelle version de Gamma, appelée Gamma Structuré, qui intègre les types graphes [14]. Il s'est avéré que les types graphes peuvent apporter des solutions à des problèmes très différents. Nous avons notamment étudié leur application pour la manipulation de structures de données avec partage dans les langages impératifs [19] ainsi que la description d'architectures de logiciels [23, 33]. Nous détaillons séparément ces trois domaines d'application.

Gamma Structuré

Participants : Pascal Fradet, Daniel Le Métayer

Mots-clés : Réaction chimique, Multi-ensemble, Structure de données, Type graphe, Vérification de type, Invariant.

Résumé : *Nous avons proposé une version de Gamma qui intègre un moyen de définir des données structurées sans pour autant remettre en*

cause le modèle de calcul de base. Les types graphes permettent une description des structures de données de nature « topologique », sous forme de relations entre les valeurs du multi-ensemble. Ces types peuvent être vus comme des invariants sur les multi-ensembles, le point crucial étant que ces invariants peuvent être prouvés automatiquement. On obtient ainsi une vérification de types pour Gamma Structuré.

Le formalisme Gamma permet une description abstraite des programmes, dépourvue de contraintes d'ordonnancement inutiles. On peut illustrer le comportement des programmes Gamma à l'aide de la métaphore de la réaction chimique : l'exécution est une succession de réactions chimiques consommant les éléments d'un multi-ensemble pour produire de nouveaux éléments selon certaines règles.

Gamma Structuré intègre dans le langage un moyen de définir des données structurées sans pour autant remettre en cause le modèle de calcul de base. La difficulté vient du fait qu'on ne peut recourir à la méthode habituelle pour définir des structures de données (les types récursifs) car ceci induirait un style de programmation récursive (avec une manipulation globale des données) incompatible avec les principes de Gamma. Les types graphes permettent une description des structures de données de nature « topologique », sous forme de relations entre les valeurs du multi-ensemble [14]. Ces types peuvent être vus comme des invariants sur les multi-ensembles, le point crucial étant que ces invariants peuvent être prouvés automatiquement. On obtient ainsi une vérification de types pour Gamma Structuré. Il faut noter que ces types ne contraignent pas les valeurs du multi-ensemble, mais les relations qui les relient. Le bénéfice est double :

- Le programmeur n'a plus, pour structurer ses données, à introduire un codage inélégant et source d'erreurs.
- Les types peuvent servir à contraindre le contrôle des programmes Gamma Structuré et ainsi obtenir des mises en œuvre plus efficaces. Une démarche possible est de partir d'un programme naïf puis d'obtenir une version efficace par raffinements successifs. Les types graphes permettent d'exprimer ces raffinements comme l'ajout de contraintes dans le type du multi-ensemble.

Shape-C

Participants : Pascal Fradet, Daniel Le Métayer, Florimond Ployette

Mots-clés : Type, Structure de donnée, Vérification, Pointeur, Sûreté.

Résumé : *Nous avons proposé une une extension de C intégrant la notion de type graphe. Elle permet de décrire précisément et de manipuler des structures classiques comme les listes circulaires, les listes avec sauts, les arbres binaires avec pointeur parent, etc. Une vérification de type permet d'assurer que la forme d'une structure spécifiée par un type graphe est un invariant du programme. Il devient ainsi possible de détecter de nombreuses erreurs de programmation, ce qui améliore la sûreté des manipulation de pointeurs.*

De nombreuses erreurs de programmation ne sont pas détectées par les systèmes de types des langages impératifs. Leur manque d'expressivité les rend incapables, par exemple, de distinguer une liste doublement chaînée d'un arbre binaire. Nous avons proposé une solution à ce problème avec Shape-C, une extension de C intégrant la notion de type graphe (ou « shape type ») [19]. On utilise en fait un sous-ensemble des types graphes de Gamma Structuré qui correspond aux structures de données manipulées dans les programmes impératifs. Il devient possible de décrire précisément et de manipuler des structures classiques comme les listes circulaires, les listes avec sauts, les arbres binaires avec pointeur parent, etc. Parmi les avantages de Shape-C on peut citer :

- La vérification de type permet d'assurer que la forme d'une structure spécifiée par un type graphe est un invariant du programme. Une vérification de types aussi précise permet de détecter de nombreuses erreurs de programmation et la manipulation de pointeurs devient plus sûre.
- Les programmes Shape-C se traduisent facilement en programmes C efficaces. Le programmeur peut exprimer ses manipulations de pointeurs pratiquement sans surcoût.
- L'intégration avec les autres constructions de C est harmonieuse. En particulier, les « shapes » peuvent contenir des valeurs de n'importe quel type C, les réactions peuvent faire appel à des expressions C, le type « pointeur sur shape » est autorisé, etc.

Un pré-processeur, traduisant des programmes Shape-C en C pur après vérification de types, a été réalisé. Il a notamment permis de déterminer plusieurs extensions souhaitables pour faciliter la programmation en Shape-C. Nous travaillons actuellement sur ce point.

Architectures de logiciels

Participants : Anne Alexandra Holzbacher, Daniel Le Métayer, Michaël Périn, Mario Südholt

Mots-clés : Architecture de logiciel, Coordination, Communication, Style, Vérification.

Résumé : *Nous avons proposé une manière de spécifier des architectures en terme de graphes. La description d'une application est séparée en deux niveaux bien identifiés : d'une part, l'ensemble de ses entités de base qui représentent des calculs autonomes ; d'autre part la coordination de ces entités. Les entités individuelles peuvent être décrites dans des langages traditionnels (par exemple séquentiels) et la coordination est assurée par un composant défini séparément : le coordinateur.*

Les principes de base de notre contribution dans le domaine des architectures de logiciels sont les suivants [8] :

- On introduit une distinction nette entre deux niveaux dans une application : d'une part, l'ensemble de ses entités de base qui représentent des calculs autonomes ; d'autre part la coordination de ces entités (création, destruction,

établissement des liens de communication). Les entités individuelles peuvent être décrites dans des langages traditionnels (par exemple séquentiels) et la coordination est assurée par un composant défini séparément : le coordinateur.
- L'ensemble des entités et leurs liens de communication forment un graphe qui doit vérifier une forme particulière, qu'on appelle le style de l'architecture. Ce style est décrit par un type graphe et on peut vérifier que le coordinateur, exprimé en terme de récritures de graphes dans le style de Gamma Structuré, assure bien l'invariance de ce type.

L'avantage de cette démarche est de concilier une vision dynamique de l'architecture avec des possibilités de vérification statique.

Ce cadre a été appliqué à un cas d'étude proposé par la société Signaal (Pays-Bas) qui consistait à modéliser un système de contrôle ferroviaire [23, 33]. L'architecture ainsi que l'évolution dynamique du système ont pu être décris à l'aide des types graphes. Il a été notamment possible d'assurer statiquement que l'évolution du système respectait les invariants exprimés par la grammaire de graphes. Une implantation en ConCoord a ensuite été dérivée de cette spécification formelle.

7 Actions industrielles

7.1 Action ARGo

Participants : Mireille Ducassé, Erwan Jahier

Mots-clés : Environnement de programmation, Mercury, Programmation logique, Débogage.

Résumé : *L'objectif du projet* ARGo *est de mettre en œuvre un environnement industriel de développement de programmes logiques Mercury. Deux applications industrielles existantes seront portées sur le nouvel environnement afin de le valider. 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 participe 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, durée : 12 mois, début 1er décembre 1997)

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 est donc nécessaire.

L'objectif du projet ARGo est de mettre en œuvre un environnement industriel de développement de programmes logiques Mercury. Deux applications industrielles

existantes seront portées sur le nouvel environnement afin de le valider. Ces applications concernent la facturation hospitalière et le diagnostic de matériel défectueux. LANDE est plus particulièrement concerné par les problèmes de débogage (cf. module 6.1.7).

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 Action CASTING

Participants : Daniel Le Métayer, 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 tests 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 uilisable aussi bien dans le cas du test structurel que fonctionnel. Les suites de tests engendrées dépendent de stratégies spécifiées par l'utilisateur, permettant ainsi d'atteindre la souplesse d'utilisation exigée pour un usage industriel.*

Nous poursuivons une collaboration avec la société rennaise AQL qui s'est concrétisée par une bourse Cifre (*Recherche et application de techniques formelles pour automatiser l'activité de tet de modules logiciels*, ref Inria : 0 94 C 390, durée : 3 ans, début 15 août 1994). Cette action conduit à la réalisation du prototype CASTING, un outil d'aide à la génération de jeux de test (cf. module 6.1.8).

L'activité de test représente une part importante du coût de développement des logiciels et les environnements existants sont encore très limités à cet égard. La phase de génération des données de test en particulier est souvent fastidieuse et mal supportée par les outils actuels. Afin de concevoir un outil de génération véritablement utilisable en milieu industriel, nous avons porté une attention particulière aux aspects suivants :

- La flexibilité : notre méthode est paramétrée par des « objectifs de test » spécifiés par l'utilisateur, ce qui permet de mettre à profit l'expertise accumulée par les testeurs et d'éviter ainsi toute rupture par rapport aux pratiques actuelles.
- Généricité : il est établi que les deux catégories principales de tests (structurels et fonctionnels) sont complémentaires. Par ailleurs, les opérations de base qu'elles mettent en jeu pour la génération des données de test reposent des techniques communes. Notre méthode a été conçue de manière à factoriser ces techniques, ce qui a permis de réaliser un outil générique comportant une partie centrale indépendante du format d'entrée (spécification ou programme). Différentes version de l'outil CASTING peuvent ainsi être proposées (B-CASTING, C-CASTING, etc.), chacune correspondant à un frontal (analyseur syntaxique paramétré par des objectifs de test) spécifique.

– Génération de suites de tests : les applications réelles ne permettent généralement pas un accès direct aux composants de l'état interne. Ceci complique l'exploitation des cas de test car il reste alors à trouver un moyen de positionner le système dans un état satisfaisant des contraintes données. Pour répondre à ce besoin, CASTING produit des suites de tests (partant de l'état initial) plutôt que des cas de test. Ces suites peuvent ensuite être soumises directement à l'application sous test.

Une première version de CASTING sera disponible en janvier 1998 qui prendra en entrée des spécifications en B. Lionel Van Aertryck en poursuivra le développement dans le cadre d'une bourse de Post-Doc industrielle Inria avec AQL.

7.3 Action Java-Sécurité

Participants : Thomas Jensen, Daniel Le Métayer, Tommy Thorn

Mots-clés : Téléchargement, Sécurité, Sûreté, Chargement dynamique, Visibilité, Typage, Java.

Résumé : *Dans le cadre de l'action* VIP *du* GIE *Dyade, nous sommes attaqués à la formalisation de la sémantique du langage Java. Nous nous focalisons 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 participons à l'action VIP du GIE Bull-Inria Dyade (avec le groupe de D. Bolignano de Bull et les projets CRISTAL et EURÉCA de l'Inria). 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 6.1.3). L'étude des problèmes de sécurité (au sens de confidentialité et d'intégrité notamment) dans le contexte du langage Java représente un défi de première importance pour plusieurs raisons :

– La sûreté (au sens du typage) et la sécurité sont présentées comme des arguments pour la promotion d'un langage qui a vocation à être utilisé dans des contextes mettant en jeu des coopérations entre des codes issus de sites différents.
– Le langage inclut des caractéristiques complexes (comme le chargement dynamique ou des règles de visibilité inhabituelles) qui justifient le besoin de définition formelle. Une telle définition permettrait de clarifier certains aspects du langage et servirait de base à un raisonnement rigoureux sur des propriétés cruciales comme la sûreté du typage ou la garantie de politiques de sécurité.

– Les développements prévisibles de 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 [34]. 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. Notre objectif actuel est de définir de la même manière des politiques de sécurité afin de pouvoir étudier leur effet (c'est à dire les exigences qu'elles imposent) sur les composants d'une application Java (comme le chargeur de classes).

8 Actions régionales, nationales et internationales

8.1 Actions régionales

M. Ducassé fait partie depuis 3 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 est impliqué dans le GDR programmation (pôles « Langages fonctionnels » et « Programmation logique »).

8.3 Actions financées par la Commission Européenne

Le projet LANDE participe au *working group* Coordina (Esprit Working Group 24512, *From Coordination Models to Applications*, ref Inria : 1 97 C 905, durée 36 mois, début le 15 août 1997) dont le 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

M. Ducassé et T. Thorn sont membres de l'ACM (Association for Computing Machinery). M. Ducassé, S. Mallet et O. Ridoux sont membres de l'ALP (Association for Logic Programming).

8.5 Relations bilatérales internationales

Nous participons à un projet bilatéral avec l'université d'Hefei-Anhui (Chine). Cette collaboration est financée par le PRA franco-chinois et porte sur les langages de haut niveau pour le génie logiciel.

8.6 Accueil de chercheurs étrangers

Le projet a reçu :

- Hong Lin, Université de Hefei-Anhui, travail sur les architectures de logiciels ; présentation au séminaire LANDE de ses résultats sur la sémantique et la mise en œuvre parallèle de Gamma ; visite d'une durée d'un mois.
- Lee Naish, Université de Melbourne, travail sur le débogage automatisé ; présentation au séminaire LANDE de *A higher order reconstruction of stepwise enhancement* ; visite d'une semaine.
- Zoltan Somogyi, Université de Melbourne, travail sur le traceur pour Mercury ; présentation au séminaire LANDE de *Termination analysis for Mercury* ; visite d'une semaine.
- Heribert Schütz, Université Ludwig-Maximilian de Munich, travail sur les bases de données déductives ; cours de 15 heures à l'Insa sur ce même sujet ; présentation au séminaire LANDE de *Model Generation with Existentially Quantified Variables and Constraints* ; visite d'une semaine.

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

M. Ducassé est membre du chapitre français du BUG (B User Group) et de l'AFFI (Association Française des Femmes Ingénieurs). Elle a également été grand électeur à l'Unesco pour le renouvellement de la commission de la République Française pour l'éducation, la science et la culture, 1996-2001, dans la section « sciences exactes et naturelles ».

M. Ducassé a été membre des comités de programme de AADEBUG'97 (3rd Workshop on Automated and Algorithmic Debugging, Linkoeping) et ICLP'97 (International Conference on Logic Programming, Leuven). Elle a, de plus été responsable de la présentation des *posters* à ICLP'97 et fait partie des comités d'organisation de deux séminaires sur les environnements, à ICLP'97 et ILPS'97 (International Symposium on Logic Programming, New York). Elle est également l'éditeur invité du numéro spécial sur le débogage automatisé de la revue *Automated Software Engineering*, numéro 4:2, 1997.

Pascal Fradet a été membre du comité de programme de IFL'97 (Implementation of Functional Languages - 9th International Workshop).

Daniel Le Métayer a été membre des comités de programme des conférences Euro-Par'97 (European conference in Parallel Processing, Passau), SAS'97 (Static Analysis Symposium, Paris), PLILP'97 (Programming Languages, Implementations, Logics and Programs, Southampton). Il a présidé le comité de programme de la conférence Coordination'97 (Berlin) avec David Garlan. Il est également éditeur de numéros spéciaux des revues ACM *Transactions on Software Engineering and Methodology* et *Theoretical Computer Science*.

Olivier Ridoux est membre du comité de direction du PRC/GDR « Programmation ». Il a organisé les journées de ce groupe de travail à Rennes les 12–14 novembre 1997. Il est membre du comité de direction de l'AFPLC (Association Française de Programmation Logique et par Contraintes) et secrétaire de l'AFPL (Association Française pour la Programmation Logique, correspondant en France de l'ALP). Il co-organise JFPLC-98 (Journées Francophones de Programmation en Logique et par Contraintes) avec Patrice Boizumault de l'École des Mines de Nantes. Il fait partie du comité de rédaction de la revue TSI et il a été membre du comité de programme de la conférence ILPS'97 (International Logic Programming Symposium, New-York).

9.2 Enseignement universitaire

Mireille Ducassé a donné un cours de 16 heures sur la méthode B à l'université Ludwig-Maximilian de Munich, chaire du professeur François Bry, en avril 1997.

Daniel Le Métayer assure une partie d'option de DEA sur les langages fonctionnels et l'analyse de programmes (15h).

Par ailleurs, le projet a encadré trois étudiants de DEA sur les sujets suivants :

- Les architectures de logiciels et la sécurité.
- L'organisation des informations structurées.
- L'aide à la mise au point de programmes Mercury.

9.3 Participation à des colloques, séminaires, invitations

Mireille Ducassé a présenté les travaux de l'équipe sur les *explications dans les bases de données* au séminaire de l'université de Melbourne, Australie. Elle a, par ailleurs, présenté ses travaux sur *la génération de traces pour programmes Prolog par transformation de programme* au séminaire australien *Declarative Languages Day* à Sydney, janvier 1997.

Mireille Ducassé a présenté ses activités d'enseignement de la méthode B à la réunion du B User Group, Paris, mai 1997.

Pascal Fradet et Ronan Gaugne ont présenté leurs travaux sur les types graphes et le débogage statique à l'Inria Rocquencourt.

Daniel Le Métayer et Lionel Van Aertryck ont effectué des présentations sur le thème du test dans le cadre des journées Irisatech (juin 1997).

Sarah Mallet, avec le soutien du projet européen MONET, a participé à DX'97 (8th International Workshop on Principals of Diagnosis), organisé par M.O. Cordier du projet REPCO de l'Irisa.

Tommy Thorn a présenté ses travaux sur la sécurité dans le langage Java lors de la journée « Confidentialité-Intégrité-Disponibilité » organisée par la Meito (Mission pour l'Electronique, l'Informatique et les Télécommunications de l'Ouest) en octobre 1997.

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] Y. Bekkers, O. Ridoux, L. Ungaro, « Dynamic Memory Management for Sequential Logic Programming Languages », *in : Int. Workshop on Memory Management, LNCS 637*, Y. Bekkers, J. Cohen (réd.), Springer-Verlag, p. 82–102, 1992.

[3] C. Belleannée, P. Brisset, O. Ridoux, « Une reconstruction pragmatique de λProlog », *Technique et Science Informatiques, Hermès 14*, 9, 1995, p. 1131–1164.

[4] S. Coupet-Grimal, O. Ridoux, « On the Use of Advanced Logic Programming Languages in Computational Linguistics », *J. Logic Programming 24*, 1&2, 1995, p. 121–159.

[5] M. Ducassé, J. Noyé, « Logic Programming Environments: Dynamic program analysis and debugging », *The Journal of Logic Programming 19/20*, mai/juillet 1994, p. 351–384, aussi Publication Interne Irisa PI910.

[6] 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.

[7] T. Jensen, « Strictness Analysis in Logical Form », *in : Proc. of 5th ACM Conference on Functional Programming Languages and Computer Architecture*, J. Hughes (réd.), *LNCS vol. 523*, Springer, p. 352–366, 1991.

[8] D. Le Métayer, « Software architecture styles as graph grammars », *in : Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, octobre 1996.

Livres et monographies

[9] D. Garlan, D. Le Métayer (eds), *Proceedings of the Second International Conference on Coordination Languages and Models*, Springer Verlag, LNCS 1282, septembre 1997.

Thèses et habilitations à diriger des recherches

[10] R. Gaugne, *Techniques d'analyse statique pour l'aide à la mise au point de programmes avec manipulation explicite de pointeurs*, thèse de doctorat, Université de Rennes 1, 1997.

[11] V. Gouranton, *Dérivation d'analyseurs dynamiques et statiques à partir de spécifications opérationnelles*, thèse de doctorat, Université de Rennes 1, 1997.

Articles

[12] R. Douence, P. Fradet, « A systematic study of functional language implementations », *ACM Transactions on Programming Languages and Systems*, 1998, à paraître.

[13] M. Ducassé, « Opium: An extendable trace analyser for Prolog », *The Journal of Logic Programming*, 1997, à paraître dans "special issue on Synthesis, Transformation and Analysis of Logic Programs, A. Bossi and Y. Deville (eds)", aussi Rapport Technique Inria 3257, Publication Interne Irisa 1127.

[14] P. Fradet, D. Le Métayer, « Structured Gamma », *Science of Computer Programming*, 1997.

[15] C. Hankin, D. Le Métayer, D. Sands, « Refining multiset transformers », *Theoretical Computer Science*, 1997.

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

[17] D. Le Métayer, « Program analysis for software engineering: new applications, new requirements, new tools », *ACM Sigplan Notices*, 1, janvier 1997, p. 86–88.

[18] T. Thorn, « Programming languages for mobile code », *ACM Computing Surveys*, 1997.

Communications à des manifestations scientifiques

[19] P. Fradet, D. Le Métayer, « Shape types », *in : Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, janvier 1997.

[20] R. Gaugne, « A static analysis for program understanding and debugging (short paper) », *in : Proceedings of the Conference on Automated Software Engineering*, IEEE, novembre 1997.

[21] V. Gouranton, D. Le Métayer, « Formal development of static program analysers », *in : Proceedings of the 8th IEEE Israeli Conference on Computer Systems and Software Engineering*, IEEE, p. 101–110, juin 1997.

[22] V. Gouranton, « Un cadre générique de définition d'analyseurs dynamiques et statiques », *in : Actes des journées du GDR Programmation*, novembre 1997.

[23] A. A. Holzbacher, M. Périn, M. Südholt, « Modeling railway control systems using graph grammars: a case study », *in : Proceedings of the Second International Conference on Coordination Languages and Models*, *LNCS*, *1282*, p. 172–186, Berlin, septembre 1997.

[24] T. Jensen, « Inference of polymorphic and conditional strictness properties », *in : Proceedings of 25th ACM Symposium on Principles of Programming Languages*, ACM Press, 1998.

[25] S. Mallet, M. Ducassé, « DDB Trees: A basis for deductive database explanations », *in : Proceedings of the 3rd Workshop on Automated and Algorithmic Debugging*, M. Kamkar (réd.), Linköping University Electronic Press, f-cis.linep.se-97-009, p. 87–102, May 1997.

[26] S. Mallet, M. Ducassé, « Generating DDB trees », *in : Proceedings of the 8th Workshop on Logic Programming Environments, In conjunction with the International Conference on Logic Programming*, 1997.

[27] S. Mallet, M. Ducassé, « An Informal Presentation of DDB Trees: A Basis for Deductive Database Explanations », *in : DDLP'97, Fifth International Workshop on Deductive Databases and Logic Programming*, U. Geske (réd.), GMD-Studien Nr. 317, ISBN 3-88457-317-9, July 1997.

[28] P. Pepper, M. Südholt, « Deriving Parallel Numerical Algorithms using Data Distribution Algebras: Wang's Algorithm », *in : Proceedings of the 30th Hawaii International Conference on System Sciences*, IEEE, janvier 1997.

[29] M. Südholt, C. Piepenbrock, K. Obermayer, P. Pepper, « Solving large systems of differential equations using Data Distribution Algebras », *in : Proceedings of the IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, Chapman & Hall, février 1997.

[30] L. van Aertryck, M. Benveniste, D. Le Métayer, « Casting: a formally based software test generation method », *in : Proceedings of the first IEEE International Conference on Formal Engineering Methods*, *IEEE*, Hiroshima, novembre 1997.

[31] L. van Aertryck, M. Benveniste, D. Le Métayer, « Casting: une méthode formelle pour la génération de tests », *in : Actes des journées AFADL (Approches Formelles dans l'Assistance du Développement de Logiciels*, mai 1997.

Rapports de recherche

[32] R. Gaugne, « Static debugging of C programs: detection of pointer errors in recursive data structures », rapport de recherche RR-3232, Inria, août 1997.

[33] A. A. Holzbacher, M. Périn, M. Südholt, « Modeling railway control systems using graph grammars: a case study », rapport de recherche n ° 3210, Inria, Rennes, France, juillet 1997.

[34] T. Jensen, D. Le Métayer, T. Thorn, « Security and dynamic class loading in Java: a formalisation », rapport de recherche PI-1137, Irisa, octobre 1997.