INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Projet LANDE

Conception et validation de logiciels

Rennes

THÈME 2A

Table des matières

1 Composition de l'équipe

Responsable scientifique

Daniel Le Métayer [DR Inria]

Assistant(e) de projet

Claire Street [vacataire Inria, jusqu'en mars 1998]

Josée Cabellan [vacataire Inria, d'avril 1998 jusqu'en septembre 1998]

Laëtitia Brenugat [vacataire Inria, depuis septembre 1998]

Personnel Inria

Pascal Fradet [CR]

Florimond Ployette [IR, Atelier]

Olivier Ridoux [CR]

Personnel CNRS

Thomas Jensen [CR]

Personnel Université

Valérie Gouranton [ATER]

Personnel Insa

Mireille Ducassé [Professeur]

Chercheurs doctorants

Jeanne Berthélemy [PRAG, Université de Bretagne Sud]

Frédéric Besson [allocataire MENRT, depuis octobre 1998]

Marc Éluard [boursier Inria, depuis octobre 1998]

Erwan Jahier [allocataire MENRT]

Julien Mallet [allocataire MENRT jusqu'en août 1998, puis ATER Insa]

Sarah Mallet [allocataire MENRT]

Valérie-Anne Nicolas [allocataire MENRT]

Michaël Périn [boursier Inria-Région]

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

Tommy Thorn [boursier Inria]

Chercheurs post-doctorant

Ewen Denney [depuis novembre 1998]

Lionel Van Aertryck [post-doc industriel AQL-INRIA]

2 Présentation et objectifs généraux

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

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 »).

Actions « aval »

Les traitements « en aval » ou *a posteriori* concernent la validation de codes existants (vérification, test, débogage). Ce type d'actions s'applique à des logiciels dont on ne maîtrise pas forcément le développement. Nous concevons dans ce cadre des techniques d'aide à la validation de programmes qui reposent essentiellement sur des *analyses de programmes* (statiques ou dynamiques). Nous avons proposé notamment un cadre pour la vérification de propriétés de sécurité de programmes. Nous nous intéressons également à l'analyse dynamique qui sert de base pour des outils de maintenance dans le cadre de la programmation impérative (COCA), de la programmation logique (OPIUM), et des bases de données déductives.

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

En nous appuyant sur des résultats récents concernant l'inférence de types d'intersection et de types polymorphes, nous avons également conçu un algorithme qui permet de déduire, pour chaque programme, une propriété principale à partir de laquelle toute autre propriété

démontrable par l'analyse peut être déterminée. Un avantage de cet algorithme est qu'il traite des fragments de programmes aussi bien que des programmes entiers. Il s'agit donc d'une démarche qui, à plus long terme, peut mener à un cadre général pour l'analyse modulaire.

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

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 développement ou de programmation*. Nous étudions dans ce cadre des langages de haut niveau comme les langages déclaratifs (logiques, fonctionnels) et les langages d'architectures de logiciels. Ces langages fournissent un pouvoir d'expression appréciable tout en offrant un niveau de description 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. Pour ce qui concerne les environnements de programmation, nous avons développé l'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.

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

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*).

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

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

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

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

Sémantiques opérationnelles

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

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

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

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

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

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

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

La sémantique naturelle de notre langage se décrit comme suit.

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

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

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

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

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

3.2 Analyse de programmes

Mots clés : Analyse dynamique, Analyse statique, Sémantique, Interprétation abstraite, compilation optimisante.

Glossaire :

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

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

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

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 programmes, Sémantique.

Glossaire :

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

gloFaute 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. gloPanneUne panne est l'incapacité d'un programme à effectuer ses fonctionnalités

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

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

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

requises. Une faute peut générer une ou plusieurs pannes [ANS]. Un exemple de panne résultant de la faute citée plus haut est que le programme ne termine pas.

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

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

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

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

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.

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

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

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

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

3.4 Test de logiciels

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

Glossaire :

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

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

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

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

correct) [XMd+94]. Tout critère valide et fiable est complet. gloHypothèse de test La complétude étant hors d'atteinte en général, on peut qualifier un jeu de test par des hypothèses de test qui caractérisent les propriétés qu'un programme doit satisfaire pour que la réussite du test entraîne sa correction

Résumé :

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

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

[XMd+94] S. XANTAKIS, M. MAURICE, A. DE AMESCUA, O. HOURI, L. GRIFFET, *Test et contrôle des logiciels. Méthodes techniques et outils*, EC2, 1994.

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

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

3.5 Langages déclaratifs

Mots clés : 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»,

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

jusqu'à obtenir une formule qui n'est plus réductible, et que l'on appelle une *forme normale*. On convient que cette forme normale est le résultat du calcul. L'opération essentielle des langages logiques est la *déduction*. On l'emploie pour construire des preuves, et on convient que le résultat du calcul est extrait de ces preuves. Il s'agit le plus souvent des valeurs données dans les preuves à certaines variables. Pour autant que la correspondance de Curry-Howard s'applique (langages fonctionnels typés), la preuve est l'objet commun à ces deux familles de langages de programmation ; les langages fonctionnels les normalisent, et les langages logiques les construisent.

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

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

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

4 Domaines d'applications

Mots clés : Sûreté, confidentialité, intégrité, logiciel critique, carte à puce, commerce électronique, génération de jeux de test, outil de débogage, base de données déductive, architecture sécurisée.

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

1. *Les applications qui exigent un degré de confiance très important justifiant l'emploi de 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 (dans le cadre de Dyade et d'une collaboration avec Bull-CP8) 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 modules 6.1.3 et 7.3). On peut citer également dans cette catégorie nos travaux sur la génération automatique de jeux de test qui se poursuivent en collaboration avec l'éditeur d'outils de tests Attol Testware et des industriels utilisateurs comme Siemens et Spacebell (cf module 7.2).

- La seconde cible de nos travaux concerne les logiciels 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 6.1.5) sont en cours d'application dans le domaine de la publicité ciblée pour la télévision à la demande (avec Next Century Media). Nous avons également entamé une collaboration industrielle (cf module 7.1) qui nous 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. Sur ce thème, nous sommes impliqués dans une action industrielle en préparation sur la conception d'architectures sécurisées. Cette action regroupera Matra SI, AQL, TNI et deux projets de l'Irisa: EP-ATR et LANDE.

5 Logiciels

Résumé : *Le projet Lande réalise un effort de développement important dans les deux catégories citées dans la présentation générale de nos objectifs (« amont» et « aval»). Nous détaillons le prototype de générateur de suites de test* CASTING *développé en collaboration avec* AQL, COCA *un débogueur automatisé pour le langage* C *et le compilateur de λProlog.*

Le projet Lande réalise un effort de développement important dans les deux catégories citées dans la présentation générale de nos objectifs (« amont » et « aval »). Nous détaillons ici le prototype de générateur de suites de test CASTING développé en collaboration avec AQL, COCA un débogueur automatisé pour le langage C et le compilateur de λProlog.

Générateur de suites de test CASTING

Correspondant: Lionel Van Aertryck

La méthode de génération de suites de test définie dans la thèse de Lionel Van Aertryck [13] a conduit à la réalisation d'un prototype en collaboration avec la société AQL. La méthode elle-même est décrite dans le module 6.1.8; nous nous focalisons ici sur sa mise en œuvre et les fonctionnalités du prototype.

Nous avons utilisé deux outils principaux pour l'implantation de CASTING qui sont Precc [1] (un compilateur de compilateur permettant de calculer des attributs hérités et synthétisés) et le solveur de contraintes Ilog Solver [2]. La version actuelle comporte un seul frontal dédié au langage de spécification de la méthode B (cf. module 6.1.8). Le prototype est constitué d'une partie générique qui regroupe tous les traitements internes (élimination des spécifications de cas de test insatisfiables, génération du graphe d'états symboliques, génération des hypothèses de test et des suites de test, etc.) et une partie liée à la spécification pour laquelle le testeur souhaite engendrer des suites de test (frontal).

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

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

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

Débogueur COCA

Correspondant: Mireille Ducassé

COCA est un débogueur automatisé pour C où le mécanisme des points d'arrêt est basé sur des événements relatifs à des constructions du langage. Ces événements ont une sémantique, alors que les lignes du code source utilisées par la plupart des débogueurs n'en ont pas.

Une trace est une séquence d'événements. Elle peut être vue comme une relation d'ordre sur une base de données. Les utilisateurs peuvent spécifier exactement les événements qu'ils veulent voir en précisant des valeurs pour les attributs des événements. À chaque événement, les variables visibles peuvent être investiguées. Le langage d'interrogation de trace est Prolog augmenté de quelques primitives.

Le mécanisme d'interrogation de trace cherche dans la trace d'exécution en utilisant à la fois des informations sur le flot de contrôle et sur les données alors que les débogueurs effectuent

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

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

habituellement leur recherche de manière exclusive en fonction du contrôle ou en fonction des données.

Contrairement aux débogueurs totalement relationnels qui utilisent effectivement une base de données, le mécanisme d'interrogation de trace de COCA repose sur une analyse à la volée : il ne nécessite donc aucun stockage.

COCA est donc plus puissant que les débogueurs dont les points d'arrêt sont des lignes du code source et plus efficace que les débogueurs relationnels [31].

Un prototype est opérationnel sous SunOS 5.6 Solaris, avec GCC et GDB ainsi que le système Prolog Eclipse 3.5.2. Il intègre des codes C, ML, Prolog et 2100 lignes de grammaire Bison. Quant à l'intégration de GDB, 400 lignes ont été ajoutées et une cinquantaine modifiées parmi les 221800 lignes de C existantes.

Pour plus de renseignements concernant COCA on peut contacter Mireille Ducassé (ducasse@irisa.fr).

Compilateur λProlog

Correspondant: Olivier Ridoux

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

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

Pour tout renseignement concernant Mali ou Prolog/Mali, le point de contact à l'Irisa est Olivier Ridoux (ridoux@irisa.fr, voir aussi [9, 3]).

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

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

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 de logiciels.

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 politiques de sécurité (module 6.1.3) et l'analyse de programmes λProlog (module 6.1.4). Nous décrivons ensuite nos travaux récents sur le le débogage dynamique (modules 6.1.5 et 6.1.6), la génération de traces (module 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, Thomas Jensen, Daniel Le Métayer, Florimond Ployette.

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

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

De nombreux travaux ont été effectués sur les fondements de l'analyse sémantique, la correction et la précision des analyseurs. On peut regretter cependant le peu d'attention accordé jusqu'à présent à la conception d'outils d'analyse génériques. Il est ainsi très difficile de factoriser les efforts en matière de réalisation d'analyseurs. Nous avons abordé ce problème en considérant la construction d'analyseurs sous l'angle de la dérivation de programmes à partir de spécifications. Le programme en l'occurrence est l'analyseur lui-même et sa spécification est composée de deux parties :

1. La sémantique opérationnelle du langage de programmation décrite sous forme de sémantique naturelle.

2. La propriété recherchée exprimée sous forme de récurrences sur les arbres de preuves de la sémantique naturelle.

Les deux composants peuvent en fait s'exprimer sous forme de fonctions et la dérivation consiste en une série de transformations (pliage/dépliage) permettant d'obtenir un programme récursif autonome qui constitue l'analyseur [23].

L'intérêt de cette démarche est qu'elle permet d'exprimer dans un même cadre des analyses variées sur différents langages. Nous en avons fait la démonstration pour des analyses de programmes impératifs (durée de vie des variables), logiques (clôture des termes) et fonctionnels (nécessité, globalisation). Certaines de ces analyses sont spécifiques à un langage de programmation : on peut citer dans cette catégorie l'analyse de clôture pour la programmation logique. D'autres, comme le *slicing* (cf. module 3.2) ont un intérêt plus général. Plutôt que d'en fournir une nouvelle définition pour chaque langage, nous avons montré qu'il est possible de définir une telle analyse de manière générique et de l'instancier pour obtenir des analyses particulières. Nous avons proposé pour ce faire un *format* de sémantique naturelle et nous avons appliqué cette technique à l'analyse de *slicing*. Nous avons pu ainsi dériver, par simple instanciation d'une définition générique, des analyses dynamiques et statiques pour un langage impératif, un langage fonctionnel et un langage de programmation logique [17]. D'un point de vue plus pratique, nous travaillons maintenant à la mise au point d'un moteur d'analyse (solveur de point fixe générique) qui pourra servir de base pour le dévéloppement de nouveaux 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 :
En nous appuyant sur des résultats récents concernant l'inférence de types d'intersection et de types polymorphes, nous avons conçu un algorithme qui permet de déduire pour chaque programme une propriété principale à partir de laquelle toute autre propriété démontrable par l'analyse peut être déterminée. Un avantage de cet algorithme est qu'il traite des fragments de programmes aussi bien que des programmes entiers. Il s'agit donc d'une démarche qui, à plus long terme, peut mener à un cadre général pour l'analyse modulaire.

Exprimer une analyse par des règles d'inférence comme un système de typage non-standard présente l'avantage de la rendre plus compréhensible mais permet également de s'appuyer sur des algorithmes de typage connus. Dans des travaux antérieurs, nous avons conçu une méthode efficace pour *vérifier* qu'un programme satisfait une propriété donnée. Se pose alors la question de savoir s'il est possible, étant donné seulement le programme, de trouver les propriétés satisfaites par ce programme, c'est-à-dire d'*inférer* les propriétés. En nous appuyant sur des résultats récents concernant l'inférence de types d'intersection et de types polymorphes, nous avons conçu un algorithme qui permet de déduire pour chaque programme une propriété principale à partir de laquelle toute autre propriété démontrable par l'analyse peut être déterminée. Un avantage de cet algorithme est qu'il traite des fragments de programmes aussi bien que des programmes entiers. Il s'agit donc d'une démarche qui, à plus long terme, peut mener à un cadre général pour l'analyse modulaire [25].

6.1.3 Vérification de politiques de sécurité

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

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

Résumé : *Nous avons proposé un cadre de définition de politiques de sécurité reposant sur une logique temporelle linéaire à deux niveaux. Nous avons montré que ce modèle est suffisamment expressif pour décrire une variété de politiques de sécurité communes et nous avons proposé une méthode automatique (et complète) de vérification de propriétés pour cette logique. L'idée de base de cette méthode est d'explorer un sous-ensemble fini de suites d'exécution dont la taille est déterminée par la complexité de la propriété à vérifier. Afin d'appliquer ce cadre à Java nous avons formalisé certains aspects de sa sémantique et nous avons montré comment la politique de sécurité du dernier environnement de développement de Java (JDK 1.2) peut s'exprimer dans notre logique.*

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

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

Nous avons étudié ces trois aspects en fournissant un cadre général de spécification de politiques de sécurité et en décrivant son instanciation au langage Java et à son environnement de développement JDK 1.2 [32, 24].

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

pile, etc.). Nous avons par ailleurs proposé une méthode automatique (et complète) de vérification de propriétés pour cette logique [32]. L'idée de base de cette méthode est d'explorer un sous-ensemble fini de suites d'exécution dont la taille est déterminée par la complexité de la propriété à vérifier.

L'application de ce cadre à Java est conditionnée par la formalisation du langage (tout du moins des aspects qui ont un impact sur la sécurité). L'une des particularités de Java est la possibilité de télécharger du code et de l'exécuter de manière transparente. Cette pratique soulève de sérieux problèmes en matière de sécurité des informations (confidentialité, intégrité notamment). Nous nous sommes donc attaqués à la formalisation de cet aspect en nous focalisant sur les règles de visibilité (des classes et de leurs membres) et leur évolution lors du chargement dynamique de classes [32]. Cette formalisation en terme de systèmes d'inférence nous a permis de décrire de manière rigoureuse l'origine d'une erreur de sécurité qui avait été découverte empiriquement par des chercheurs d'ATT.

Nous avons ensuite montré comment la politique de sécurité du dernier environnement de développement de Java (JDK 1.2) peut se formaliser dans notre logique. La technique citée plus haut permet donc de vérifier automatiquement qu'une application Java satisfait la politique affichée. Nous travaillons actuellement à lever certaines restrictions de la méthode (concernant notamment la récursivité mutuelle) et à en dériver un analyseur modulaire (mieux adapté à un environnement ouvert comme celui de Java).

6.1.4 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 avons proposé 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 [30].

Un développement complémentaire et qui concerne aussi le domaine de calcul de λProlog consiste à prendre en compte le type des termes pour la constitution du domaine abstrait dans lequel s'exprime la propriété analysée. Ainsi, si la propriété analysée est l'absence de variable existentielle libre (la clôture, ou *groundness* en anglais), l'expression *(list (pair vrai faux))* sera vraie d'une liste dont le squelette ne comporte pas de variable, dont aucun élément n'est une variable, et dont tous les éléments comportent un «champ gauche» sans variable libre et un «champ droit» qui peut en comporter.

Ce domaine d'abstraction et sa combinaison avec les fonctions de transfert seront complètement développés dans la thèse de Frédéric Malésieux (co-encadrée par Olivier Ridoux et Patrice Boizumault de l'École des Mines de Nantes), dont la soutenance aura lieu au début de l'année 1999.

6.1.5 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 ne correspondent pas toujours au bon profil d'exécution. Nous proposons une technique de traçage qui consiste à intégrer une trace "relationnelle" avec un méta-interprète instrumenté utilisant des ensembles de substitutions. Les aspects coûteux de la méta-interprétation sont réduits par l'utilisation de la trace qui évite beaucoup de calculs. La flexibilité de la méta-interprétation est conservée. Elle permet de produire facilement des traces de profils différents.*

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

La forme déclarative du langage de règles permet aux concepteurs de la base de données de se concentrer sur sa logique (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, qui peut être utile dans certains cas, provoque, dans le cas général, une explosion du nombre d'arbres de preuve produits. Il s'agit donc de concevoir un outil capable de présenter différents points de vue aux utilisateurs.

Nous proposons une technique de traçage qui consiste à intégrer une trace "relationnelle" avec un méta-interprète instrumenté utilisant des ensembles de substitutions. La trace relationnelle donne, de manière efficace, de l'information précise sur l'extraction de données de la base relationnelle. Le méta-interprète ensembliste gère des ensembles de substitutions et donne des explications sur la déduction. Les aspects coûteux de la méta-interprétation sont réduits par l'utilisation de la trace qui évite beaucoup de calculs. La flexibilité de la méta-interprétation est conservée. Elle permet de produire facilement des traces de profils différents [28, 29].

Ce travail fait l'objet d'une coopération avec Next Century Media. Cette société fournit Validity, un produit reposant sur la technologie des bases de données déductives, et vise les applications dans la publicité ciblée. Un prototype de recherche, développé dans LANDE, met en œuvre la technique de traçage mentionnée ci-dessus en utilisant des traces relationnelles produites par le traceur de Validity. Des discussions sont en cours pour affiner la spécification des informations qui doivent apparaître dans les explications de haut niveau.

6.1.6 Analyse de trace automatisée

Participants : Mireille Ducassé, Erwan Jahier.

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

Résumé : *Nous avons développé un outil,* OPIUM*, qui analyse les traces d'exécution de programmes générées par un traceur Prolog existant. Les programmeurs peuvent spécifier de manière précise ce qu'ils veulent observer du comportement du programme à l'aide de requêtes 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. La meilleure preuve en est les deux nouveaux prototypes de recherche qui appliquent ces idées aux langages Mercury et C.*

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* permet de traiter un ensemble plus riche de requêtes mais nécessite de stocker les événements dans une base de données interne, ce qui prend un temps non négligeable. L'analyse proprement dite, par contre, ne prend pas plus de temps que l'analyse à la volée. Du point de vue de l'utilisateur, les requêtes se posent de la même manière dans les deux cas.

À partir du langage de requêtes, nous avons bâti des analyses qui 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 [15].

Si le contenu de la trace utilisée et les analyses proposées sont dédiés à Prolog, les techniques de base pour mettre en œuvre le langage de requêtes exploitent une trace qui peut être produite pour n'importe quel langage séquentiel. Nous mettons à profit cette généralité pour concevoir des débogueurs pour les langages Mercury (cf. module 7.1) et C [31].

6.1.7 Génération de traces

Participante : Mireille Ducassé.

Mots clés : Débogage, traceur, transformation de programmes, mesures de performances, Prolog.

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

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

Typiquement, les traceurs modifient en profondeur le compilateur, les structures de données et les schémas d'exécution. Les problèmes avec ce type de traceurs sont de plusieurs ordres : tout d'abord, on perd souvent les optimisations qui donnent tout leur intérêt aux compilateurs ; ensuite, l'information est donnée à l'utilisateur en termes de trop bas niveau, il peut même y avoir perte significative d'information ; enfin, ces traceurs ne sont pas portables d'un compilateur à l'autre pour un même langage.

Nous étudions actuellement la génération de traces d'exécution pour Prolog par transformation source à source de programmes. Cette technique permet d'éviter les problèmes mentionnés ci-dessus. Il se pose alors la question de son efficacité. Nous avons implémenté un prototype. Des mesures montrent que les performances des programmes tracés par transformation de programmes sont au pire deux fois moins bonnes que celles des programmes tracés par deux traceurs opérationnels "bas niveau" [20]. Cette réduction de performance est acceptable pour un traceur "standard" surtout lorsque l'on considére le gain significatif en termes de portabilité.

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

6.1.8 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, Casting, Jeu de test, Suite de test.

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

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

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

Les supports d'entrée peuvent être constitués de spécifications formelles ou informelles, de programmes sources ou de propriétés fournies directement par l'utilisateur. Les opérations peuvent être des machines abstraites dans le cas du langage B, des schémas pour le langage Z, des programmes dans le cas d'un langage de programmation, etc. Dans tous les cas, les critères de test sont implantés par des *stratégies de test* et se traduisent in fine par des *hypothèses d'uniformité et hypothèses de régularité*. Ces hypothèses permettent de préciser le sens (et les limites) des jeux de test qui seront engendrés (cf. module 3.4). D'un point de vue pratique, une stratégie de test correspond à un mode d'extraction de contraintes à partir du texte source. Ces contraintes caractérisent les jeux de données qui devront être engendrés pour chaque opération

du système. Le graphe d'accessibilité symbolique indique l'ordre dans lequel les opérations peuvent être appliquées pour satisfaire toutes les contraintes. Dans le cas général en effet on ne peut faire l'hypothèse qu'une opération est toujours applicable : selon l'état du système, il peut être nécessaire d'effectuer plusieurs opérations intermédiaires avant de pouvoir appliquer une opération donnée. La dernière phase consiste à explorer ce graphe en résolvant les contraintes associées pour générer les données de test effectives.

Ces travaux ont conduit au développement de l'outil d'aide à la génération de jeux de test CASTING[3] [13] (cf. module 5). La version actuelle de CASTING prend en entrée des spécifications dans la notation AMN de la méthode B [Abr96] et fait appel à Ilog Solver[4] pour résoudre les contraintes engendrées. Diverses démonstrations de ce prototype ont été réalisées (notamment à la conférence B). Nous travaillons maintenant à la transposition de cette technique pour la génération de programmes C et C++ (test structurel) dans le cadre du projet européen TWO (cf. module 7.2).

Nous nous intéressons conjointement à la manière d'assurer que des programmes sont conformes à des hypothèses de test. Nous considérons dans ce but des schémas de programmes qui peuvent être vus comme une manière de définir des classes de fautes : identifier un programme de manière non ambiguë dans une classe revient à assurer que toutes les fautes ayant pour effet de produire une version de programme dans cette classe seront détectées. À tout schéma de programme est associé un jeu de test fini et robuste [18] (c'est à dire permettant de distinguer deux fonctions quelconques du schéma). Ce résultat permet de montrer automatiquement qu'un programme satisfait une propriété donnée (à condition que l'un et l'autre puissent être situés dans notre hiérarchie de schémas). La propriété attendue est définie comme une fonction et la propriété effective est dérivée du programme par interprétation abstraite. Il s'agit ensuite de déterminer le plus petit schéma de la hiérarchie qui les contient toutes les deux. On sait alors que le jeu de test associé à ce schéma est suffisant pour établir l'égalité des deux fonctions. Ce jeu de test peut être concrétisé (opération inverse de l'interprétation abstraite) pour être soumis au programme [12].

6.2 Actions « amont »

Nous décrivons dans cette partie les travaux à plus long terme sur les langages fonctionnels (module 6.2.1), la programmation par aspects (module 6.2.2), et les architectures de logiciels (module 6.2.3).

6.2.1 Implantation des langages fonctionnels

Taxonomie des implantations séquentielles

Participant : Pascal Fradet.

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

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

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

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 de langages fonctionnels, d'étudier 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 [14]. 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. Nous avons identifié pour chaque étape de compilation plusieurs choix fondamentaux et nous les avons décrits comme des transformations de programmes. La description dans un cadre unique a permis d'établir une taxinomie des mises en œuvre de langages fonctionnels. 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 [14]. Ce travail a été réalisé en collaboration avec Rémi Douence, actuellement à l'École des Mines de Nantes.

Langages restreints pour machines parallèles

Participants : Pascal Fradet, Julien Mallet.

Mots clés : Langages dédiés, Schémas de programme, Compilation, Analyse de coût, Parallélisme.

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

Il n'existe pas de langage idéal pour programmer les machines parallèles : on trouve d'une part des langages à parallélisme explicite efficaces mais non portables et très complexes à utiliser; et d'autre part des langages simples et portables mais dont la compilation est complexe et relativement inefficace. La démarche que nous avons adoptée est celle d'un langage spécialisé permettant une estimation exacte du coût de l'implantation parallèle ([27],[33]). Le résultat de cette analyse de coût, qui prend en compte les temps de calcul et les temps de communication,

conditionne des choix de compilation comme la distribution de données, l'ordonnancement, et certaines optimisations.

Notre langage source est un langage fonctionnel du premier ordre où la récursivité générale et les conditionnelles sont remplacées par des collections de schémas de programmes (patrons) [11]. Ces restrictions permettent de choisir automatiquement la meilleure distribution de données parmi un ensemble de distributions standards. Ce choix repose sur une analyse de coût exacte et symbolique (les tailles effectives des données n'ont pas à être connues).

Ce travail est un exemple de fertilisation croisée entre trois domaines différents : la parallélisation automatique de Fortran, les langages fonctionnels et les langages à patrons. Les paralléliseurs Fortran ont introduit des analyses exactes de temps d'exécution symbolique pour des sous-ensembles du langage. Nous avons étendu ces travaux pour définir notre analyse de coût et nous avons explicité les restrictions nécessaires pour assurer l'exactitude de l'analyse. Nous avons également adapté des résultats obtenus dans le cadre des langages fonctionnels pour manipuler les tableaux de façon efficace. Les langages à patrons fournissent un cadre pour définir les restrictions du langage source en introduisant des schémas de calculs parallèles. Nous avons étendu cette approche en introduisant des patrons de communication (dénotant les routines optimisées des machines parallèles) et des patrons de masque (qui permettent d'évaluer exactement la répartition des calculs sur les processeurs). Les résultats préliminaires sont prometteurs puisque, sur les quelques exemples traités, nos programmes se sont révélés avoir les mêmes performances que les programmes HPF équivalents (avec des distributions manuelles). L'analyse de coût calcule des temps d'exécution très proches de la réalité et les distributions sélectionnées automatiquement se sont révélées être les meilleures en pratique.

6.2.2 Programmation par aspects

Participant : Pascal Fradet.

Mots clés : Aspects, Cadre générique, Programmation robuste, Exceptions.

Résumé : *La programmation par aspects propose de décrire un logiciel comme un ensemble formé d'un composant principal et d'une collection d'aspects. Un outil, appelé tisseur, est chargé de produire automatiquement un programme intégrant les différents aspects au composant principal. Nous avons commencé à nous intéresser à cette approche en proposant à la fois un cadre générique de programmation par aspects et son application à la production de logiciels robustes.*

La programmation par aspects est un paradigme de programmation proposé tout récemment par une équipe du PARC (Xerox). Dans cette approche, un logiciel est formé d'un composant principal et d'une collection d'aspects décrivant des tâches comme la gestion mémoire, la synchronisation, les optimisations, etc. Un outil, appelé tisseur, est chargé de produire automatiquement un programme intégrant les différents aspects au composant principal. L'intérêt de cette approche est de localiser (dans les aspects) des choix de mise en œuvre qui seraient sinon dispersés dans le code source. Nous avons commencé à nous intéresser à cette approche en proposant à la fois un cadre générique de programmation par aspects et son application à un exemple concret [22].

Dans notre cadre, un aspect est une collection de transformations syntaxiques de programmes. Le tisseur effectue un calcul de point fixe appliquant ces transformations autant que faire se peut sur le composant principal. Ce cadre permet également d'associer des analyses de programmes au tisseur afin que les aspects puissent reposer sur des critères sémantiques aussi bien que syntaxiques.

Nous nous intéressons à l'utilisation de ce paradigme pour la production de logiciels robustes. Un programme robuste possède un domaine standard d'entrées sur lequel il termine (et satisfait ses spécifications) et un domaine exceptionnel sur lequel il termine en produisant une erreur ou une exception. L'écriture de programmes robustes implique le plus souvent de nombreux tests répartis dans le code (par exemple pour vérifier les débordements ou les accès aux tableaux). Il est difficile de prévoir et d'insérer tous les tests nécessaires. De plus, ces tests rendent le programme source difficile à lire et à maintenir. De fait, peu de programmes sont robustes. Dans le cadre de la programmation par aspects, un programme robuste peut être décomposé en un programme (par exemple codé en C) qui respecte sa spécification pour le domaine standard mais pas forcément pour le domaine exceptionnel et un aspect décrivant le domaine exceptionnel. L'aspect prend la forme d'un ensemble de propriétés (e.g., $1 \leq X \leq 100$ qui spécifie que la variable X doit être comprise entre 1 et 100 durant toute l'exécution). Le programme résultant du tissage est un programme robuste, équivalent au composant pour le domaine standard mais qui termine par un message d'erreur pour le domaine exceptionnel. Pour cette application, le tisseur se doit d'intégrer une analyse de programmes afin d'éviter l'insertion de tests superflus. Nous terminons actuellement la formalisation du cadre générique et la définition du langage d'aspects pour la programmation robuste. Ce travail est effectué en collaboration avec Mario Südholt de l'École des Mines de Nantes.

6.2.3 Les architectures de logiciels

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 [16]. 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 description d'architectures de logiciels [19].

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.

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 [16]. 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 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.

Architectures de logiciels

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

Mots clés : Architecture de logiciel, Coordination, Communication, Style, Vue, Analyse, 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.*

Le domaine des architectures de logiciels a connu un développement important ces dernières années. La tendance actuelle consiste à proposer des langages spécifiques pour la définition de l'architecture (ou organisation globale) de gros logiciels et à fournir des outils pour manipuler ces architectures et en prouver des propriétés. Il s'avère en fait que les propriétés qu'on souhaite vérifier dans ce contexte sont de natures très diverses et que chacune d'elle peut demander une

présentation différente de l'organisation du logiciel. La définition d'une architecture comme un ensemble de *vues* complémentaires s'impose donc mais il est alors nécessaire de traiter le délicat problème de la cohérence de ces vues multiples. Nous avons abordé cette question en proposant une notion de cohérence locale définie à l'aide de relations binaires et de contraintes structurelles sur des vues représentées par des graphes non interprétés [26]. Par rapport à la cohérence «d'implémentation» utilisée dans le cadre des langages de spécification, notre définition facilite le traitement de vues hétérogènes. Cet aspect est crucial dans le contexte des architectures de logiciels où les vues doivent pouvoir être exprimées naturellement dans des formalismes très différents.

Une des vues les plus étudiées dans le domaine des architectures de logiciels est celle qui concerne les communications et les synchronisations entre composants. Nous avons abordé cette question en proposant une manière de décrire les applications en séparant deux niveaux bien distincts : 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 [19].

En collaboration avec le projet SOLIDOR, nous étudions également le raffinement d'architectures de logiciels vers des systèmes à agents mobiles. L'idée est de raisonner sur une vue de l'architecture faisant apparaître les volumes de données échangés afin de choisir la mise en œuvre la plus adaptée (agent mobile ou RPC).

Nos travaux sur les architectures de logiciels s'appuient sur des études de cas (système de contrôle de trains proposé par la société néerlandaise Signaal, sécurité du système d'informations de l'Irisa). Ils se concrétiseront dans le cadre d'une action industrielle conduite en collaboration avec le projet EP-ATR et qui impliquera Matra Systèmes Informatiques AQL et TNI. Cette action, qui est commanditée par le Celar, porte sur la conception d'un outil d'aide à la mise au point d'architectures sécurisées.

7 Contrats industriels (nationaux, européens et internationaux)

7.1 Action ARGo

Participants : Mireille Ducassé, Erwan Jahier.

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

Résumé : *L'objectif du projet* ARGo *est de mettre en œuvre un environnement industriel de développement de programmes logiques Mercury. Deux applications in-*

dustrielles 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) qui a été lancé en novembre 1997 pour une durée de 18 mois.

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.

Nous avons mis en oeuvre un prototype d'analyse de trace (cf. module 6.1.6). Un manuel utilisateur et des démonstrations ont été présentés à la revue officielle du projet en décembre 1998.

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 TWO

Participants : Daniel Le Métayer, Olivier Ridoux, Lionel Van Aertryck.

Mots clés : Test en boîte noire, Test en boîte blanche, Test structurel, Objectif de test, Contrainte, Jeu de test, Suite de test.

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

Nos travaux sur la génération automatique de jeux de test ont été initiés dans le cadre d'une collaboration avec la société rennaise AQL à travers une bourse Cifre puis le stage de post-doc industriel de Lionel Van Aertryck. Ils ont conduit à la réalisation du prototype CASTING, un

outil qui permet de générer des jeux de test à partir de spécifications B. Cette génération peut être automatique ou interactive en fonction des besoins de l'utilisateur. Le prototype en question est maintenent robuste et il en a été fait plusieurs démonstrations, notamment lors de la dernière conférence B.

Ces travaux sur la génération automatique de jeux de test prennent une nouvelle dimension avec la collaboration entamée dans le cadre du projet européen TWO (*Test and Warning Office*, Industrial RTD Project No 25503, ref. Inria : 1 98 C 344). Ce projet a commencé en octobre 1998 et s'étendra sur deux années et demie. Il implique le centre de recherche du CEA, le Politecnico di Milano (Italie), les sociétés Elsag Bailey (Italie), Siemens (Allemagne), Spacebel Informatique (Belgique) et l'éditeur d'outils de tests Attol Testware (France) qui pilote le projet.

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

7.3 Action Java-Sécurité

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

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

Résumé : *Dans le cadre de l'action* VIP *du* GIE *Dyade, nous avons proposé une formalisation de la sémantique de certains aspects du langage Java. Nous nous sommes focalisés sur les règles de visibilité (des classes et de leurs membres) et leur évolution lors du chargement dynamique de classes. Nous avons ensuite proposé un cadre pour spécifier des politiques de sécurité basé sur une logique temporelle linéaire à deux niveaux. Nous avons conçu une méthode pour vérifier qu'un programme Java, réprésenté comme un système de transition, satisfait une politique exprimée dans ce cadre. La méthode répose sur un résultat qui permet de limiter la taille du système en fonction de la propriété à vérifier.*

Nous participons à l'action VIP du GIE Bull-Inria Dyade. Le thème de cette collaboration est la formalisation de certains aspects de la sémantique de Java et la preuve de propriétés de sécurité de programmes (cf. module 6.1.3). 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.

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

Nous nous sommes attaqués à cette formalisation en nous focalisant sur les règles de visibilité (des classes et de leurs membres) et leur évolution lors du chargement dynamique de classes [32]. Il s'agit en effet de caractéristiques particulières de Java qui ont un impact direct sur la sécurité et dont les définitions informelles ne sont pas exemptes d'ambiguïtés ou d'insuffisances. Cette formalisation en terme de systèmes d'inférence nous a permis de décrire de manière rigoureuse l'origine d'une erreur de sécurité qui avait été découverte empiriquement par des chercheurs d'ATT.

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

Notre effort de formalisation des différents aspects de Java et de sa mise en œuvre se concrétise également à travers une collaboration entamée avec le fabricant de cartes à puces Bull-CP8 sur la vérification d'optimisations pour Java Card. Ce projet (ref. Inria : 1 98 C 144), commencé en février 1998, doit durer une année.

8 Actions régionales, nationales et internationales

8.1 Actions régionales

Mireille 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

Thomas Jensen est responsable de l'Action Collaborative Java Card qui associe les projets CRISTAL, CROAP, et LANDE à deux partenaires extérieurs: le groupe Sécurité du CERT (Toulouse) et l'action VIP de Dyade. Son objectif est de fédérer les activités en cours en France sur

la sémantique, l'optimisation et la sécurité de Java Card, la version de Java dédiée aux cartes à puces.

8.3 Actions financées par la Commission Européenne

Le projet LANDE participe aux projets européens ARGO, TWO et Coordina. Les deux premiers sont décrits dans la section Actions industrielles (modules 7.1 et 7.2 respectivement). Coordina est un *working group* (Esprit Working Group 24512, *From Coordination Models to Applications*, ref Inria : 1 97 C 905) qui a été lancé en août 1997 pour une période de trois années. Son but est l'étude des langages et des modèles de coordination et d'architectures de logiciels. Les activités du groupe s'articulent autour d'études de cas fournies par les sociétés Signaal (Pays-Bas) et Xerox (France). Le projet inclut comme partenaire académiques le CWI, les universités de Berlin, Berne, Bologne, Chalmers, Genève, Leiden, Lisbonne, Londres (Imperial College) et Pise.

8.4 Réseaux et groupes de travail internationaux

Mireille Ducassé et Tommy Thorn sont membres de l'ACM (Association for Computing Machinery). Mireille Ducassé, Erwan Jahier, Sarah Mallet et Olivier Ridoux sont membres de l'ALP (Association for Logic Programming). Daniel Le Métayer participe au lancement du groupe de travail de l'IFIP sur les architectures de logiciels.

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 :

- Norbert Eisinger, Université Ludwig-Maximilian de Munich, travail sur la visualisation d'exécutions ; visite d'une semaine.
- Konstantinos Sagonas, Université de Leuven, travail sur l'analyse statique de programmes ; présentation au séminaire de l'Irisa de ses résultats sur la recherche de points fixes par la méthode de la résolution tabulée.
- Wolfram Schulte, de l'université d'Ulm, travail sur la sémantique et l'analyse de programmes Java Card ; visite d'une durée de deux mois.
- Chun Yuan, University of Science and Technology of China (Hefei-Anhui), travail sur les architectures de logiciels ; visite d'une durée de trois mois.

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

9 Diffusion de résultats

9.1 Animation de la communauté scientifique

Mireille Ducassé est membre du chapitre français du BUG (B User Group) et de l'AFFI (Association Française des Femmes Ingénieurs). Elle a été membre du comité de programme de JFPLC-98 (Journées Francophones de Programmation en Logique et par Contraintes).

Mireille Ducassé a été rapporteur sur les thèses de Jacques Pelet (*Évaluation de l'intégrité des logiciels à caractère sécuritaire*, université de St Etienne), préparée sous la responsabilité de M. Jourlin, et de Sophie Cherki (*Rétro-ingénierie de programmes Fortran utilisant des spécifications algébriques*, université de Paris Sud), préparée sous la responsabilité de C. Choppy.

Pascal Fradet a été membre du comité de programme de la conférence ICCL'98 (IEEE International Conference on Computer Languages, Chicago).

Thomas Jensen a été membre du comité de programme de SAS'98 (Static Analysis Symposium, Pise).

Daniel Le Métayer a été membre des comités de programme des conférences ACM SIGSOFT FSE-6 (Sixth Symposium on the Foundations of Software Engineering, Orlando), IEEE ACM ICSE'99 (International Conference on Software Engineering, Los Angeles), WICSA1 (First Working IFIP Conference on Software Architecture), AFADL'98 (Approches Formelles dans l'Assistance au Développement de Logiciel, Poitiers). Il est également éditeur de numéros spéciaux des revues ACM *Transactions on Software Engineering and Methodology* et *Theoretical Computer Science*. Par ailleurs, il a été rapporteur sur la thèse de Christophe Bidan préparée sous la responsabilité de Valérie Issarny (*Sécurité des systèmes distribués: apport des architectures logicielles*).

Olivier Ridoux a été rapporteur sur la thèse d'Alain Hui-Bon-Hoa, (*Principe de résolution pour un langage de programmation en logique avec contraintes de portée*, université de Paris 7), préparée sous la responsabilité de Bernard Lang (Inria Rocquencourt), et examinateur pour la thèse de Christine Sinoquet (*Grammaires à transformations morphiques. Recherche de motif - exacte ou approchée - adaptée aux séquences génétiques : le système* GTM), préparée sous la responsabilité de Jacques Nicolas (ex-projet REPCO, Irisa).

Olivier Ridoux est membre du comité de direction du PRC/GDR « ALP ». Il est membre (secrétaire) du comité de direction de l'AFPLC (Association Française de Programmation Logique et par Contraintes, correspondant en France de l'ALP). Il a co-organisé JFPLC-98 (Journées Francophones de Programmation en Logique et par Contraintes) avec Patrice Boizumault de l'École des Mines de Nantes [10]. Ces journées organisées conjointement avec les Journées française sur la résolution pratique des problèmes NP-complets (JNPC'98) ont rassemblé une centaine de participants pendant 3 jours. Des actes ont été publiés (éditions Hermes) avec l'aide de l'Irisa. Il fait également partie du comité de rédaction de la revue TSI.

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 1998.

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

Olivier Ridoux a fait partie du comité de réflexion pour le renouvellement de l'habilitation du DEA d'informatique de l'Ifsic.

Par ailleurs, le projet a reçu Ranjit Jhala, étudiant de l'Indian Institute of Technology à Delhi pour un stage de deux mois sur le typage et l'analyse de flot de contrôle en Java et il a encadré deux étudiants de DEA sur les sujets suivants :

- Analyse sémantique de programmes Signal.
- Calcul de points fixes pour la résolution des équations produites lors de l'analyse statique de programme.

9.3 Participation à des colloques, séminaires, invitations

Mireille Ducassé a présenté ses activités d'enseignement de la méthode B à la session sur l'éducation de la conférence international sur B [21]. Elle a également passé une semaine à IC-Parc, Londres, au sujet du débogage dans Eclipse, un système de programmation logique avec contraintes en novembre 1998.

Pascal Fradet et Daniel Le Métayer ont présenté leurs travaux sur les types graphes, les architectures de logiciels et le test à l'Institut de Mathématiques de l'Academia Sinica (Pékin), l'Institut du Logiciel (Pékin) et l'University of Science and Technology of China (Hefei). Daniel Le Métayer a par ailleurs été invité au workshop NSF-CNR ROSATEA (Role of Software Architectures for Testing and Analysis) à Marsala. Il a également présenté, avec Thomas Jensen, les activités du projet sur la vérification de politiques de sécurité à la journée de rencontre Inria-Industrie "Telecom et multimedia".

Thomas Jensen a présenté l'Action de Collaboration Java Card à la conférence européenne EMMSEC'98 à Bordeaux. Il a également représenté le projet LANDE au workshop Ercim sur les langages de programmation à Pise.

Erwan Jahier, dans le cadre du projet ARGO, a passé un mois à l'université de Melbourne, Australie, au sujet du traceur de base de Mercury, en février 1998. Il a par ailleurs assisté à l'école "Jeunes chercheurs" du GDR Programmation à Nantes et aux journées "Doctoriales" à Dinard.

Michaël Périn a participé à l'École "Jeunes chercheurs" MOVEP'98 à Nantes. Siegfried Rouvrais a assisté à "Summer School on specification, verification and refinement" à Turku (Finlande) et à l'école "Jeunes chercheurs" du GDR Programmation à Nantes.

Tommy Thorn a présenté les travaux de notre groupe sur la formalisation de la sémantique de Java à l'université de Princeton (États-Unis).

Lionel Van Aertryck a effectué des présentations et démonstrations du prototype CASTING à la conférence B (Montpellier) et aux journées du groupe de travail BUG (Paris).

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 (éditeurs), Springer-Verlag, p. 82–102, 1992.

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

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

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

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

[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.

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

Livres et monographies

[10] O. Ridoux (éditeur), *Actes de Journées Francophones de Programmation logique et programmation par Contraintes*, Hermes, mai 1998.

Thèses et habilitations à diriger des recherches

[11] J. Mallet, *Compilation d'un langage spécialisé pour machine massivement parallèle*, thèse de doctorat, Université de Rennes I, Ifsic, Irisa, décembre 1998.

[12] V.-A. Nicolas, *Preuve de propriétés de classes de programmes par dérivation systématique de jeux de tests*, thèse de doctorat, Université de Rennes I, Ifsic, Irisa, décembre 1998.

[13] L. Van Aertryck, *Une méthode et un outil pour l'aide à la génération de jeux de tests de logiciels*, thèse de doctorat, Université de Rennes I, Ifsic, Irisa, janvier 1998.

Articles et chapitres de livre

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

[15] M. Ducassé, « Abstract views of Prolog executions with Opium », *in : Learning to Build and Comprehend Complex Information Structures: Prolog as a Case Study*, P. Brna, B. du Boulay, et H. Pain (éditeurs), *Cognitive Science and Technology*, Ablex, 1999.

[16] P. Fradet, D. Le Métayer, « Structured Gamma », *Science of Computer Programming 31*, 2-3, juillet 1998, p. 263–289.

[17] V. Gouranton, D. Le Métayer, « Dynamic slicing: a generic analysis based on a natural semantics format », *Journal of Logic and Computation*, à paraître, version préliminaire parue en rapport de recherche Inria No 3375, mars 1998.

[18] D. Le Métayer, V.-A. Nicolas, O. Ridoux, « Exploring the software development trilogy », *IEEE Software*, novembre 1998.

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

Communications à des congrès, colloques, etc.

[20] M. Ducassé, J. Noyé, « Tracing Prolog programs by source instrumentation is efficient enough », *in : IJCSLP'98 Post-conference workshop on Implementation Technologies for Programming Languages based on Logic.*, K. Sagonas (éditeur), juin 1998.

[21] M. Ducassé, « Teaching B at a technical university is possible and rewarding », *in : B'98, Proceedings of the Educational Session*, H. Habrias, S. E. Dunn (éditeurs), Association de Pilotage des Conférences B, avril 1998.

[22] P. Fradet, M. Südholt, « Aspect-Oriented Programming: towards a generic framework using program transformation and analyses », *in : Workshop on Aspect-Oriented Programming, ECOOP'98 Workshop Reader*, *Lecture Notes in Computer Science*, Springer-Verlag, juillet 1998.

[23] V. Gouranton, « Deriving analysers by folding/unfolding of natural semantics and a case study: slicing », *in : Proceedings of the int. Static Analysis Symposium*, *Lecture Notes in Computer Science*, *1503*, Springer Verlag, p. 115–133, septembre 1998. Version préliminaire parue en rapport de recherche Inria No 3413, avril 1998.

[24] T. Jensen, D. Le Métayer, T. Thorn, « Coarse grained Java security policies », *in : 4th Workshop on Mobile Object Systems, ECOOP'98 Workshop Reader*, *Lecture Notes in Computer Science*, Springer-Verlag, juillet 1998.

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

[26] D. Le Métayer, M. Périn, « Multiple views in software architecture: consistency and conformance », *in : First Working IFIP Conference on Software Architecture (WICSA1)*, à paraître.

[27] J. Mallet, « Symbolic cost analysis and automatic data distribution for a Skeleton-based Language », *in : Euro-Par'98 Parallel Processing*, *Lecture Notes in Computer Science*, Springer-Verlag, p. 688–697, septembre 1998.

[28] S. Mallet, M. Ducassé, « Pilotage d'un méta-interprète ensembliste par une trace "relationnelle" pour le débogage de bases de données déductives », *in : Journées Francophones de Programmation Logique et programmation par Contraintes*, O. Ridoux (éditeur), JFPLC'98, Hermes, p. 151–165, mai 1998.

[29] S. Mallet, M. Ducassé, « A set-oriented meta-interpreter driven by a "relational" trace for deductive database debugging », *in : LOgic-based Program Synthesis and TRansformation*, P. Flener (éditeur), LOPSTR'98, juin 1998.

[30] F. Malésieux, O. Ridoux, P. Boizumault, « Abstract compilation of λProlog », *in : Proceedings of the Joint Int. Conf. and Symp. Logic Programming*, J. Jaffar (éditeur), MIT Press, juin 1998. Version française dans [10].

Rapports de recherche et publications internes

[31] M. Ducassé, « Coca: a debugger for C based on fine grained control flow and data events », *rapport de recherche n° 3489*, Inria, septembre 1998.

[32] T. Jensen, D. Le Métayer, T. Thorn, « Semantic verification of programming language based security policies », *rapport de recherche PI-1210*, Irisa, octobre 1998.

[33] J. Mallet, « Compilation of a skeleton-based parallel language through symbolic cost analysis and automatic distribution », *rapport de recherche n° 3436*, Inria, mai 1998.