INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Projet LANDE

Logiciel : analyse et développement

Rennes

THÈME 2A

Table des matières

1. Composition de l'équipe

Responsable scientifique
Thomas Jensen [CR CNRS]

Assistantes de projet
Catherine Godest [TR CNRS, jusqu'en octobre 2002]
Lydie Letort [TR INRIA, à partir de novembre 2002]

Personnel Inria
Pascal Fradet [CR]
Arnaud Gotlieb [CR, à partir de novembre 2002]
Florimond Ployette [IR, Atelier]

Personnel Université Rennes 1
Olivier Ridoux [professeur]
Thomas Genet [maître de conférences]

Personnel Insa
Mireille Ducassé [professeur]

Personnel ENS Cachan
David Cachera [maître de conférences, à partir de juin 2002]

Chercheurs doctorants
Frédéric Besson [allocataire MENRT]
Sébastien Ferré [boursier CNRS/RÉGION, jusqu'en octobre 2002]
Yoann Padioleau [allocataire MENRT]
Jean-Philippe Pouzol [boursier Inria]
Valérie Viet Triem Tong [allocataire MENRT]
Thomas de Grenier de Latour [allocataire MENRT]
Thomas Lefort [boursier Inria]
Stéphane Hong Tuan-Ha [boursier Inria-Region]
David Pichardie [allocation couplée, ENS Cachan]
Gurvan Le Guernic [allocataire MENRT]

Ingénieurs experts
François Monin [ingénieur expert, jusqu'en août 2002]
Marc Éluard [ingénieur expert]

Chercheur post-doctorant
Wendelin Serwe

2. Présentation et objectifs généraux

Le thème de recherche central du projet Lande est la conception d'outils d'aide au développement et à la validation de logiciels. Notre approche est fondée sur une collection de méthodes formelles permettant de spécifier ou d'extraire une *vue partielle* de l'architecture et du comportement d'un logiciel. Cette approche nous a amené à étudier deux types de problèmes. D'une part, la spécification d'un logiciel en vues partielles nécessite une vérification de la *cohérence* entre plusieurs vues afin que celles-ci puissent être synthétisées. D'autre part, l'extraction d'une vue demande des techniques d'analyse statique et dynamique précises. Nous insistons sur la nécessité que les réponses apportées à ces problèmes reposent sur des bases formelles (sémantique du langage étudié, définitions de propriétés à vérifier dans des logiques formelles) afin d'obtenir une garantie sur les résultats produits. De plus, il est important que les outils construits soient les plus automatiques possible car les utilisateurs visés ne sont pas nécessairement des experts en méthodes formelles.

Lande est un projet commun avec le CNRS, l'Université de Rennes 1 et l'Insa de Rennes.

2.1.1. Axes de recherche

La **description multi-vues** de l'architecture de gros logiciels a comme objectif de spécifier l'organisation globale de systèmes afin d'améliorer la maîtrise de leur développement (spécification, analyse, programmation, 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. Nos travaux visent à assurer une forme de cohérence de ces descriptions hétérogènes. Le défi principal est de trouver comment mettre en relation des vues qui mettent en jeu des propriétés de nature très différentes ou qui se situent à des niveaux d'abstraction différents. Une fois les vues mises en relation, il devient possible d'utiliser des techniques standards d'analyse statique ou de vérification afin d'assurer des propriétés de cohérence.

La nouvelle technique de programmation appelée « **programmation par aspects** » consiste à décrire un logiciel comme un ensemble formé d'un composant principal et d'une collection de vues ou 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. Après avoir proposé un aspect dédié à la sécurisation de code mobile, nous avons étudié les problèmes d'interactions qui se posent lorsque l'on doit tisser plusieurs aspects. En effet, comme pour les vues, les aspects ne sont pas obligatoirement orthogonaux et des conflits ou ambiguïtés peuvent apparaître lors du tissage. Nous travaillons également sur un langage d'aspect dédié à la composition de composants.

Pour faciliter la navigation et l'organisation de logiciels de taille importante nous cherchons à définir **un cadre logique pour les systèmes d'information** qui décrit uniformément la navigation, l'interrogation et l'analyse des données. Ce cadre est générique par rapport à la logique utilisée pour naviguer et interroger ; en particulier il peut être appliqué à plusieurs types de logiques de programme, comme les types ou des propriétés statiques. Ces travaux se basent sur une extension de la théorie de l'analyse de concepts.

La validation d'un logiciel utilise des méthodes d'**analyses et de test de programmes**. Nous nous intéressons à différents aspects de l'**analyse statique** de programmes, aussi bien sur le plan des fondements (spécification d'analyses à partir de règles d'inférence) que des applications (détection des pointeurs pendants pour l'aide à la mise au point de programmes C, analyse de flots de données et de contrôle dans des programmes Java et Java Card, analyse de protocoles cryptographiques) et à la mise en œuvre d'analyses statiques par des techniques de résolution itérative de systèmes d'équations et de réécriture d'automates d'arbres.

Pour faciliter l'**analyse dynamique** de programmes, nous développons un outil d'analyse de traces d'exécutions permettant à l'utilisateur d'exprimer des requêtes dans un langage de programmation logique. Ces requêtes peuvent être traitées à la volée, ce qui permet d'analyser des traces de grande taille. L'outil peut être utilisé pour le débogage de programmes séquentiels et nous étudions maintenant son application au problème de la détection d'intrusion.

En collaboration avec la société AQL nous poursuivons le développement de l'outil Casting dont le noyau utilise une méthode de **génération de suites de tests**. L'outil peut prendre des entrées dans des formats variés et produit des suites de tests selon des stratégies spécifiées par l'utilisateur. Nous adaptons actuellement Casting pour la génération de suites de test à partir de spécifications UML.

La **sécurité logicielle** constitue un domaine d'application privilégié pour le projet. Nous élaborons un cadre pour la définition de propriétés de sécurité et une technique pour leur vérification automatique. Cette technique intègre des techniques d'analyse statique et de vérification de modèle (« *model checking* »). Ce cadre a été appliqué à la formalisation et la vérification de politiques de sécurité d'applications programmées avec la nouvelle architecture de sécurité de Java 2 et à la vérification de propriétés de sécurité des cartes à puce multi-applicatives programmées avec le langage Java Card.

3. Fondements scientifiques

3.1. Introduction

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.

3.2. Sémantique des langages de programmation

Mots clés : *sémantique*, *sémantique dénotationnelle*, *sémantique opérationnelle*.

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

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

$E \quad ::= \quad N \quad | \quad E_1 \quad + \quad E_2$ où N représente un entier

3.2.1. Sémantiques dénotationnelles

Une sémantique dénotationnelle [66] attribue un sens aux programmes d'un langage à l'aide d'une fonction qui associe à chaque construction syntaxique du langage une valeur dans un domaine de définition. La sémantique d'une expression est construite à partir de celle de ses sous-expressions ; on dit que la sémantique est compositionnelle. La technique de preuve classique quand on travaille avec de telles sémantiques est la récurrence sur la structure (*structural induction*).

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

ε	:	Expression $\longrightarrow Nat$
$\varepsilon[\![N]\!]$	=	$Val(N)$
$\varepsilon[\![E_1 \quad + \quad E_2]\!]$	=	$Plus \quad (\varepsilon[\![E_1]\!], \varepsilon[\![E_2]\!])$

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*).

3.2.2. 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) [70] 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 :

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

$$\frac{E_1 \Longrightarrow E_1'}{E_1 \quad + \quad E_2 \Longrightarrow E_1' \quad + \quad E_2} \qquad \frac{E_2 \Longrightarrow E_2'}{N \quad + \quad E_2 \Longrightarrow N \quad + \quad E_2'}$$

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* [59] 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.

$$N \Longrightarrow N$$

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

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

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

3.3. 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.

L'analyse de programmes désigne l'ensemble des techniques qui permettent de déduire mécaniquement des propriétés des programmes. Ses principaux domaines d'application sont la compilation et l'aide à la mise au point de programmes. Comme exemples d'analyses existantes, nous pouvons citer l'analyse d'alias, le *slicing*, les analyses de dépendances. Une analyse peut être dynamique, elle porte alors sur une trace d'exécution particulière, ou statique, et valable pour toute exécution de programme. Dans les deux cas, sa correction doit être assurée, ce qui signifie que les informations qu'elle procure doivent être cohérentes avec la sémantique du programme analysé.

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

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

On distingue deux classes d'analyses : les analyses dynamiques et les analyses statiques. L'analyse dynamique déduit des propriétés d'un programme à partir d'une trace d'exécution particulière [49]. En revanche, l'analyse statique [60] 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 3.2). La correction et la précision des analyses ont été étudiées de manière extensive dans le cadre de l'interprétation abstraite [50]. 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.

3.4. Débogage

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

Glossaire

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

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

Panne Une panne est l'incapacité d'un programme à effectuer ses fonctionnalités requises. Une faute peut générer une ou plusieurs pannes [44]. Un exemple de panne résultant de la faute citée plus haut est que le programme ne termine pas.

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

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

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

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

1. La première tâche consiste à déterminer les informations qui doivent apparaître dans la trace. La trace est calquée sur la sémantique opérationnelle du langage sans forcément en donner tous les détails. Elle fournit une abstraction des étapes de calcul dont l'objectif est la compréhension par l'utilisateur du comportement des programmes. Elle dépend donc du langage et du type d'utilisateur potentiel.
2. La deuxième tâche est la mise en œuvre des traceurs qui nécessite l'insertion d'instructions de trace dans les mécanismes d'exécution des programmes (appelée *instrumentation* dans la suite). Cette instrumentation peut se faire à différents niveaux : dans le code source, dans le compilateur ou dans l'émulateur quand il en existe un. La pratique courante consiste à instrumenter à un niveau bas [64] 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 3.3) 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 3.2) et le filtrage réalise une approximation permettant de rendre décidable la propriété recherchée.

3.5. Test de logiciels

Mots clés : *critère de test*, *hypothèse de test*, *test unitaire*, *test d'intégration*, *test fonctionnel*, *test système*, *test en boîte noire*, *test en boîte blanche*, *test structurel*.

Glossaire

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

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

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

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

Complétude Un critère est dit complet pour un programme s'il produit uniquement des jeux de test qui suffisent à déterminer la correction du programme (pour lequel tout programme passant le jeu de test avec succès est correct) [68]. Tout critère valide et fiable est complet.

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

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

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

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 [68]. 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 [45]. Ces techniques reposent sur l'examen de la structure du programme et le calcul de flots de contrôle ou de données. Les tests produits sont dits *structurels*.

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

3.6. Langages déclaratifs

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

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 [65] et le calcul des prédicats. Dans les deux cas, les programmes sont des *formules* mais elles sont interprétées différemment. L'opération essentielle des langages fonctionnels est la *réduction* qui permet de remplacer une formule par une autre formule équivalente, mais plus « simple », jusqu'à obtenir une formule qui n'est plus réductible, et que l'on appelle une *forme normale*. On convient que cette forme normale est le résultat du calcul. L'opération essentielle des langages logiques est la *déduction*. On l'emploie pour construire des preuves, et on convient que le résultat du calcul est extrait de ces preuves. Il s'agit le plus souvent des valeurs données dans les preuves à certaines variables. Pour autant que la correspondance de Curry-Howard s'applique (langages fonctionnels typés), la preuve est l'objet commun à

ces deux familles de langages de programmation ; les langages fonctionnels les normalisent, et les langages logiques les construisent.

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

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

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

3.7. Analyse de concepts

Mots clés : *contexte formel*, *concept formel*, *intention/extension*, *treillis de concepts*.

L'analyse de concepts est une formalisation des notions philosophiques de concept, extension de concept (les individus qui sont sensés appartenir à un concept) et compréhension de concept (les notions communes à tous les individus d'un concept) [54].

La formalisation part d'un *contexte formel*, qui est en fait une relation entre individus et descriptions. Les individus sont les objets du discours, par exemple des entrées dans un catalogue, des fonctions d'un programme, ou des points d'une carte. On considérera que les descriptions sont des ensembles d'attributs, par exemple « 330 Mhz et 500 Moctets », même si nous avons montré qu'on peut généraliser les descriptions à une logique arbitraire pour peu que sa relation de déduction forme un treillis [53]. On peut alors trouver les relations suivantes : « fréquence :330 Mhz et capacité :500 Moctets » ou « type :$bignum \longrightarrow int \longrightarrow bignum$ et pré :$Arg_1 = 0 \Longrightarrow Arg_2 \geq 0$ ».

Il est naturel de se demander quels sont les individus qui satisfont telle description, car ils ont tous les attributs de cette description, et quelle est la description commune à un ensemble d'individus, l'intersection des attributs de ces attributs. Cela définit une fonction τ des descriptions vers les ensembles d'individus, et une fonction σ des ensembles d'individus vers les descriptions. Théorème : ces fonctions forment une connexion de Galois :

$$I \leq_i \tau(D) \longleftrightarrow D \leq_d \sigma(I)$$

où $\leq_i$ est l'inclusion ensembliste (des ensembles d'individus) et $\leq_d$ est la relation inverse de l'inclusion ensembliste (des ensembles d'attributs). Dans le cas où les descriptions sont généralisées à une logique arbitraire, $\leq_d$ est la relation de déduction des descriptions.

Les fonctions $\tau\sigma$, sur les ensembles d'individus, et $\sigma\tau$, sur les descriptions, sont donc des opérations de fermeture par rapport à un contexte formel. Cela veut dire qu'un contexte formel détermine des ensembles d'individus et des descriptions qui sont plus canoniques que les autres. Par contre, les autres ensembles d'individus ou descriptions peuvent toujours être canonisés par application de $\tau\sigma$ ou $\sigma\tau$. Noter que des ensembles d'individus différents peuvent avoir la même fermeture (de même pour les descriptions). On dispose donc de plusieurs moyens pour exprimer les mêmes choses.

Un concept formel est une paire (I, D) telle que $D = \sigma(I)$ et $I = \tau(D)$. On appelle I l'*extension* du contexte formel, et D son *intention*[1]. I et D sont donc des fermetures. En fait, on peut désigner un concept formel par n'importe quel ensemble d'individus i ou description d de la façon suivante : $(\tau(d), \sigma\tau(d))$ ou $(\tau\sigma(i), \sigma(i))$. On dit alors que i, ou d, étiquète le concept.

[1] Le vocabulaire de la philosophie utilise aussi le mot *compréhension*. Certains écrivent *intension*.

Le théorème fondamental de l'analyse de concepts formels est que l'ensemble des concepts qu'on peut former dans un contexte formel constitue un treillis quand on l'ordonne par inclusion des extensions. Dans un tel treillis, un individu étiquète un concept si et seulement si l'individu est présent dans tous les concepts supérieurs. De même, une description étiquète un concept si elle est satisfaite par tous les concepts inférieurs.

Traditionnellement, l'analyse de concept est mise en œuvre dans des systèmes d'analyse a posteriori de faits bruts constituant le contexte formel. C'est alors une méthode d'analyse des données, mais nous préférons étudier son emploi dans des systèmes plus dynamiques où elle est considérée comme un modèle d'organisation. Par exemple, munie d'une logique permettant de décrire les différentes vues d'une architecture logicielle, l'analyse de concepts devient un environnement de développement de logiciels qui intègre la navigation dans l'architecture. D'où l'idée de la mettre en œuvre dans un système de fichiers qu'on appellera « système de fichiers logique » [52].

Dans un système de fichiers logique, l'analogue du chemin Unix est la description. Une description désigne un répertoire (virtuel) qui est le concept qu'elle étiquète. On dit qu'un individu appartient à un répertoire si il l'étiquète. Sont considérés comme des sous-répertoires tous les concepts inférieurs au répertoire. On peut ainsi reconstruire interrogation ($ls \quad d$) et navigation ($cd \quad d$) dans une perspective unifiée.

4. Domaines d'application

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

Les deux cibles privilégiées du projet Lande sont :

1. Les applications qui exigent un degré de confiance très important justifiant l'emploi de méthodes formelles. L'accent est mis en particulier sur la sécurité des informations (confidentialité, intégrité).
2. Les logiciels complexes ou qui nécessitent des modifications fréquentes : il s'agit du domaine d'excellence des langages de programmation déclaratifs.

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

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

- La première concerne les applications complexes ou exigeant un degré de confiance très important justifiant l'emploi de méthodes formelles. Il peut s'agir de logiciels critiques pour la confidentialité ou l'intégrité des informations, la sécurité des personnes. Sur ce thème, nous travaillons (dans le cadre de Dyade et des projets européens « Secsafe » et « Verificard ») à l'analyse et la vérification de transformateurs de bytecodes Java Card, avec comme domaine d'application la sécurité des cartes à puce (cf modules 6.2.4, 7.4 et 7.5).
  Les descriptions multi-vues ainsi que les langages d'architectures de logiciels constituent une démarche plus récente pour le développement de logiciels complexes. Sur ce thème, nous sommes impliqués dans une action industrielle (l'action CASTOR cf 7.6) qui porte sur la conception d'architectures sécurisées. Elle s'effectue en collaboration avec AQL, EADS Sycomore, TNI et le projet VERTECS de l'Irisa.

On peut citer également dans cette catégorie nos travaux sur la génération automatique de jeux de test qui s'est poursuivi dans le cadre du projet RNTL « Cote » avec les projets Inria VERTECS et TRISKELL ainsi que Softeam, IMAG, France Télécom R&D et Gemplus (cf module 7.2).

- La seconde cible de nos travaux concerne les logiciels qui nécessitent des modifications fréquentes : on peut citer par exemple les systèmes qui mettent en œuvre des ensembles de règles (de facturation, de réservation, etc.) devant suivre les évolutions du marché ou de la législation. Il s'agit du domaine d'excellence des langages de programmation déclaratifs. Par exemple, 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 et pour la reconnaissance de partitions musicales (cf module 5.3).

5. Logiciels

5.1. Générateur de suites de test Casting

Participant : Thomas Jensen.

Nous avons proposé une méthode de génération de suites de test qui forme le noyau de l'outil Casting[2] développé en collaboration avec Lionel Van Aertryck de 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 3.5). 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 3.5). D'un point de vue pratique, une stratégie de test correspond à un mode d'extraction de contraintes à partir du texte source. Ces contraintes caractérisent les jeux de données qui devront être engendrés pour chaque opération du système. Le graphe d'accessibilité symbolique indique l'ordre dans lequel les opérations peuvent être appliquées pour satisfaire toutes les contraintes. Dans le cas général en effet on ne peut faire l'hypothèse qu'une opération soit toujours applicable : selon l'état du système, il peut être nécessaire d'effectuer plusieurs opérations intermédiaires avant de pouvoir appliquer une opération donnée. La dernière phase consiste à explorer ce graphe en résolvant les contraintes associées pour générer les données de test effectives.

Ces travaux ont conduit au développement d'un outil d'aide à la génération de jeux de test. La version actuelle de Casting prend en entrée des spécifications dans la notation AMN de la méthode B [69] et fait appel à Ilog Solver[3] pour résoudre les contraintes engendrées.

[2] *Computer Assisted Software Testing*
[3] Ilog Solver est une marque déposée par Ilog.

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 Thomas Jensen (jensen@irisa.fr).

5.2. Débogueurs Coca et Morphine

Participant : Mireille Ducassé [correspondant].

Coca [51] est un débogueur automatisé pour C où le mécanisme des points d'arrêt est basé sur des événements relatifs à des constructions du langage. Ces événements ont une sémantique, alors que les lignes du code source utilisées par la plupart des débogueurs n'en ont pas. Morphine [56] est un débogueur automatisé pour le langage de programmation en Logique Mercury[67].

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

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

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

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

5.3. Compilateur λProlog

Participant : Olivier Ridoux [correspondant].

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 [48] et sur la mémoire Mali [62]. 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'Edimbourg 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 [63][46]).

5.4. Solveur de point fixes Reqs

Participants : Thomas Jensen, Florimond Ployette [correspondant].

De nombreux travaux ont été effectués sur les fondements de l'analyse sémantique, la correction et la précision des analyseurs. On peut regretter cependant le peu d'attention accordé jusqu'à présent à la conception d'outils d'analyse génériques. Nous travaillons à la mise au point d'un solveur de point fixe générique qui peut servir de base pour le développement de nouveaux analyseurs. Une analyse produit souvent comme résultat un système d'(in)équations sur un domaine de propriétés dont la solution représente l'information recherchée. La phase de résolution est indépendante du programme analysé ; une méthode de résolution de systèmes d'équations peut donc s'appliquer à des systèmes provenant de différentes analyses. Cette observation suggère la possibilité de réaliser un « moteur d'analyse », c'est-à-dire un outil générique de résolution de systèmes d'(in)équations qui peut servir de base pour implanter des analyseurs. La généricité évoquée ici est relative aux propriétés et aux domaines abstraits considérés. La solution d'un tel système peut être calculée de façon itérative comme le plus petit point fixe d'un opérateur défini par le système d'équations à résoudre.

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

Nous avons réalisé plusieurs expériences avec REQS pour implémenter des analyses provenant de différents types de langage (langages à objets, langage Signal). Dans le domaine de langages à objets, REQS a été utilisé pour implémenter une analyse de classes Java qui tient compte du traitement des exceptions. Cette analyse de classes produit un arbre de flot de contrôle qui sert de base à des analyses plus spécifiques comme la vérification de propriétés de sécurité.

Nous nous sommes intéressé à la phase « amont » du solveur d'équations c'est à dire au passage d'un programme à un système d'équations. Cette phase est spécifique à un langage et à une propriété à analyser. Nous avons voulu offrir à l'utilisateur plus de généricité en utilisant l'environnement de langage de programmation SmartTools du projet Oasis à l'Inria Sophia. Cet outil permet, à partir de la description du langage à analyser sous forme d'une syntaxe abstraite de produire un éditeur structuré ainsi que des outils

d'analyses sémantiques de langage. D'un point de vue développement, cette plateforme utilise la technique des « visiteurs »(Visitor patterns) automatisée et étendue permettant de décrire de manière structurée des analyses. Elle utilise également la technologie XML. L'expérimentation a porté sur l'analyse de classes de programmes Java à partir du bytecode. Nous avons défini une syntaxe abstraite du bytecode dans le formalisme SmartTools et nous avons utilisé des visiteurs pour générer les équations.

REQS a été utilisé dans deux analyses de programmes JavaCard :

- L'une, réalisée au SICS en Suède, utilise notre analyse de classe Java à partir du Bytecode pour l'analyse d'applets JavaCard. Elle est distribuée sur le web (http://www.sics.se/fdt/projects/vericode/jcave.html).
- L'autre concerne l'analyse de programmes Carmel (un sous ensemble de JavaCard) dans le cadre du projet SecSafe.

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

5.5. Librairie d'automates d'arbres : Timbuk

Participants : Thomas Genet [correspondant], Valérie Viet Triem Tong.

Timbuk [55] est une librairie de fonctions OCAML destinées à manipuler des automates d'arbres. La classe des automates d'arbres pouvant être décrits est la classe des automates finis ascendants déterministes et non déterministes. Cette librairie fournit les opérations classiques sur les automates d'arbres :

- opérations booléennes : intersection, union, complément,
- décision du vide, décision de l'inclusion,
- nettoyage, renommage,
- déterminisation,
- normalisation des transitions,
- calcul de l'automate reconnaissant l'ensemble des termes irréductibles pour un système de réécriture linéaire à gauche.

Mais, elle implante également plusieurs opérations plus spécifiques que nous utilisons pour la vérification de protocoles cryptographiques :

- la complétion d'un automate d'arbre par rapport à un système de réécriture donné,
- l'approximation de l'ensemble des descendants et de l'ensemble des formes normales d'un automate pour un système de réécriture donné,
- le filtrage dans les automates.

Ce logiciel est distribué sous la Gnu Library General Public License et disponible à l'URL suivante : http://www.irisa.fr/lande/genet/timbuk/

Une version 2.0 de Timbuk est en préparation. L'algorithme de complétion a été optimisé en temps d'exécution et en occupation mémoire. Cette nouvelle version proposera deux versions de la complétion : une version dynamique qui permet la construction pas à pas des approximations et une version statique effectuant une précompilation du filtrage et des approximations qui offre de meilleures performances. Cette nouvelle version (non encore distribuée) est actuellement utilisée dans une étude réalisée avec Thomson-Multimédia (voir section 6.2.2). D'autre part, Timbuk est utilisé par Frédéric Oehl et David Sinclair de l'université de Dublin également pour la vérification de protocoles cryptographiques dans une approche combinant un assistant de preuve (Isabelle/HOL) et des approximations (réalisées avec Timbuk) [61].

6. Résultats nouveaux

6.1. Méthodes de conception et de structuration de logiciel

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

6.1.1. Descriptions de systèmes multi-vues

Participants : Pascal Fradet, Thomas Lefort.

Mots clés : *description de systèmes*, *vues*, *mise en relation*, *cohérence*, *vérification*, *analyse*.

Nous travaillons sur des descriptions composées de multiples points de vue. Le but principal est d'assurer des propriétés de cohérence sur ce style de descriptions hétérogènes. Nous avons proposé et implanté un vérificateur d'une classe très riche de propriétés structurelles. Cet outil a également été utilisé pour vérifier l'absence de failles sur une vue sécurité. Nous travaillons actuellement sur la mise en relation et la vérification de propriétés de vues plus complexes décrites par des systèmes de transition.

Le développement de systèmes d'information complexes est particulier en ceci qu'il implique :

- des descriptions décomposant le système en sous-parties et le spécifiant à différents niveaux d'abstraction (de l'architecture globale aux composants de base),
- des descriptions de propriétés ou d'aspects très différents (flots de données ou de contrôle, communications, exigences non fonctionnelles),
- la participation de nombreux acteurs spécialisés dans une étape du développement, un sous-système ou un aspect particulier du système (par ex. la sécurité),
- l'utilisation d'outils (analyse de performance, de gestion de configuration) impliquant une description particulière du système.

Clairement, un unique outil, notation ou formalisme ne peut convenir à une telle variété de besoins et de participants. En conséquence, les gros systèmes sont le plus souvent décrits selon différents points de vue. Cependant, il est rare que les vues soient totalement indépendantes. Il est même possible que deux vues se contredisent. Si ce type d'incohérences persiste, on peut aboutir à des systèmes erronés ou impossible à implémenter. La multiplicité et l'hétérogénéité des descriptions rend nécessaire d'expliciter leurs dépendances et de les exploiter de manière à assurer une forme de cohérence.

Nous nous sommes intéressés à la vérification de propriétés de cohérence dans le cas où chaque vue est représentée par un graphe simple. La simplicité du modèle considéré (graphes annotés) a permis d'implanter un vérificateur pour une classe très riche de propriétés structurelles. Ce travail s'est concrétisé dans le cadre d'une action industrielle conduite en collaboration avec les sociétés AQL, EADS Sycomore et TNI, ainsi que les projets VERTECS et Lande de l'Irisa. L'objectif de ce projet est de fournir un environnement permettant de décrire l'organisation globale d'un système d'information sous forme de vues et d'étudier ses propriétés de sécurité. Notre vérificateur de propriétés a été intégré à l'environnement de modélisation de Castor et à également été utilisé pour l'analyse de propriétés (synthèse de parades/attaques) sur la vue sécurité [43].

Nous nous intéressons maintenant à la vérification automatique de propriétés de cohérence sémantique (ou de comportement) de descriptions hétérogènes. Les vues considérées sont décrites par des systèmes de transition dont les états sont annotés de formules logiques. Ce formalisme permet de décrire une grande variété de vues (structurelles ou comportementales). Le défi principal est de trouver comment mettre en relation des vues qui mettent en jeu des propriétés de nature très différentes ou qui se situent à des niveaux d'abstraction différents.

6.1.2. Agents mobiles pour services distribués

Participants : Pascal Fradet, Siegfried Rouvrais.

Mots clés : *architectures de logiciel*, *interactions*, *agent mobile*, *performance*, *sécurité*, *fiabilité*.

Nous avons étudié l'analyse de propriétés d'interactions exprimées comme des compositions d'agents mobiles, d'invocations distantes (RPC, Remote Procedure Call, et RMI, Remote Method Invocation), ou d'évaluations à distance. Les propriétés analysées sont la performance (en terme de volume de données échangées), la sécurité (confidentialité et intégrité) et la fiabilité des interactions. Nous avons illustré comment ce type d'information permet de guider le concepteur de systèmes distribués dans ses choix d'implantation.

Les agents mobiles ont été récemment proposés comme une nouvelle forme d'interaction des systèmes distribués. Une des raisons qui rendent les agents difficiles à utiliser est que leurs avantages ou inconvénients, comparés aux interactions classiques comme les RPC, portent sur des aspects non fonctionnels (comme les performances). Les propriétés non fonctionnelles sont souvent difficiles à appréhender et choisir la bonne combinaison d'interactions pour implanter un service complexe est une tâche délicate.

Nous avons étudié cette question en analysant et en comparant les différents styles d'interaction selon trois types de propriétés : les performances, la sécurité et la fiabilité [17][21]. Nous avons proposé un cadre linguistique pour spécifier et implanter les services complexes. Les agents mobiles, les invocations distantes, l'évaluation à distance ou toute combinaison de ces protocoles sont représentés comme des expressions fonctionnelles. Ces expressions peuvent alors être analysées et comparées simplement. Ces analyses permettent de guider le concepteur de systèmes distribués dans ses choix de protocoles. Ce cadre a été intégré à un environnement de développement basé sur un langage de description d'architectures de logiciels. Il intègre des outils destinés à guider le concepteur de services distribués complexes au regard des propriétés traitées.

Ce travail a été effectué en collaboration avec Valérie Issarny du projet Solidor (UR de Rocquencourt).

6.1.3. Programmation par aspects

Participants : Pascal Fradet, Stéphane Hong Tuan-Ha.

Mots clés : *interactions entre aspects*, *analyse et transformation de programme*, *construction de programme*.

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 étudié les problèmes posés par le tissage de plusieurs aspects (c'est à dire les interactions entre aspects). Nous étudions également un aspect dédié à la composition de composants. La notion d'« aspect » présente des similarités avec celle de « vue » présentée dans le module 6.1.1. En programmation par aspects, un logiciel est formé d'un composant principal et d'une collection d'aspects décrivant des tâches comme la sécurité, la synchronisation, le traitement des exceptions, 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 déclarer séparément des aspects dont la mise en œuvre serait sinon dispersée dans tout le programme.

Comme pour les vues, les aspects ne sont pas obligatoirement orthogonaux. Des interactions (conflits ou ambiguïtés) peuvent apparaître lorsque l'on tisse plusieurs aspects. Ce point est généralement traité de façon manuelle et ad-hoc. Le programmeur doit identifier les interactions et coder la résolution des conflits. Nous avons étudié les interactions possibles selon différents types d'aspects dans le but :

- de détecter à la compilation les conflits potentiels,
- d'assister le programmeur dans la résolution des conflits.

Afin d'explorer ces problèmes d'interactions indépendamment d'un langage de programmation ou d'aspect précis, l'étude se place dans un cadre formel et générique. Nous avons proposé deux analyses statiques détectant les interactions entre aspects ainsi que des constructions linguistiques permettant au programmeur de les résoudre [25][37]. Ce travail est l'objet d'une collaboration avec Rémi Douence et Mario Südohlt de l'école des Mines de Nantes.

La recherche en programmation par aspects consiste également à proposer des langages (et tisseurs) dédiés à certaines tâches. Nous avons proposé précédemment une méthode pour imposer des politiques de sécurité par tissage. Nous étudions maintenant un aspect d'assemblage de composants. L'aspect spécifie des ports d'entrées/sorties pour chaque composant ainsi que leurs interconnexions ; le tisseur, lui, se doit de fusionner, selon l'assemblage spécifié, les composants aussi efficacement que possible.

6.1.4. Systèmes d'information logiques

Participants : Sébastien Ferré, Yoann Padioleau, Olivier Ridoux.

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

Nous étudions la conception de systèmes d'information non-hiérarchiques offrant à la fois des possibilités de navigation et d'interrogation. Nous avons développé pour cela un modèle d'analyse de concepts logiques qui généralise l'analyse de concepts formels de Wille et Ganter [54][53].

Nous étudions l'extension de ce modèle pour la prise en compte de relations entre objets, une métaphore de navigation dans le treillis de concepts logiques, et les algorithmes de consultation, navigation et mise à jour. Sur ce dernier point, l'enjeu principal est de ne jamais construire entièrement le treillis de concepts logiques. La métaphore de navigation envisagée possède les caractéristiques suivantes : une formule désigne un endroit où l'on souhaite lire *ou* écrire, l'interrogation du contenu d'un endroit retourne entre autres la désignation d'endroits proches par des formules (combinant ainsi *interrogation* et *navigation*), et la manipulation des formules se fait aussi bien en prenant en compte leur intention que leur extension. Ce dernier point permet de faire des déductions qui sont correctes dans le contexte courant, mais pas en général. La plupart de ces résultats sont décrits dans la thèse de Sébastien Ferré [16].

Nous avons développé une méthode d'apprentissage des descriptions d'après le contenu des objets [29]. Cette méthode débouche de façon a priori inattendue sur un nouvel algorithme de formation incrémentale du treillis des concepts [38]. Celui-ci a les meilleures performances pour les contextes creux (*sparse*).

Le concepteur d'un système d'information logique doit définir la logique de description des individus. Afin de soulager sa tâche, nous avons étudié le moyen de définir une logique par le moyen de composants élémentaires que nous avons appelé *foncteurs logiques* [28]. Nous avons défini et mis en œuvre plusieurs foncteurs qui vont de la logique propositionnelle à la déduction sur les intervalles en passant par un foncteur pour une logique modale épistémique. Nous avons déterminé des critères de compatibilité (des types) qui spécifient quels sont les agencements de foncteurs qui forment des systèmes logiques corrects.

Un ensemble de foncteurs a été spécifiquement développé pour modéliser les types de Java modulo isomorphisme de type [22]. L'utilisation de ces foncteurs depuis un système d'information logique constitue un système de recherche de composants Java par leurs types. Un point important par rapport aux propositions de ce type est que le système d'information logique permet aussi la navigation. Ainsi, la réponse à une demande de composant d'un certain type n'est pas toujours une liste de composants, parfois longue, mais peut aussi être une liste de bribes de types qu'il faut composer avec la requête pour la préciser.

Nous avons réalisé un système de fichiers (sous Linux) qui offre les fonctions de l'analyse de concepts formels sous l'interface standard d'un système de fichiers virtuel [42]. Dans cette réalisation, seul le système de fichiers est spécifique, et les couches supérieures comme le *shell* restent inchangées. Nous recherchons maintenant comment mettre en œuvre un système de fichiers basé sur l'analyse de concepts logiques de façon la plus générique possible. Nous avons aussi développé un autre prototype qui met en œuvre toute la généralité des systèmes d'information logique, mais simule un système de fichiers en mémoire. Notre objectif est de réconcilier les deux types de prototype à moyen terme.

6.1.5. Transformations certifiées de programmes Java Card

Participants : Thomas Genet, Thomas Jensen, François Monin, David Pichardie.

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

Nous avons conçu et breveté une méthode qui permet de prouver la correction des transformations effectuées sur le code intermédiaire Java Card pour permettre son installation sur une carte à puce. Cette preuve a été

vérifiée avec l'assistant de preuve Coq. La formalisation de ces transformations forme la base de travaux qui visent à construire un transformateur Java Card certifié en Coq.

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

Pour prouver la correction de ces optimisations, nous avons développé un cadre qui permet de décrire la sémantique des deux formats [18]. La différence entre les deux formats est encapsulée dans des fonctions auxiliaires, ce qui permet d'utiliser le même système d'inférence pour spécifier les deux sémantiques. En utilisant la notion de relation logique, il a été possible de définir une relation liant une entité dans un format à l'entité lui correspondant dans l'autre format. La tâche restant à accomplir pour établir l'équivalence consiste alors à montrer que lesdites fonctions auxiliaires respectent ces relations ce qui représente une simplification majeure de la preuve. Cette technique a permis d'effectuer la preuve de correction avec l'assistance de l'outil Coq.

Dans le cadre du projet européen « Verificard » nous nous sommes servi de cette formalisation pour développer et prouver la correction des transformations du byte code Java Card. L'objectif est de fournir des transformations certifiées, développées à l'aide d'un assistant de preuve. Nous nous sommes focalisés sur deux transformations :

- la « tokenisation » d'une hiérarchie de classes Java Card, qui consiste à remplacer des noms (de classes, de méthodes *etc.* avec des « tokens » (des entiers) qui permettent à la fois une représentation plus compacte du code et une mise en œuvre plus efficaces des appels de méthodes virtuelles.
- la « componentisation », qui consiste à regrouper plusieurs fichiers de classe en un seul fichier CAP pour partager des données communes entre classes et pour permettre un adressage direct entre entités de différentes classes.

Les transformations ont été spécifiées dans le langage de spécification de l'assistant de preuve PVS. Les théorèmes énonçant la correction des transformations ont ensuite été prouvés formellement à l'aide de ce même outil.

6.2. Validation de logiciel

Nous présentons d'abord des résultats fondamentaux sur l'analyse de flot de contrôle de programmes fonctionnels et à objets (module 6.2.1) avant de détailler un certain nombre de recherches portant sur des analyses particulières : la vérification de politiques de sécurité (module 6.2.4) et la preuve de protocoles cryptographiques à l'aide de l'interprétation abstraite (module 6.2.2). Nous décrivons ensuite nos travaux sur la détection d'intrusion. Nous concluons avec la présentation de nos activités en matière de génération de jeux de test (module 6.2.8).

6.2.1. Fondements de l'analyse de flot de contrôle

Participants : Frédéric Besson, Thomas Jensen.

Mots clés : *analyse de flot de contrôle*, *types non-standard*, *interprétation abstraite*, *programmation fonctionnelle et à objets*.

L'analyse de flot de contrôle est fondamentale pour la vérification de programmes fonctionnels et à objets dont le flot de contrôle ne se déduit pas par une analyse syntaxique du programme. Nos travaux ont porté sur l'analyse de flot de contrôle pour les langages à objet comme Java et Java Card. Nous avons en particulier étudié le problème d'analyse modulaire d'un fragment d'une hiérarchie de classes Java.

L'analyse de flot de contrôle est fondamentale pour la vérification de programmes fonctionnels et à objets dont le flot de contrôle ne se déduit pas avec une analyse syntaxique du programme. Pour les programmes à objets avec méthodes virtuelles, une analyse de flot de contrôle doit déterminer une sur-approximation de l'ensemble de méthodes qui peuvent effectivement être appelées par l'invocation d'une méthode virtuelle. À partir de cette information, l'analyse produit un graphe de flot de contrôle qui à chaque appel de méthode indique où dans le programme l'exécution va se poursuivre. Ces graphes servent ensuite pour rendre d'autres analyses plus précises ou pour vérifier des propriétés comme les propriétés de sécurité (voir 6.2.4).

Nous travaillons autour de l'analyse de flot de contrôle pour les langages à objet comme Java et Java Card. Ce travail concerne à la fois la conception de nouvelles analyses adaptées à traiter des fragments de programmes et la mise en œuvre d'analyses existantes pour fournir des graphes de flot de contrôle pour les vérifications de propriétés de sécurité (voir 6.2.4 et 6.2.5).

Nous avons conçu une analyse qui permet d'analyser le flot de contrôle inter-procédurale d'un fragment d'une hiérarchie de classes Java. L'analyse s'exprime sous forme d'un schéma de traduction qui à partir du code Java engendre un ensemble de contraintes DATALOG qui décrivent le flot d'objets. Les contraintes peuvent ensuite être normalisées. La composition des résultats de l'analyse de deux fragments de code Java se réduit ensuite à la composition de systèmes de contraintes suivie par une normalisation [15].

6.2.2. Preuve et interprétation abstraite pour la vérification de protocoles cryptographiques

Participants : Valérie Viet Triem Tong, Thomas Genet.

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

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

La vérification des protocoles cryptographiques nécessite des techniques d'analyse particulièrement fines et sophistiquées. En effet, à cause des enjeux notamment dans le secteur bancaire, la sécurité de ces protocoles doit être garantie contre tout type d'attaque et ceci dans un contexte d'utilisation très général : pas de limites sur le nombre de sessions ou sur le nombre d'acteurs exécutant le protocole en même temps. Actuellement, la principale technique de vérification des protocoles cryptographiques - le model-checking - permet de découvrir certaines attaques, mais ne permet pas de garantir un protocole contre leur existence.

En utilisant des techniques de réécriture et d'automates d'arbres, nous avons proposé une méthode de vérification de protocoles cryptographique [9] utilisant une forme d'interprétation abstraite et un mécanisme de déduction simple sur des contraintes ensemblistes. À partir d'une description d'un protocole sous la forme d'un système de réécriture et d'un ensemble de requêtes possibles décrit par un automate d'arbre, nous calculons automatiquement une sur-approximation du comportement du protocole sur les requêtes possibles. Cette approximation est obtenue sous la forme d'un automate d'arbre reconnaissant un sur-ensemble des messages pouvant être échangés entre un nombre quelconque d'acteurs, pendant un nombre quelconque de sessions. À partir de ce sur-ensemble, pour montrer que le protocole a les propriétés souhaitées, il est suffisant de montrer que l'intersection entre l'ensemble approximation et un ensemble contenant tous les cas d'erreurs est vide. Ce travail a été réalisé en collaboration avec Francis Klay de France Telecom R&D.

Actuellement, nous affinons cette technique d'un point de vue théorique comme d'un point de vue pratique. Du point de vue théorique, tout d'abord, nous avons proposé une première version d'un système de déduction dédié à la vérification combinant naturellement preuve par induction et interprétation abstraite [41]. Le système de déduction proposé s'appuie sur un mécanisme d'induction par réécriture très proche de celui utilisé dans le système SPIKE (projet CASSIS) mais spécialisé dans la preuve de conjectures négatives. L'interprétation abstraite, quant à elle, est injectée dans la preuve sous la forme d'équations additionnelles qui, en identifiant des termes a priori différents, construit des classes d'équivalences (i.e. le domaine abstrait).

D'un point de vue plus pratique, cette technnique est actuellement utilisée au travers de l'outil Timbuk (voir section 5.5) dans une étude réalisée avec Thomson-Multimédia sur la vérification de protocoles de la norme SmartRight de protection des contenus numériques contre la copie. Sur ces protocoles nous nous intéressons notamment à la preuve de secret, d'authentification et de « non-rejeu » modélisant l'impossibilité de visionner un contenu numérique plusieurs fois (fonctionnalité de type *pay per view*).

En dehors du cas particulier de la vérification des protocoles cryptographiques et de l'interprétation abstraite, l'intérêt d'un système comme Timbuk est de prouver facilement des propriétés sur des systèmes de réécriture non terminants qui sont généralement en dehors du spectre des outils classiques de démonstration pour lesquels la terminaison est indispensable [55]. Enfin, Timbuk doit prochainement être intégré à l'outil Reqs (voir section 5.4), afin d'offrir des facilités pour la résolution d'équations récursives sur des domaines d'arbres.

6.2.3. Algorithmes itératifs pour le calcul de points fixes

Participants : Thomas Jensen, Florimond Ployette, Olivier Ridoux.

Mots clés : *analyse de programmes*, *treillis*, *itération*, *systèmes d'équations recursives*.

Nous avons étudié comment optimiser la résolution de systèmes d'équations recursives par itération de points fixes. L'algorithme classique de « *work-set* » a été optimisé en utilisant les dépendances entre variables pour ordonnancer l'évaluation des équations. L'algorithme fait partie de l'outil REQS (*cf. Section 5.4*).

Nous avons étudié comment optimiser la résolution de systèmes d'équations récursives par itération de points fixes [30]. L'algorithme classique de « *work-set* » gère un ensemble d'équations à itérer. Nous avons étudié comment utiliser les dépendances entre variables pour ordonnancer l'évaluation des équations. L'observation de base est qu'il est possible d'éviter des calculs inutiles si on suspend l'évaluation des expressions qui dépendent d'autres variables présentes dans le « *work-set* ». Des variations sur ce schéma sont possibles en fonction du type de dépendance (dépendance directe, dépendance indirecte, dépendance sémantique ou syntaxique,...). Nous avons effectué des expériences avec deux types d'analyse de programmes : une analyse d'alias pour les programmes C et une analyse de flot de contrôle pour Java. Les expériences montrent que l'optimisation apporte une réduction en nombre d'itérations effectuées mais que cette réduction ne se traduit pas toujours dans une amélioration dans le temps d'exécution absolu parce que la gestion du graphe de dépendances rend l'ordonnancement trop coûteux. Il est donc nécessaire d'identifier d'autres types d'approximations des dépendances réelles qui se calculent facilement.

6.2.4. Analyse de propriétés de sécurité

Participants : Frédéric Besson, Marc Eluard, Thomas Jensen, Thomas de Grenier de Latour.

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

Nous avons proposé un cadre de définition de politiques de sécurité reposant sur une logique temporelle linéaire à deux niveaux et nous avons proposé une méthode de « model checking » automatique pour la vérification de propriétés de cette logique. Ce cadre a été utilisé pour vérifier la sécurité d'applications Java utilisant le mécanisme de l'*inspection de la pile* pour mettre en œuvre le contrôle d'accès. Nous avons notamment étudié comment analyser des fragments de code afin d'extraitre des pre-conditions qui garantissent que l'execution d'une méthode se fait sans qu'une politique de sécurité soit violée.

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 [58][57].

Dans une communication précédente, nous avons proposé une méthode automatique (et complète) de vérification de propriétés pour cette logique [47]. Cette méthode de vérification était soumise à certaines restrictions ; notamment elle ne s'appliquait qu'aux programmes entiers. Nous avons montré comment modulariser cette méthode de vérification afin de mieux analyser des fragments de code et des bibliothèques de code [23]. Cette technique est basée sur la génération et la résolution de contraintes sur des domaines de formules de logiques temporelles. Couplée avec l'analyse de flot de contrôle modulaire (décrite en 6.2.1), cette technique constitue un pas important vers une vérification complétement modulaire de la sécurité du code utilisant l'inspection de la pile.

6.2.5. Analyse de sécurité d'applettes Java Card

Participants : Marc Eluard, Thomas Jensen.

Mots clés : *cartes à puce multi-applicatives*, *Java Card*, *pare feu*, *sémantique opérationnelle*, *résolution de contraintes*.

Le langage Java Card définit un modèle de sécurité différent du modèle de Java. Par défaut, les applettes installées sur une carte Java Card sont séparées par un pare-feu et ne peuvent communiquer entre elles. La communication entre applettes se fait par la création et les échanges des objets partagés. Nous nous sommes attaqués au problème de vérifier que les accès aux données rendus possibles par des objets partagés ne divulguent pas des informations confidentielles. Pour calculer une approximation sûre des accès effectués pendant l'exécution, nous nous appuyons sur des analyses statiques de flot de données et de contrôle. Nous avons conçu une analyse de flot de contrôle pour Java Card qui prend en compte les actions du pare feu sur les interactions entre applettes pour modéliser finement les comportements possibles d'une applette Java Card. Cette analyse nous a amené à modifier l'algorithme itératif de résolution de systèmes de contraintes pour qu'il génère le système à resoudre de façon paresseuse en fonction des resultats obtenus par les itérations précedentes [36]. Ces travaux se poursuivent dans le projet européen IST SECSAFE (cf module 7.4) où nous participons à la mise en œuvre d'une analyse complète pour un langage intermédiaire de Java Card.

6.2.6. Analyse de trace automatisée

Participant : Mireille Ducassé.

Mots clés : *débogage*, *analyse dynamique*, *traceur abstrait*, *Prolog*, *Mercury*, *CLP(FD)*, *langage C*.

Nous avons développé des outils qui analysent les traces d'exécution de programmes générées par un traceur bas niveau. Les programmeurs peuvent spécifier de manière précise ce qu'ils veulent observer du comportement du programme à l'aide de requêtes exprimées en Prolog. À partir du langage de requêtes, nous avons bâti des analyses qui fournissent des vues abstraites des exécutions selon certains critères. Il a également été possible de mettre en œuvre à faible coût des traceurs abstraits pour des langages de haut niveau. Les

techniques de base exploitent une trace dont le seul pré-requis est d'être séquentielle. Nous avons montré cette généralité en appliquant ces principes aux langages Prolog, Mercury, C et CLP(FD) (programmation logique avec contraintes sur domaines finis).

Nous avons développé des outils (cf. module 5.2) qui analysent les traces d'exécution de programmes générées par un traceur bas niveau (cf. module 3.4). Le programmeur peut poser des questions sur les exécutions à l'aide de Prolog et de quelques primitives dans une forme concise en s'appuyant sur la logique et les mécanismes de recherche du langage. Les programmeurs peuvent donc, à l'aide de requêtes de haut niveau, spécifier de manière précise ce qu'ils veulent observer du comportement du programme.

Le prototype initial, Opium [4], était essentiellement dédié au débogage. Nous avons montré avec Morphine comment il est possible d'utiliser un traceur doté d'un module d'analyse de traces pour faire davantage que du débogage. Par exemple, nous pouvons calculer un taux de couverture de jeu de test afin d'en évaluer la qualité. Des « moniteurs » qui surveillent le comportement des programmes peuvent également être facilement mis en œuvre. Ainsi, au lieu de fabriquer des instrumentations *ad hoc* comme c'est le cas actuellement pour de tels outils, on peut utiliser un environnement uniforme, ce qui permet une synergie entre les outils. De plus, les instrumentations *ad hoc* nécessitent de bien connaître le système à instrumenter, que l'instrumentation se fasse au niveau bas ou par transformation de source à source. Même si elle n'est pas techniquement très difficile, cette tâche demande un effort de programmation non négligeable. Dans notre cas, l'instrumentation étant générique, elle est faite une fois pour toutes et les analyses spécifiques peuvent être relativement simples [19].

Nous avons un projet RNTL en cours sur l'analyse de traces d'exécutions de programmes logiques et avec contraintes (cf. module 7.1). Dans ce cadre, nous collaborons avec l'INRIA Rocquencourt sur la conception d'un modèle de trace et la méthodologie de conception de traceurs [24][31]. Un prototype d'analyse de traces a également été expérimenté [26][27]. À cette occasion, M. Ducassé co-encadre avec P. Deransart la thèse de Ludovic Langevine.

6.2.7. Détection d'intrusions

Participants : Mireille Ducassé, Jean-Philippe Pouzol.

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

Dans la continuité des travaux sur l'analyse de traces, Lande démarre une activité autour de l'analyse d'audits dans le cadre de la détection d'intrusions. La détection d'intrusions est une analyse de l'activité d'un système visant à rapporter à l'officier de sécurité toute utilisation suspecte. Nous concevons un langage de haut niveau de spécification de signatures d'attaques.

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

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

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

Nous concevons un langage de haut niveau de spécification de signatures d'attaques. Les signatures d'attaques sont les manifestations des attaques dans les systèmes attaqués. Dans les systèmes existants, les signatures d'attaques sont très proches des algorithmes de détection. De ce fait, elles contiennent trop de détails de bas niveau. Des approches récentes ont proposé des langages déclaratifs de spécification de signatures qui élèvent le niveau d'abstraction. Cependant, ces langages n'ont pas toujours de support opérationnel.

Nous montrons comment transformer des signatures déclaratives en signatures opérationnelles. Un langage de spécification de signatures, appelé *Sutekh*, est proposé. Un algorithme qui produit automatiquement des règles de détection pour des systèmes existants est défini [33].

Ces travaux se situent dans le contexte du projet RNTL Dico (cf module 7.3). Dans ce cadre nous collaborons également avec France Télécom R&D de Caen et Supelec de Rennes pour concevoir un modèle de données permettant la corrélation d'alertes issues d'outils de détection d'intrusions avec des informations issues d'autres sources, par exemple des analyseurs de vulnérabilités [32]. Sur ce sujet, M. Ducassé co-encadre la thèse de Benjamin Morin avec Hervé Debar et Ludovic Mé.

6.2.8. Génération automatique de cas de test par programmation (logique) avec contraintes

Participants : Arnaud Gotlieb, Thomas Jensen.

Mots clés : *test structurel*, *test fonctionnel*, *extraction de contraintes*, *OCL.*

Les techniques de test logiciel qui s'appuient sur l'utilisation de la programmation par contraintes constituent actuellement un sujet de recherche prometteur. Nous avons proposé de générer automatiquement des cas de test structurel à partir de la résolution de systèmes de contraintes extraits d'un programme source, écrit dans un langage impératif. Les travaux que nous menons actuellement visent à etendre cette approche aux extensions objet de ces langages. En particulier, le traitement des références, de l'héritage et du polymorphisme sont des points durs qui nous intéressent particulièrement, bien que la mise au point d'un noyau stable et performant pour notre approche soit encore à l'ordre du jour. Nous avons défini des opérateurs de contraintes qui modélisent de manière exacte la sémantique opérationnelle des instructions d'accès et de mise à jour de la mémoire. Ces opérateurs s'appuient sur une représentation abstraite de la mémoire à l'aide de tableaux. La résolution des systèmes de contraintes engendrés pour chaque programme (par des techniques de filtrage par consistance partielle) permet l'obtention de données de test qui respectent un objectif de test structurel.

Nous travaillons également à la mise au point d'un outil capable de générer automatiquement des données de test fonctionnel à partir d'expressions OCL présentes dans les diagrammes de classe et les diagrammes d'état d'une spécification UML. Une décomposition est opérée sur les préconditions, postconditions et invariants présents dans ces diagrammes, ce qui permet d'obtenir une formule représentant des classes d'équivalence issues de la spécification. La résolution des systèmes de contraintes obtenus après la mise en forme DNF de cette formule permet de générer des cas de test fonctionnels [34].

Ces travaux se font en partie dans le cadre du projet RNTL « Cote » (voir 7.2) et constituent un pas supplémemtaire vers l'automatisation complète des procédures de test fonctionnel et structurel des logiciels.

7. Contrats industriels

7.1. Projet RNTL OADYMPPAC

Participant : Mireille Ducassé.

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

Le projet OADYMPPAC a démarré officiellement le 15 novembre 2000, pour 3 ans. Un des objectifs du projet est de mettre en forme et d'expérimenter des techniques de trace génériques pour des programmes logiques avec contraintes. Cela facilitera la définition d'outils d'observation et de mise au point. Un autre aspect est d'étudier l'apport, à la conception de tels outils, des progrès réalisés en matière de visualisation d'information sur de grands ensembles de données. Cela permettra, en parallèle, la définition et l'expérimentation de nouveaux outils de mise au point. Il y a deux champs d'application : d'une part, la programmation avec contraintes et, en particulier, une meilleure compréhension du comportement des solveurs ; d'autre part le développement de techniques génériques de visualisation d'information adaptées à l'analyse visuelle des traces produites par les phénomènes dynamiques.

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

Le projet travaille, actuellement, sur les formats de traces communs aux divers outils. (cf module 6.2.6). Un modèle de trace [24][31] ainsi que des formats d'échange concrets (HTML) ont été spécifiés. Des analyses de traces ainsi que des outils de visualisation sont en cours de conception [26][27]. Le projet maintient une page d'information :
http://contraintes.inria.fr/OADymPPaC/.

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

7.2. Projet RNTL « Cote » : Test de composants

Participant : Thomas Jensen.

Mots clés : *Test*, *UML*.

Le projet Lande en collaboration avec Lionel van Aertryck de la société AQL participe avec les projets Triskell et Vertecs de l'Irisa au projet « Cote » qui a comme autres partenaires Softeam, IMAG, France Télécom R&D et Gemplus. Le projet est de nature « pré-compétitif » et a comme objectif de fournir des méthodes, techniques et outils pour tester et vérifier les composants logiciels. Le projet vise à réunir et étendre des techniques de test appliquées sur des modèles, afin de les transposer à l'approche de conception de logiciels par composants. Le cadre choisi est celui du langage UML qui fournit un langage suffisamment riche pour décrire à la fois l'architecture des composants et les cas de test. La participation du projet Lande concerne particulièrement la génération de cas de test à partir d'une stratégie de test choisie (voir aussi la description de l'outil Casting 5.1).

7.3. Projet RNTL Dico

Participants : Mireille Ducassé, Jean-Philippe Pouzol.

Mots clés : *Détection d'intrusions coopérative*.

Le projet DICO a démarré officiellement le 5 décembre 2001, pour 2 ans. L'objectif du projet est double. Il s'agit de proposer de nouvelles techniques pour, d'une part, détecter des malveillances et, d'autre part, réduire le nombre et améliorer la qualité des alertes générées par corrélation.

Les techniques de détection d'intrusions, bien que prometteuses, sont encore imparfaites. Elles génèrent en effet des alertes de granularité trop fine, provoquent beaucoup de faux positifs (alerte générée alors qu'il n'y a pas d'attaque) et de faux négatifs (pas d'alerte générée alors qu'il y a une attaque). Tout cela complique l'analyse et le diagnostic final d'un administrateur de sécurité, en le submergeant d'alertes parmi lesquelles il est difficile, voire impossible, d'extraire les plus pertinentes. Il est donc nécessaire de proposer de nouvelles approches pour améliorer le taux d'attaques détectées, la qualité du diagnostic ainsi que les performances des outils de détection d'intrusions.

Dans ce cadre, Lande participe plus particulièrement à la définition d'un langage de description de haut niveau de scénarios, permettant la description des attaques [33]. En collaboration avec l'équipe Armor, Lande se propose de développer une sonde de détection d'intrusions qui utilisera les résultats précédents. Le projet RNTL mettra en commun toutes les sondes implantées par les partenaires. Il développera des bases de données de corrélation [32] et un environnement d'expérimentation pour la Détection d'intrusions coopérative (cf module 6.2.7).

Cette action bénéficie d'un soutien du ministère de la recherche, sous la forme d'un contrat RNTL. Nos partenaires industriels sont NetSecure Software et France Telecom R&D. Nos partenaires universitaires sont

l'école SUPELEC de Rennes, l'école normale supérieure de Cachan (laboratoire LSV) et la FERIA de Toulouse avec ses composants ONERA et IRIT.

7.4. Proejt européen IST FET/Open Secsafe

Participants : Thomas Jensen, Marc Éluard, Frédéric Besson, Florimond Ployette, Thomas de Grenier de Latour.

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

Le projet européen Secsafe qui a commencé en 2000 porte sur la sécurité de logiciels et l'analyse statique avec une focalisation sur les langage Java et Java Card. Le choix de Java comme langage à étudier permet d'appliquer nos analyses à deux domaines différents : le code mobile et les cartes à puce. Les analyses sont fondées sur nos travaux de formalisation de Java et Java Card (voir module 6.2.4). Le projet a deux partenaires académiques (Imperial College de l'Université de Londres et l'Université technique du Danemark) et la PME Trusted Logic, spécialisée en sécurité et cartes à puce.

7.5. Projet européen IST R&D Verificard

Participants : Thomas Jensen, Thomas Genet, François Monin.

Mots clés : *cartes à puce*, *Java Card*, *transformation et compilation de programmes*, *assistants de preuves*.

Le projet Verificard (IST R&D 2000-26328) est un projet entre l'université de Nijmegen, l'université technique de Munich, l'université de Hagen, Swedish Institute of Computer Science, Schlumberger CP8, Gemplus et l'Inria. Le projet a comme objectif de fournir des méthodes de validation et de vérification d'une plate-forme et d'applications Java Card. Pour ce faire, on propose d'intégrer des techniques de vérification de modèles dans des assistants de preuves comme Coq, PVS ou HOL. Comme la sécurité d'une application est conditionnée par le bon fonctionnement de la plate-forme d'exécution, une partie du projet concerne le développement d'outils certifié pour la manipulation de programmes Java Card. Notre participation porte sur la construction d'un transformateur certifié pour le byte code Java Card (voir 6.1.5).

7.6. Action Castor

Participants : Pascal Fradet, Thomas Lefort.

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

L'objectif du projet Castor est de fournir un environnement permettant de décrire l'architecture d'un système d'informations et d'étudier ses propriétés de sécurité. L'analyse d'une telle architecture doit permettre de modéliser ou synthétiser des scénarios d'attaques, de détecter des failles de sécurité et de guider dans la mise en œuvre de parades. Le projet Castor, financé par le Celar, regroupe les sociétés Sycomore EADS, AQL et TNI et les projets EP-ATR et Lande de l'Irisa. Son objectif est de fournir un environnement permettant de décrire l'organisation globale d'un système d'informations sous forme de vues et d'étudier ses propriétés de sécurité. Le projet Lande s'est concentré sur les problèmes de cohérence liés aux descriptions multi-vues et à l'analyse de sécurité (voir section 6.1.1). Un prototype de vérification et d'analyse a été intégré à l'environnement de modélisation Castor. Il a été utilisé pour vérifier des propriétés de cohérence inter-vues et pour l'analyse de propriétés (synthèse de parades/attaques) sur la vue sécurité [43]. Le projet Castor s'est terminé en septembre 2002.

8. Actions régionales, nationales et internationales

8.1. Actions nationales

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

Le projet Lande participe à l'Action de Recherche Coopérative « Modocop » sur la vérification de programmes à objets concurrents. L'action regroupe les projets Inria Lande, Lemme, Vasy, Vertecs et Oasis et le laboratoire Verimag de Grenoble

8.2. Réseaux et groupes de travail internationaux

Mireille Ducassé est membre de l'ACM (Association for Computing Machinery) et de l'ALP (Association for Logic Programming).

8.3. Relations bilatérales internationales

Thomas Jensen est, avec S. Peyton Jones de Microsoft Research, chercheur consultant sur le projet australien « Constraint-based program analysis » conduit par les universités de Monash et de Melbourne.

9. Diffusion des résultats

9.1. Animation de la communauté scientifique

Mireille Ducassé a édité un numéro spécial du « Journal of Automated Software Engineering » sur le débogage automatisé [14]. Elle a été membre des comités de programme de DX'02 (International Workshop on Principles of Diagnosis), JFPLC'02 (Journées Francophones de Programmation en Logique et avec Contraintes), et de WLPE'02 (International Workshop on Logic Programming Environments). Elle est membre du comité de pilotage du workshop AADEBUG (Automated Debugging) ainsi que du bureau de l'Association Française de Programmation en Logique et avec Contraintes. Elle est présidente de la commission de spécialistes 27e section de l'INSA de Rennes et membre des commissions de spécialistes 27e section des universités de Rennes 1 et de La Réunion. Elle a été présidente du jury de thèse de Sébastien Ferré (direction Olivier Ridoux).

Olivier Ridoux a fait partie du comité de programme de LOPSTR2002 (LOgic-based Program Synthesis and TRansformation). Il a été membre des jurys des thèses de Teresa Higuera (direction Valérie Issarny), Le Vu Hahn (direction Jean-Marc Jézéquel), Frédéric Besson (direction Thomas Jensen), Hélène François (direction Laurent Miclet), et Mickaël Kerbœuf (direction (Paul Le Guernic), et des jurys d'HDR de Christian Rétoré (université de Nantes) et de Pascale Sébillot (Rennes 1). Il est membre des commissions de spécialiste 27e section de l'université de Rennes 1 et de l'Insa, et de la commission de spécialiste 72e section (Epistémologie, histoire des sciences et des techniques) de l'université de Rennes 1.

Thomas Jensen a fait partie des comités de programme de European Symposium on Programming (ESOP'2002) et du Workshop on Model Checking and Abstract interpretation (VMCAI'2002) et de IFIP International conference on Smart Cards (CARDIS'2002). Par ailleurs, il a été rapporteur sur la these de L. Casset (Universite de Marseille) et il a été examinateur sur la thèse d'Eva Rose (Université de Paris 7) et sur la thèse de Anne-Françoise Le Meur (Université de Rennes 1). Thomas Jensen est responsable scientifique de l'école « Jeunes Chercheurs » en Programmation organisée sous l'égide du GDR ALP. Avec M. Huisman de l'Inria Sophia-Antipolis il a organisé le workshop « Verisafe » sur les méthodes formelles pour cartes à puce à Nice, septembre 2002.

9.2. Enseignement universitaire

Pascal Fradet et Thomas Jensen assurent un module de DEA de l'IFSIC, Université de Rennes 1, sur la sémantique et l'analyse de programmes.

Pascal Fradet est responsable de la filière « Génie Logiciel et méthodes formelles » du DEA de l'IFSIC. Il donne un mini-cours sur la sécurité des logiciels en 5ème année INSA et intervient dans un cours d'initiation à la programmation en DEUG.

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

d'informatique. Il co-encadre à l'Université de Bretagne Sud un doctorat sur « une approche logique pour la compréhension des énoncés oraux » [35].

Mireille Ducassé a donné un cours d'une journée sur la méthode formelle B à l'école « Jeunes Chercheurs » en Programmation à Rennes. Elle est intervenue au séminaire IFSIC, Rennes1, sur le thème de la diminution des étudiantes en informatique. Elle a été responsable de l'option industrielle de la dernière année de la formation d'ingénieur en informatique de l'INSA de Rennes jusqu'en juin 2002. Elle enseigne la compilation et la méthode formelle « B » au niveau bac+4, ainsi que la qualité du logiciel au niveau bac+5. Elle encadre un projet annuel de 8 étudiants, niveau bac+4, qui travaillent sur le logiciel Coca (cf module 5.2).

Thomas Genet enseigne la programmation fonctionnelle en DEUG, la méthode formelle « B » en maîtrise d'informatique. Il assure également un module de DEA sur les méthodes déductives pour la vérification (en collaboration avec Vlad Rusu du projet VERTECS).

Arnaud Gotlieb assure une partie d'un module en tronc commun du DEA, sur le test logiciel. Il intervient également en 5ème année INSA dans un cours sur la qualité du logiciel.

Le projet a encadré les étudiants de DEA suivants : Stéphane Hong Tuan-Ha (*assemblage et fusion de composants*), Soazig Bars (*recherche de composants logiciels par leurs types*), Guillaume Feuillade (*Automates d'arbres pour l'approximation régulière des états accessibles dans les systèmes de réécriture conditionnels*), Elisa Fromont (*Slicing de programmes logiques avec contraintes*), et Gurvan Le Guernic (*analyse de programmes Java concurrents*).

Par ailleurs, le projet a reçu, pour des séjours d'une semaine, Joachim Schimpf d'IC-PARC (UK), et Gyongyi Szilagyi chercheuse à l'académie des sciences hongroise, sur l'analyse de programmes logiques avec contraintes dans le cadre du projet RNTL OADYMPPAC (cf module 7.1).

10. Bibliographie

Bibliographie de référence

[1] F. BESSON, T. JENSEN, D. L. MÉTAYER, T. THORN. *Model ckecking security properties of control flow graphs.* in « Journal of Computer Security », volume 9, 2001, pages 217-250.

[2] T. COLCOMBET, P. FRADET. *Enforcing trace properties by program transformation.* in « Proc. of Principles of Programming Languages », ACM Press, pages 54-66, Boston, janvier, 2000.

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

[4] M. DUCASSÉ. *Opium : An extendable trace analyser for Prolog.* in « Elsevier Journal of Logic programming », volume 39, 1999, pages 177-223, http://www.inria.fr/rrrt/rr-3257.html, Special issue on Synthesis, Transformation and Analysis of Logic Programs, A. Bossi and Y. Deville (eds) (également RR-3257 INRIA).

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

[6] S. FERRÉ, O. RIDOUX. *A File System Based on Concept Analysis.* in « DOOD2000, 1st Int. Conf. Computational Logic, LNAI 1861 », éditeurs Y. SAGIV., 2000.

[7] S. FERRÉ, O. RIDOUX. *A logical Generalization of Formal Concept Analysis.* in « 8th Int. Conf. Conceptual Structures, LNAI 1867 », éditeurs B. GANTER, G. MINEAU., 2000.

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

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

[10] T. JENSEN. *Disjunctive Program Analysis for Algebraic Data Types.* in « ACM Transactions on Programming Languages and Systems », numéro 5, volume 19, 1997, pages 752-804.

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

[12] T. JENSEN, D. LE MÉTAYER, T. THORN. *Verification of control flow based security properties.* in « Proc. of the 20th IEEE Symp. on Security and Privacy », New York : IEEE Computer Society, pages 89-103, mai, 1999.

[13] 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

[14] *Journal of Automated Software Engineering 9(1), Special issue on automated debugging.* éditeurs M. DUCASSÉ., Kluwer Academic Publishers, janvier 2002.

Thèses et habilitations à diriger des recherche

[15] F. BESSON. *Analyse modulaire de programmes.* thèse de doctorat, Université de Rennes 1, 2002.

[16] S. FERRÉ. *Systèmes d'information logiques : un paradigme logico-contextuel pour interroger, naviguer et apprendre.* thèse de doctorat, Université de Rennes 1, 2002.

[17] S. ROUVRAIS. *Utilisation d'agents mobiles pour la construction de services distribués.* thèse de doctorat, Université de Rennes 1, juillet, 2002, N. d'ordre : 2614.

Articles et chapitres de livre

[18] E. DENNEY, T. JENSEN. *Correctness of Java Card method lookup via logical relations.* in « Theoretical Computer Science », volume 283, 2002, pages 305-331.

[19] E. JAHIER, M. DUCASSÉ. *Generic Program Monitoring by Trace Analysis.* in « Theory and Practice of Logic Programming », numéro 4-5, volume 2, September, 2002.

[20] T. JENSEN. *Types in program analysis.* éditeurs T. MOGENSEN, D. SCHMIDT, I. H. SUDBOROUGH., in « The Essence of Computation : Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones », série Lecture Notes in Computer Science 2566, Springer-Verlag, 2002, pages 204-222.

[21] S. ROUVRAIS. *Construction de services distribués : une approche à base d'agents mobiles.* in « TSI-HERMES / RSTI - TSI. Volume 21 - n° 7 », 2002, pages 985-1007.

Communications à des congrès, colloques, etc.

[22] S. BARS, S. FERRÉ, O. RIDOUX. *Logic Functors for Types as Search Keys.* in « 1st Int. Workshop on Isomorphisms of Types », 2002.

[23] F. BESSON, T. DE GRENIER DE LATOUR, T. JENSEN. *Secure calling contexts for stack inspection.* in « Proc. of 4th Int Conf. on Principles and Practice of Declarative Programming (PPDP 2002) », ACM Press, pages 76-87, 2002.

[24] P. DERANSART, L. LANGEVINE, M. DUCASSÉ. *A Generic Trace Model for Finite Domain Solvers.* in « Proc. of the International Workshop on User-Interaction in Constraint Satisfaction », Cornell University, éditeurs B. O'SULLIVAN., September, 2002.

[25] R. DOUENCE, P. FRADET, M. SÜDHOLT. *A framework for the detection and resolution of aspect interactions.* in « Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering », Springer-Verlag, Lecture Notes in Computer Science 2487, 2002.

[26] M. DUCASSÉ, L. LANGEVINE. *Analyse automatisée de traces d'exécution de programmes CLP(FD).* in « Actes des Journées Francophones de Programmation en Logique avec Contraintes », HERMES science publications, éditeurs M. RUEHER., pages 119-134, Mai, 2002.

[27] M. DUCASSÉ, L. LANGEVINE. *Automated analysis of CLP(FD) program execution traces.* in « Proceedings of the International Conference on Logic Programming », Lecture Notes in Computer Science 2401, Springer-Verlag, éditeurs P. STUCKEY., July, 2002, http://www.irisa.fr/lande/ducasse/.

[28] S. FERRÉ, O. RIDOUX. *A Framework for Developing Embeddable Customized Logics.* in « LOPSTR'01 : Logic-based Program Synthesis and Transformation, LNCS 2372 », 2002.

[29] S. FERRÉ, O. RIDOUX. *The Use of Associative Concepts in the Incremental Building of a Logical Context.* in « 10th Int. Conf. Conceptual Structures, LNAI 2393 », éditeurs U. PRISS, D. CORBETT, G. ANGELOVA., 2002.

[30] T. JENSEN, F. PLOYETTE, O. RIDOUX. *Iteration schemes for fixed point conputation.* in « Proc. of 4th Int workshop on Fixed Points in Computer Science (FICS'02) », éditeurs A. INGOLFSDOTTIR, Z. ESIK., pages 69-76, 2002.

[31] L. LANGEVINE, P. DERANSART, M. DUCASSÉ, E. JAHIER. *Prototypage de traceurs CLP(FD).* in « Actes des Journées Francophones de Programmation en Logique avec Contraintes », HERMES science publications, éditeurs M. RUEHER., pages 135-150, Mai, 2002.

[32] B. MORIN, L. MÉ, H. DEBAR, M. DUCASSÉ. *M2D2 : A Formal Data Model for IDS Alert Correlation.* in « Recent Advances in Intrusion Detection », Springer-Verlag, Lecture Notes in Computer Science 2516, éditeurs A. WESPI, ET AL.., pages 97-104, October, 2002.

[33] J.-P. POUZOL, M. DUCASSÉ. *Formal specification of intrusion signatures and detection rules.* in « Proc. of 15th IEEE Computer Security Foudations Workshop », IEEE Press, éditeurs S. SCHNEIDER., pages 64-76, 2002.

[34] L. VAN AERTRYCK, T. JENSEN. *UML-CASTING : Test synthesis from UML models using constraint resolution.* in « Proc. Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'2003) », INRIA, éditeurs J.-M. JÉZÉQUEL., à paraître.

[35] J. VILLANEAU, J.-Y. ANTOINE, O. RIDOUX. *LOGUS : un système formel de compréhension du français parlé spontané - présentation et évaluation.* in « TALN'02 : Traitement Automatique de la Langue Naturelle », 2002.

[36] M. ÉLUARD, T. JENSEN. *Secure object flow analysis for Java Card.* in « Proc. of 5th Smart Card Research and Advanced Application Conference (Cardis'02) », IFIP/USENIX, 2002.

Rapports de recherche et publications internes

[37] R. DOUENCE, P. FRADET, M. SÜDHOLT. *Detection and resolution of aspect interactions.* Rapport de recherche, numéro 4435, Inria, Avril, 2002, http://www.inria.fr/rrrt/rr-4435.html.

[38] S. FERRÉ. *Incremental Concept Formation Made More Efficient by the Use of Associative Concepts.* RR-4569, Inria, 2002, http://www.inria.fr/rrrt/rr-4569.html.

[39] S. FERRÉ, O. RIDOUX. *Introduction to Logical Information Systems.* RR-4540, Inria, 2002, http://www.inria.fr/rrrt/rr-4540.html.

[40] S. FERRÉ, O. RIDOUX. *Logic Functors : a Framework for Developing Embeddable Customized Logics.* RR-4457, Inria, 2002, http://www.inria.fr/rrrt/rr-4457.html.

[41] T. GENET, V. VIET TRIEM TONG. *Proving Negative Conjectures on Equational Theories using Induction and Abstract Interpretation.* rapport technique, numéro RR-4576, INRIA, 2002, http://www.inria.fr/rrrt/rr-4576.html.

[42] Y. PADIOLEAU, O. RIDOUX. *A Logic File System.* RR-4656, Inria, 2002, http://www.inria.fr/rrrt/rr-4656.html.

Divers

[43] *Castor : Dossier de définition & dossier de justification (Tome 3).* Rapport d'études, AQL, Irisa, Sycomore, TNI, septembre, 2002.

Bibliographie générale

[44] *ANSI/IEEE Standard 729-1983.* Glossary of Software Engineering Terminology.

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

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

[47] F. BESSON, T. JENSEN, D. L. MÉTAYER, T. THORN. *Model ckecking security prperties of control flow graphs.* in « Journal of Computer Security », volume 9, 2001, pages 217-250.

[48] P. BRISSET, O. RIDOUX. *Continuations in λProlog.* in « 10th Int. Conf. Logic Programming », MIT Press, éditeurs D. WARREN., pages 27-43, 1993.

[49] B. BRUEGGE, T. GOTTSCHALK, B. LUO. *A framework for dynamic program analyzers.* in « Proc. of the OOPSLA'93 Conference », pages 65-82, 1993.

[50] P. COUSOT. *Abstract Interpretation Based Static Analysis Parameterized by Semantics.* in « Proc. of 4th Static Analysis Symposium », Springer Verlag, LNCS vol. 1302, éditeurs P. VAN HENTENRYCK., pages 388-394, 1997.

[51] M. DUCASSÉ. *Coca : An automated Debugger for C.* in « Proceedings of the 21st International Conference on Software Engineering », ACM Press, pages 504-513, mai, 1999, http://www.inria.fr/rrrt/rr-3489.html, (également RR-3489 INRIA).

[52] S. FERRÉ, O. RIDOUX. *A File System Based on Concept Analysis.* in « DOOD2000, 1st Int. Conf. Computational Logic, LNAI 1861 », éditeurs Y. SAGIV., 2000.

[53] S. FERRÉ, O. RIDOUX. *A logical Generalization of Formal Concept Analysis.* in « 8th Int. Conf. Conceptual Structures, LNAI 1867 », éditeurs B. GANTER, G. MINEAU., 2000.

[54] B. GANTER, R. WILLE. *Formal Concept Analysis : Mathematical Foundations.* Springer, 1999.

[55] T. GENET, V. VIET TRIEM TONG. *Reachability Analysis of Term Rewriting Systems with Timbuk.* in « Proceedings of the 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning », série Lecture Notes in Artificial Intelligence, volume 2250, Springer-Verlag, pages 691-702, 2001.

[56] E. JAHIER, M. DUCASSÉ. *An automated debugger for Mercury - Morphine 0.1 User and reference manuals.* mai, 1999, http://www.inria.fr/rrrt/rt-0231.html, RT-0231 INRIA (également PI-1234 IRISA).

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

[58] T. JENSEN, D. LE MÉTAYER, T. THORN. *Verification of control flow based security properties.* in « Proc. of the 20th IEEE Symp. on Security and Privacy », New York : IEEE Computer Society, pages 89-103, mai, 1999.

[59] G. KAHN. *Natural semantics.* in « Proceedings of STACS'87 », série LNCS 247, Springer Verlag, pages 22-39, 1987.

[60] F. NIELSON, H. NIELSON, C. HANKIN. *Principles of Program Analysis.* Springer, 1999.

[61] F. OEHL, D. SINCLAIR. *Combining two approaches for the formal verification of cryptographic protocols.* in « Proceedings of ICLP Workshop on Specification, Analysis and Validation for Emerging technologies in computational logic », 2001.

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

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

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

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

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

[67] Z. SOMOGYI, F. HENDERSON, T. CONWAY. *The execution algorithm of Mercury, an efficient purely declarative logic programming language.* in « Journal of logic Programming », volume 29, October-December, 1996, pages 17-64, http://www.cs.mu.oz.au/research/mercury/information/papers.html.

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

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

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