INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Project-Team Lande

Logiciel : ANalyse et DEveloppement

Rennes

THEME 2A

Activity Report

2003

Table of contents

1. Team

Head of project-team
Thomas Jensen [Research scientist, DR CNRS]

Administrative assistant
Lydie Letort [Administrative assistant, TR Inria]

INRIA personnel
Pascal Fradet [Research scientist, CR, until September 2003]
Arnaud Gotlieb [Research scientist, CR]
Florimond Ployette [Technical staff, IR]

Université Rennes 1 Personnel
Olivier Ridoux [Professor]
Thomas Genet [Assistant Professor]

Insa Personnel
Mireille Ducassé [Professor]

ENS Cachan Personnel
David Cachera [Assistant Professor]

PhD students
Yoann Padioleau [MENRT grant]
Jean-Philippe Pouzol [Inria grant]
Valérie Viet Triem Tong [MENRT grant]
Thomas de Grenier de Latour [MENRT grant]
Stéphane Hong Tuan-Ha [Inria-Region Bretagne grant]
David Pichardie [ENS Cachan]
Gurvan Le Guernic [MENRT grant]
Benjamin Sigonneau [MENRT grant, from October 2003]

Technical staff
Frédéric Besson [Inria project technical staff, until October 2002]

2. Overall Objectives

2.1. Project overview

The Lande project is concerned with formal methods for constructing and validating software. Our focus is on providing methods with a solid formal basis (in the form of a precise semantics for the programming language used and a formal logic for specifying properties of programs) in order to provide firm guarantees as to their correctness. In addition, it is important that the methods provided are highly automated so as to be usable by non-experts in formal methods.

2.1.1. Research activities:

The project conducts activities aimed at analysing the behaviour of a given program. These activities draw on techniques from static and dynamic program analysis, testing and automated theorem proving. In terms of **static program analysis**, our foundational studies concern the specification of analyses by inference systems, the classification of analyses with respect to precision using abstract interpretation and reachability analysis for software specified as a term rewriting system. Particular analyses such as pointer analysis for C and control flow analysis for Java and Java Card have been developed. For the implementation of these and other analyses, we are improving and analysing existing iterative techniques based on constraint-solving and rewriting of tree automata.

In order to facilitate the **dynamic analysis of programs** we have developed a tool for manipulating execution traces. The distinctive feature of this tool is that it allows the user to examine the trace by writing queries in a logic programming language. These queries can be answered on-line which enables the analysis of traces of large size. The tool can be used for program debugging and we are now investigating its use in intrusion detection.

Concerning the **testing** of software, we have in particular investigated how flow analysis of programs based on constraint solving can help in the process of generating test cases from programs and from specifications. More recent work include the study of testing a program based on symmetries that its semantics is supposed to exhibit.

To address the problem of analysing large software systems, we are studying how the various verification techniques adapt to the different ways of writing *modular software*, for example how to perform static analysis of program fragments. Another, more recent program-structuring technique is **aspect-oriented programming** where new properties are added incrementally to a program by *weaving* them into the code. This allows in particular to enforce properties upon a program in a cost-effective manner, by mixing static analysis with run-time supervision.

An additional issue in the verification of large software systems is the exploitation of the results of an analysis which associates a variety of properties with each program entity (variable, function, class,...). To facilitate the design of tools for navigating and extracting particular views of a code based on the information found by the analysis, we have defined the abstract notion of **logical information systems**, in which activities such as navigation, querying and data analysis can be studied at the same, logic-based level. This framework presents the advantage of being generic in the logic used for navigation and querying; in particular it can be instantiated with different types of program logics such as types or other static properties.

An important application domain for these techniques is that of **software security**. Our activity in the area of **programming language security** has lead to the definition of a framework for defining and verifying security properties based on a combination of static program analysis and model checking. This framework has been applied to software for the Java 2 security architecture, to multi-application Java Card smart cards, and to cryptographic protocols. Similarly, the trace-based program analysis techniques has been shown to be useful in intrusion detection where it provides a language for describing and determining attacks.

Lande is a joint project with the CNRS, the University of Rennes 1 and Insa Rennes.

3. Scientific Foundations

3.1. Static program analysis

Key words: *static program analysis*, *semantics*, *abstract interpretation*, *optimising compilers*.

Glossary

Abstract interpretation Abstract interpretation is a framework for relating different semantic interpretations of a program. Its most prominent use is in the correctness proofs of static program analyses, when these are defined as a non-standard semantic interpretation of a language in a domain of abstract program properties

Fixpoint iteration The result of a static analysis is often given implicitly as the solution of a system of equations $\{\overline{x} = f_i(\overline{x})\}_{i=1}^{n}$ where the f_i are monotone functions over a partial order. The Knaster-Tarski Fixpoint Theorem suggests an iterative algorithm for computing a solution as the limit of the ascending chain $f^n(\perp)$ where $\perp$ is the least element of the partial order.

3.1.1. Static program analysis

[45][58]
covers a variety of methods for obtaining information about the run-time behaviour of a program without actually running it. It is this latter restriction that distinguishes static analysis from its dynamic counterparts (such as debugging or profiling) which are concerned with monitoring the execution of the program. It is common to impose a further requirement *viz.*, that an analysis is decidable, in order to use it in program-processing tools such as compilers without jeopardizing their termination behaviour.

Static analysis has so far found most of its applications in the area of program optimisation where information about the run-time behaviour can be used to transform a program so that it performs a calculation faster and/or makes better use of the available memory resources. Examples of static analysis include:

- Data-flow analysis as it is used in optimising compilers for imperative languages. The properties can either be approximations of the values of an expression ("the value of variable x is greater than 0") or invariants of the computation trace done by a program. This is for example the case in "reaching definitions" analysis that aims at determining what definitions (in the shape of assignment statements) are always valid at a given program point.
- Alias analysis is another data flow analysis that finds out which variables in a program addresses the same memory location. This information is significant *e.g.*, when trying to recover unused memory statically ("compile-time garbage collection").
- Strictness analysis for lazy functional languages is aimed at detecting when the lazy call-by-need parameter passing strategy can be replaced with the more efficient call-by value strategy. This transformation is safe if the function is strict, that is, if calling the function with a diverging argument always leads to a diverging computation. This is *e.g.*, the case when the function is guaranteed to use this parameter; strictness analysis serve to discover when this is the case.
- Dependency analysis determines those parts of a program whose execution can influence the value of a particular expression in a program. This information is used in *program slicing*, a debugging technique that permits to extract the part of a program that can be at the origin of an error, and to ignore the rest. Dependency information can also be used for determining when two instructions are independent, and hence can be executed in any order, or in parallel. Finally, dependency analysis plays an important role in software security where it forms the core of most information flow analyses.
- Control flow analysis will find a safe approximation to the order in which the instructions of a program are executed. This is particularly relevant in languages where parameters or functions can be passed as arguments to other functions, making it impossible to determine the flow of control from the program syntax alone. The same phenomenon occurs in object-oriented languages where it is the class of an object (rather than the static type of the variable containing the object) that determines which method a given method invocation will call. Control flow analysis is an example of an analysis whose information in itself does not lead to dramatic optimisations (although it might enable in-lining of code) but is necessary for subsequent analyses to give precise results.

3.1.2. The structure of a static analysis

A static analysis can often be viewed as being split into two phases. The first phase performs an *abstract interpretation* of the program producing a system of equations or constraints whose solution represents the information of the program found by the analysis. The second phase consists in finding such a solution. The first phase involves a formal definition of the abstract domain *i.e.*, the set of properties that the analysis can detect, a technique for extracting a set of equations describing the solution from a program, and a proof of semantic correctness showing that a solution to the system is indeed valid information about the program's behaviour. The second phase can apply a variety of techniques from symbolic calculations and iterative algorithms to find the solution. An important point to observe is that the resolution phase is decoupled from the analysis and that the same resolution technique can be combined with different abstract interpretations.

3.2. Debugging

Key words: *programming environments*, *dynamic program analysis*, *trace schema*, *trace generation*, *trace analysis*, *fault localization*.

Glossary

Error An error is a human action which results in an incorrect program. For example, an error can be to invert two variables A and B.

Fault A fault is an erroneous stage, process or definition of data in a program. An error can generate one or more faults. For example, a fault induced by the above mentioned error can be that the test to stop a loop is done on A, which is never updated.

Failure A failure is the incapacity of a program to carry out its necessary functionalities. A fault can generate one or more failures [43]. An example of failure resulting from the above mentioned fault is that the program execution never finishes.

Debugging consists in locating and correcting the faults which are responsible for software failures. Debugging is a complex cognitive activity which requires, in general, to go up to the human error to understand the reasons of the faults which generated the observed failures. There are tools, commonly called *debuggers*, which help programmers identify the unexpected behaviors of programs. These tools give an image (called *trace*) of the execution of programs. There are three principal tasks in order to make a real debugger. The first task consists in specifying which information must appear in the trace. The second task is the implementation of a tracer. The third task consists in automating the analysis of execution traces in order to give relevant information to the programmers, who can thus concentrate on the cognitive process.

Debugging consists in locating and correcting the faults which are responsible for software failures. A failure can be detected after an execution, for example during test phases (cf module 3.3). Debugging is a complex cognitive activity which requires, in general, to go up to the human error to understand the reasons of the faults which generated the observed failures. A failure is a symptom of fault which appears in an erroneous behavior of the program. Very often, the programmer does not control all the facets of the behavior of a program. For example, operational semantics details of the language can escape to him, the program can be too complex, or the used tools can have obscure documentations. There are tools, commonly called *debuggers*, which help programmers to identify the unexpected behaviors of programs. These tools, which should be rather called *tracers*, give an image (called *trace*) of program executions. A trace consists of remarkable computation events.

There are three principal tasks in order to make a real debugger:

1. The first task consists in specifying which information must appear in the trace, namely the trace schema. The trace is an abstraction of the operational semantics of the language. Its objective is to help users understand the behavior of programs. It thus depends on the language and the type of potential users.
2. The second task is the implementation of a tracer. It requires the insertion of trace instructions in the mechanisms of program executions (this insertion is called *instrumentation* in the following). Instrumentation can be done at different levels: in the source code, in the compiler or in the emulator when there is one. The current practice consists in instrumenting at the lowest level [60] but the more the instrumentation is made on a high level, the more it is portable.
3. The third task consists in automating the analysis of execution traces in order to give relevant information to the programmers, who can thus concentrate on the cognitive process. At present, tracers too often provide very poor analysis capabilities. Users have to face too many irrelevant details.

3.3. Program testing

Key words: *Test data*, *Testing criterion*, *Testing hypothesis*, *oracle*, *unit testing*, *integration testing*, *structural and functional testing*.

Glossary

Test data A test datum is a complete valuation of all the input variables of a program.

Test set A test set is a non-empty finite set of test data.

Testing criterion A testing criterion defines finite subsets of the input domain of a program. Testing criteria can be viewed as testing objectives.

Successful test set A test set is successful when the execution of the program with all the test data of a test set has given expected results

A reliable criterion A testing criterion is reliable iff it only selects either successful test set or unsuccessful test set.

A valid criterion A testing criterion is valid iff it produces unsuccessful test set as soon as there exists at least one test datum on which the program gives an incorrect result.

Ideal test set A test set is ideal iff it is unsuccessful or if it is successful then the program is correct over its input domain.

Fundamental theorem of structural testing A reliable and valid criterion selects only ideal test sets

Program Testing involves several distinct tasks such as test data selection, program executions, outcome checking, non-regression test data selection, etc. Most of them are based on empirical choices and as a consequence the current tools lack help for the testers. One of the main goal of the Research works undertook by the Lande Project into this field consists in finding ways to automate the testing process. Current works focus on automatic test data generation and automatic oracle checking. Difficulties include testing criteria formalization, automatic source code analysis, low-level specification modeling and automatic constraint solving. The benefits that are expected include a better understanding of these techniques and the design of automated testing tools.

Program Testing requires to select test data from the input domain, to execute the program with the selected test data and finally to check the correctness of the computed outcome. One of the main challenge of this field consists in finding techniques that allow the testers to automate this process. They face two problems that are referenced as the *test data selection problem* and *the oracle problem*.

Test data selection is usually done with respect to a given structural or functional testing criterion. Structural testing (or white-box testing) relies on program analysis to find automatically a test set that guarantees the coverage of some testing objectives whereas functional testing (or black-box testing) is based on software specifications to generate the test data. These techniques both require a formal description to be given as input : the source code of programs in the case of structural testing ; the formal specification of programs in the case of functional testing. For example, the structural criterion *all_statements* requires that every statement of the program would be executed at least once during the testing process. Unfortunately, generating automatically a test set that entirely cover a given criterion is in general a formally undecidable task. Hence, it becomes necessary to compromise between completeness and automatisation in order to set up practical automatic testing methods. This problem is called the *test data selection problem*.

Outcome checking is usually done with the help of a procedure called an oracle that computes a verdict of the testing process. The verdict may be either pass or fail. The former case corresponds to a situation where the computed outcome is equal to the expected one whereas the latter case demonstrates the existence of a fault within the program. Most of the techniques that tend to automate the generation of input test data consider that a complete and correct oracle is available. Unfortunately, this situation is far from reality and it is well known that most programs don't have such an oracle. Testing these programs is then an actual challenge. This is called *the oracle problem*.

Partial solutions to these problems have to be found in order to set up practical testing procedures. Structural testing and functional testing techniques are based on a fundamental hypothesis, known as the *uniformity hypothesis*. It says that selecting a single element from a proper subdomain of the input domain suffices to explore all the elements of this subdomain. These techniques have also in common to focus on the generation of input values and propositions have been made to automate this generation. They fall into two main categories :

- Deterministic methods aim at selecting a priori the test data in accordance with the given criterion. These methods can be either symbolic or execution-based. These methods include symbolic evaluation, constraint-based test data generation, etc.
- Probabilistic methods aim at generating a test set according to a probability distribution on the input domain. They are based on actual executions of the program. They include random and statistical testing, dynamic-method of test data generation, etc.

In the Lande project-team, we are studying these techniques and we are trying to improve them. The main goal consists in designing automated tools able to test complex imperative and sequential programs. An interesting technical synergy comes from the combination of several techniques including program analysis and constraint solving to handle difficult problems of Program Testing.

3.4. Logic information system

Key words: *Logic*, *Information system*, *Formal concept analysis*, *Logic concept analysis*, *Data-mining*, *Information retrieval*.

Glossary

Extension The extension of a formula in a domain, is the subset of the domain elements that satisfy the formula. Some authors say extent.

Intention The intention of a set of elements of a domain is the most specific formula that they all satisfy. Some authors say intent.

Formal context A set of object and their intensions. In formal concept analysis, intensions are usually sets of attributes, while in logic concept analysis they are formula of some logic.

Formal concept Given a formal context, and extensions and intentions taken from the formal context, a formal concept is a pair of an extension and an intention that are mutually complete; the intention of the extension is the extension of the intention.

Main result Given a formal context, the set of all formal concepts form a complete lattice.

Formal concept analysis A form of data-analysis that aims at exhibiting formal concepts hidden in data.

Logic concept analysis A form of formal concept analysis that is generic with respect to the logic used in intentions.

Logic information system An information system in which all operations (i.e., navigation, querying, data-analysis, updating, and automated learning) are based on logic concept analysis.

The framework of Logic information systems offers means for processing and managing data in a generic and flexible way. It is an alternative to hierarchical systems (à la file systems), boolean systems (à la web browsers), and relational systems (à la data-bases). It is based on logic concept analysis, a variant of formal concept analysis.

Formal concept analysis [53] is the formal counter-part of the philosophical ideas of intention and extension (e.g., Leibniz, Pascal and the Logic of Port-Royal). It has received attention to model various situations of data-analysis in mathematics, social sciences, and also in computer science. These applications are stereotyped in two ways: they use attributes as intention, and they aim at building the lattice of formal concept. It is the concept lattice that support the analysis of data. The drawbacks are that expressivity is low, and the required computer power is high.

These two stereotypes can be circumvented as follows. First, Logic concept analysis allows intentions that can be any kind of logic formula provided the underlying logic is monotonous [50]. This yields high expressivity for handling domain knowledge and rules. Second, Logic information systems (LIS) perform data-analysis based on Logic concept analysis without ever building a concept lattice [49][48][47].

Logic information systems offer a range of operations that are all based on concept analysis.

querying — Querying amounts to computing extensions of intentions.

navigation — Navigating amounts to computing differences between a intention (the query), and the intentions of all objects that satisfy the query. This is not trivial as soon as intentions are not sets of attributes. If differences are carefully computed, they form the basis of a progressive exploration tool. Note that combining navigation and querying in a single is in itself a contribution [51].

data-mining — Some operations of data-mining like computing association rules amount to a variant of navigation, where extensions of intentions are used.

automated learning — In a LIS, objects have two kinds of intention. Intrinsic intentions are computed from the object itself (e.g., from its content), and extrinsic intentions are given by a user according to causes that are not in the object (or that cannot be computed). For instance, an intrinsic intention of a music file can be the composer, and an extrinsic intention can be a judgment on the music. When introducing a new object in a LIS, intrinsic intentions can be computed automatically, but extrinsic intensions cannot. However, it is possible to learn from already existing objects relations between intrinsic and extrinsic intentions [52].

concept lattice construction — Though not needed by a LIS, constructing the concept lattice can be useful, at least for an illustration. A variant of the automated learning algorithm can be used for the incremental construction of concept lattices.

It has been shown that provided the size of intentions does not depend of the number of objects, these operations can be implemented in a time quasi-linear with the number of objects. This makes it possible to envisage efficient implementations of a LIS. Two styles of implementation are possible. First, to design a system under its own specific interface that handles properly all its facets. Second, to design a system under an existing interface taking the chance that some facets do not fit the interface. The advantage of the latter style is to offer the LIS service to every application of the interface. One of the most frequently used interface is the file system. So, to design a file system implementation of LIS is an important challenge.

Prototype applications of LIS have been done in software engineering, publishing, genetics, classification of personal document...

For further information, consult http://www.irisa.fr/LIS.

3.5. Reachability analysis over term rewriting systems

Participants: Thomas Genet, Valérie Viet Triem Tong.

Key words: *Term rewriting systems*, *reachability analysis*, *tree automata*.

Term rewriting rystems are a very general, simple and convenient formal model for a large variety of computing systems. For instance, it is a very simple way to describe deduction systems, functions, parallel processes or state transition systems where rewriting models respectively deduction, evaluation, progression or transitions. Furthermore rewriting can model every combination of them (for instance two parallel processes running functional programs).

In rewriting, the problem of reachability is well-known: given a term rewriting system $\mathcal{R}$ and two ground terms s and t, t is $\mathcal{R}$-reachable from s if s can be finitely rewritten into t by $\mathcal{R}$, which is formally denoted by

$s \rightarrow^{\star}_{\mathcal{R}} t$. On the opposite, t is $\mathcal{R}$-unreachable from s if s cannot be finitely rewritten into t by $\mathcal{R}$, denoted by $s \neg \rightarrow^{\star}_{\mathcal{R}} t$.

Depending on the computing system you model using rewriting, a deduction system, a function, some parallel processes or state transition systems, reachability (and unreachability) permit to achieve some verifications on your system: respectively prove that a deduction is feasible, prove that a function call evaluates in a particular value, show that a processes configuration may occur, or that a state is reachable from the initial state. As a consequence, reachability analysis has several applications in equational proofs used in the theorem provers or in the proof assistants as well as in verification where term rewriting systems can be used to model programs.

We are interested in proving (as automatically as possible) reachability or unreachability on term rewriting systems for verification and automated deduction purposes. The reachability problem is known to be decidable for terminating term rewriting systems. However, in automated deduction and in verification, systems considered in practice are rarely terminating and, even when they are, automatically proving their termination is difficult. On the other hand, reachability is known to be decidable on several syntactic classes of term rewriting systems (not necessarily terminating nor confluent). On those classes, the technique used to prove reachability is rather different and is based on the computation of the set $\mathcal{R}^{\star}(E)$ of $\mathcal{R}$-reachable terms of an initial set of terms E. For those classes, $\mathcal{R}^{\star}(E)$ is a regular tree language and can thus be represented using a *tree automaton*. Tree automata offer a finite way to represent infinite (regular) sets of reachable terms when a non terminating term rewriting system is under concern.

For the negative case, i.e. proving that $s \neg \rightarrow^{\star}_{\mathcal{R}} t$, we already have some results based on the over-approximation of the set of reachable terms [54][56]. Now, we focus on a more general approach dealing with the positive and negative case at the same time. We propose a common, simple and efficient algorithm for computing exactly known decidable regular classes for $\mathcal{R}^{\star}(E)$ as well as to construct some approximation when it is not regular. This algorithm is essentially a *completion* of a *tree automata*, thus taking advantage of an algorithm similar to the Knuth-Bendix [57] *completion* in order not to restrict to a specific syntactic class of term rewriting systems and *tree automata* in order to deal efficiently with infinite sets of reachable terms produced by non-terminating term rewriting systems [38].

5. Software

5.1. Recursive Equations Solver

Participants: Thomas Jensen, Florimond Ployette [contact point], Olivier Ridoux.

Key words: *fixed point*, *solver*, *recursive equations*.

Whereas the foundational aspects (correctness and precision) of semantic program analysis have been highly investigated, less attention has been paid to the construction of generic analysis tools. Data flow analysis usually involves a phase of fixed point computation whose algorithmics is largely independent of the particular property analysed for. Our aim is to obtain a generic fixed point solver that can serve as “back ends” in program analysers.

REQS is a tool for solving recursive equations systems in the framework of program static analysis (imperative, logic or functional). The equations are expressed in a simple language and the user has to define and implement the domain operations used in the equations. However predefined domains are available for standard domains. A crucial point in solving these systems is the strategy used by the solver to minimize the number of variable evaluations. This strategy is influenced by the dependency relation that describe how the value of one variable depends on the value of another. REQS proposes different strategies from naive to more sophisticated ones based on the dependency graph of the variables.

REQS has been used as a back end for various experiments in program analysis (object language analysis, synchronous language analysis, namely the Signal language). A Java class analysis (including exceptions) is available and generates a control flow graph which serves as a basis for other specific analysis. REQS has been used in the context of two JavaCard analysis:

- A JavaCard applets analysis, implemented in SICS(Sweden), uses our Java class analysis as a front end: (http://www.sics.se/fdt/projects/vericode/jcave.html).
- A Carmel(a JavaCard subset) analysis in the framework of the Secsafe project.

REQS is written in Java and can be freely downloaded.

For further information concerning REQS, contact Florimond Ployette (ployette@irisa.fr), or consult (http://www.irisa.fr/lande/reqs).

5.2. Logic File System

Participants: Yoann Padioleau, Olivier Ridoux [contact point].

Key words: *Logic information system*, *file system*.

The Logic file system (LISFS) implements a logic information system at the file system level (see other sections).

LISFS is a freely down-loadable Linux 2.4 kernel module. It has been used for various experiments in the retrieval of software components, documentation, music files, etc.

For further information concerning LISFS, contact Olivier Ridoux (ridoux@irisa.fr), or consult (http://www.irisa.fr/LIS).

5.3. Timbuk: a Tree Automata Library

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

Key words: *Tree automata*, *approximations*, *term rewriting systems*.

Timbuk [56] is a library of OCAML functions for manipulating tree automata. More precisely Timbuk deals with finite bottom-up tree automata (deterministic or not). This library provides the classical operations over tree automata:

- boolean operations: intersection, union, inversion,
- emptiness decision, inclusion decision,
- cleaning, renaming,
- determinisation,
- transition normalisation,
- building the tree automaton recognizing the set of irreducible terms for a left-linear TRS.

This library also implements some more specific algorithm that we use for verification (of cryptographic protocols in particular):

- tree automata completion w.r.t. a given term rewriting system,
- approximation of reachable terms and normal forms for a given term rewriting system,
- matching in tree automata.

This software is distributed under the Gnu Library General Public License and is freely available at http://www.irisa.fr/lande/genet/timbuk/.

A version 2.0 of Timbuk will soon be available. This new version contains several optimisations and utilities. The completion algorithm complexity has been optimised for better performance in space and time. Timbuk now provides two ways to achieve completion: a dynamic version which permits to compute approximation step by step and a static version which pre-compiles matching and approximation in order to enhance speed of completion. Timbuk 2.0 also provide a graphical interface called Tabi for browsing tree automata and figure out more easily what are the recognized language. Timbuk 2.0 was used for a case study done with Thomson-Multimedia for cryptographic protocol verification (see 6.10).

On the other hand, Timbuk is used by Frédéric Oehl and David Sinclair of Dublin University for verifying cryptographic protocols with an approach combining a proof assistant (Isabelle/HOL) and approximations (done with Timbuk) [59].

6. New Results

6.1. Modular static program analysis

Participants: Frédéric Besson, Thomas Jensen.

Key words: *control-flow analysis*, *abstract interpretation*, *constraints*, *non-standard type systems*.

Glossary

modular analysis Technique in which fragments of a program can be analysed separately and then pieced together to yield information about the entire program. As opposed to global analysis.

We have proposed techniques for modularising program analysis and verification techniques that so far were carried out as global analyses. The technique is a novel approach to control flow analysis of object-oriented programs using DATALOG clauses to represent systems of constraints [21]. The standard iterative fixpoint computation of a least solution to a constraint system is here complemented by symbolic contraint transformation techniques for normalising a set of DATALOG clauses. This work was partly funded by the European IST FET/Open project “Secsafe” (see 7.1).

Another approach to modular control flow analysis has been developed with A. Banerjee from Kansas State University [14]. This approach is based on enriching an intersection type system so that the type of an expression includes information about the origin of the values consumed and produced by the expression (here, labels of lambda expressions in a lambda calculus term). One distinctive feature of this type system is its use of and restriction to rank 2 intersection types which renders the problem of type inference tractable. We show how a standard inference algorithm for rank 2 intersection types can be extended to infere control flow types, and prove correctness of this algorithm.

The foundations of control-flow analysis for object oriented languages have been investigated in a joint work with F. Spoto from the University of Verona [19] in which a number of *class analyses* have been derived systematically from a trace semantics using the notions of abstract interpretation. This derivation allows to compare the relative precision of the analyses. Furthermore, it is shown that efficient implementations of the static analyses can be derived from this formalisation

6.2. Java Card - certified compilation and security analysis

Participants: Thomas Genet, Thomas Jensen, David Pichardie.

The Java Card language is a trimmed down dialect of Java aimed at programming smart cards. Java Card specifies its own class file format (the Java Card Converted APplet (CAP) format) that is optimised with respect to the limited space resources of smart cards. The purpose of the work (reported in [27]) was to show how to do the certified development of algorithms necessary for the conversion of ordinary Java class files into the CAP format. More precisely, these algorithms are concerned with constructing and compressing method tables and constant pools. The main challenge has been to specify and prove the correctness of these algorithms using the theorem prover PVS.

We have also developed a static analysis for verifying the proper protection of resources on a multi-application Java Card [18]. The access control to resources exercised by the Java Card firewall can be bypassed by the use of shareable objects. To help detecting unwanted access to objects, we propose a static analysis that calculates a safe approximation of the possible flow of objects between Java Card applets. The analysis deals with a subset of the Java Card bytecode focussing on aspects of the Java Card firewall, method invocation, field access, variable access, shareable objects and contexts. The technical vehicle for achieving this task is a new kind of constraints: quantified conditional constraints, that permits us to model precisely yet cost-effectively the effects of the Java Card firewall by only producing a constraint if the corresponding operation is authorized by the firewall.

6.3. Intrusion detection

Participants: Mireille Ducassé, Jean-Philippe Pouzol.

Key words: *security*, *audit trail analysis*, *intrusion*, *attack scenario*, *correlation*.

Lande has recently started an activity around audit trail analysis within the framework of the detection of intrusions. A traditional strategy to secure an information processing system consists in building a protective shield around it. The users wishing to reach the data or resources of the system must cross safety fences, for example, identification or cryptography mechanisms. However, the development of open systems, as well as the proliferation of heterogeneous machines connected to networks, make complex and not very practicable the implementation of completely reliable mechanisms of security. A possible defense against the abusive uses of a system is the *detection of intrusions*. Intrusion detection is an analysis of system activity aiming at bringing back to the security officer any suspect use. The system activity is recorded in an audit trail which provides a sequence of events related to a trace of program execution.

We have designed a high-level language of specification of signatures of attacks named Sutekh. The signatures of attacks are the manifestations that can be found in the audit trails recorded on attacked systems. The aim of a signature language is to provide security officers with a way of descibing what kind of activity must be considered as an "intrusion". The expressiveness of a signature language is correlated with the capability of the intrusion detection engine that actually performs the analysis of the audit trail. Indeed, an expressive signature language that descibes complex combinations of events requires a sophisticated engine. Because of the strong link between languages and engines, in many existing systems, the description of the signatures are close to the algorithms of detection and hence contain too many low level details. Recent approaches proposed declarative languages of specification of signatures which raise the level of abstraction. The aim of these language is to focus on the description of the signatures (ie. what combination of event is relevant of an intrusion) and to hide operational details. Sutekh belongs to this class of declarative languages. The gap between the declarative languages and the operationnal engines requires a formal framework to enforce the correctness of the approach. Furthermore, even if hiding operational details is required in many cases, giving high level operational control is sometimes needed. To adress these points, we propose an algorithm based on the "parsing schemata" formalism and show the correctness of the approach [39]. We are currently implementing this work, using the ELAN rewriting system developped at the LORIA [44], to perform an online analysis of system call traces in a Solaris environment.

This work is done in the context of the Dico RNTL project (see module 7.3). We particularly collaborate with France Telecom R &D of Caen and Supelec of Rennes in particular on alamrm correlation [15]. M. Ducassé supervises the theses of Benjamin Morin and Elvis Tombini from France Telecom R &D of Caen with Hervé Debar and Ludovic Mé.

6.4. Debugging for constraint logic programming

Participant: Mireille Ducassé.

Key words: *Debugging*, *Tracer design methodology*, *formal trace schema*, *Gnu-Prolog*, *CLP(FD)*.

This work is a joint activity with the INRIA project CONTRAINTES from Rocquencourt. Together with P. Deransart, M. Ducassé is the PhD supervisor of Ludovic Langevine.

Tracers give partial insight into the behaviour of a program: with an execution trace a programmer can debug and tune programs. Traces can also be used by analysis tools, for example to produce statistics or build graphical views of program behaviors. Constraint propagation tracers are especially needed because constraint propagation problems are particularly hard to debug. Yet, there is no satisfactory tracer for CLP(FD) systems. Some do not provide enough information, others are very inefficient. We have formally designed a tracer for Gnu-Prolog [31]. It provides more complete information than existing propagation tracers. Benchmarks show that its implementation is efficient. Its formal specification is useful both to implement the tracer and to understand the produced trace. It is designed to cover many debugging needs.

In order to design and implement tracers, one must decide *what* exactly to trace and *how* to produce this trace. On the one hand, trace designs are too often guided by implementation concerns and are not as useful as they should be. On the other hand, an interesting trace which cannot be produced efficiently, is not very useful either. The design and implementation of the tracer for Gnu-Prolog has given us the opportunity to work on the methodology to build debugging tools, from formal specification to prototyping, implementation and testing [25].

This work is done in the context of OADYMPPAC, a French RNTL project (see module 7.4. Several solvers of different types have to be connected to different tools. We have generalized the trace schema in order to enable debugging tools to be defined almost independently from finite domain solvers, and conversely, tracers to be built independently from these tools [24][42].

6.5. UML-Casting : Test case generation by data flow analysis of UML specifications

Participants: Lionel Van Aertryck, Thomas Jensen.

UML-Casting is a methodology (coupled with a prototype tool) for computer-assisted generation of test suites from UML models [35]. The concept underlying Casting is that a large part of test cases corresponding to a given test strategy can be obtained automatically from the text of the specification. A test strategy consists of a set of decomposition rules that are applied to the Boolean expressions in a specification in order to reveal potentially interesting test cases. In the case of UML-Casting, the decomposition is applied to class and state machine diagrams in order to yield an intermediate representation (also in the shape of a state machine) from which test objectives (here, method invocations) can be extracted. The user can then specify two kinds of goals: either a coverage percentage that he wants to achieve or specific test objective that he wants to see executed. In either case, UML-Casting will use constraint solving techniques to produce test cases to achieve these goals. UML-Casting was developed as part of the French RNTL project « Cote ».

6.6. Automated Metamorphic Testing

Participant: Arnaud Gotlieb.

Key words: *Metamorphic Testing*, *oracle problem*, *automatic test data generation*, *Constraint Logic Programming*.

Usual techniques for automatic test data generation are based on the assumption that a complete oracle will be available during the testing process. However, there are programs for which this assumption is unreasonable. Recently, Chen et al. proposed to overcome this obstacle by using known relations over the input data and their unknown expected outputs to look for a subclass of faults inside imperative programs. In this work, we introduced an automatic testing framework able to check these so-called metamorphic relations. The framework makes use of Constraint Logic Programming techniques to find test data that violate a given metamorphic–relation. Circumstances where it can also prove that the program satisfies this relation were presented. The first experimental results we got with a prototype tool build on the top of the test data generator INKA, showed that this methodogy can be completely automated [29]. This work has been made in cooperation with Bernard Botella from Thales Airborn Systems.

6.7. Exploiting Symmetries to Test Programs

Participant: Arnaud Gotlieb.

Key words: *Program Testing*, *symmetry*, *oracle problem*, *permutations*, *Group theory*.

In Program Testing, symmetries be viewed as properties of programs and can be given by the tester to serve as oracles or to check the correctness of the computed outcome. In this work, we considered symmetries that are permutation relations between program executions and used them to automate the testing process. We introduced a software testing paradigm called Symmetric Testing, where automatic test data generation is coupled with symmetries checking to uncover faults inside imperative and sequential programs. A practical procedure for checking that a program satisfies a given symmetry relation was designed. The paradigm makes use of Group theoretic results as a formal basis to minimize the number of outcome comparisons required by the method. This approach appeared to be of particular interest for programs for which neither an actual oracle, nor any formal specification is available. We implemented Symmetric Testing by using the primitive operations of the Java unit testing tool *Roast*. The experimental results we got on faulty versions of classical programs of the Software Testing community tend to show the effectiveness of the approach [30].

6.8. Logic information system

Participants: Yoann Padioleau, Olivier Ridoux, Benjamin Sigonneau.

Key words: *Logic information system*, *file system*.

The first file system implementation of a logic information system has been presented at USENIX [32][40]. It is a fully-featured file system working on the virtual file system (VFS) interface of Linux. It offers navigation among concepts under the usual `cd`/`ls` protocol. This system is called LISFS.

A sub-system of LISFS, called PofFS for "Parts-of-file" file system, offers mechanisms for navigating *inside* file using the same concept-based semantics as LISFS [33][41]. This allows one to extract almost arbitrary *views* of a program, and possibly to update them. In the latter case, updates are back-propagated to the original file. This introduces a *coherent view-update problem* which we have solved for PofFS.

In the mean-time we have explored applications of LISFS and PofFS, especially in the software engineering domain. For instance, we have used LISFS to support a type-based organization of Java methods. Since in a LIS, and also in LISFS, all kinds of organizations may cohabit, we have also made experiments with non-formal organizations, based on keywords extracted from identifiers or comments.

Further search directions are to improve the performance of LISFS and PofFS, to extend them to handle a greater part of a LIS capabilities, and to examine their possible impact on applications that manipulate complex sets of data (e.g., software engineering, geographic information systems). A more fundamental direction is to formalize explicit relations among object as well as implicit relations. The current theory only formalizes implicit relations between formal concepts.

6.9. Semi-automatic verification methods for Systems of Affine Recurrence Equations

Participants: David Cachera, Katell Morin-Allory, David Pichardie.

We have investigated the use of formal methods to prove functional properties about systems of affine recurrence equations over polyhedral multidimensional domains. We take advantage of the abstraction provided by the polyhedral model to define semi-automatic verification tools.

On one hand, we considered using theorem provers, and have implemented a translation from SAREs into the **Coq** system [23][37]. We overcome the lack of automation of theorem provers by developing proof tactics specific to our model. We take advantage of the regularity of our model to automatically generate induction schemes adapted to each particular system. For the verification of complex systems, we developed an importation/exportation mechanism that allows for a simple and efficient reuse of theorems in modular proofs.

On the other hand, we have propose a combination of heuristic methods to prove safety properties in parameterized SAREs [22][36]. These heuristics combine syntactic substitutions with automatic invariant and pattern detection. Thanks to the intrinsic regularity of our model, we are able to identify those cases where these technics apply successfully. This lead us to discover some errors in actual systems.

6.10. Verification of Cryptographic Protocols by Reachability Analysis

Participants: Thomas Genet, Valérie Viet Triem Tong.

Key words: *Verification*, *cryptographic protocols*, *reachability analysis*.

Verification of cryptographic protocols require precise and sophisticated analysis techniques in order to guarantee their security under very general verification hypothesis: no limit on the number of runned sessions, no limit on the number of actors running the protocol at the same time. Many works are devoted to attack discovery and generally use model-checking. When attacks are no longer found, it is necessary to prove than there is no more whatever be the number of sessions to be runned or the number of actors involved in the protocol executions. Several ways to achieve such a proof have already been explored. First of all, it is possible to make the proof by hand or with a proof assistant. However, the proof is generally tedious, not automatic and require a lot of user intuition. On the other hand, for some specific syntactic sub-classes of cryptographic protocols, the secrecy property is decidable. This aspect is being also widely investigated. However, even if some real protocols are in this decidable class, this is not the case in general.

Our approach to cryptographic protocol verification [55] is in between since it is not restricted to a specific class of protocols neither to a specific property to prove, and it is semi-automatic. We apply reachability analysis over term rewriting systems (see 3.5) to the verification of cryptographic protocols. We encode protocols into term rewriting systems and construct an over-approximation of the set of reachable terms. Then, in this over-approximation, if there is no term representing a violation of one of the security property then the protocol is secure.

On these aspects, we have obtained both theoretical and practical results. On a theoretical point of view, we have extended reachability analysis and the tree automata completion to conditional term rewriting systems [26]. Conditional term rewriting systems will offer a more general and more expressive framework for modelling programs in general and cryptographic protocols in particular.

On a practical point of view, reachability analysis have been successfully used for verifying the “view-only” protocol of the SmartRight digital rights management system developped by Thomson-Multimedia [61]. On this protocol, we have shown the secrecy, the authentication and the “no-replay” properties necessary to ensure that a digital content cannot be played twice (as in pay per view systems) [28].

6.11. Aspect-oriented programming

Participants: Pascal Fradet, Stéphane Hong Tuan-Ha.

Key words: *Aspects*, *interaction*, *composition*, *fusion*.

The goal of Aspect-Oriented Programming (AOP) is to isolate aspects (such as security, synchronization or error handling) which cross-cut the program basic functionality and whose implementation would otherwise yield tangled code. In AOP, such aspects are specified separately and integrated into the program by an automatic transformation process called *weaving*.

We have advocated an approach which — by means of expressive trace-based aspect languages and restrictions on inserts — enables reasoning about different aspect properties [17]. *Trace-Based Aspects* are defined on traces of events occurring during program execution. These aspects are more expressive than those based on atomic points because relations between execution events – possibly involving information from the corresponding execution states — can be expressed.

Aspects are not always orthogonal and aspect interaction is a fundamental problem of AOP. By restricting aspect to regular expressions, we have shown that it is possible to statically detect whether several aspects interact [17]. Recently, we have worked on three extensions of trace-based aspects:

- Support for variables in aspects. This allows the definition of even more precise aspects and avoid detecting spurious conflicts.
- Generic composition operators for aspects. This enables us to provide expressive support for the resolution of conflicts among interacting aspects.
- Applicability conditions for aspects. This makes interaction analysis more precise and paves the way for reuse of aspects by making explicit requirements on contexts in which aspects must be used.

This work is done in collaboration with Rémi Douence and Mario Südholt from the Obasco project-team at Ecole des Mines de Nantes.

We have also studied the application of aspects to the efficient implementation of component-based systems. In our approach, a component-based software is represented by a *Kahn Process Network* (KPN). Each component is modelised by a sequential program communicating with other components using (non-blocking) writes and (blocking) reads on ports. The whole system is represented of a network of components whose ports are interconnected by FIFO channels (i.e. a KPN). To implement efficiently such networks, we make use of aspects of composition and invasive composition techniques. The network and aspects are woven into a unique sequential, and hopefully, efficient program. More precisely:

- Even if a network of components defines a deterministic program, it has many possible executions. The programmer can define constraints on the execution (scheduling, size of FIFOS, etc.) in composition aspects.
- An automatic process finds a schedule satisfying the programmer's constraints and the semantics of the network (i.e. a maximal and fair schedule).
- Using this schedule, the network of components is woven into a unique sequential program.

This AOP-inspired technique permits to keep separate the construction of component-based systems from scheduling and efficiency issues.

6.12. Programming models and calculi

Participant: Pascal Fradet.

Key words: *λ-calculus*, *abstract machine*, *Gamma*, *chemical reaction metaphor*.

The Krivine machine is a simple and natural implementation of the call-by-name λ-calculus. While its original description has remained unpublished, this machine has served as a basis for many variants, extensions and theoretical studies. We have presented the Krivine machine and some well-known variants in a common framework [16]. We have characterized the essence of the Krivine machine and have located it in the design space of functional language implementations. We have showed that, even within the particular class of Krivine machines, hundreds of variants can be designed. This work is based on the framework that we have previously developed for the systematic study of functional language implementations [46].

This is joint work with Rémi Douence from Ecole des Mines de Nantes.

Another strand of work has concerned the γ-calculus and higher-order chemical reactions. Gamma is a formalism in which programs are expressed in terms of multiset rewriting often referred to as the Chemical Reaction Model. We have proposed a formal and basic calculus, the γ-calculus [20], which allows the definition of γ-abstractions as first class citizens (in the same sense as λ-abstractions in the λ-calculus). In this formalism, the execution of a program can be seen as the evolution of a solution of molecules (i.e. γ-expressions) which react until the solution becomes inert. This calculus can of course express the classical Gamma formalism but its higher-order nature makes it easy to describe notions such as code mobility, distribution, adaptation, etc.

This work is done in collaboration with Jean-Pierre Banâtre and Yann Radenac from the Paris team at IRISA.

7. Contracts and Grants with Industry

7.1. The IST FET/Open project SECSAFE

Participants: Frédéric Besson, Thomas Jensen, Florimond Ployette.

Key words: *static program analysis*, *software security*, *constraint solving*, *smart card*.

The European project SECSAFE (Secure and Safe Systems based on Static Analysis) has had as its objectives to assess the scalability of static analysis technology to the validation of security and safety aspects of realistic languages and applications. We have identified two domains where security is all-important: smart cards and internet programming. Thus, the project has been focussed on using static program analysis techniques to improve and validate the security of software, in particular software for smart cards written in the Java programming language and its dialect Java Card , treating source-level as well as bytecode-level applications. This has lead to a precise definition of the Java Card semantics, a detailed static analysis for a representative subset of the Java Card byte code, including an accurate modelling of the Java Card firewall. In order to implement this analysis, several constraint solvers, based on different resolution techniques, have been developed.

7.2. The IST R&D project VERIFICARD

Participants: Thomas Jensen, David Pichardie, Thomas Genet.

Key words: *smart cards*, *formal methods*, *automated deduction*, *certified compilation*.

The European project VERIFICARD has been concerned with using formal methods to improve and validate the safety and security of software for smart cards written in the Java Card programming language. The Lande project has participated together with two other Inria project-teams (Logical and Lemme) in a consortium involving U. Nijmegen, SICS, U. Kaiserslautern, Tech. U. Munich and Schliumberger. Our part of the project has been concerned with provinding a provable correct compiler from the Java Card byte code format to the intermediate, optimised representation called the Java Card CAP format. The technical details are described in Section 6.2.

7.3. The DICO RNTL project

Participants: Mireille Ducassé, Jean-Philippe Pouzol.

Key words: *cooperative detection of intrusions*.

The DICO project officially started on December 5, 2001, for 2 years. Its extension is currently under discussion. The objective of the project is double. It proposes new techniques, on the one hand, to detect intrusions and, on the other hand, to reduce by correlation the number of generated alarms. The existing intrusion detection techniques, although promising, are still imperfect. They generate alarms of too fine granularity, cause many false positives (generated alarms whereas there is no attack) and false negatives (no generated alarm whereas there is an attack). All that complicates the analysis and the final diagnosis of a security administrator, by submerging him with alarms among which it is difficult to extract the most relevant information. It is thus necessary to propose new approaches to improve the rate of detected attacks, the quality of the diagnosis as well as the performances of the tools for detection of intrusions. Within this framework, Lande takes part more particularly in the definition of a language to specify attack scenarios at a high level(see module 6.3). Lande develops an intrusion detection tool which will use the preceding results. This action benefits from a support of the French Ministry for research, in the form of an RNTL contract. Our industrial partners are NetSecure Software and France Telecom R &D. Our university partners are SUPELEC of Rennes, the "Ecole Normale Supérieure" of Cachan (laboratory LSV) and the FERIA of Toulouse with its components ONERA and IRIT.

7.4. The OADYMPPAC RNTL project

Participant: Mireille Ducassé.

Key words: *programming environment*, *logic programming with constraints*, *debugging*, *visualization*.

The OADYMPPAC Project officially started on November 15, 2000, for 3 and a half years. One of the objectives of the project is to design and experiment generic debugging techniques for logic programs with constraints. That will facilitate the definition of observation and development tools. Another aspect is to study the contribution of the progress of visualization techniques for large sets of data. The aim is twofold: on the one hand, a better comprehension of the behavior of the solveurs is expected ; in the other hand generic techniques of visualization of information will be adapted to the visual analysis of the traces produced by the dynamic phenomena.

The project currently works on the formats of traces common to the various tools. A generic trace schema has been specified (see module 6.4). The project maintains a page of information:
http://contraintes.inria.fr/OADymPPaC/. This action benefits from a support of the French Ministry for research, in the form of an RNTL contract. The industrial partners of the consortium are Ilog and Cosytec. Our university partners are the INRIA Rocquencourt, the university of Orleans and the "Ecole des Mines" of Nantes.

The project hosted, for 2 months, Bil Lewis who contributed to the implementation of a connection between a tracer and a trace analyzer.

7.5. Contract with the Brittany region: SACOL

Participants: Pascal Fradet, Stéphane Hong Tuan-Ha, Thomas Jensen.

Key words: *Security*, *safety*, *weaving*, *modularity*, *component*.

Usually, programs are securized manually by inserting tests and controls. The SACOL (*Sécurisation Automatique de Composants Logiciels*) project aims at designing automatic securization techniques for component-based systems. The two main objectives are:

- Allowing the programmer to specify security properties separately from programs.
- Providing an automatic tool to enforce properties to programs.

An important challenge is to make the approach modular. In particular, it must apply to large software built as interconnected components. Our approach relies on programming (aspects), transformation (weaving) and modular analysis techniques. The work on aspects of composition related in Section already defines the components, their interface and their fusion. The next phase is to design modular analyses and transformations in order to enforce security properties on such networks.

8. Other Grants and Activities

8.1. The Inria ARC MODOCOP : Model checking of concurrent object-oriented programs

Participant: Thomas Jensen.

Key words: *static program analysis*, *software security*, *model checking*, *object-oriented programming*.

The Modocop project is concerned with automatic specification, verification and symbolic testing of concurrent object-oriented programs. It combines two important lines of research:

- verification of single-threaded Java Card programs, which is mainly theorem-prover based; and
- application of symbolic techniques to generate tests for embedded applications, which can be analysed using either finite model checking techniques or symbolic methods that can deal with infinite state systems.

In particular, the project aims at the development of a framework for specification, verification and symbolic testing of concurrent object-oriented programs. This framework consist of a specification language, which is tailored to specify properties about concurrent object-oriented programs; abstraction techniques, which allow a program to be transformed into an abstract program model that is amenable to model checking; verification techniques which can be used to verify both the functional behaviour and the interactive behaviour of the different objects within concurrent object-oriented programs; and symbolic test generation techniques, which allow to check that an actual, physical implementation still satisfies the behaviour as described by the verified formal model.

The framework is designed in such a way that it can be applied to arbitrary concurrent object-oriented programs. JavaCard with multi-threading will be used as an example instantiation for our framework, as it is a complete, but still manageable programming language.

8.2. The CNRS Action Spécifique SECURITE LOGICIELLE

Participants: Thomas Jensen, Thomas Genet, David Cachera, Pascal Fradet.

Key words: *software security*, *static analysis*, *protocol verification card*.

This research action on software security is a joint collaboration between a number of French research groups working on different aspects of software security. It is coordinated by ENS Cachan, ENST Bretagne, Verimag and IRISA/Lande. One of the primary goals of the action is to federate the activites in software security at a national level. To achieve this, a number of research objectives have been identified, including:

- specification,validation and coherence of security policies,
- security of software on embedded devices (in particular smart cards),
- verification of cyrptographic protocols.

The action was held two meetings (in Paris and Grenoble). It is part of the CNRS Research Networks on Embedded Systems and on Information Security.

8.3. The ACI Sécurité Informatique project DISPO : Disponibilité de services dans des composants logiciels

Participants: Pascal Fradet, Thomas Jensen, Stéphane Hong Tuan-Ha.

Key words: *Availability*, *software components*, *aspects*.

8.3.1. The DISPO project

, coordinated by Lande, is concerned with specifying, verifying and enforcing security policies governing the *availability* of services offered by software components. Contrary to the other two kinds of security properties (integrity and confidentiality), there are only few attempts on formalising availability policies. Furthermore, the results obtained for availability has so far not been connected with the software engineering process that manufactures the components. The aim of the project is therefore to develop suitable specification formalisms together with formal methods for program verification and transformation techniques taking advantage of modern program structuring techniques such as component-based and aspect-oriented software development.

The project is composed of three sub-activities:

- Developing a formalism for specifying availability properties. This formalism will be based on temporal and deontic logics. We will be paying particular attention to the problem of verifying the *coherence* of policies specified in this formalism.
- We will use a combination of static and dynamic techniques for enforcing a particular availability property on a particular software component. In the purely static view, properties are enforced by a combination of static program analysis and model checking, in which the code of a component is abstracted into a finite or finitary model on which behavioral properties can be model-checked. When such verifications fail (either because the property does not hold or because the analysis is incapable of proving it), we will employ the technique of aspect-oriented programming to transform the program so as to satisfy a given property.
- At the architectural level, the project ains at developing a component model equipped with a notion of availability interface, describing not only static but also dynamic properties of the component. The challenge here is to define suitable composition operators that allow to reason at the level of interfaces. More speculatively, we intend to investigate how the notion of aspects apply at the architecture level, and in particular how to specify aspects of components in such a way that properties can be enforced at component assembly time.

The project consortium consists of four partners: Ecole des Mines de Nantes, IRISA (Rennes), IRIT (Toulouse) and ENST-Bretagne.

8.4. The ACI Sécurité Informatique project V3F: Validation and Verification of programs with floating-point numbers computations

Participant: Arnaud Gotlieb.

Key words: *Validation*, *Verification*, *constraint solving*, *floating-point numbers computations*.

Computations with floating-point numbers are a major source of failures of critical software systems. It is well known that the result of the evaluation of an arithmetic expression over the floats may be very different from the exact value of this expression over the real numbers. Formal specifications languages, model-checking techniques, and testing are currently the main issues to improve the reliability of critical systems. A significant effort over the past year was directed towards development and optimization of these techniques, but few works has been done to tackle applications with floating-point numbers. A correct handling of a floating point representation of the real numbers is very difficult because of the extremely poor mathematical properties of floating-point numbers; moreover the results of computations with floating-point numbers may depends on the hardware, even if the processor complies with the IEEE 754 norm.

The goal of this project is to provide tools to support the verification and validation process of programs with floating-point numbers. More precisely, project V3F will investigate techniques to check that a program satisfies the calculations hypothesis on the real numbers that have been done during the modelling step. The underlying technology will be based on constraint programming. Constraints solving techniques have been successfully used during the last years for automatic test data generation, model-checking and static analysis. However in all these applications, the domains of the constraints were restricted either to finite subsets of the integers, rational numbers or intervals of real numbers. Hence, the investigation of solving techniques for constraint systems over floating-point numbers is an essential issue for handling problems over the floats.

So, the expected results of project V3F are a clean design of constraint solving techniques over floating-point number, and a deep study of the capabilities of these techniques in the software validation and verification process. More precisely, we will develop an open and generic prototype of a constraint solver over the floats. We will also pay a special attention on the integration of floats into various formal notations (e.g., B, Lustre, UML/OCL) to allow an effective use of the constraint solver in formal model verification, automatic test data generation (functional and structural) and static analysis.

The V3F project is funded by the French National Science Fund. It is a 3 years project and 4 partners are involved : the INRIA Cassis project from LORIA, the INRIA Coprin project from RU of Sophia-Antipolis, the INRIA Lande and Vertecs projects from IRISA and the LSL group of CEA.

9. Dissemination

9.1. Dissemination of results:

9.1.1. *Conferences: program committees, organization, invitations*

Olivier Ridoux served on the LOPSTR'2003 (Logic-based Program Synthesis and Transformation) program committee.

Mireille Ducassé has been the program committee chair of the JFPLC'03 (Journées Francophones de Programmation en Logique et avec Contraintes), and is the editor of the proceedings published by Hermès [11].

Mireille Ducassé served in the program committee of AADEBUG'03 (International Workshop on Automated Debugging) and ICLP'03 (International conference on Logic Programming). At ICLP'03 she organizes a panel on teaching constraint logic programming.

Mireille Ducasse is general chair and organizer of the Int. Conference on Logic Programming 2004. Arnaud Gotlieb acts as publicity chair for the same conference.

Thomas Jensen (together with Marieke Huisman of Inria Sophia-Antipolis) has co-edited a special issue of J. Algebraic and Logic Programing dedicated to formal methods for smart card software [12].

Olivier Ridoux has given an invited talk at ICLP'2003 (International Conference on Logic Programming) [34].

Thomas Jensen has been invited to the Dagstuhl semainar on Language-based Security, October 2003. Mireille Ducassé has been invited to the Dagstuhl seminar on dynamic analysis organized in Dagstuhl, december 2003.

Fausto Spoto, assistant professor at the Department of Informatics at the University of Verona, visited the project in the period June-September 2003.

9.1.2. *PhD theses defended*

Valérie Viet Triem Tong, PhD University of Rennes 1, on tree automata and rewriting for reachability problems, supervised by Thomas Genet [13].

Jeanne Villaneau PhD Université de Bretagne Sud (Vannes-Lorient) on logic approaches to the understanding of spoken language [62], supervised by Olivier Ridoux.

9.1.3. PhD and Habilitation committees

Mireille Ducassé is *rapporteur* on the PhD thesis of Fabien Gaucher, U. Jospeph Fourier, Grenoble and on the habilitation of Daniel Deveaux, U. of Vannes. She has been president of the jury of the habilitation of Ludovic Mé Supelec, Rennes.

Thomas Jensen is *rapporteur* on the PhD theses of Hervé Grall (Ec. Ponts et Chaussées) and Guillaume Dufay (University of Nice Sophia Antipolis). He has been on the jury of the habilitation of Sophie Pinchinat and the theses of Simon Pickin and Valérie Viet Triem Tong, all University of Rennes 1.

9.1.4. Teaching: university courses and summer schools

Olivier Ridoux teaches compiling, operating system, logic programming, and software engineering at IFSIC (*Institut de Formation Supérieure en Informatique et Communication* — the computer science department at University of Rennes 1). He is the coordinator of the Maîtrise level (4th university year). He works with the professional training committee of IRISA ("Commission de formation permanente"), and organised a 2-day research school on data-mining, for which he contributed a session. The plan is to have such research schools on a regular basis, on different subjects.

Thomas Genet teaches formal methods for software engineering (with "B") for Maîtrise level (4th university year). He also teaches formal methods and proof assistants to DEA students (5th university year) in collaboration with Vlad Rusu (VERTECS project).

Thomas Jensen teaches semantics, type systems and static analysis at the 5th year DEA level in collaboration with Bertrand Jeannet (VERTECS project).

Thomas Jensen is scientific leader of the *Ecole Jeunes Chercheurs en Programmation*, an annual summer school for graduate students on programming languages and verification, organized under the auspicies of the CNRS GdR ALP. This year's event was held at Aussois and organized together with Verimag, Grenoble.

Jean-Philippe Pouzol gave an introduction course on intrusion detection at the pedagogical seminar of the IFSIC (Journées pédagogiques de l'IFSIC), Lannion, June 2003.

10. Bibliography

Major publications by the team in recent years

[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] M. DUCASSÉ. *Opium: An extendable trace analyser for Prolog.* in « Journal of Logic programming », volume 39, 1999, pages 177-223.

[4] S. FERRÉ, O. RIDOUX. *Introduction to Logic Information Systems.* in « Elsevier J. Information Processing & Management », 2004, http://www.sciencedirect.com/science.

[5] 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.

[6] T. GENET, F. KLAY. *Rewriting for Cryptographic Protocol Verification.* in « Proceedings 17th International Conference on Automated Deduction », series Lecture Notes in Artificial Intelligence, volume 1831, Springer-Verlag, 2000.

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

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

[9] 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.

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

Books and Monographs

[11] *Actes des Journées Francophones de Programmation en Logique avec Contraintes.* M. DUCASSÉ, editor, Revue des Sciences et Technologies de l'Information, Hors série/JFPLC 2003, HERMES Science Publications, June, 2003.

[12] *J. Logic and Algebraic Programming. Speical issue on Formal Methods for Smart Cards.* M. HUISMAN, T. JENSEN, editors, Elsevier, 2003, to appear.

Doctoral dissertations and "Habilitation" theses

[13] V. VIET TRIEM TONG. *Automates d'arbres et réécriture pour l'étude de problèmes d'accessibilité.* Ph. D. Thesis, Université Rennes 1, 2003.

Articles in referred journals and book chapters

[14] A. BANERJEE, T. JENSEN. *Control-flow analysis with rank-2 intersection types.* in « Mathematical Structures in Computer Science », number 1, volume 13, 2003, pages 87–124.

[15] H. DEBAR, B. MORIN, F. C. F. AUTREL, L. MÉ, B. V. S. BENFERHAT, M. DUCASSÉ, R. ORTALO. *Corrélation d'alertes en détection d'intrusions.* in « Technique et Science Informatiques, TSI », volume To appear, 2003.

[16] R. DOUENCE, P. FRADET. *The next 700 Krivine Machines.* in « Higher-Order and Symbolic Computation », 2003, to appear.

[17] R. DOUENCE, P. FRADET, M. SÜDHOLT. *Trace-Based Aspects.* M. A. ET AL., editor, in « Aspect-Oriented Software Development », Addison-Wesley, 2003.

[18] M. ELUARD, T. JENSEN. *Vérification du contrôle d'accès dans des cartes à puce multi-application.* in « Technique et Science Informatiques », 2003, to appear.

[19] F. SPOTO, T. JENSEN. *Class Analyses as Abstract Interpretations of Trace Semantics.* in « ACM Transactions on Programming Languages and Systems (TOPLAS) », number 5, volume 25, 2003, pages 578–630.

Publications in Conferences and Workshops

[20] J.-P. BANÂTRE, P. FRADET, Y. RADENAC. *Higher-order chemistry.* in « Preproceedings of the Workshop on Membrane Computing », July, 2003.

[21] F. BESSON, T. JENSEN. *Modular Class Analysis with DATALOG.* in « Proc. of 10th Static Analysis Symposium (SAS 2003) », Springer LNCS vol. 2694, R. COUSOT, editor, pages 19–36, 2003.

[22] D. CACHERA, K. MORIN-ALLORY. *Verification of Control Properties in the Polyhedral Model.* in « Proc. 1st MEMOCODE conference », Mont-St-Michel, France, jun, 2003.

[23] D. CACHERA, D. PICHARDIE. *Embedding of Systems of Affine Recurrence Equations in Coq.* in « Proc. TPHOLs 2003, 16th International Conference on Theorem Proving in Higher Order Logics », series LNCS, Rome, Italy, sep, 2003.

[24] P. DERANSART, L. LANGEVINE, M. DUCASSÉ. *A Generic Trace Schema for the Portability of CP(FD) Debugging Tools.* in « Proceedings of the ERCIM workshop on Constraint and Logic Programming », 2003.

[25] M. DUCASSÉ, L. LANGEVINE, P. DERANSART. *Rigorous design of tracers: an experiment for constraint logic programming.* in « Proceedings of the Fifth International Workshop on Automated Debugging », CoRR cs.SE/0309027, M. RONSSE, editor, September, 2003.

[26] G. FEUILLADE, T. GENET. *Reachability in conditional term rewriting systems.* in « FTP'2003, International Workshop on First-Order Theorem Proving », series ENTCS, volume 86 n. 1, Elsevier, June, 2003.

[27] T. GENET, T. JENSEN, V. KODATI, D. PICHARDIE. *A Java Card CAP Converter in PVS.* in « Proc. of 2nd International Workshop on Compiler Optimization Meets Compiler Verification (COCV 2003) », ENTCS 82(2), 2003.

[28] T. GENET, Y.-M. TANG-TALPIN, V. VIET TRIEM TONG. *Verification of Copy Protection Cryptographic Protocol using Approximations of Term Rewriting Systems.* in « In Proceedings of Workshop on Issues in the Theory of Security », 2003.

[29] A. GOTLIEB, B. BOTELLA. *Automated Metamorphic Testing.* in « Proc. of the 27th IEEE Annual International Computer Software and Applications Conference (COMPSAC) », Dallas, TX, USA, 2003, 3th to 7th November.

[30] A. GOTLIEB. *Exploiting Symmetries to Test Programs.* in « Proc. of 14th IEEE International Symposium on Software Reliability Engineering (ISSRE 2003) », Denver, Colorado, USA, 2003, 17th to 20th November.

[31] L. LANGEVINE, M. DUCASSÉ, P. DERANSART. *A Propagation Tracer for Gnu-Prolog: from Formal Definition to Efficient Implementation.* in « Proceedings of the 19th Int. Conf. in Logic Programming », Springer-Verlag, Lecture Notes in Computer Science, C. PALAMIDESSI, editor, December, 2003.

[32] Y. PADIOLEAU, O. RIDOUX. *A Logic File System.* in « Proc. USENIX Annual Technical Conference », 2003.

[33] Y. PADIOLEAU, O. RIDOUX. *Présentation du « Parts-of-file File System »*. in « Proc. Conférence Française sur les Systèmes d'exploitation », 2003.

[34] O. RIDOUX. *Logic Information Systems for Logic Programmers*. in « Proc. Int. Conf. Logic Programming », Springer-Verlag, LNCS, C. PALAMIDESSI, editor, 2003.

[35] 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, J.-M. JÉZÉQUEL, editor, 2003.

Internal Reports

[36] D. CACHERA, K. MORIN-ALLORY. *Verification of Control Properties in the Polyhedral Model*. Technical report, number 4756, INRIA, 2003, http://www.inria.fr/rrrt/rr-4756.html.

[37] D. CACHERA, D. PICHARDIE. *Proof Tactics for the Verification of Structured Systems of Affine Recurrence Equations*. Technical report, number 1511, IRISA, 2003, ftp://ftp.irisa.fr/techreports/2003/PI-1511.ps.gz.

[38] G. FEUILLADE, T. GENET, V. VIET TRIEM TONG. *Reachability Analysis over Term Rewriting Systems*. Technical report, number RR-4970, INRIA, 2003, http://www.inria.fr/rrrt/rr-4970.html.

[39] J. GOUBAULT-LARRECQ, S. DEMRI, M. DUCASSÉ, L. MÉ, J. OLIVAIN, C. PICARONNY, J.-P. POUZOL, E. TOTEL, B. VIVINIS. *Algorithmes de détection et langages de signatures*. Livrable, number 3.3, Projet RNTL DICO, Octobre, 2003.

[40] Y. PADIOLEAU, O. RIDOUX. *A logic file system*. Rapport de recherche, number 4656, INRIA, 2003, http://www.inria.fr/rrrt/rr-4656.html.

[41] Y. PADIOLEAU, O. RIDOUX. *The Parts-of-file File System*. Rapport de recherche, number 4783, INRIA, 2003, http://www.inria.fr/rrrt/rr-4783.html.

Miscellaneous

[42] P. DERANSART, L. LANGEVINE, M. DUCASSÉ. *Debugging Constraint problems with Portable Tools*. Demonstration presented at the 13th Workshop on Logic Programming Environments, December, 2003.

Bibliography in notes

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

[44] P. BOROVANSKY, C. KIRCHNER, H. KIRCHNER, P. E. MOREAU, M. VITTEK. *ELAN: A Logical Framework Based on Computational Systems*. in « Proc. of the First Int. Workshop on Rewriting Logic », volume 4, Elsevier, 1996.

[45] P. COUSOT, R. COUSOT. *Abstract Interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints*. in « Proc. of 4th ACM Symposium on Principles of Programming Languages », ACM Press, New York, pages 238–252, 1977.

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

[47] S. FERRÉ, O. RIDOUX. *Introduction to Logic Information Systems.* in « Elsevier J. Information Processing & Management », 2004, http://www.sciencedirect.com/science.

[48] S. FERRÉ. *Systèmes d'information logiques : un paradigme logico-contex tuel pour interroger, naviguer et apprendre.* Ph. D. Thesis, Université de Rennes 1, 2003.

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

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

[51] S. FERRÉ, O. RIDOUX. *Searching for Objects and Properties with Logical Concept Analysis.* in « Int. Conf. Conceptual Structures », series LNCS 2120, Springer, 2001.

[52] S. FERRÉ, O. RIDOUX. *The Use of Associative Concepts in the Incremental Building of a Logical Context.* in « Int. Conf. Conceptual Structures », series LNCS 2393, Springer, U. PRISS, D. CORBETT, G. ANGELOVA, editors, pages 299–313, 2002.

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

[54] T. GENET. *Decidable Approximations of Sets of Descendants and Sets of Normal forms.* in « Proceedings 9th International Conference on Rewriting Techniques and Applications », series Lecture Notes in Computer Science, volume 1379, Springer-Verlag, pages 151–165, 1998.

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

[56] 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 », series Lecture Notes in Artificial Intelligence, volume 2250, Springer-Verlag, pages 691–702, 2001.

[57] D. E. KNUTH, P. B. BENDIX. *Simple word problems in universal algebras.* J. LEECH, editor, in « Computational Problems in Abstract Algebra », Pergamon Press, Oxford, 1970, pages 263–297.

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

[59] 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.

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

[61] THOMSON. *SmartRight Technical White Paper V1.0.* Thomson, October, 2001, http://www.smartright.org.

[62] J. VILLANEAU. *Contribution au traitement syntaxico-pragmatique de la langue naturelle parlée : approche logique pour la compréhension de la parole.* Ph. D. Thesis, Université de Bretagne Sud, 2003.