INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Project-Team Lande

Logiciel : ANalyse et DEveloppement

Rennes

THEME SYM

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
Frédéric Besson [Research scientist, CR, from September 2004]
Arnaud Gotlieb [Research scientist, CR]
Florimond Ployette [Technical staff, IR, until September 2004]

Université Rennes 1 Personnel
Olivier Ridoux [Professor]
Sébastien Ferré [Assistant Professor, from September 2004]
Thomas Genet [Assistant Professor]

Insa Personnel
Mireille Ducassé [Professor]

ENS Cachan Personnel
David Cachera [Assistant Professor]

PhD students
Yoann Padioleau [ATER Université Rennes 1, until September 2004]
Jean-Philippe Pouzol [ATER Université Rennes 1, until September 2004]
Thomas de Grenier de Latour [MENRT grant]
Stéphane Hong Tuan-Ha [Inria-Region Bretagne]
David Pichardie [ENS Cachan]
Gurvan Le Guernic [MENRT grant]
Benjamin Sigonneau [MENRT grant]
Matthieu Petit [Bretagne grant]
Tristan Denmat [MENRT grant]
Luka Leroux [Inria-France Télécom R&D grant]

Post-doctoral researchers
Valérie Viet Triem Tong [ATER U. Rennes 1, until September 2004]
Sandrine Gouraud [Post-doc CNRS]
Gerardo Schneider [Post-doc CNRS]

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. 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.

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.

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

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

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 $f^n(\bot)$ where $\bot$ is the least element of the partial order.

3.1.1. Static program analysis

[35][49]

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 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

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

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 [33]. 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 [53] 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

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

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 generation, program execution, outcome checking, non-regression test data selection, etc. Most of them are based on heuristics or empirical choices and current tools lack help for the testers. One of the main goal of the research undertaken by the Lande Project in this field consists in finding ways to automate some parts of the testing process. Current works focus on automatic test data generation and automatic oracle checking. Difficulties include test criteria formalization, automatic source code analysis, low-level specification modeling and automatic constraint solving. The benefits that are expected from these works include a better and deep understanding of the testing process 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 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, automatically generating 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 do not 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.

The main goal of the Lande prooject in this area 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

Keywords: *Logic*, *data-mining*, *formal concept analysis*, *information retrieval*, *information system*, *logic concept analysis*.

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; i.e., 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 [42] 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 [39]. 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 [38][37][36].

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 an 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 framework is in itself a contribution [40].

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 aLogic information system, 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 [41].

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

Keywords: *Term rewriting systems*, *reachability analysis*, *tree automata*.

Term rewriting systems 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 $s \rightarrow_{\mathcal{R}}^{\star} t$. On the opposite, t is $\mathcal{R}$-unreachable from s if s cannot be finitely rewritten into t by $\mathcal{R}$, denoted by $s \not\rightarrow_{\mathcal{R}}^{\star} t$.

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 to a particular value, show that a process 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 $s \not\rightarrow_{\mathcal{R}}^{\star} t$, we already have some results based on the over-approximation of the set of reachable terms [43][46]. 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 [18] 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 [48] *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.

4. Software

4.1. Recursive Equations Solver

Keywords: *fixed point*, *recursive equations*, *solver*.

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

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 equation 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.
- A Carmel (a JavaCard subset) analysis in the framework of the Secsafe project.

REQS is written in Java and can be freely downloaded. REQS has been registered at the APP with number IDDN.FR.001.310017.000.S.P.2004.000.10600.

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

4.2. Logic File System

Keywords: *Logic information system*, *file system*.

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

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. The latest autumn 2004 version can handle contexts of more than 100,000 files.

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

4.3. Timbuk: a Tree Automata Library

Keywords: *Tree automata*, *approximations*, *term rewriting systems*.

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

Timbuk [46] 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):

- exact computation of reachable terms for most of the known decidable classes of term rewriting system,
- approximation of reachable terms and normal forms for any 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/. Timbuk has been registered at the APP with number IDDN.FR.001.20005.00.S.P.2001.000.10600.

A version 2.1 of *Timbuk* is now 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 2.1 also provides a graphical interface called *Tabi* for browsing tree automata and figure out more easily what are the recognized language, as well as *Taml* an Ocaml toplevel with basic functions on tree automata. Timbuk 2.1 was used for a case study done with Thomson-Multimedia for cryptographic protocol verification.

Timbuk is also used by some other research groups to achieve cryptographic protocol verification. Frédéric Oehl and David Sinclair of Dublin University use it in an approach combining a proof assistant (Isabelle/HOL) and approximations (done with Timbuk) [51][50]. Pierre-Cyrille Heam, Yohan Boichut and Olga Kouchnarenko of the Cassis INRIA project use Timbuk as a verification back-end for verification of protocols [47] defined in high level protocol specification format.

5. New Results

5.1. Modular static program analysis

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

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

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.

The Lande project has for several years made an effort to develop modular analysis and verification techniques, with a particular emphasis on their application to validating the security of Java software. A particular attention has been given to the stack inspection mechanism for programming secure applications in the presence of code from various protection domains. Run-time checks of the call stack allow a method to obtain information about the code that (directly or indirectly) invoked it in order to make access control decisions. This mechanism is part of the security architecture of Java and the .NET Common Language Runtime. A central problem with stack inspection is to determine to what extent the *local* checks inserted into the code are sufficient to guarantee that a *global* security property is enforced. A further problem is how such verification can be carried out in an incremental fashion. Incremental analysis is important for avoiding re-analysis of library code every time it is used, and permits the library developer to reason about the code without knowing its context of deployment. We propose a technique for inferring interfaces for stack-inspecting libraries in the form of *secure calling context* for methods. By a secure calling context we mean a pre-condition on the call stack sufficient for guaranteeing that execution of the method will not violate a given global property. The technique is a constraint-based static program analysis implemented via fixed point iteration over an abstract domain of linear temporal logic properties [15].

5.2. Flow analysis of Java Card

Participant: Thomas Jensen.

The Java Card language is a trimmed down dialect of Java aimed at programming smart cards. We have developed a static analysis for verifying the proper protection of resources on a multi-application Java Card [17]. 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, local 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.

5.3. Logic Information Systems and Logic File System

Participants: Sébastien Ferré, Yoann Padioleau, Olivier Ridoux, Benjamin Sigonneau.

The Logic File System that implements parts of the Logic Information System concept [52] has been further developped to gain better performance and more services. On the performance side, the latest release of LISFS (autumn 2004) can handle more than 100,000 files with reasonable throughput. This is still less than what a standard file system does (e.g., EXT3), but it can handle real size data. On the service side, LISFS offer the same services applied to a set of files as applied to the set of parts of one or several files. Again, the second service is affordable for up to 100.000 parts in a file. The issue tackled by this second service is to manage coherently different read/write views of the same file. So doing, the hiatus between pushing everything in a single file, as say in log files, of scattering data in multiple files, as say in Java programs, no longer exists; all kinds of data layouts are treated the same way.

Further works is to yield still more performance, and incorporate other operations that have been defined for Logic Information Systems.

On the Logic Information System the current work is on its application to Software Engineering. The issue is to apply Knowledge Engineering to Software Engineering as a Formal Method. This is made possible by the very design of Logic Information Systems; it is based on logic. We have already applied these methods to the retrieval of software components based on their types, name structure, and documentation [28]. This shows that formal elements, like types, and informal elements, like documentation, can be handled in a single framework.

Further works is to cover more Software Engineering activities and formalisms, and to handle the formal relations that exist between Software Engineering objects. This is a fundamental issue for Logic Information System because its current design is based on unary predicates; it must be extended to handle binary predicates and more.

Olivier Ridoux has a collaboration with Jeanne Villaneau (U. Bretagne Sud) and Jean-Yves Antoine (U. Tours) on the understanding of spoken natural language [21]. As the concept of Logic Information System contains a logical formalization of a user-system dialog we will investigate its application to a man-machine dialog.

5.4. Certified static analyser

Keywords: *Program analysis*, *constraint solving*, *constructive logic*, *lattices*, *theorem proving*.

Participants: David Cachera, Thomas Jensen, David Pichardie.

We have formalised a constraint-based data flow analysis in the specification language of the *Coq* proof assistant [22]. Our approach relies on the definition of a dependent type of lattices together with a library of lattice functors for modular construction of complex abstract domains. The analysis is the defined by a collection of constraints on these complex lattices. Constraints are represented in a way that allows for both

efficient constraint resolution and correctness proof of the analysis with respect to an operational semantics. Proofs in *Coq* are constructive and correspond, via the Curry-Howard isomorphism, to programs in a functional language with a rich type system. The program extraction mechanism in *Coq* provides a tool for automatic translation of these proofs into a functional language with a simpler type system, namely *Ocaml*. We thus extract, from the proof of existence of a correct program analysis result, a static analyser that maps any given program to its static analysis. We have chosen to develop our methodology in the concrete setting of a flow analysis for Java Card byte code. Nevertheless, the library of lattices and the representation of constraints are defined in an analysis-independent fashion that provides a basis for a generic framework for proving and extracting static analysers in *Coq*.

5.5. Component fusion

Keywords: *Software components*, *composition*, *fusion*.

Participant: Stéphane Hong Tuan-Ha.

We have 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 modeled 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 technique, inspired by aspect-oriented programming, permits to keep separate the construction of component-based systems from scheduling and efficiency issues [23]. This work has been done in collabration with Pascal Fradet at Inria Rhône-Alpes.

5.6. Debugging for constraint logic programming and other languages

Keywords: *CLP(FD)*, *Debugging*, *formal trace format*, *trace analysis*.

Participant: Mireille Ducassé.

The work on debugging for constraint logic programming is a joint activity with the INRIA project CONTRAINTES from Rocquencourt in the context of OADYMPPAC, a French RNTL project (see module 6.3. Together with P. Deransart, M. Ducassé is the PhD supervisor of Ludovic Langevine [13].

Tracers give some insight of program executions: 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. Furthermore, each solver environment implements its own analysis and debugging tools which are not easy to port to another environment.

We have formally designed a generic trace format which can be specialized for dedicated constraint solvers. This enables debugging tools to be defined almost independently from finite domain solvers, and conversely, tracers to be built independently from these tools. This trace format has been successfully used to trace three different trace solvers. For all these solvers the same set of debugging tools has been plugged [20].

In order to efficiently analyze the produced trace, we have designed a *tracer driver* which drives the production and filtering of the execution traces according to the debugging demands. With this mechanism, only the necessary information is then produced and forwarded to the debugging tools [25][24]. The major advantage of this approach is that the trace format can be very rich, the analyses will only "pay" for what they need. The PhD thesis of Ludovic Langevine shows that the response times of the tracer driver with the tracer of Gnu-Prolog are consistent with the performance of the state-of-the-art debuggers with a much richer functionality [13].

A technical report also summarizes the lessons learnt while implementing a number of efficient trace query systems for C and various logic programming languages [30].

5.7. Intrusion détection

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

Participants: Mireille Ducassé, Jean-Philippe Pouzol.

The work on intrusion detection has been done in the context of the Dico RNTL project (see module 6.2). We particularly collaborated with France Telecom R &D of Caen and Supelec of Rennes. M. Ducassé supervises the theses of Benjamin Morin [14] and Elvis Tombini from France Telecom R &D of Caen with Hervé Debar and Ludovic Mé.

A possible defense against the abusive uses of a system is the *detection of intrusions*. The system activity is recorded in an audit trail which provides a sequence of events related to a trace of program execution. Audit trails are then analyzed in order to detect suspicious activity.

Current intrusion detection systems generate too many alerts. These alerts are imprecise and partial. Furthermore, they contain low level information. These alerts are therefore of limited interest for a human operator. Alert correlation is a promising technology to reduce the number of alerts, improve the diagnostic and provide a better vision of the security of the system in the case of an intrusion. We contributed to an overview of different alert correlation technologies which shows how these technologies can be applied to intrusion detection [16].

Intrusion detection can be performed with two methods: misuse detection and anomaly detection. Misuse detection recognizes attacks using a set of signatures while anomaly detection declares safe events that fit its model of behavior. Combining an "anomaly" and a "misuse" intrusion detection systems offers therefore the advantage of a refined classification of the monitored events into safe, intrusive or unqualified classes (ie notknown as an attack, but not recognize as safe either). We contributed to the definition of a framework to systematically reason about the combination of anomaly and misuse components. This framework applied to web servers led us to propose a serial architecture, using a drastic anomaly component with a sensitive misuse component. This architecture provides the operator with better qualification of the detection results, raises lower amount of false alarms and unqualified events [29].

We also studied a high-level specification language for signatures of attacks, named Sutekh. The aim of this language is to focus on the description of the signatures (ie. what combination of event is characteristic of an intrusion) and to hide operational details. We propose an algorithm based on the "parsing schemata" formalism. We made a prototype using the ELAN rewriting system developped at the LORIA [34], to perform an online analysis of system call traces in a Solaris environment. The preliminary results in terms of performance are not as promising as expected and require further optimizations [31].

5.8. Statistical Testing via Probabilistic Concurrent Constraint Programming

Keywords: *Statistical testing*, *probability distribution*, *structural testing*.

Participants: Matthieu Petit, Arnaud Gotlieb.

Statistical Testing techniques aim at selecting the test data according to a given probability distribution over the input domain. These techniques appear to be of particular interest when testing parts of the program under test that have a very low probability to be activated. As a part of these techniques, structural statistical

testing is based on a probabilistic generation according to some structural criteria based on control– or data–flows. The goal of structural statistical testing is to define a non-uniform probability distribution by solving a set of probabilistic constraints extracted from the program. Several theoretical and practical frameworks have been proposed to handle probabilistic constraints into constraint solving processes. We started to explore the possibility of using Probabilistic Constraint Logic Programming and proposed a first model to generate automatically test data for statistical testing [27][26]. Our goal is to apply this framework to the test of Java Card and J2ME applets.

5.9. Approximate Reachability Analysis and Cryptographic Protocol Verification

Keywords: *cryptographic protocol*, *reachability analysis*, *security protocols*, *term rewriting systems*, *verification*.

Participants: Thomas Genet, Valérie Viet Triem Tong.

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 that 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 [44] 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 successfully applied reachability analysis to the verification of the "view-only" protocol of the SmartRight digital rights management system developped by Thomson-Multimedia [45][54].

On a more theoretical point of view, we have refined the reachability analysis technique. We have proposed a unique algorithm to deal both with the exact and the approximated reachability analysis [18]. The proposed algorithm is able to produce an exact representation of infinite sets of reachable terms (for most of the known decidable classes of the litterature) and produce an over-approximation otherwise (for non decidable classes). This algorithm also extends the class of term rewriting systems to consider since it can handle conditional ones.

The extension to the conditional term rewriting systems will offer a more general and more expressive framework for modelling programs in general and cryptographic protocols in particular. On the other hand, computing exactly reachable terms now permits our protocol verification tool to discover attacks on flawed cryptographic protocols rather than only proving that there is no one on correct protocols. Our collaboration with Thomson showed that having both aspects (attack discovery and proof) in the *same verification technique* is crucial for convincing the protocol designer to rely on the proof when it is successful.

The algorithm dealing both with the exact and approximated reachability has been implemented in Timbuk 2.0 (see 4.3) and is part of the current distribution.

6. Contracts and Grants with Industry

6.1. The CASTLES RNTL project

Keywords: *Java Card*, *testing and certified analysis*.

Participants: Arnaud Gotlieb, Sandrine Gouraud, Thomas Jensen, Gerardo Schneider.

This project aims at defining an automated environment for the certification of a platform Java Card and the applets that are intended to be executed on it. This environment will be based on abstraction tools, automated proofs checkers, static analysis and software testing techniques. The originality of the proposed approach comes from the integration of these distincts tools into a single framework able to deal with the specification step, the development and the validation steps. Parts of this environment already exist and this project will alleviate the two remaining difficulties :

- the formal verification of the Java Card plateform requires a major effort due to the absence of automated verification techniques and mecanisms that allow modular and reusable proofs ;
- the applet validation process is based complex static analysis that must be justified with regards to the certification process.

The choice of Java Card, which is the open smart card reference language, maximizes the industrial benefits and provides a realistic application domain. However, the proposed solutions will overcome the context of Java Card and could be eventually adapted to other smart card platforms for small secured objects and mobile codes, such as Java or .NET.

The CASTLES project is funded by the French Ministry of Reserach as part of the RNTL funding scheme. It is a 3 years project and 4 partners are involved : the INRIA Everest project from Inria Sophia-Antipolis, the Lande project, Oberthur Card Systems and Alliance Qualité Logicielle. The works to do include :

- Package “SP0: Requirements in certification“ provides a precise analysis of needs in certification of the Java Card platform.
- Package “SP1: Java Card platform Validation” is concerned with techniques and tools for the validation of the Java Card platform will be defined. In particular, formal proof checkers and automatic test data generator will be proposed.
- Package “SP2: Static analysis of security properties”. This package will target the certification of Java Card applets. It will be based on the definition of formal security analysis for confidentiality and disponibility.

6.2. The DICO RNTL project

Keywords: *cooperative detection of intrusions*.

Participants: Mireille Ducassé, Jean-Philippe Pouzol.

The DICO project started on December 5, 2001 and ended on June 14,2004. The project proposes new techniques, on the one hand, to detect intrusions and, on the other hand, to reduce by correlation the number of generated alarms. This action benefited from a support of the French Ministry for Research, as partof the RNTL funding scheme. 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. The main outcome of the project in which Lande contributed is the implementation of a collection of nine intrusion detection systems which work at different levels, namely operating system, network and http servers [31].

6.3. The OADymPPaC RNTL project

Keywords: *debugging*, *logic programming with constraints*, *programming environment*, *visualization*.

Participant: Mireille Ducassé.

The OADymPPaC Project started on November 15, 2000 and ended on May 14, 2004. The project designed and experimented generic debugging and visualization techniques for logic programs with constraints. The aim is twofold: on the one hand, a better comprehension of the behavior of the solvers, on the other hand, the adaptation of generic techniques of visualization of information to the visual analysis of the traces produced by the dynamic phenomena. This action benefited 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 main outcome of the project in which Lande contributed is the definition of a generic trace format common to the various tools [32]. The project still maintains a page of information: http://contraintes.inria.fr/OADymPPaC/.

6.4. The MEFORSE collaborative research contract with France Télécom R&D

Keywords: *Java on mobile devices J2ME*, *cryptographic protocols*, *static analysis*.

Participants: Frédéric Besson, Thomas Genet, Thomas Jensen, Luka Leroux.

The Lande project has initiated a formalized collaboration with the France Télécom R&D team TAL/VVT based in Lannion. The collaboration is concerned with the modeling and analysis of software for telecommunication, in particular cryptographic protocols and Java (J2ME) applets written using the profile dedicated to mobile devices. The collaboration has so far lead to a list of features to verify on Java-enabled mobile telephones in order to ensure their security. We are notably interested in validating properties pertaining to the proper use of resources (eg. sending of SMS messages) for which we have developed a static analysis that allows to assert that a given applet will not use an unbounded amount of resources.

In another strand of the collaboration we analyse cryptographic protocols by over-approximating the protocol's and intruder's behavior. In general, the over-approximation is computable, whereas the exact behavior is not. To prove that there is no possible attack on the protocol we show that there is no attack on the over-approximation of its behavior. This leaves the problem of false positives: if the approximation contains an attack, it is not possible to say if it is a real attack or if it is due to the over-approximation. We thus work on attack reconstruction from the over-approximation of protocol's and intruder's behavior in order to discriminate between real and false attacks. We have already proposed a first algorithm which have been implemented and tested under the Timbuk library.

7. Other Grants and Activities

7.1. The ACI Sécurité Informatique project DISPO

Keywords: *Availability*, *aspects*, *software components*.

Participants: Thomas Jensen, Stéphane Hong Tuan-Ha.

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 is 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 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 (the OBASCO project), IRISA (Rennes), IRIT (Toulouse) and ENST-Bretagne.

7.2. The ACI Sécurité Informatique project V3F

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

Participant: Arnaud Gotlieb.

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.

7.3. Contract with the Brittany region: SACOL

Keywords: *Security*, *component*, *modularity*, *safety*, *weaving*.

Participants: Stéphane Hong Tuan-Ha, Thomas Jensen.

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 described in Section 5.5 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. This work is carried out in collaboration with Pascal Fradet from Inria Rhône-Alpes.

7.4. The ACI Sécurité Informatique project SATIN

Keywords: *cryptographic protocols*, *security protocols*, *verification*.

Participant: Thomas Genet.

The SATIN ACI (http://lifc.univ-fcomte.fr/~heampc/SATIN/) started on July 2004, for 3 years. SATIN means Security Analysis for Trusted Infrastructures and Network protocols. This project gathers some academic (Loria, LIFO, LIFC, Irisa) and industrial researchers (CEA-DAM and France Telecom R&D). The main goal of the project is to bring academic tools closer to industrial expectations. Indeed, there are more and more academic tools, based on process algebras and on rewriting techniques dedicated to protocol verification. Furthermore, several interesting security analysis results have recently been obtained in these directions, but some fundamental issues remain before these results can be applied to practical problems of industrial type: more efficient decision procedures and closer approximation results, taking into account the incidence of time, modeling imperfect cryptographic primitives and the notion of denial of service suitability to consider attacks based on them.

In this ACI, Timbuk is used as one of the verification back-end. For instance, researchers of LIFC use Timbuk to prove security properties on protocol specification in HLPSL format (High Level Protocol Specification Language). HLPSL has been designed in the European Project AVISS (see http://www.informatik.uni-freiburg.de/~softech/research/projects/aviss/ for details). During the ACI SATIN, Lande is going to strenghen several aspects of the Timbuk tool. First of all we plan to refine the system so that it can produce more precise trace information when it discovers an attack on a protocol. Then, we aim at implementing the completion algorithm for conditional term rewriting systems (see 5.9) so that it can handle more detailed protocol models and more specific protocols behaviors, like conditional protocols. Finally, since the tree automata built with Timbuk, representing the set of reachable terms, is used to prove properties on critical programs – security protocols – we would like to certify this result. We propose to build with Coq a checker proving that the tree automaton, produced by Timbuk, is complete w.r.t. reachable terms.

8. Dissemination

8.1. Conferences: program committees, organization, invitations

Mireille Ducassé has been the general chair and organizer of ICLP'04, the International Conference on Logic Programming 2004. She also served in the program committee. Arnaud Gotlieb acted as publicity chair for the same conference.

Mireille Ducassé has co-organized the First International Workshop on Teaching Logic Programming: TeachLP [10].

Arnaud Gotlieb served in the program committee of ISSRE'04, International Symposium on Software Reliability, and Engineering, and JFPLC'04, Journées Francophones de Programmation en Logique et par Contraintes.

Thomas Jensen served on the program committee for the International Static Analysis Symposium 2004 (SAS'04) and for the international Conference for Formal Engineering Methods (ICFEM'04).

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 [11]. Together with Jean-Louis Lanet he has written an overview article of formal methods for modeling smart card software [19].

8.2. Long-term visitors

Dave Schmidt, professor at the Department of Computer Science at Kansas State University visited the project in the period March-June 2004.

8.3. PhD theses defended

Benjamin Morin, PhD Insa of Rennes, on February 5th, 2004 [14], co-supervised by Mireille Ducassé.

Katell Alloy-Morin, PhD University of Rennes 1, on October 27th, 2004 [12], supervised by David Cachera.

Ludovic Langevine, PhD Insa of Rennes, on December 6th, 2004 [13], co-supervised by Mireille Ducassé.

8.4. PhD and Habilitation committees

Thomas Jensen was *rapporteur* on the PhD thesis of Vincent Simonnet (U. Paris 7) and on the habilitation thesis of Jean-Louis Lanet, (U. de la Méditerranée). He was president of the jury for the PhD of Alexandre Frey (Ecole des Mines de Paris) and has been member of the jury for the habillitation of Yves Le Traon (U. Rennes 1) and the PhD jury for Nestor Catano (U. Paris 7) and Rabéa Boulifa, (U. Nice).

Mireille Ducassé has been *opponent* to the PhD thesis of Peter Bunus (University of Linkoeping, Sweden).

Olivier Ridoux has been *rapporteur* on the PhD thesis of David Chemouil (U. Toulouse III), and *président* on the PhD theses of Ludovic Langevine (Insa de Rennes), Elena Leroux-Zinovieva and Benoît Gaudin (U. Rennes 1).

8.5. Teaching: university courses and summer schools

Mireille Ducassé teaches compilation, formal methods for software engineering (with "B") at Master 1 level of Insa. She also teaches software engineering at Master 2 level in collaboration with Arnaud Gotlieb. She is the president of the *commission de spécialistes* in computer science of Insa of Rennes.

Thomas Genet teaches formal methods for software engineering (with "B") and Cryptographic Protocols for M1 level (4th university year). He also teaches formal methods and proof assistants to M2R students (5th university year) in collaboration with Vlad Rusu (VERTECS project).

Arnaud Gotlieb teaches structural testing at the Research Master 2 level in collaboration with Thierry Jéron (VERTECS project) and Yves Le Traon (TRISKELL project). He also teaches at the 5th year of INSA Rennes in collaboration with Mireille Ducassé.

Thomas Jensen teaches semantics, type systems and static analysis at Master 2 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 Ecole de Mines de Nantes and Le Croisic and was organized together with the Inria project Obasco from Ecole de Mines de Nantes.

Olivier Ridoux is the head of IFSIC (Institut de Formation Supérieure en Informatique et Communication - the Computer Science department of U. Rennes 1). He teaches compilation, logic and constraint programming, as well as software engineering at the Master level of IFSIC.

9. 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", vol. 9, 2001, p. 217–250.

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

[3] M. DUCASSÉ. *Opium: An extendable trace analyser for Prolog*, in "Journal of Logic programming", vol. 39, 1999, p. 177-223.

[4] S. FERRÉ, O. RIDOUX. *Introduction to Logic Information Systems*, in "Elsevier J. Information Processing & Management", vol. 3, nº 40, 2004.

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

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

[7] 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, mai 1999, p. 89–103.

[8] Y. PADIOLEAU, O. RIDOUX. *A Logic File System*, in "USENIX Annual Technical Conference, General Track", 2003.

[9] O. RIDOUX. *Logic Information Systems for Logic Programmers*, in "ICLP — Int. Conf. Logic Programming", 2003.

Books and Monographs

[10] M. DUCASSÉ, U. NILSSON, D. SEIPEL (editors). *Proceedings of the First International Workshop on Teaching Logic Programming: TeachLP*, ISSN 1650-3686 (print), 1650-3740 (www), Linköping Electronic Conference Proceedings, Issue No. 12, September 2004, http://www.ep.liu.se/ecp/012/.

[11] M. HUISMAN, T. JENSEN (editors). *J. Logic and Algebraic Programming. Special issue on Formal Methods for Smart Cards, vol. 58(1-2)*, Elsevier, 2004.

Doctoral dissertations and Habilitation theses

[12] K. ALLOY-MORIN. *Vérification formelle dans le modèle polyédrique*, Ph. D. Thesis, Université de Rennes 1, Octobre 2004.

[13] L. LANGEVINE. *Observation de programmes avec contraintes et traces d'exécutions : une sémantique générique et une architecture d'analyse dynamique*, Ph. D. Thesis, INSA de Rennes, Décembre 2004.

[14] B. MORIN. *Corrélation d'alertes issues d'outils de détection d'intrusions avec prise en compte d'informations sur le système surveillé*, Ph. D. Thesis, INSA de Rennes, Février 2004.

Articles in referred journals and book chapters

[15] F. BESSON, T. DE GRENIER DE LATOUR, T. JENSEN. *Interfaces for stack inspection*, in "Journal of Functional Programming", to appear, 39 pp..

[16] H. DEBAR, B. MORIN, F. CUPPENS, F. AUTREL, L. MÉ, B. VIVINIS, S. BENFERHAT, M. DUCASSÉ, R. ORTALO. *Corrélation d'alertes en détection d'intrusions*, in "Technique et Science Informatiques", vol. 23, n° 3, 2004, p. 323–358.

[17] M. ELUARD, T. JENSEN. *Vérification du contrôle d'accès dans des cartes à puce multi-application*, in "Technique et Science Informatiques", vol. 23, n° 3, 2004, p. 323–358.

[18] G. FEUILLADE, T. GENET, V. VIET TRIEM TONG. *Reachability Analysis over Term Rewriting Systems*, in "Journal of Automated Reasoning", to appear, 49 pp..

[19] T. JENSEN, J.-L. LANET. *Modélisation et vérification dans les cartes à puce*, in "Revue d'électricité et de l'électronique", vol. 6–7, 2004, p. 89–94.

[20] L. LANGEVINE, P. DERANSART, M. DUCASSÉ. *A Generic Trace Schema for the Portability of CP(FD) Debugging Tools*, in "Recent advances in Constraint Programming", J. VANCZA, K. APT, F. FAGES, F. ROSSI, P. SZEREDI (editors)., Springer-Verlag, Lecture Notes in Artificial Intelligence 3010, 2004, p. 171–195.

[21] J. VILLANEAU, O. RIDOUX, J.-Y. ANTOINE. *LOGUS : un système formel de compréhension de l'oral spontané*, in "RIA — Revue d'Intelligence Artificielle, (à paraître)", 2004.

Publications in Conferences and Workshops

[22] D. CACHERA, T. JENSEN, D. PICHARDIE, V. RUSU. *Extracting a data flow analyser in constructive logic*, in "Proc. ESOP'04", Springer LNCS, n° 2986, 2004, p. 385 – 400.

[23] P. FRADET, S. HONG TUAN HA. *Network Fusion*, in "Programming Languages and Systems: Second Asian Symposium, APLAS 2004", W.-N. CHIN (editor)., vol. Springer LNCS vol. 3302, 2004, p. 21–40.

[24] L. LANGEVINE, M. DUCASSÉ. *A tracer driver to enable debugging, monitoring and visualization of CLP executions from a single tracer*, in "Proceedings of the International Conference on Logic Programming", B. DEMOEN, V. LIFSCHTITZ (editors)., Poster, Lecture Notes in Computer Science 3132, Springer-Verlag, September 2004, p. 462-463.

[25] L. LANGEVINE, M. DUCASSÉ. *Un pilote de traceur pour la PLC. Déboguer, auditer et visualiser une exécution avec un même traceur*, in "Actes des Journées Francophones de Programmation en Logique avec

Contraintes", F. MESNARD (editor)., HERMES Science Publications, Juin 2004, p. 19-36.

[26] M. PETIT, A. GOTLIEB. *An ongoing work on statistical structural testing via probabilistic concurrent constraint programming*, in "IEEE MODEVA-SIVOES Workshop of ISSRE'04", 2004.

[27] M. PETIT, A. GOTLIEB. *Probabilistic choice operators as global constraints : application to statistical software testing*, in "Poster presentation in ICLP'04", Springer LNCS, n° 3132, 2004, p. 471 – 472.

[28] B. SIGONNEAU, O. RIDOUX. *Indexation multiple et automatisée de composants logiciels orientés objet*, in "AFADL — Approches Formelles dans l'Assistance au Développement de Logiciels", 2004.

[29] E. TOMBINI, H. DEBAR, L. MÉ, M. DUCASSÉ. *A serial combination of anomaly and misuse IDSes applied to HTTP traffic*, in "Proceedings of the Annual Computer Security Applications Conference", D. THOMSEN, C. SCHUBA (editors)., December 2004.

Internal Reports

[30] M. DUCASSÉ, B. SIGONNEAU. *Building efficient tools to query execution traces*, Also Publication Interne IRISA 1638, Rapport de Recherche, n° RR-5280, INRIA, July 2004, http://www.inria.fr/rrrt/rr-5280.html.

[31] J.-P. POUZOL, S. BENFERHAT, H. DEBAR, M. DUCASSÉ, E. FAYOL, S. GOMBAULT, J. GOUBAULT-LARRECQ, Y. LAVICTOIRE, L. MÉ, L. NOÉ, J. OLIVAIN, E. TOTEL, B. VIVINIS. *Rapport de synthèse sur la création de sondes de détection d'intrusions*, 121 pages, Livrable du projet RNTL DICO, Juillet 2004.

[32] THE OADYMPPAC CONSORTIUM. *GENTRA4CP: Generic Trace Format for Constraint Programming*, Livrable du projet RNTL OADymPPaC, INRIA, July 2004, http://contraintes.inria.fr/OADymPPaC.

Bibliography in notes

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

[34] 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", vol. 4, Elsevier, 1996.

[35] 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, 1977, p. 238–252.

[36] S. FERRÉ, O. RIDOUX. *Introduction to Logic Information Systems*, in "Elsevier J. Information Processing & Management", vol. 3, n° 40, 2004.

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

[38] S. FERRÉ, O. RIDOUX. *A File System Based on Concept Analysis*, in "dood2000, 1st Int. Conf. Computational Logic, lnai 1861", Y. SAGIV (editor)., 2000.

[39] 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.

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

[41] S. FERRÉ, O. RIDOUX. *The Use of Associative Concepts in the Incremental Building of a Logical Context*, in "Int. Conf. Conceptual Structures", U. PRISS, D. CORBETT, G. ANGELOVA (editors)., LNCS 2393, Springer, 2002, p. 299–313.

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

[43] T. GENET. *Decidable Approximations of Sets of Descendants and Sets of Normal forms*, in "Proceedings 9th International Conference on Rewriting Techniques and Applications", Lecture Notes in Computer Science, vol. 1379, Springer-Verlag, 1998, p. 151–165.

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

[45] 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.

[46] 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", Lecture Notes in Artificial Intelligence, vol. 2250, Springer-Verlag, 2001, p. 691–702.

[47] P.-C. HEAM, Y. BOICHUT, O. KOUCHNARENKO, F. OEHL. *Improvements on the Genet and Klay Technique to Automatically Verify Security Protocols*, in "Proceedings of AVIS", 2004.

[48] D. E. KNUTH, P. B. BENDIX. *Simple word problems in universal algebras*, in "Computational Problems in Abstract Algebra, Oxford", J. LEECH (editor)., Pergamon Press, 1970, p. 263–297.

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

[50] F. OEHL, G. CÉCÉ, O. KOUCHNARENKO, D. SINCLAIR. *Automatic Approximation for the Verification of Cryptographic Protocols*, in "Proc. of FASE'03", vol. 2629, 2003, p. 34-48.

[51] 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.

[52] Y. PADIOLEAU, O. RIDOUX. *A Logic File System*, in "USENIX Annual Technical Conference, General Track", 2003.

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

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