INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Project-Team Lande

Logiciel : ANalyse et DEveloppement

Rennes

THEME SYM

Activity Report

2005

Table of contents

1. Team **1**
2. Overall Objectives **1**
2.1. Project overview 1
2.1.1. Research activities: 1
3. Scientific Foundations **2**
3.1. Static program analysis 2
3.1.1. Static program analysis 3
3.1.2. The structure of a static analysis 3
3.2. Debugging 4
3.3. Program testing 4
3.4. Logic information system 6
3.5. Reachability analysis over term rewriting systems 7
4. Software **8**
4.1. A Coq library of lattices 8
4.2. Logic File System and Camelis 8
4.3. Logfun: a Logic Functor Library 8
4.4. Timbuk: a Tree Automata Library 9
5. New Results **10**
5.1. Modular static program analysis 10
5.2. Certified static analyser 10
5.3. Resource-oriented analysis 11
5.4. Proving safety properties in the polyhedral model 11
5.5. Constraint-based test data generation for pointer programs 12
5.6. Using Constraint Handling Rules to generate test cases for the JCVM 12
5.7. Invariant refutation with constraint reasonning 12
5.8. Dynamic and static information flow analysis 12
5.9. Symbolic execution of floating-point computations 13
5.10. Debugging for constraint logic programming 13
5.11. Data mining and trace cross-checking 13
5.12. Logic Information Systems 14
6. Contracts and Grants with Industry **14**
6.1. The CASTLES RNTL project 14
6.2. Static analysis of Java Midlets with rewriting and reachability analysis 15
6.3. The MEFORSE collaborative research contract with France Télécom R&D 15
6.4. European FET-Integrated project MOBIUS 16
7. Other Grants and Activities **16**
7.1. The ACI Sécurité Informatique project DISPO 16
7.2. The ACI Sécurité Informatique project V3F 17
7.3. Contract with the Brittany region: SACOL 17
7.4. The ACI Sécurité Informatique project SATIN 18
8. Dissemination **18**
8.1. Conferences: program committees, organization, invitations 18
8.2. PhD theses defended 19
8.3. PhD and Habilitation committees 19
8.4. Teaching: university courses and summer schools 19
8.5. Administrative responsibilities 20
9. Bibliography **20**

1. Team

Head of project-team

Thomas Jensen [Research scientist, DR CNRS]

Administrative assistant

Lydie Letort [Administrative assistant, TR Inria]
Céline Ammoniaux [Administrative assistant, TR CNRS]

Inria personnel

Frédéric Besson [Research scientist, CR]
Arnaud Gotlieb [Research scientist, CR]

Université Rennes 1 Personnel

Olivier Ridoux [Professor]
Sébastien Ferré [Assistant Professor]
Thomas Genet [Assistant Professor]

Insa Personnel

Mireille Ducassé [Professor]

ENS Cachan Personnel

David Cachera [Assistant Professor]

PhD students

Yoann Padioleau [SRF, until September 2005]
Stéphane Hong Tuan-Ha [Inria-Region Bretagne]
David Pichardie [ENS Cachan, until September 2005]
Gurvan Le Guernic [MENRT grant]
Benjamin Sigonneau [MENRT grant]
Matthieu Petit [Region Bretagne grant]
Tristan Denmat [MENRT grant]
Luka Leroux [Inria-France Télécom R&D]
Peggy Cellier [MENRT grant]
Olivier Bedel [Region Bretagne grant]
Pascal Sotin [BDI CNRS-DGA]

Post-doctoral researchers

Sandrine Gouraud [Post-doc CNRS, until September 2005]
Gerardo Schneider [Post-doc CNRS, until March 2005]

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, to Java applets for mobile telephones 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 $f^n(\perp)$ where $\perp$ is the least element of the partial order.

3.1.1. Static program analysis

[36], [51] cover 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 [35]. 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. 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 [54] 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 deeper 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 automatization 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 project 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*.

Logic A language of formulas equipped with a semantics, an entailment relation, and possibly other operations like conjunction, disjunction, etc.

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 objects and their intentions. In formal concept analysis, intentions are usually sets of attributes, while in logic concept analysis they are formulas 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 [45] 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 [41]. 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 [40], [39], [38].

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 [42].

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 Logic 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 intentions cannot. However, it is possible to learn from already existing objects relations between intrinsic and extrinsic intentions [43].

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

Depending on the computing system modelled using rewriting, a deduction system, a function, some parallel processes or state transition systems, reachability (and unreachability) permit to achieve some verifications on the 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 $\mathcal{R}^*(E)$ of $\mathcal{R}$-reachable terms of an initial set of terms E. For those classes, $\mathcal{R}^*(E)$ 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 $s \neg \rightarrow^*_{\mathcal{R}} t$, we already have some results based on the over-approximation of the set of reachable terms [46], [47]. 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 [44] for computing exactly known decidable regular classes for $\mathcal{R}^*(E)$ 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 [50] *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. A Coq library of lattices

Keywords: *Constructive logic*, *Coq modules*, *lattices*.

Participant: David Pichardie.

We have developed a library of Coq modules for implementing lattices, the fundamental data structure of most static analysers. The motivation for this library was the development and extraction of certified static analysis in the Coq proof assistant—see Section 5.2. Using the abstract interpretation methodology, static analyses are specified as least solution of system of equations (inequations) on lattice structures. The library of Coq modules allows to construct complex and efficient lattices by combination of functors and base lattices. The lattice signature possesses a parameter which ensure termination of a generic fixed-point solver. The delicate problem of termination of fixpoint iterations is hence dealt with once and for all when building a lattice as a combination of the different lattice functors.

4.2. Logic File System and Camelis

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

Participants: Yoann Padioleau, Olivier Ridoux [contact point], Sébastien Ferré [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 2005 version can handle contexts of more than 100,000 files.

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

CAMELIS is another implementation of logic information system. Unlike LISFS, it works entirely at the user level, which means it cannot improve existing applications. The counterpart is that it makes it available on most platforms, and it is equipped with a dedicated interface that enables us to implement all the facets of logic information systems.

CAMELIS is distributed under the GNU General Public License and is freely available at http://www.irisa.fr/lande/ferre/camelis/.

4.3. Logfun: a Logic Functor Library

Keywords: *Logics*, *data structures*, *functors*.

Participant: Sébastien Ferré [contact point].

LOGFUN is a library of logic functors implemented as OCAML functors. It enables people to build logics (in the sense given in foundations) that are dedicated to particular applications, without requiring them to be logic experts. Logic functors can be seen as primitive types and type constructors, from which arbitrary complex logics can be designed. The generation of a logic from the composition of logic functors includes the generation of a theorem prover, as well as properties about this theorem prover (e.g., consistency, completeness).

LOGFUN is used in CAMELIS applications for designing and building their logic, and can also be used to build logic plugins for LISFS. LOGFUN is distributed under the GNU General Public License and is freely available at http://www.irisa.fr/lande/ferre/logfun/.

4.4. Timbuk: a Tree Automata Library

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

Participant: Thomas Genet.

Timbuk [47] 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 has been used for a case study done with Thomson-Multimedia for cryptographic protocol verification.

Timbuk is also used by 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) [53], [52]. Pierre-Cyrille Heam, Yohan Boichut and Olga Kouchnarenko of the Cassis Inria project use Timbuk as a verification back-end for verification of protocols [48] 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 [14].

5.2. Certified static analyser

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

Participants: David Cachera, Thomas Jensen, David Pichardie, Gerardo Schneider.

We have developed a methodology and a tool to produce certified static analysers from their formal specification in constructive logic. The purpose of this is to use the same formal definition of an analysis when proving its correctness and when deriving its implementation. For this, we rely on the theory of abstract interpretation, where static analyses are specified as an approximated semantics of a program. We use the Coq proof assistant for conducting the proofs of correctness and existence of solutions. This later allows for extracting a Caml implementation of the analyser from the computational content of the proofs [16].

We have presented a theoretical framework based on abstract interpretation allowing for the formal development of a broad range of static analyses. A Coq library for the modular construction of lattices is then proposed. Complex proofs for termination of iterative fix-point computations can hence be constructed by simple composition of lattice functors.

This formalism has been used to develop several certified analyses for Java bytecode-like languages, namely

- an interval-based analysis for the verification of array bounds;
- an interprocedural data flow analysis that uses the library of lattice functors together with an intermediate representation for constraints; this constraint representation allows for both efficient constraint resolution and correctness proof of the analysis;
- a reference analysis for bytecode programs, where references are abstracted by their creation point;
- a memory usage analysis for embedded systems [19] see Section5.3.

5.3. Resource-oriented analysis

Keywords: *Memory usage, aspect-oriented programming, availability, quantitative program analysis*.

Participants: David Cachera, Thomas Jensen, Pascal Sotin, Stéphane Hong Tuan-Ha.

A program's resources usage (time, memory or energy) is a valuable information about that program, even more when these resources are limited, as in the case of embedded software. There exist numerous methods for estimating these consumptions, going from monitoring executions to the exact computation of the complexity of the program, passing by techniques for determining Worst Case Execution Time.

We have developed a memory usage analysis for languages in the style of Java byte code. The algorithm verifies that a program executes in bounded memory. The algorithm is destined to be used in the development process of applets and for enhanced byte code verification on embedded devices. We have therefore aimed at a low-complexity algorithm derived from a loop detection algorithm for control flow graphs. The algorithm works at two levels: an interprocedural level to detect mutually dependent methods, and an intraprocedural level to detect method calls performed within loops.

The algorithm itself is a loop-detection algorithm for inter-procedural control-flow graphs. It is expressed as a constraint-based static analysis of the program over simple lattices. This provides a link with abstract interpretation that allows to state and prove formally the correctness of the analysis with respect to an operational semantics of the program. The certification is based on an abstract interpretation framework implemented in the Coq proof assistant which has been used to provide a complete formalisation and formal verification of all correctness proofs—see Section 5.2.

We have proposed a technique inspired by Di Piero and Wicklicky's Quantitative Abstract Interpretation for computing quantitative non-functional properties of a program's behaviour. The particular property we are interested in in this case study is estimating the number of cache misses in a computation described by a Java byte code program [20]. The analysis is based on a quantitative semantics for a subset of the Java language for Java Card, where we model the data cache impact on the execution cost. We then give ways to abstract this semantics, to allow for an effective computation of an over-approximation of the real costs. This abstraction offers promising albeit preliminary results when the semantics describes the worst case cache misses.

Finally, we have studied the use of aspect-oriented programming for resource management with the aim of improving resource availability. In this approach, aspects specify time limits or orderings in the allocation of resources and can be seen as the specification of an availability policy. The approach relies on timed automata to specify services and aspects. This allows us to implement weaving as an automata product and to use model-checking tools to verify that aspects enforce the required availability properties [24] (joint work with Pascal Fradet from Inria Rhône-Alpes).

5.4. Proving safety properties in the polyhedral model

Keywords: *Polyhedral model, affine recurrence equations, parameterised systems*.

Participant: David Cachera.

The polyhedral model provides a unified framework for expressing hardware and software parts of regular systems. Systems are described in a generic manner through the use of symbolic parameters. We have proposed a combination of heuristic methods to prove properties of control signals for regular systems defined by means of affine recurrence equations. We benefit from the intrinsic regularity of the underlying polyhedral model to handle parameterized systems in a symbolic way. Our techniques apply to safety properties. The general proof process consists in an iteration that alternates two heuristics. We are able to identify the cases when this iteration will stop in a finite number of steps. We then have extended our proof system with the detection of pseudo-pipelines, that are particular patterns in the variable definitions generalizing the notion of pipeline. The combination of pseudo-pipeline detection with the use of a simple widening operator greatly improves the effectiveness of our proof techniques. These techniques have been implemented in a high level synthesis environment based on the polyhedral model (MMAlpha) [17].

5.5. Constraint-based test data generation for pointer programs

Keywords: *Constraint-based testing*, *pointer programs*, *structural testing*.

Participants: Arnaud Gotlieb, Tristan Denmat.

Constraint-based test data generation (CBT) exploits constraint satisfaction techniques to generate test data able to kill a given mutant or to reach a selected branch in a program. When pointer variables are present in the program, aliasing problems may arise and may lead to the failure of current CBT approaches. In our work, we propose a CBT method that exploits the results of an intraprocedural points-to analysis and provides two specific constraint combinators for automatically generating test data able to reach a selected branch [26]. Our approach correctly handles multi-levels stack-directed pointers that are mainly used in real-time control systems. The method has been fully implemented in the test data generation tool INKA and first experiences in applying it to a variety of existing programs are promising [25].

5.6. Using Constraint Handling Rules to generate test cases for the JCVM

Keywords: *CHR*, *Java Card Virtual Machine*, *Testing the Java Card platform*, *structural testing*.

Participants: Sandrine Gouraud, Arnaud Gotlieb.

Automated functional testing consists in deriving test cases from the specification model of a program to detect faults within an implementation. In our work, we investigate using Constraint Handling Rules (CHRs) to automate the test cases generation process of functional testing. In the context of the CASTLES project, our case study is a formal model of the Java Card Virtual Machine (JCVM) written in a sub-language of the Coq proof assistant. We have defined an automated translation from this formal model into CHRs and have proposed to generate test cases for each bytecode definition of the JCVM [27]. Using CHRs allows to faithfully model the formally specified operational semantics of the JCVM. The approach has been implemented in Eclipse Prolog and a full set of test cases have been generated for testing the JCVM.

5.7. Invariant refutation with constraint reasonning

Keywords: *Constraint-based testing*, *disproving*, *likely invariants*, *proving*, *structural testing*.

Participants: Tristan Denmat, Arnaud Gotlieb, Mireille Ducassé.

In the context of software validation and verification, program invariants are properties that hold for every execution of the program and that can be used to detect program faults in the early stages of program development. Unfortunately, inferring accurate invariants is challenging as it is required to perform a global analysis of the program. Recent work suggest to infer *likely*-only invariants [37]. A likely invariant is a property that holds for some executions but is not guaranteed to hold for all executions. In our work, we investigate the challenging problem of automatically verifying that likely invariants are actual invariants, which is the well-known drawback of this technique. We have developed a constraint-based framework that is able, unlike other approaches, to both prove or disprove likely invariants [22]. The framework is based on constraint reasonning techniques in the context of dynamic program analysis. Very first experimental results suggest to focus on improving the unsatisfiability detection capacities of classical finite domains constraint solvers. We currently explore the use of polyhedra libraries to perform such detection at runtime.

5.8. Dynamic and static information flow analysis

Keywords: *Information flow*, *dependency*, *dynamic analysis*, *security levels*.

Participants: Gurvan Le Guernic, Thomas Jensen.

A standard way of formalising confidentiality is via the notions of information flow and non-interference. We have developed an information flow monitoring mechanism for sequential programs. The monitor executes a program on standard data that are tagged with labels indicating their security level. The originality of the approach is that we formalize the monitoring mechanism as a big-step operational semantics that integrates a static information flow analysis to gather information flow properties of non-executed branches of the program. This essentially shows how to mix static and dynamic non-interference analysis. Using the information flow monitoring mechanism, it is then possible to partition the set of all executions in two sets. The first one contains executions which *are safe* and the other one contains executions which *may be unsafe*. Based on this information, we show that, by resetting the value of some output variables, it is possible to alter the behavior of executions belonging to the second set in order to ensure the confidentiality of secret data [30].

5.9. Symbolic execution of floating-point computations

Keywords: *Symbolic execution, floating-point numbers*.

Participant: Arnaud Gotlieb.

Symbolic execution is a classical program testing technique which evaluates a selected control flow path with symbolic input data. A constraint solver can be used to enforce the satisfiability of the extracted path conditions as well as to derive test data. When path conditions contain floating-point computations, a common strategy is to use a constraint solver over the rationals or the reals. Unfortunately, even in a fully IEEE-754 compliant environment, this leads not only to approximations but also can compromise correctness: a path can be labelled as infeasible although there exists floating-point input data that satisfy it, or labeled as feasible when no floating-point input data can satisfy it. In our work, we address the peculiarities of the symbolic execution of program with floating-point numbers. Issues in the symbolic execution of this kind of programs are carefully examined and a constraint solver that supports constraints over floating-point numbers has been described. The overall approach has been implemented in a tool named FPSE (Floating-Point Symbolic Execution). Preliminary experimental results demonstrate the value of our approach [15].

5.10. Debugging for constraint logic programming

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

Participant: Mireille Ducassé.

Tracers provide users with useful information about program executions. We propose a “tracer driver” which from a single tracer provides a powerful front-end for multiple dynamic analysis tools while limiting the overhead of the trace generation. The tracer driver can be used both synchronously and asynchronously. The relevant execution events are specified by flexible event patterns and a large variety of trace data can be given either systematically or “on demand”. The proposed tracer driver has been designed and experimented in the context of constraint logic programming, within GNU-Prolog. Its principles are, however, independent of the traced programming language. Experimental measures show that the flexibility and power of the described architecture are also the basis of reasonable performances [28], [34] and leads to good performance in the context of constraint logic programming, where a large variety of tools exists and the trace is potentially huge [29]. This is a joint work with Ludovic Langevine at the Swedish Institute for Computer Science (SICS).

5.11. Data mining and trace cross-checking

Keywords: *Debugging, data mining, trace analysis*.

Participants: Peggy Cellier, Tristan Denmat, Mireille Ducassé, Sébastien Ferré, Olivier Ridoux.

The current trend in debugging and testing is to cross-check information collected during several executions. Jones *et al. [49]*, for example, propose to use the instruction coverage of passing and failing runs in order to visualize suspicious statements. This seems promising but lacks a formal justification. We show that the method of Jones *et al.*, can be re-interpreted as a data mining procedure. More particularly, the suspicion indicator they define can be rephrased in terms of well-known metrics of the data-mining domain. These metrics characterize *association rules* between data. With this formal framework we are able to explain limitations of the above indicator. Three significant hypotheses were implicit in the original work: 1) there exists at least one statement that can be considered as faulty ; 2) the values of the suspicion indicator for different statements should be independent from each others; 3) executing a faulty statement leads most of the time to a failure. We show that these hypotheses are hard to fulfill and that the link between the indicator and the correctness of a statement is not straightforward. The underlying idea of association rules is, nevertheless, still promising, and our conclusion emphasizes some possible tracks for improvement [21], [33].

5.12. Logic Information Systems

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

On the foundation side, Logic Concept Analysis has been extended by qualified relations between objects [23]. The querying language has been extended so as to search for objects being in some kind of relation with some kind of other objects, and recursively. This means we can specify paths between objects. This extension led to a redefinition of the notions of extent and intent, which are the basis for Concept analysis. The new theory obtained as a result retains all good properties of Logic concept analysis, which is important as Logic information systems depend on them. In addition to the conceptual navigation that already existed, it is now also possible to navigate through relations, i.e., from object sets to related object sets (e.g., from old men to their children in a genealogical context).

On the implementation side, a full-fledged system has been designed [31], [32] that implements the core operations of logic information systems (i.e., querying, navigation, and updating). An interesting feature is that it implements these operations at the file system level as well as the file level. This allows navigating *inside* files. This is especially useful for large files like some data-bases and for complex files like programs. The implementation of other operations of logic information systems is in order. More specifically, the prototype implementations of relations and of extraction of association rules are under test.

On the application side, the storage and retrieval of software components in a logic information system has been developed [18]. It provides an efficient interface to a component repository in which the different facets of the description of components are integrated in a single navigation scheme. At the present stage, the descriptions are unary, i.e., they do not relate several components. We plan to exploit the newly defined relations in logic information systems to handle n-ary description, that relate several components (e.g., call graph). This opens the road to navigating into complex relational descriptions like UML schemas.

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 distinct 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 platform requires a major effort due to the absence of automated verification techniques and mechanisms 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 are not limited to 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 Research 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—see 5.6.
- Package "SP2: Static analysis of security properties". This package targets the certification of Java Card applets. It is based on the definition of formal security analysis for confidentiality and disponibility—for results see 5.3.

6.2. Static analysis of Java Midlets with rewriting and reachability analysis

Keywords: *Java MIDP, Reachability Analysis, Term Rewriting*.

Participants: Thomas Jensen, Thomas Genet, Luka Leroux.

This contract with France Telecom R & D started on October 1, 2004, for 3 years. The objective is to prove security properties on Java Midlets to be downloaded and executed on a cell phone. The first goal is to extract the flow graph of the graphical interface (i.e. flow graph between the screens) of the midlet directly from the bytecode. Then, the security property in question is *e.g.*, whether critical methods of the Java MIDP library are called under some restrictions. A typical property to prove is that every sent message has been authorized by the user, i.e. every call to the message sending JAVA MIDP method should be done after a screen where the user has to confirm that he agrees with the sending of a message.

Another originality of this project consists in the way to tackle this goal. Since the specification of the JAVA MIDP library is constantly evolving, it is impossible to define once for all a static analyser able to deal with the analysis described above. We thus aim at taking advantage of the reachability analysis technique over term rewriting systems (see module 3.5) to define an analyser that can easily evolve with MIDP. The idea is to specify the semantics of the Java virtual machine, the semantics of MIDP and of a given midlet using term rewriting systems and then to use the automatic approximation of term rewriting systems to over-approximate the flow graph of the midlet. Then, for making the analyser evolve with MIDP it should be enough to adapt the term rewriting system describing the semantics of MIDP. The main challenges of this project are, first to define those semantics into term rewriting in a simple and usable way, second to express the usual static analysis achieved on Java by means of approximations on term rewriting systems and last to produce a complete analyser efficient enough to prove properties on real midlets.

6.3. 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.

6.4. European FET-Integrated project MOBIUS

Keywords: *Java, Proof Carrying Code, Security, smart phones, static analysis*.

Participants: Thomas Jensen, Frédéric Besson, Tiphaine Turpin, Guillaume Dufay.

Mobius (IST-15905) is an Integrated Project launched under the FET Global Computing Proactive Initiative. The project has started on September 1st 2005 for 48 months and involves 16 partners. The goal of this project is to develop the technology for establishing trust and security for mobile devices using the Proof Carrying Code (PCC) paradigm. Proof Carrying Code is a technique for downloading mobile code on a host machine while ensuring that the code adheres to the host's security policy. The basic idea is that the code producer sends the code with a formal proof that the code is secure. Upon reception of the code, the receiver uses a simple and fast proof validator to check, with certainty, that the proof is valid and hence the untrusted code is safe to execute.

In this project, we participate in the specification of security requirements and resource policies to be studied throughout the project. In parallel, we develop resource-aware static analyses to enforce the proposed resource policies.

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 aims 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: École 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 techniques used 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 has already defined the components, their interface and their fusion. The current phase is to design modular analyses and transformations in order to enforce security properties on such networks—see Section 5.3 . 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. During the ACI SATIN, Lande is going to strengthen 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 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

Thomas Jensen was program co-chair for the ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE'05) [11] and was on the program committee for the International Static Analysis Symposium 2005 (SAS'05), for the international Conference for Formal Engineering Methods (ICFEM'05) and for the 18th IEEE Computer Science Foundations Workshop (CSFW'18). He is on the editorial board of the French journal Technique et Science Informatiques.

Mireille Ducassé was on the program committees of AADEBUG'05 (International Workshop on Automated Debugging), JFPC'05 (Journées Francophones de Programmation par Contraintes), WLPE'05 (International Workshop on Logic-based Programming Environments) and PADL'06 (Practical Applications of Declarative Languages).

Mireille Ducassé has visited the Georgia Technical University in the USA, to present the results on data mining and cross-checking of execution traces. She has been invited to give a presentation at the Dagstuhl seminar "Beyond Program Slicing" organized by D. Binkley (Loyola College - Baltimore, US), M. Harman (King's College London, GB), J. Krinke (FernUniversität in Hagen, D), November 2005.

Frédéric Besson was on the program committee for the ETAPS workshop Bytecode'05.

Olivier Ridoux was on the program committee for the International Conference on Conceptual Structure 2005 (ICCS'2005).

8.2. PhD theses defended

Yoann Padioleau, *Logic File System, un système de fichier basé sur la logique*, PhD University of Rennes 1, on February 17th, 2005 [12].

David Pichardie: *Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés*, PhD University of Rennes 1, on December 6th, 2005 [13]

8.3. PhD and Habilitation committees

Mireille Ducassé has reported on the Master thesis of Ian Maclarty on *Practical Declarative Debugging of Mercury Programs*, Zoltan Somogyi supervisor, University of Melbourne, Australia, November 2005. She was president of the jury of the thesis of Elisa Fromont on *Learning rules from multi-source data using inductive logic programming: application to cardiac arrythmia recognition*, Marie-Odile Cordier and René Quiniou supervisors, Rennes, December 2005.

Thomas Genet was examiner on the PhD theses of Julie Vuotto (U. d'Orléans) and of Yu Zhang (École Normale Supérieure Cachan).

Thomas Jensen was *rapporteur* on the PhD thesis of Massimo Bartoletti (University of Pisa), Alexandre Miège (École Nat. Sup. Telecommunication) and Felip Luna de Agiula (University of Nice Sophia-Antipolis).

8.4. 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.

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 M2 students in collaboration with Vlad Rusu (VERTECS project).

Sébastien Ferré teaches software engineering at M1 and L2 level and functional programming at L1 level. He also teaches graph algorithmics and sequence algorithmics for bioinformatics at M1 level.

Arnaud Gotlieb teaches structural testing at M2 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.

Olivier Ridoux teaches compilation, logic and constraint programming, as well as software engineering at the Master level of IFSIC.

Thomas Jensen teaches semantics, type systems and static analysis at Master 2 level in collaboration with Bertrand Jeannet (VERTECS project). In addition, he has taught a course n Formal Methods for Java Card software at the Smart Card summer University, held in Sophia-Antipolis in September.

Thomas Jensen is scientific leader of the *École Jeunes Chercheurs en Programmation*, an annual summer school for graduate students on programming languages and verification, organized under the auspices of the CNRS GdR ALP. This year's event was organised by the Lande project and took place at Dinard and at Irisa. The school attracted 40 participants for two weeks during June.

8.5. Administrative responsibilities

Mireille Ducassé is the president of the *commission de spécialistes* in computer science at the Insa of Rennes

Olivier Ridoux is the head of IFSIC (Institut de Formation Supérieure en Informatique et Communication - the Computer Science department of U. Rennes 1).

Thomas Jensen is vice-president of the Scientific Committee (*comité des projets*) at Irisa.

Thomas Jensen was chairman of the hiring committee for junior researchers at Inria Rennes.

9. Bibliography

Major publications by the team in recent years

[1] A. BANERJEE, T. JENSEN. *Control-flow analysis with rank-2 intersection types*, in "Mathematical Structures in Computer Science", vol. 13, n° 1, 2003, p. 87–124.

[2] F. BESSON, T. DE GRENIER DE LATOUR, T. JENSEN. *Interfaces for stack inspection*, in "Journal of Functional Programming", vol. 15, n° 2, 2005, p. 179–217.

[3] 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.

[4] D. CACHERA, T. JENSEN, D. PICHARDIE, V. RUSU. *Extracting a Data Flow Analyser in Constructive Logic*, in "Theoretical Computer Science", vol. 342, n° 1, 2005, p. 56–78.

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

[6] S. FERRÉ, O. RIDOUX. *A Framework for Developing Embeddable Customized Logics*, in "Int. Work. Logic-based Program Synthesis and Transformation", A. PETTOROSSI (editor). , LNCS 2372, Springer, 2001, p. 191–215.

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

[8] G. FEUILLADE, T. GENET, V. VIET TRIEM TONG. *Reachability Analysis over Term Rewriting Systems*, in "Journal of Automated Reasoning", vol. 33, n° 3–4, 2004, p. 341–383.

[9] T. GENET, F. KLAY. *Rewriting for Cryptographic Protocol Verification*, in "Proc. of the 17th International Conference on Automated Deduction", LNAI, vol. 1831, Springer-Verlag, 2000.

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

Books and Monographs

[11] M. ERNST, T. JENSEN (editors). *Proc. of 6th ACM SIGSOFT-SIGPLAN Workshop on Program Analysis for Software Tools and Enginnering*, ACM, September 2005.

Doctoral dissertations and Habilitation theses

[12] Y. PADIOLEAU. *Logic File System, un système de fichier basé sur la logique*, Ph. D. Thesis, Université de Rennes 1, February 2005.

[13] D. PICHARDIE. *Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certiés*, Ph. D. Thesis, Université Rennes 1, Rennes, France, dec 2005.

Articles in refereed journals and book chapters

[14] F. BESSON, T. DE GRENIER DE LATOUR, T. JENSEN. *Interfaces for stack inspection*, in "Journal of Functional Programming", vol. 15, n° 2, 2005, p. 179–217.

[15] B. BOTELLA, A. GOTLIEB, C. MICHEL. *Symbolic execution of floating-point computations*, in "The Software Testing, Verification and Reliability journal", to appear.

[16] D. CACHERA, T. JENSEN, D. PICHARDIE, V. RUSU. *Extracting a Data Flow Analyser in Constructive Logic*, in "Theoretical Computer Science", vol. 342, n° 1, 2005, p. 56–78.

[17] D. CACHERA, K. MORIN-ALLORY. *Verification of safety properties for parameterized regular systems*, in "Trans. on Embedded Computing Systems", vol. 4, n° 2, 2005, p. 228–266.

[18] B. SIGONNEAU, O. RIDOUX. *Indexation multiple et automatisée de composants logiciels*, in "Technique et Science Informatiques", to appear.

Publications in Conferences and Workshops

[19] D. CACHERA, T. JENSEN, D. PICHARDIE, G. SCHNEIDER. *Certified Memory Usage Analysis*, in "Proc. of 13th International Symposium on Formal Methods (FM'05)", LNCS, Springer-Verlag, 2005.

[20] D. CACHERA, T. JENSEN, P. SOTIN. *Estimating Cache Misses with Semi-Modules and Quantitative Abstraction*, in "Proc. of IST-APPSEM II Workshop on Applied Semantics", 2005.

[21] T. DENMAT, M. DUCASSÉ, O. RIDOUX. *Data mining and cross-checking of execution traces. A re-interpretation of Jones, Harrold and Stasko test information visualization*, in "Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering", T. ELLMAN, A. ZISMAN (editors). , 4 pages, short paper, ACM Press, November 2005.

[22] T. DENMAT, A. GOTLIEB, M. DUCASSÉ. *Proving or Disproving Likely Invariants with Constraint Reasoning*, in "Proc. of the 15th Workshop on Logic-based Method for Programming Environments, Sitges, SPAIN", A. SEREBRENIK (editor). , Satelite event of International Conference on Logic Programming (ICLP'2005). Published in Computer Research Repository cs.SE/0508108, October 2005, http://arxiv.org/abs/cs.pl/0508108.

[23] S. FERRÉ, O. RIDOUX, B. SIGONNEAU. *Arbitrary Relations in Formal Concept Analysis and Logical Information Systems*, in "ICCS", LNCS 3596, Springer, 2005, p. 166-180.

[24] P. Fradet, S. H. T. Ha. *Systèmes de gestion de ressources et aspects de disponibilité*, in "2ème Journée Francophone sur le Développement de Logiciels par Aspects (JFDLPA 2005), Lille, France", 2005.

[25] A. Gotlieb, T. Denmat, B. Botella. *Constraint-based test data generation in the presence of stack-directed pointers*, in "20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05), Long Beach, CA, USA", 4 pages, short paper, Nov. 2005.

[26] A. Gotlieb, T. Denmat, B. Botella. *Goal-oriented test data generation for programs with pointer variables*, in "29th IEEE Annual International Computer Software and Applications Conference (COMPSAC'05), Edinburh, Scotland", 6 pages, July 2005, p. 449-454.

[27] S. Gouraud, A. Gotlieb. *Using CHRs to generate test cases for the JCVM*, in "Eighth International Symposium on Practical Aspects of Declarative Languages, PADL 06, Charleston, South Carolina", January 2006.

[28] L. Langevine, M. Ducassé. *A Tracer Driver for Hybrid Execution Analyses*, in "Proc. of the 6th Automated Debugging Symposium", ACM Press, September 2005, http://doi.acm.org/10.1145/1085130.1085149.

[29] L. Langevine, M. Ducassé. *A Tracer Driver for Versatile Dynamic Analyses of Constraint Logic Programs*, in "Proc. of the 15th Workshop on Logic-based Method for Programming Environments, Sitges, SPAIN", A. Serebrenik (editor). , Satelite event of International Conference on Logic Programming (ICLP'2005). Published in Computer Research Repository cs.SE/0508105, October 2005, http://arxiv.org/abs/cs.pl/0508105.

[30] G. Le Guernic and T. Jensen. *Monitoring Information Flow*, in "Workshop on Foundations of Computer Security", A. Sabelfeld (editor). , 2005.

[31] Y. Padioleau, O. Ridoux. *A Parts-of-File File System*, in "USENIX Annual Technical Conference, General Track (Short Paper)", 2005, http://www.usenix.org/events/usenix05/tech/general/padioleau.html.

[32] Y. Padioleau, B. Sigonneau, O. Ridoux, S. Ferré. *LISFS: a Logical Information System as a File System*, in "Bases de données avancées", V. Benzaken (editor). , Université de Rennes 1, October 2005, p. 393–398.

Internal Reports

[33] T. Denmat, M. Ducassé, O. Ridoux. *Data Mining and Cross-checking of Execution Traces. A re-interpretation of Jones, Harrold and Stasko test information visualization (Long version)*, Also Publication Interne IRISA PI-1743, Research Report, INRIA, August 2005, http://www.inria.fr/rrrt/rr-5661.html.

[34] L. Langevine, M. Ducassé. *A Tracer Driver to Enable Concurrent Dynamic Analyses*, Research Report, INRIA, June 2005, http://www.inria.fr/rrrt/rr-5611.html.

Bibliography in notes

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

[36] 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.

[37] M. D. ERNST, J. COCKRELL, W. G. GRISWOLD, D. NOTKIN. *Dynamically Discovering Likely Program Invariants to Support Program Evolution.*, in "Transactions on Software Engineering", vol. 27, n^o 2, 2001, p. 99-123.

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

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

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

[41] 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.

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

[43] 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.

[44] G. FEUILLADE, T. GENET, V. VIET TRIEM TONG. *Reachability Analysis over Term Rewriting Systems*, in "Journal of Automated Reasoning", vol. 33, n^o 3–4, 2004, p. 341–383.

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

[46] T. GENET. *Decidable Approximations of Sets of Descendants and Sets of Normal forms*, in "Proc. 9th International Conference on Rewriting Techniques and Applications", LNCS, vol. 1379, Springer-Verlag, 1998, p. 151–165.

[47] T. GENET, V. VIET TRIEM TONG. *Reachability Analysis of Term Rewriting Systems with Timbuk*, in "Proc. of the 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning", LNAI, vol. 2250, Springer-Verlag, 2001, p. 691–702.

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

[49] J. A. JONES, M. J. HARROLD, J. STASKO. *Visualization of test information to assist fault localization*, in "Proc. of the 24th International Conference on Software Engineering", ACM Press, 2002, p. 467–477.

[50] 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.

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

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

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

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