Acronyme	**Certo**		
Titre du projet en français	Certification des outils de vérification de logiciels		
Titre du projet en anglais	Certification of software verification tools		
CSD principale	1		
CSD secondaire ***(si interdisciplinarité)***			
Aide totale demandée	117765 €	**Durée du projet**	36 mois

SOMMAIRE

Table of Contents

Contexte et positionnement du projet /Context and positioning of the proposal…1
From low confidence tools to high confidence results in software verification….1
Description scientifique et technique /Scientific and technical description……3
État de l'art /Background, state of art……3
Objectifs et caractère ambitieux/novateur du projet /Rationale highlighting the originality and novelty of the proposal……6
Programme scientifique et technique, organisation du projet / Scientific and technical programme, project management……7
Programme scientifique et structuration du projet / Scientific programme, specific aims of the proposal……7
Coordination du projet /Project management……12
Organisation du projet /Consortium organisation and description……16
Description, adéquation et complémentarité des participants / Relevance and complementarity of the partners within the consortium……16
Justification scientifique des moyens demandés / Scientific justification of requested budget……18

Contexte et positionnement du projet / Context and positioning of the proposal

From low confidence tools to high confidence results in software verification

Computer-aided verification technology aims at proving the correctness of software or hardware components with respect to a specification. In the case of software, the specification may be as simple

as “the program does not crash”, but it can also include higher-level requirements. The goal of verification is to prove that in no circumstance the software or hardware being analyzed can violate its specification. In comparison, “dynamic analysis” or “testing” approaches look for violations over a finite number of test cases, which the testers hope are representative of the usage of the component ; they cannot prove correctness in all usages.

Examples of verification tools (VTs, for short) include BDD-based model checkers (e.g. NuSMV, Cadence-SMV, enjoying much success in VLSI design), SAT-solvers (e.g. zChaff, Minisat, also used in VLSI design), refinement-based proof systems (e.g. the Atelier B, which was used for the development of the Paris Metro line 14), and static analyzers based on abstract interpretation (e.g. the Polyspace verifier, sold by Mathworks, or the Astrée tool, used by Airbus for its fly-by-wire controls).

A common objection is, however, that while the theory of software verification is mature enough, its implementation in tools may contain bugs. Indeed, verification tools are relative large software items, implementing subtle algorithms based on a large theoretical background. They are often quasi-prototypes, developed inside research projects. Users are rightly concerned that such tools may themselves contain bugs, and are reluctant to trust their results too much.

The purpose of this project is to develop techniques so that VTs produce, in addition to a ***verdict*** *(say, that the program conforms to some specification), a* ***certificate*** *for this fact. This certificate could then be used as evidence of the quality of the software. In the case of critical software with a certification process (e.g. avionics), the certificate could be given to the certification authorities.*

A high-level of confidence can be reached by providing a certificate for each particular verdict of the tool. The idea of adding certificates to verdict of VTs was introduced in [Namjoshi] for model-checkers of temporal formula. The tool is not trusted to be safe, and its output is considered as a mere guess. The certificate produced by the tool is then checked using a ***trusted certificate checker*** to demonstrate the accuracy of that guess.

Certificates provide a incredible gain of confidence in the verdict of a VT. The gain is confidence relies on the two following remarks:

- The certificate-checker uses much simpler algorithms than a verification tool. It does not search for a solution; instead, it only checks the correctness of the arguments that justify the verdict. In human terms, this is the difference between looking for the proof of a theorem (difficult and long) and checking that a given proof is correct (much simpler and shorter). Since standards such as DO-178B require that tools upon which the safety of the system reside be certified to the same level of assurance as the system itself, it is important that the proof checker is small and simple enough for its certification to be possible. Small proof-checkers (such as Coq or LF) are privileged candidates to play the role of certificate checkers. Coq has already been used in certification of security application at the highest level of assurance of the Common Criteria (an international norm for certification of security concerned applications).
- The certificate is checked independently of the tool that produced it. Hence the confidence in the result of the tool only depends on the confidence one have in the certificate-checker. A verdict and certificate

produced by an untrusted, unstable, prototype tool that is accepted by the certificate-checker can thus be used in certification.

The generation of certificates can bring prototypes of verification tools to the level of confidence required for the certification of critical systems.

Description scientifique et technique / Scientific and technical description

État de l'art / Background, state of art

Principles of verification tools

Software verification aims, on one hand, at ensuring that no execution of a given program can reach an error state (a state in which some local correctness condition has been violated), and, on the other hand, at ensuring that all executions of the program are finite. The first correctness criterion is called safety, whereas the second criterion (termination) belongs to the more general class of liveness properties (stating that infinitely often, something good must happen).

A safety condition is provided to a verification tool as a (possibly infinite set of states), deemed to be safe. This condition can be expressed as an invariant in a logic used to reason about program states. The task of the verification tool is to verify that the set of reachable states of the program is included in the set of safe states. In other words, a verification condition reduces to (1) computing the set of reachable states of a program (or an over-approximation of it) and (2) verifying that the set of reachable states is included in the set of safe states. The first step is usually performed as a fixed point iteration of the transition relation of the program, whereas the second point reduces typically to verifying the validity of a logical entailment, or inclusion between the languages of two automata.

The termination condition is, in general established by finding suitable ranking functions (mappings from the domain of program states to a well-founded domain). While guessing ranking functions is merely a heuristic task, proving that they decrease by each program step (more precisely, at least once in each program loop) can be done by checking the validity of an inequality in a domain-dependent logic. Equivalently, this inequality can be formulated as a language inclusion property (for a suitable class of automata) as well.

Both safety and termination checks are based on (i) guessing (or inferring by fixpoint iteration) a formula describing a set of states or transitions, and then (ii) checking either validity of a logical entailment, or inclusion of the recognizable sets (languages). These problems are equivalent to checking satisfiability (or, equivalently, language emptiness) of the negated conditions. The given condition is valid if and only if the negated condition is unsatisfiable. Consequently, most verification tools use decision procedures (either SAT/SMT solvers or automata libraries).

In almost all current automatic program analysis systems, the algorithms implementing the entailment checks provide a “yes” or “no” answer that the user has to trust, almost without means for guaranting that this answer is correct. For instance, if the VT proves relations between numerical variables using

convex polyhedra, then the entailment check amounts to polyhedral inclusion, performed using the double representation of polyhedra (generators or constraints) — which involves some rather complex algorithms. This check can also be performed using a general purpose SMT-solver (in theory of linear inequalities). Recent work on abstract interpretation [Costan et al., CAV'05] [POPL'09] has tried to obtain invariants without using so-called widening operators in order to avoid the imprecisions and inefficiencies entailed by widening. Again, the correctness of these invariants can be proved by SMT (Satisfiability Modulo Theories) techniques.

Hence, software verification can often be reduced to to proving the unsatisfiability of a formula in a suitable logic. Indeed, many verification tools use a SAT/UNSAT solver modulo theory (SMT-solver for short). After 30 years of verification, two main classes of SAT/UNSAT algorithms have emerged: those based on standard mathematics and deductive reasoning and those based on automata. In the Certo project we identified a logic that is used in many program verification tools [Sumit+]: it can handle properties on simple data (boolean, integers, rationals) and arrays (or equivalently vectors of indexes and uninterpreted functions). Then, our effort to extend tools SAT/UNSAT algorithms for this logic can benefit to many verification tools.

A verification tools can be split in two parts (1. searching then 2. checking). Step 1 consists in *finding (or guessing using heuristics)* the reason for unsatisfiability ; then Step 2 uses the guess to *checks* the unsatisfiability. The former is difficult and time consuming ; the latter is simpler, efficient but it needs to be justified to certify the verdict of the tools. In the Certo project we shall focus on Step 2, extending tools to produces justifications.

Since the seminal work of Floyd and Hoare [Floyd, Hoare], it has been known that the total correctness of a program can be split into: safety and termination. Checking safety properties of programs (e.g., assertion checking) amounts to (1) finding inductive invariants for loops and recursive functions, (2a) checking that these invariants are inductive, (2b) and then checking that these invariants entail the desired properties. On the other hand, proving program termination is done (traditionally) by (1) finding ranking functions and (2) checking that they decrease at least once in each program loop.

Remark that the soundness of the final result does not reside in the correct guessing of the invariant (or the ranking function), but in the correctness of the verification done in Step 2: invariant inductiveness and entailment (or ranking function decreasing).

Intensive use of SMT-solvers in software verification

Static analyzers based on predicate abstraction, steps (2a,b) use automated proving, often by SMT-solving. SMT-solving is also used for bounded model checking, and for model generation for unbounded model-checking.

In almost all current automatic program analysis systems, the algorithms implementing steps (2a,b) provide a “yes” or “no” answer that the user has to trust, almost without means for guaranteeing that this answer is correct. For instance, if the VT proves relations between numerical variables using convex polyhedra, then the entailment check amounts to polyhedral inclusion, performed using the double representation of polyhedra (generators or constraints) — which involves some rather complex algorithms. This check can also be performed using a general purpose SMT-solver (in theory of linear inequalities). Recent work on abstract interpretation [Costan et al., CAV'05] [POPL'09] has tried to obtain invariants without using so-called widening operators in order to avoid the imprecisions and inefficiencies entailed by widening. Again, the correctness of these invariants can be proved by SMT.

SMT solvers, and, more generally, automatic provers, are also used as building blocks for analysis

systems based on predicate abstraction and counterexample-based refinement (CEGAR). Notable examples of such analyzers include Microsoft's SLAM tool, integrated into the product “device driver verifier”. Again, the soundness of the tool relies on the soundness of the prover. Many analysis systems use the prover Simplify as a “black box”.

State-of-the-art SMT solvers include Barcelogic[1] (University of Barcelona), Yices[2] (SRI), and Z3[3] (Microsoft Research). The source code of these tools is not available. Moreover, they do not provide proofs. Users are thus requested to trust “black boxes”.

When a formula is satisfiable, all SMT-solvers return a witness: an assignment of the free variables of the formula that makes the formula true. In cases where a formula is unsatisfiable, the existing tools do not output any (unsatisfiability) arguments. Because of this asymmetry, we must, in general, trust their implementation. Some SAT solvers (e.g. Minisat) can provide unsatisfiability cores and some log that can be used to reconstruct a proof. Some SMT solvers (e.g., Yices) can be made to list the theory lemmas that they use.

Nevertheless, the gap between the existing state of art and an SMT tool that is able to output checkable proofs, is still big.

Crisis of confidence in software /Reliable software using certificates

More generally, the problem described for verification tools (how can we trust their output for critical systems?) also applies to compilers. There have been several projects of formally verified compilers. Xavier Leroy is working on a project to build a C to PowerPC compiler with formal proofs of correctness [Leroy]. Interestingly, instead of proving the full compiler to be correct, he sometimes only proves the correctness of some verification phase. For instance, the register allocation phase colors an interference graph, and this complex procedure is not proved correct; but then, its output is checked by a certified checker.

Optimizing compilers also use automatic provers to simplify numerical constraints. The Omega procedure is a partial decision procedure for quantified Presburger arithmetic that was originally developed to perform loop optimization in compilers. It has known bugs. A restricted version of Omega has been implemented as a tactic inside the Coq proof assistant. It produces a proof when it succeeds. However no interface is provided to use the Omega tactic out of Coq as an automatic decision procedure. In many tools, the Omega procedure is still used (e.g. in Dfinder, see further) with no guarantee.

The CompCert projet and other research projects have focused on building correct-by-construction compilers [Leroy] and static analyzers [Pichardie+] based on a development in Coq. This is a salutary approach, but it cannot benefit to all valuable existing verification tools. One difficulty with correct-by-construction static analyzers is that each abstract domain must be proved correct, by hand. The Certo projet aims at providing a *justified SMT-solver* that could be reused in many VTs and compiler optimizer instead of the current untrusted solvers.

Some mathematical theorems, most notably the 4-color theorem (all planar graphs can be colored with 3 colors) and Kepler's conjecture (the best stacking for spheres is hexagonal), have been proved using computers : typically, the problem is reduced to a finite number of cases, and each case is checked algorithmically. Mathematicians have voiced concerns that such “proofs” could be erroneous due to bugs. In the case of Kepler's conjecture, Thomas Hales' proof relied on many numerical inequalities. Hales has since then launched a worldwide effort to produce formal proofs of these inequalities,

1 http://www.barcelogic.org/

2 http://yices.csl.sri.com/

3 http://research.microsoft.com/projects/Z3/

known as *project Flyspeck*.[4] One crucial difference though between Flyspeck and the problems that we plan to look into is that they target a specific class of formulas and can use human guidance (in a recent PhD thesis, a few nonlinear inequalities were proved), while we want to obtain automatic techniques suitable for a wide range of applications.

Objectifs et caractère ambitieux/novateur du projet / Rationale highlighting the originality and novelty of the proposal

Transfer and [Valorisation] of the research in software verification

VERIMAG has been a pioneer in the development of model-checking techniques – software verification based on mathematical models. Those techniques go beyond testing: Whereas a system cannot be exhaustively tested – the number of possible executions is in general infinite – exhaustivity can be achieved by mathematical means. Then, VERIMAG has, throughout time, developed a number of verification and analysis tools, based on various techniques (automata, polyhedral constraints, BDDs, SAT-solving...).

None of these tools produced any independently checkable proofs, which limits their usefulness for some industrial applications. Industrial partners may be reluctant to trust academic tools developed without the costly procedures and testing enforced by the standards of critical industries. This lack of confidence limits the potential transfer and valorisation from research to industry. The proposal therefore builds upon laboratory expertise, but brings in a definite difference.

We plan to: evaluate various methods for producing proofs from commonly used proving techniques (invariant generation, SMT-solving, automata), instrument existing tools and develop prototypes tools implementing these verification techniques and generating proofs.

The main difficulty is bridging the gap between the "proof sketches" obtained by the instrumentation, and the level of details required by proof checker such as Coq. It may be necessary to design some intermediate models. Another difficulty is producing proofs of a reasonable length, within acceptable time, and for which the proof-checking is efficient. Experimentations on actual verification tools are needed to drive research on proof reduction.

In addition to scientific publication, final products should include:
- A system for producing proofs from SMT-solving. None of the SMT-solvers available now (Yices, Barcelogic, Z3...) produce such proofs.
- Extension of two verification tools developed in Verimag. These experimentations should demonstrate the feasibility of instrumentation and will be used in demonstrations to industrial partners and certification authorities.

4 http://code.google.com/p/flyspeck/

Programme scientifique et technique, organisation du projet / Scientific and technical programme, project management

Programme scientifique et structuration du projet / Scientific programme, specific aims of the proposal

1. Algorithms

There are mainly two approaches to satisfiability checking. For Boolean Logic (SAT), graph-based methods are used. For Quantifier-free First Order Logic with linear real arithmetic propositions (SMT-LRA) a combination of classical SAT solving and linear programming is used; other kinds of propositions may use e.g. a combination of SAT and integer linear programming. For quantified logics, such as Monadic Second Order Logic interpreted over words and trees automata-based techniques are used. When developing certification methods, we must take into account the two existing approaches to satisfiability checking.

1.1 SAT/SMT Constraint Solving/Linear Programming Algorithms

Many current verification systems, internally use satisfiability testing (SAT) combined with a theory prover (satisfiability modulo theory, SMT). SMT algorithms generally combine DPLL SAT-solving with a decision procedure for theory conjunctions.

SMT first replaces each atomic proposition by a propositional variable (if necessary following some canonization), then solves the resulting propositional problem using SAT-solving (DPLL algorithm).The resulting propositional instantiation is then interpreted as a conjunction in the theory, using a theory decision procedure. Either the conjunction is non contradictory and we have a model (actually, a collection of models), either it is contradictory; a minimal contradiction is extracted, and then its negation is added to the problem as a "theory lemma" and the SAT procedure is restarted. If the SMT problem is unsolvable, the SMT procedures terminates by proposing an unsolvable SAT problem consisting of the original problem plus the theory lemmas.[5]

SMT unsatisfiability proof certificates will thus need to include both a SAT unsatisfiability certificate and theory lemma certificates. For instance, a theory lemma certificate for the unsatisfiability of a system of linear real inequalities $L_1,...,L_n$ consists in positive coefficients $\alpha_1,...,\alpha_n$ such that $\alpha 1\ L1 + ... + \alpha n\ Ln$ is trivially unsatisfiable. SAT unsatisfiability certificates are more complex, and one task of the research project will be to find ways to use them in tractable ways.

Several approaches are possible, in isolation or combination : instrumenting an existing SMT solver, such as Yices, creating our own SMT solver, using the unsatisfiability core extraction and proof logging features of existing SAT solvers such as Minisat. Experiments will be needed to know the best method.

1.2 Automata

An important class of satisfiability checking methods is based on automata theory. Given a formula F we build an automaton A which recognizes (suitable encodings of) the models of F. The satisfiability problem is thus reduced to checking emptiness of automata. To apply this method, one usually needs

5 This is the description of the "lazy theory" approach, where SAT is solved eagerly and the resulting conjunction is then checked. In practice, one uses a more "eager" approach, where partial instantiations are tested for satisfiability. For instance, for the theory of linear inequalities, incremental simplex algorithms are used. This is only for efficiency reasons and does not change the final result, this is why we explained the simpler algorithm.

to work within a class of automata closed under boolean operations (union, intersection, complement) and has a decidable emptiness problem.

This approach to satisfiability stems from the seminal works of Büchi [Büchi] and Rabin [Rabin], who established in this way powerful decidability results for the Monadic Second Order Logic interpreted over (infinite) words and trees, respectively. Subsequently, Vardi and Wolper [VardiWolper] developed the automata-based approach to model checking for linear-time temporal logic, which has become the mainstream technique for the verification of concurrent and/or reactive systems. Their approach uses a proof-theoretic like tableaux construction that produces Büchi automata from Linear Temporal Logic formulas.

Recently, in [HIV08a,HIV08b] we have applied automata-based techniques to checking the satisfiability of logics interpreted over vectors of integers. In this work, we use flat counter automata to encode (possibly infinite) strings of integers.

From a practical point of view, the main difficulty in applying automata-based techniques to satisfiability problems (that are raised usually as verification conditions) is twofold:

1. Guaranteeing the faithfulness of encodings.
2. Ensuring the correct implementation of boolean operations (union, intersection or complementation) on automata. As an example, recent tests performed with a mainstream tree automata library revealed a situation in which the intersection between an automaton and its complement was not empty.

One of the goals of this project is certifying satisfiability checkers developed for integer vector logics. Since we use counter automata to encode the set of models of a formula, an unsatisfiability result is given in terms of a counter automaton that has no run leading to a final state. The emptiness of a counter automaton is usually due to the unsatisfiability of a guard at certain control location, which can be detected automatically. The challenge is to use the emptiness argument for the counter automaton in order to build a certificate of unsatisfiability for the given formula. This certificate can be expressed, for instance, as a derivation in quantifier-free Presburger arithmetic with uninterpreted functions, and can be automatically checked by a dedicated proof checker.

Even though the techniques used to generate certificates of unsatisfiability seem to pertain to a certain logic/class of automata, we aim at inferring general guidelines from particular techniques such as e.g., the ones used to attest emptiness of Büchi automata [KupfermanVardi]. A major obstacle here is that formal proofs usually rely on axiomatized domains, whereas automata work with recognizable structures. To date, there are no feasible attempts to axiomatize general-purpose recognizable domains (e.g. words, trees, etc.) in order to generate proofs of emptiness automatically. This is currently an important open problem that we would like to tackle in this project.

2. Applications

As shown by the pioneering works of Floyd [Floyd] and Hoare [Hoare], verification of sequential programs rely on the two key concepts of (1) invariance and (2) progress. In this project, we aim at applying general unsatifiability certificates.

2.1 Soundness of abstractions

Abstraction is a technique that is used to reduce the mathematical model to check; it is intensively used in verification tools. When doing program analysis by either abstract interpretation or by model checking combined with predicate abstraction, it is important that the abstract model really represents all behaviors of the concrete system, otherwise the system may infer false properties. SMT-solving techniques can be used to check that an abstraction is correct by showing the unsatisfiability of a formula defining the behaviors not captured by the abstraction. If we have a proof-logging version of the SMT solver, then we automatically obtain a proof of correctness of the abstraction. This contrasts

with more manual assisted-proof approaches [Pichardie+].

2.2 Correctness of invariants

An assertion in a program is an invariant property if it always holds when the execution reaches that program point. The verification of a safety property is based on the computations of an invariant property, followed by the verification that it entails the desired property. A certificate should contains (1) a proof of this entailment which is obtained using a SMT-solving algorithm (noticing that the validity of $Inv \Rightarrow Safe$ is equivalent to the unsatisfiability of $Inv \wedge \text{not}(Safe)$), and (2) a proof of the invariance of the *Inv* property. The latter consists in a proof by induction based on the

The latter is based on Floyd-Hoare's proof techniques [Hoare,Floyd] which requires to prove that *Inv* holds as initial conditions of the program and that it is preserved by each program step, meaning that, if *Inv* holds as a precondition of an instruction then it still holds as a postcondition. All these subgoals of the proof are done using Hoare's triple *{Precondition} Instruction {Postcondition}*, they are acheived using Hoare's calculus of weakest liberal precondition (*wlp*) that transforms the triple into the equivalent implication $Precondition \Rightarrow wlp(Inst,Postcondition)$, followed once again by a call to a SMT-solver.
This well-known proof technique relies on an induction principle, thus it only works for invariants that can be proved by induction (they are called *inductive invariants*). Whereas the proof of inductive invariant can easily be generated, precise inductive invariants are difficult to discover. Often VTs computes *non-inductive invariants* that still are invariant properties and precise enough to get the proof (1) of entailment. *Non-inductive invariants* are easier to obtain but there is no straightforward technique for conducting the proof (2) of their invariance. So, one has to strengthen the invariants to make it inductive using refinement by counter examples proposed by [Bradley07].

We plan to use the standard proof technique for *inductive invariants* and we propose a new technique to prove *non inductive invariants* which avoid strengthening. In general the inductive quality of an invariants is lost during abstraction step that are used in VTs to reduce the formula using non conservative simplifications. The tools computes a simplified formula F' that is implied by the original formula F. Recording the implication $F \Rightarrow F'$ and its proofs of validity it can then be used in the proof of invariance, avoiding the appeal to an inductive invariant. This novel technique can be though as successive refinements of invariants; it requires a strong connection with the computation steps of the VT. It should result in less compact and elegant proofs than the standard technique for inductive invariants but it can be automated and avoid the difficult problem of strengthening invariants to make them inductive.

2.3 Termination

The termination problem asks, for a given program and a (possibly infinite) set of initial configurations, whether there exists an initial configuration in the set, starting with which the program has an infinite execution. Even though this problem is different in nature from the invariance problem (the difference is the same as between safety and liveness properties), the existing approaches are similar.

The ranking functions method [Bradley] consists in defining a function that maps program states into a well-founded domain, and showing that, for all program loops and for all states, the value of the function decreases with each iteration of the loop. As the range of the ranking function is usually a domain which is known to be well-founded (positive integers, tuples of positive integers, multisets, etc.) the progress test is usually implemented as a validity condition. If the negation of the verification condition is unsatisfiable, then the choice of the ranking function is valid, and the program is shown to terminate. The transition invariants method [PodelskiRybalchenko] is a generalization of the ranking functions method.

Proving termination of non-deterministic systems exhibits problems that are beyond the reach of the techniques above. Such is the case of counter automata with non-deterministic transition relations (e.g., difference bound constraints) whose termination can be proved by different means. Namely, the termination problem for a counter automaton with difference bound transition relations is reduced to the presence of a cycle of negative weight in the constraint graph representing the execution of the automaton.

In [BIL06], we have developed a method for detecting the presence of such negative cycles in constraint graphs of unbounded size. The method is based on a translation of difference bound relations to weighted finite automata. A certification method for this translation will be used to automatically generate termination proofs for counter automata. More generally, this would increase the confidence of the users in the implementation of the analysis method provided in [BIL06].

2.4 Exchange

Many verification tools deal with transition systems (a mathematical representation of programs) and variants of the logic of [Sumit+] on arrays and simple data. They internally use different formats preventing cooperation between tools. Producing certificates for the same proof-checker naturally provide an exchange format. It becomes possible for a tool to call another tool specialized on specific sub-problems (like a SMT-solver) and to integrate the resulting certificate in the construction of the main certificate. This generalizes the cooperation principle of [NelsonOppen] which was originally restricted to the exchange of equality relations. We shall use this feature in our experimentations on VTs using SMT-solvers.

[NelsonOppen] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. ACM Transactions on Programming Languages and Systems, 1(2):245–257, . (TOPLAS)

3. Experiments

3.1 Péron-Halbwachs' analyser of programs manipulating arrays

Péron & Halbwachs developed a static analyzer that discovers properties of programs manipulating arrays [PéronHalbwachs]. The PH's analyzer is based on a fixpoint computation of inductive invariants using abstract interpretation. We plan to instrument the tool to make it producing on-demand a certificate of invariance of the discovered property. The certificate will take the form of a proof based on the standard Floyd-Hoare's proof techniques for inductive invariant.

The source code of the tools is available and the developer of the tools is located in Verimag. Therefore we can then experiment a direct instrumentation of the code: we distinguish functions that implement the heuristics of the tools (they are responsible of precision, termination) and semantic functions which are accountable for correctness. Only those functions need to be instrumented. A justified result is a couple $(e, \nabla\phi)$ made of a data e and a proof that the property ϕ holds for e. A semantic function $f(e_1,\ldots, e_n) = e$ is instrumented by replacing each parameter with a justified entry, leading to $f((e_1,\nabla\phi_1),\ldots,(e_n,\nabla\phi_n)) = (e,\nabla\phi)$. The proof $\nabla\phi$ is build from the justifications $\nabla\phi_1, \ldots, \nabla\phi_n$ of the entries. All the efforts then lie in combining the appropriate deduction rules of the deduction system to demonstrate ϕ from $\phi_1 \wedge \ldots \wedge \phi_n$.

We can use a two phases process to produce a certificate. First, we run the tool without instrumentation to preserve efficiency in searching invariant. The tools ends up with a invariant *Inv* that is a fixpoint of its computation, meaning that if we run the tools once again *Inv* as an initial invariant, it will stop after 1-iteration of its computation (of *post(Inv)*), discovering that this invariant is stable henceforth inductive. Second ,we run the instrumented version of the tool on the discoved invariant *Inv*, the instrumented computation of *post(Inv)* and the instrumented test of stability should provides the arguments for the proof that *Inv* is a n inductive invariant.

To summarize: *We believe that direct instrumentation will be simpler that a posteriori generation of proof in a proof assistant. The goal of this experimentation is to evaluate this claim, by instrumenting the code of a verification tool so that it builds certificate of its results on-demand while not sacrificing its performance.*

[PéronHalbwachs] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In ACM Conference on Programming Language Design and Implementation, pp.339–348. ACM Press, 2008. (PLDI'08)

3.2 The Deadlock Finder

DFinder [Bensalem+] is a symbolic model-checker developed in Verimag that verifies the deadlock freedom in multi-threaded applications. It is used in a tools chain that generates embedded code for robotic system from mathematical model [Basu+]. DFinder computes invariants by refinement based on the initial conditions and the symbolic computation of the *post* operator. DFinder uses two external ools, the Yices SMT-solver and the Omega simplification procedure. They are used to simplify symbolic constraints (eliminating useless quantifier and discovering unsatisfiable constraints).

The case of DFinder is interesting and differ from PH's analyzer for several reasons: (1) The logic of Dfinder is strict subset of the one used in PH's analyzer but the systems to verify have a richer semantics that deals with threads and synchronizations; (2) DFinder is already a gray box, since it sends requests to Omega and Yices; (3) DFinder is a large tool that consists in several stages where it applies aggressive simplifications to reduce the size of the mathematical models; (4) It often produces *non inductive invariant* due to lost of precision in simplifications, henceforth forbidding the standard proof by induction.

DFinder computes a invariant *Inv* that entails the deadlock freedom. Again the proof can be decomposed into (1) a proof of invariance and (2) a proof of entailment. The latter can be obtained using the *justified SMT-solver* that will be developed in the project. We plan to focus on the proof of invariance.

In the case where the property *Inv* is not inductive the proof of preservation $post^*(Inv) \Rightarrow Inv$ will fail. We can still try to prove the invariance of Inv following the refinement steps of DFinder. It starts from the trivial invariant $Inv0 := true$ and computes finer invariants *Inv1,Inv2,...* until it reaches a invariant, say *Inv*, precise enough to proof the deadlock freedom. Using the definition of the invariant refinement process it is possible to prove at each step that *Inv[i]* is a invariant (reusing the fact that *Inv[i-1]* was an invariant). This proof includes a justification that the abstraction process (done by Omega) always preserves implication (meaning that $F \Rightarrow Abst(F)$). This technique shall result in less compact and elegant proofs than the standard technique for inductive invariants but it can be automated and avoid the difficult problem of strengthening invariants to make them inductive.

In order to conduct this experiment we have to develop a justified version of the Omega procedure (or part of it) and a justified SMT-solver. Both can be reused in many verification tools, since simplification, elimination of useless quantifier, unsatisfiability of constraints are largely used in software verification.

To summarize: *The challenge in this experimentation is to develop a proof technique for non inductive invariant which avoids the difficult problem of strengthening the invariant to make it inductive.*

[Bensalem+] S.Bensalem, M.Bozga, J.Sifakis, T.Nguyen. Compositional Verification for Component-based Systems and Application. In: *International Symposium on Automated Technology for Verification and Analysis, 2008, (ATVA'08)*

[Basu+] A.Basu, M.Gallien, C.Lesire, T.Nguyen, S.Bensalem, F.Ingrand and J.Sifakis. Incremental Component-Based Construction and Verfication of a Robotic System. In: *European Conference on Artificial Intelligence, 2008, (ECAI'08)*

3.3 Applications to program analysis

Program analyzers by abstract interpretation do fairly complex computations in order to infer program invariants. For instance, the ASOPT project at VERIMAG aims at inferring invariants using numerical algorithms, or even at producing abstract transfer functions automatically. In general, checking invariants is easier than inferring them; simpler algorithms may be used. We plan to use SMT-solving to obtain proofs of correctness of invariants and transfer functions.

3.4 FLATA

The FLATA system is a tool developed at VERIMAG for the analysis of counter automata models. This tool relies essentially on the results of [ComonJurski,BIL06] for non-deterministic flat counter automata with difference bound constraints. The tool is moreover used as analysis engine for the counter automata produced from formulae in the integer vector logic of [X]. In the near future, we plan to extending FLATA by developing an entire program analysis framework on top of it.

Since FLATA is developed by a part of this project's consortium, it is feasible to envisage certification techniques for the implementation of the tool. In order to describe the directions we intend to follow in what concerns certification, we need to give an overview of the system's workflow:

a) Given a formula in the logic of [HIV08b], it is translated into a counter automaton. The translation needs to be instrumented in order to turn a proof of emptiness of the counter automaton in a certificate of unsatisfiability of the formula. This certificate can be e.g., a derivation in quantifier-free Presburger arithmetic with uniniterpreted function symbols.

b) A given counter automaton is abstracted into a flat counter automaton, by merging transitions, eliminating unfeasible paths, and eventually over-approximating strongly-connected components with elementary loops. These transformations need to preserve the semantics of the automaton. The implementation of the abstraction techniques needs to be certified.

c) sFinally, for each loop in the counter automaton we compute the transitive closure of its transition relation. The implementation of this computation is to be also certified.

Together, the above three certification steps can be combined in order to obtain a trusted analysis tool for counter automata models.

Coordination du projet / Project management

The project has been divided in relativity independent tasks, in order to ease the work-flow. Yet, strong synergies exist. For instance, one possible option for the certificates of the automata-theoretic approach is to reduce certain problems with respect to numerical constraints in the automaton to some SMT-problem, which can then be decided using a proof-

logging SMT-solver.
The researchers involve also organize a “program analysis and verification” workgroup, where researchers from VERIMAG and surrounding laboratories meeting.

Description des travaux par tâche / Detailed description of the work organised by tasks

Tâche 1 / Task 1

Generation of certificates as foundational proofs. Participant: Michaël Périn.

Several forms of certificates have been proposed for model-checking [KupfermanVardi,Tan+]. They all require dedicated checking algorithms. Meanwhile it has been recognized in the Proof-Carrying Code community that foundational proof-checkers are more reliable [Appel]. *Foundational proofs* are deductive proofs conducted in restricted proof systems such as the axiomatization of first order logic and standard mathematics. The challenge is then to produce foundational proofs form VTs. Each evidence must be provided in term of a few elementary deduction rules. We plan to use the Coq environment developed at Inria as the trusted foundational proof-checker.

Our goal is to extend two verification tools to produce certificate in the form of Coq proofs.

- We shall consider two types of verification tools with two different domains of application: HP's static analyser for program manipulating arrays and Dfinder, a model-checker that discovers deadlocks in multi-threaded applications used in robotics.
- We'll experiment two instrumentation techniques: (1) a white box approach on HP's analyser, that is a direct instrumentation of the source code so that it builds the certificate along the computations, and (2) a gray box instrumentation of Dfinder asking the developper to output some additional informations needed to a posteriori build the certificate.
- We'll apply two techniques for proving invariance: (1) the standard Floyd-Hoare's induction for inductive invariants provided by HP's analyser, and (2) the invariant refinment technique presented in Section (2.2 Correctness of invariants) for non-inductive invariants computed by DFinder.
- We'll factorize the work by making use of the SMT developed in the project for subgoals that require proofs of unsatisfiability (they are intensively used in both tools).
- We'll develop a justified version of a simplification procedure based on Omega.
- We'll develop an intermediate format of certificate suitable for proof involving Hoare's triple. This will help reducing the size of proofs and getting "understandable" proofs. We'll provide an automatic translation into Coq format. This translation will automatically add (with justification) the uninteresting simplification and normalisation steps that need to be justified in foundational proofs.

To summarize: *We shall conduct actual experimentations instrumenting two existing verification tools based on invariant comptations and using SMT-solvers to make them produce foundational proofs for the Coq proof-checker.*

Tâche 2 / Task 2

SMT solving. Participants: David Monniaux, MichaÎl Périn.

The goal is to obtain a SMT-solver that can produce proofs, and then to postprocess these proofs for

use in a proof assistant.

We envision several methods for obtaining these proofs from SMT-solving:
- Instrument a SMT solver such as Yices and get it to print out theory lemmas, then feed both the propositional model the the theory lemmas into a SAT solver that does proof logging.
- Implement a SMT solver from scratch. This would give us better control on the system, given that most available SMT-solvers are closed-source, proprietary "black boxes". The disadvantage is that these tools have been implemented by groups who have worked on these issues for years and have introduced many improvements, not necessarily documented in open publications, so it seems difficult to reach comparable performance..

The output from this step should then be transformed into a model acceptable by a proof assistant such as Coq or PVS. It seems desirable to have an intermediate model, if only so as to be able to target multiple assistants.

Applications targeted include the analysis tools developed within the ANR ASOPT project: we plan to evaluate whether it is possible to get them to output proofs (by making their final step log proofs or by running an additional checking phase with proof logging).

Since David Monniaux has already developed small SMT-solvers and works with Yices for his tools based quantifier elimination, he will be heading this task.

The risks are very limited if the goal is to develop a prototype without regard to scalability – we do not envision any strong difficulty there. What is yet unknown, though, is how to instrument the tool so as not to penalize too much its efficiency, and how to keep the proofs small — both characteristics are needed for the method to be applicable beyond mere academic toy cases.

To summarize: We will implement a SMT-solver (for, say, the theory of linear inequalities), or instrument an existing one, so that it prints proof sketches, and implement a postprocessing scheme to convert these proofs into proofs accepted by a proof assistant.

Tâche 3 / Task 3

Participants: Radu Iosif

The goal of this task is to provide certificates for the satisfiability checking method for the integer vector logic described in [HIV08b]. This technique consists in translating a formula on integer vectors into a flat counter automaton with difference bound transition relation and then checking the automaton for emptiness. The latter problem is solved using a method described in [BIL06]. On one hand, we need to certify the correctness of the emptiness checking of counter automata. On the other hand, and more importantly, an empty automaton provides an argument of unsatisfiability of the integer vector formula. This argument has to be translated into a mechanical-checkable proof that can be given to a proof checker. The techniques developed by this task will be implemented as part of the FLATA system.

Radu Iosif is in charge of this task.

To summarize: We will investigate how to produce proofs out of these computations on automata, and possibly implement research prototypes.

Calendrier des tâches, livrables et jalons / Planning of tasks, deliverables and milestones

[t0,t0+12]

T1.1.1 Design of a certificate exchange format as foundational proofs in a friendly, human readable syntax

T1.1.2 Translation into Coq proof format
T1.1.3 Development of an justified version of the Omega procedure
T2.1 Experiments with a prototype SMT-solver for linear inequalities with logging features and/or instrumentation of Yices.
T3.1 From empty counter automata to unsatisfiable integer vector formulae.
- This subtask deals with the generation of unsatisfiability certificates of an array formula, given a proof of emptiness of the corresponding counter automaton.

[t0+12,t0+24]
T1.2.1 Instrumentation of PH's analyser and DFinder.
T2.2.1 Experiments with SMT-solver for other logics (integer linear relations...)
T2.2.2 Experiments with scalability and optimization of SMT-solver.
T3.2 Certification of counter automata decision procedures.
- The analysis of counter automata is based on a transitive closure algorithm for simple loops with difference constraints. This subtask aims at certifying the reduction steps involved by this decision procedure.

[t0+24,t0+36]
T1.3.1 Experimentations running the instrumented tools on systems to verify
T1.3.2 Evaluation of the size of certificates and development of reduction techniques
T1.3.3 Dissemination to industrial partners
T2.3: Use of proof-logging SMT-solver as back-end to tools developed within project ASOPT (synthesis of abstract transfer functions).
T3.3 Certification of automata-based satisfiability techniques.
- This task is of a more exploratory nature. Its goal is to investigate certification techniques for general-purpose automata-based satisfiability techniques, such as decision procedures for Presburger arithmetic or Monadic Second Order Logic.

Stratégie de valorisation des résultats et mode de protection et d'exploitation des résultats / Data management, data sharing, intellectual property and results exploitation

We plan the usual scientific dissemination through peer-reviewed international journals and peer-reviewed proceedings of international conferences.
In addition, we plan to make our software available under a free license, so as to maximize the impact on the scientific community. Since we plan only academic prototypes, not industrial-strength software packages (which we do not have the manpower to develop, anyway), we think this licensing is appropriate.

We expect the following industrial relations:

1) VERIMAG is is contact with Airbus regarding their static and dynamic analysis needs. In december 2008, Airbus is organizing a grand meeting with tool suppliers and academic laboratories to help them defined their long-term strategy regarding software reliability. David Monniaux will present the ASOPT project, whose goal is to introduce numerical optimization techniques and geometrical methods into static analysis. We envision that within the CERTO project, we can build automated techniques capable of certifying some classes of

invariants or transfer functions produced by the ASOPT technologies.

2) VERIMAG is in contract with ASTRIUM, the aerospace service of EADS, to develop a prototype of code generation for robotic applications from mathematical models written in the Verimag's BIP language (Project MARAE). This prototype will then be evaluated as a candidate for programming satellites and drone applications. We plan to apply our certification techniques to a crucial verification tools (DFinder) used in the BIP framework to guaranty the absence of deadlock in multi-threaded applications.

3) VERIMAG has been involved in two RNTL projects (EDEN1, EDEN2) on Certification of security applications for smart cards in the Common Criteria.The Common Criteria is an international norm for certification of applications with security requirements. At the highest level of assurance, the developer must formally demonstrate that the application conforms to the security policy. Recently, a certification has been achieved by Gemalto on the basis of a development in the Coq proof assistant. Coq is then trusted by the DCSSI (the French certification authorities). It then becomes worthwhile to produce Coq proofs from verification tools. The benefits of certificates is discussed with the industrial partners of EDEN2 (Gemalto, Trusted logic) and the involved certification authorities (CEA-LETI, DCSSI).

Organisation du projet / Consortium organisation and description

Description, adéquation et complémentarité des participants / Relevance and complementarity of the partners within the consortium

David Monniaux has done extensive work in static analysis by abstract interpretation. Between 2001 and 2007, he was involved in the development of the Astrée static analyzer, a tool used at the aircraft manufacturer Airbus in order to prove the absence of runtime errors in critical embedded programs, including fly-by-wire controls. Many publications have ensued, both on theoretical and practical aspects.[6]
Since moving to VERIMAG in 2007, David Monniaux has been interested in, on the one hand, methods for analyzing programs in a modular way (Astrée implements monolithic analysis), and on the other hand deciding numerical formulas. He works on including techniques such as SMT, numerical optimization, BDDs, into abstract analyzers [LPAR '08, POPL '09].

David Monniaux has past experience with the use of proof assistants, including Coq and PVS, for the formalization of abstract interpretation, thus his interested in making such proofs automatic.

Radu Iosif obtained his MSc in 1997 from the Polytechnic University of Bucharest and his PhD in 2001 from the "Politecnico" University of Turin, in the area of software model checking. Between 2001 and 2002, he worked as a post-doctoral fellow within the BANDERA program analysis project at Kansas State University. Since Radu Iosif has been hired as a CNRS researcher in the VERIMAG laboratory in 2002, he has done work in the fields of program verification (both safety and termination checking) and automated reasoning. His main interests are applying logics and automata theory to program verification, by identifying classes of programs for which verification is decidable (and

6 A complete list of publications is available from http://www-verimag.imag.fr/~monniaux/biblio/David_Monniaux.html

tractable), and designing suitable logics and classes of automata to fit the verification problems raised by these classes. His research mainly targeted programs with dynamic recursive data structures (e.g. lists, trees, arrays) and the analysis of counter automata models.

Qualification du porteur du projet / Qualification of the principal investigator

Michaël Périn (36 years old) is associated professor since 2001 at the University of Grenoble 1 and Verimag lab. His research interests are program verification and certification using formal proof, applied to security applications and concurrent programming. He participated in the developpement of a verification tool for cryptographic protocols, that he instrumented to produce certificates for the Coq proof-checker. His recent works focus on certification.

He was responsible of the RNTL EDEN 1 (2002-2005) and EDEN 2 (2006-2009) project on Certification of Security Applications for Smart Cards in Common Criteria. These two projects involved industrial partners (Gemalto, Trusted Logic), research labs (CEA-LIST, VERIMAG) and certification autorithies (CEA-LETI, DCSSI). The goal was to the design of a methology supported by tools for establishing the conformance of an application to a security policy, fullfilling the requirements of the Common Criteria. The project Eden1 focused on simulation which helps finding bugs whereas the Eden2 project generalizes the methodology to symbolic model-checking that is more amenable to produce formal certificates of conformance. As a result of Eden2 we proposed a methodology for establishing the conformance relation between two mathematical models, one representing the application (described in the JavaCard language) and the other representing the security policy (described as an extended state machine). The method is an extension of the Floyd-Hoare's technique of Section 2.2 for proving *invariant assertions* between a *pair of models* [SAC'09]. The DCSSI (*Direction centrale de la sÈcuritÈ des systËmes d'information*, the french certification authorities) recently delivers a certificate on the basis of a development in Coq. This offers opportunities to get certified verdicts of verification tools accepted in certification process. The tool developed in EDEN2 is currently not mature enough to be instrumented, but it will benefit from result of the Certo project. Indeed, it is based on generation of invariants and uses the Yices SMT-solver.

Here is a list of his recent publications:

- *[SAC'09] Certification of Smart-Card Applications in Common Criteria,* with I.Narasamdya. In ACM Symposium on Applied Computing / Software Verification and Testing track (SAC/SVT'09), 2009.

- *[SafeCert'08] Convincing proofs for program certification*, with M.Garnacho. In International Workshop on Certification of Safety-Critical Software Controlled Systems (SafeCert'08), 2008.

- *[ITCES'06] Certification of Cryptographic Protocols by Abstract Model-Checking and Proof Concretization*, with R.Janvier, Y.Lakhnech. In Workshop on Innovative Techniques for the Certification of Embedded Systems (ITCES'06), 2006.

- [STTT'06] P*attern-based Abstraction for verifying Secrecy in Protocols*, with L. Bozga, Y. Lakhnech. In International Journal on Software Tools for Technology Transfer, vol. 8, 2006.

Qualification, rôle et implication des participants / Contribution and qualification of each project participant

	Nom	Prénom	Emploi actuel	Unité de rattachement et Lieu	Discipline *	Personne. mois	Rôle/Responsabilité dans le projet 4 lignes max
Exemple	*LATIFI*	*Fatima*	*Professeur*				*CaractÈrisation des facteurs de transcription recombinants en systËme in vitro ...*
Coordinateur	Périn	MichaÎl	MCF	Université Grenoble 1			Instrumentation d'outils. Génération de preuves. en Coq
Autres membres	Monniaux	David	CR1 CNRS	UMR5104		4	Aspects SMT-solving
	Iosif	Radu	CR1 CNRS	UMR5104		4	Aspects automates

* à renseigner uniquement pour les Sciences Humaines et Sociales

Justification scientifique des moyens demandés / Scientific justification of requested budget

Personnel / Staff

Requested budget: **93300€** for recruiting of a post-doctoral researcher for two years (or 2 post-docs each for one year).
In addition to purely scientific work (e.g. inventing algorithms or schemes), the project involves substantial programming tasks in order to produce reasonably practical tools. It does not seem possible to achieve these goals with only the limited manpower of the permanent scientific staff.

Missions / Missions

Requested budget: **20000€**. The project aims at dissemination of new scientific results, and in computer science, international conferences with editorial committees and peer reviewers are the major vector of dissemination. A one-week conference in the United States costs approximately 2500€ overall. There are four researchers on this project (3 permanent staff and 1 post-doc).

In addition, if industry (e.g. Airbus) gets interested in the project, then it will be necessary to visit them.

Autres dépenses de fonctionnement / Other expenses

Toute dÈpense significative relevant de ce poste devra être justifiÈe.

Requested budget 11500€ as follows:

- 3 new personal computers: 6000€
- Various operating costs of the laboratory (estimated at 500€ per permanent staff per year): 4500€
- Books, stationery, printing costs for the poster: 1000€

Annexes

Références bibliographiques / References

Inclure la liste des rÈfÈrences bibliographiques utilisÈes dans ce document et les rÈfÈrences bibliographiques des partenaires ayant trait au projet.

Our articles:

[BIL06] Marius Bozga, Radu Iosif, Yassine Lakhnech: Flat Parametric Counter Automata. ICALP (2) 2006: 577-588

[HIV08a] P. Habermehl, R. Iosif and T. Vojnar. What else is decidable about integer arrays? Proc. FoSSaCS 2008, pp. 474-489

[HIV08b] P. Habermehl, R. Iosif and T. Vojnar. A Logic of Singly Indexed Arrays. Proc. LPAR 2008 to appear.

[Essance'02] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation: Complexity, Analysis, Transformation*, number 2566 in Lecture Notes in Computer Science, pages 85-108. Springer Verlag, 2002.

[PLDI'03] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196-207. ACM, 2003.

[CAV'05] David Monniaux. Compositional analysis of floating-point linear numerical filters. In *Computer-aided verification: CAV '05*, number 3576 in Lecture Notes in Computer Science, pages 199-212. Springer Verlag, 2005.

[ESOP'05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In *ESOP*, number 3444 in Lecture Notes in Computer Science, pages 21-30, 2005.

[LPAR'08] David Monniaux. A quantifier elimination algorithm for linear real arithmetic. In *LPAR (Logic for Programming Artificial Intelligence and Reasoning)*, Lecture Notes in Computer Science. Springer Verlag, 2008.
[POPL'09] David Monniaux. Automatic modular abstractions for linear constraints. In *POPL (Principles of programming languages)*. ACM, 2009

Other articles:

[Appel] Andrew W. Appel. Foundational proof-carrying code. In IEEE Symposium on Logic in Computer Science, pages 247–258, 2001. (LICS’01).

[Costan et al., CAV'05] Alexandru Costan, Stephane Gaubert, Eric Goubault, Matthieu Martel, Sylvie Putot: A Policy Iteration Algorithm for Computing Fixed Points in Static Analysis of Programs. CAV 2005: 462-475

[Bradley] A. R. Bradley, Z. Manna, and H. B. Sipma. The polyranking principle. In ICALP'05: International Colloquium on Automata, Languages and Programming, 2005.

[Bradley07] Bradley, A. R. and Manna, Z. Checking Safety by Inductive Generalization of Counterexamples to Induction. In: Formal Methods in Computer-Aided Design, 2007, pp. 173-180.

[Buechi] Büchi, J.R., On a decision method in restricted second order arithmetic. In: Nagel, E. (Ed.), Methodology and Philosophy of Science, Stanford University Press, Stanford, CA. pp. 1-11.

[ComonJurski] Comon, H. and Jurski, Y. 1998. Multiple Counters Automata, Safety Analysis and Presburger Arithmetic. In Proceedings of the 10th international Conference on Computer Aided Verification (June 28 - July 02, 1998). A. J. Hu and M. Y. Vardi, Eds. Lecture Notes In Computer Science, vol. 1427. Springer-Verlag, London, 268-279.

[Floyd] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, Mathematical Aspects of Computer Science, volume 19 of Proceedings of Symposia on Applied Mathematics, pages 19--32. American Mathematical Society, 1967.

[Hoare] Hoare, C. A. 1969. An axiomatic basis for computer programming. Commun. ACM 12, 10 (Oct. 1969), 576-580.

[KupfermanVardi] Kupferman, O. and Vardi, M. Y. 2005. From complementation to certification. Theor. Comput. Sci. 345, 1 (Nov. 2005), 83-100.

[Leroy] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42-54. ACM Press, 2006.

[Namjoshi] Namjoshi, K. S. 2001. Certifying Model Checkers. In Proceedings of the 13th international Conference on Computer Aided Verification (July 18 - 22, 2001). G. Berry, H. Comon, and A. Finkel, Eds. Lecture Notes In Computer Science, vol. 2102. Springer-Verlag, London, 2-13.

[Pichardie+] Frédéric Besson, Thomas Jensen, and David Pichardie. Proof-carrying code from certified abstract interpretation and fixpoint compression. Computer Science, 364(3):273–291, 2006. (TCS).

[PodelskiRybalchenko] Podelski, A. and Rybalchenko, A. 2004. Transition Invariants. In Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (July 13 - 17, 2004). LICS. IEEE Computer Society, Washington, DC, 32-41.

[Rabin] MichaÎl O. Rabin, Decidability of second-order theories and automata on infinite trees, Bull. Amer. Math. Soc. Volume 74, Number 5 (1968), 1025-1029.

[Sumit+] Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In ACM Conference on Principles of Programming Languages, pages 235–246. ACM Press, janvier 2008. (POPL'08).

[Tan+] T. Tan and R. Cleaveland. Evidence-based model checking. In Computer Aided Verification, volume 2404 of LNCS, pages 455–470, 2002. (CAV'02).

[VardiWolper] Vardi, M. Y. and Wolper, P. 1994. Reasoning about infinite computations. Inf. Comput. 115, 1 (Nov. 1994), 1-37.

Biographies / CV, Resume

David Monniaux studied as a normalien undergraduate at École normale supérieure de Lyon, starting in 1995. He obtained a research master's degree from University Paris-7 in 1998 on programming languages, semantics and proofs; the subject of his master's thesis was the formalization of abstract interpretation using the Coq proof assistant. He made several stays at SRI International, in the group developing the PVS assistant. His PhD, defended in 2001, was on the analysis of probabilistic programs. He also worked on proofs on cryptographic protocols.

Between 2001 and 2007, David Monniaux was involved in the development of the Astrée static analyzer[7], a tool developed at the request of the avionics division of the aircraft manufacturer Airbus for proving the absence of runtime errors in critical programs, most notably fly-by-wire controls for A340-600 and A380. His work on the subject includes among others analysis of numerical properties, parallel implementations, analysis of asynchronous code.

David Monniaux moved to VERIMAG in 2007, where he is a CNRS researcher.

Michaël Périn studied mathematics and computer science at the University of Rennes. During is master he moved to computer science and does his master thesis on termination of partial evaluation. He defended his PhD in 2000 on the formalization and consistency checking of UML diagrams using logic and family of graphs. He started working on security and certification during his post-doctoral research at the University of L'Aquila (Italy). In 2001, he became assistant professor the University of Grenoble 1 and Verimag. He is in charge of courses on deductive systems and software verification. His research interests are software verification applied to security applications and concurrent programming. He participated in the development of a verification tool for cryptographic protocols, that he instrumented to produce certificates for the Coq proof-checker. His recent work focuses on certification using formal proof.

Radu Iosif obtained his MSc in 1997 from the Polytechnic University of Bucharest and his PhD in 2001 from the "Politecnico" University of Turin, in the area of software model checking. Between 2001 and 2002, he worked as a post-doctoral fellow within the BANDERA program analysis project at Kansas State University. Since Radu Iosif has been hired as a CNRS researcher in the VERIMAG laboratory in 2002, he has done work in the fields of program verification (both safety and termination checking) and automated reasoning. His main interests are applying logics and automata theory to program verification, by identifying classes of programs for which verification is decidable (and tractable), and designing suitable logics and classes of automata to fit the verification problems raised by these classes. His research mainly targeted programs with dynamic recursive data structures (e.g. lists, trees, arrays) and the analysis of counter automata models.

Implication des personnes dans d'autres contrats / Involvement of project participants to other grants, contracts, etc…

Cf. § .

Mentionner ici les projets en cours d'Èvaluation soit au sein de programmes de l'ANR, soit auprËs d'organismes, de fondations, à l'Union EuropÈenne, etc. que ce soit comme coordinateur ou comme partenaire. Pour chacun, donner le nom

7 http://www.astree.ens.fr/

de l'appel à projets, le titre du projet et le nom du coordinateur.

Part.	Nom de la personne participant au projet	Personne. mois	Intitulé de l'appel à projets Source de financement Montant attribué	Titre du projet	Nom du coordinateur	Date début & Date fin
N°1	D. Monniaux		ANR ARPEGE 15600€ (discussion ANR en cours)	ASOPT	Bertrand Jeannet	1/2009-12/2011
N°						