Graph-based Reduction of Program Verification Conditions

Jean-François Couchot, Alain Giorgetti, Nicolas Stouls

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Graph-based Reduction of Program Verification Conditions

Jean-François Couchot — Alain Giorgetti — Nicolas Stouls

N° 6702

Octobre 2008

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--6702--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
SACLAY - ÎLE-DE-FRANCE

Graph-based Reduction of Program Verification Conditions *

Jean-François Couchot† , Alain Giorgetti‡† , Nicolas Stouls§¶

Thème SYM — Systèmes symboliques
Équipes-Projets ProVal et CASSIS

Rapport de recherche n° 6702 — Octobre 2008 — 22 pages

Abstract: In the verification of C programs by deductive approaches based on automated provers, some heuristics of separation analysis are proposed to handle the most difficult problems. Unfortunately, these heuristics are not sufficient when applied on industrial C programs: some valid verification conditions cannot be automatically discharged by any automated prover mainly due to their size and a high number of irrelevant hypotheses.

This work presents a strategy to reduce program verification conditions by selecting their relevant hypotheses. The relevance of a hypothesis is the combination of separated static dependency analyzes based on graph constructions and traversals. The approach is applied on a benchmark issued from industrial program verification.

Key-words: Program verification, proof, hypothesis selection

* This work is partially funded by the French Ministry of Research, thanks to the CAT (C Analysis Toolbox) RNTL (Reseau National des Technologies Logicielles), and by the SYSTEM@TIC Paris Region French cluster, thanks to the PFC project (Plateforme de Confiance, trusted platforms).

† LIFC, University of Franche-Comté, 16 route de Gray, Besançon, F-25030
‡ INRIA Nancy - Grand Est, CASSIS project
§ INRIA Saclay, Île-de-France, ProVal, Parc Orsay Université, F-91893
¶ LRI, Univ Paris-Sud, CNRS, Orsay, F-91405

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université
4, rue Jacques Monod, 91893 ORSAY Cedex
Téléphone : +33 1 72 92 59 00

Réduction de conditions de vérification de programmes par graphes

Résumé : Diverses heuristiques de séparation d'hypothèses facilitent l'approche déductive de la vérification de programmes C, dans la perspective d'utiliser un prouveur automatique. Malheureusement, ces heuristiques ne suffisent pas pour vérifier des programmes C issus de l'industrie : certaines conditions de vérification valides ne sont établies par aucun prouveur automatique, en raison de leur trop grande taille et d'un trop grand nombre d'hypothèses non pertinentes.

Ce travail présente une stratégie pour réduire les conditions de vérification de programmes par la sélection d'hypothèses pertinentes. La pertinence d'une hypothèse résulte de la combinaison de deux analyses statiques de dépendance, basées sur la construction et le parcours de graphes. Cette approche est appliquée à une étude de cas issue du monde industriel.

Mots-clés : Vérification de programme, preuve, sélection d'hypothèses

Contents

1 Introduction

Deductive software verification aims at verifying program properties with the help of theorem provers, by static analysis of program source code. It has gained more interest with the increased use of software embedded in, for instance, plane commands, cars or smart cards, requiring a high-level of confidence, but with hardware constraints that limit the usability of a constructive formal approach, such as refinement.

In the Hoare logic framework, program properties are expressed by first-order logical assertions on program variables (preconditions, postconditions, invariants, ...). The deductive verification method consists in transforming a program, annotated with sufficiently many assertions, into so-called *verification conditions* (VCs) that, when proved, establish that the program satisfies its assertions. This method is supported by effective tools such as ESC/Java [13], a toolkit for Java programs annotated using the Java Modeling Language [6], Boogie [1] for the C# programming language, and Caduceus/Why [16] for C programs.

A theorem prover is invoked to establish the validity of each verification condition. A challenge in deductive software verification is to automatically discharge as many verification conditions as possible. A key issue is that the whole context of a verification condition is a huge set of axioms modelling not only the property and the program under verification, but also many features

of the programming language. Naively passing this large context to an automated prover induces a combinatorial explosion, preventing the prover from terminating in reasonable time.

Possible solutions to reduce the VC size and complexity are to optimize the memory model (e.g. by introducing separations of zones of pointers [19]), to improve the weakest precondition calculus [20] and to apply strategies for simplifying VCs [18, 11, 21]. This work focuses on the latter. We suggest heuristics to select axioms to feed to automated theorem provers (ATPs). Instead of invoking ATPs blindly with a large VC, we present reduction strategies that significantly prune their search space. The idea behind these strategies is quite natural: an axiom is relevant if it contains predicates and constants needed to establish the conclusion. Relevance criteria are computed by the combined traversal of two graph-based representations of dependencies between axioms and a conclusion. On one hand, a graph of constants analyzes similarities in occurrences of constants between the conclusion and each hypothesis. On the other hand, a graph of predicates analyzes logical dependencies between predicates in the conclusion and predicates either in these program-dependent hypotheses or in a global theory of the programming language.

In a former work [9], selection was limited to ground hypotheses and comparison predicates were not taken into account. This led to unsatisfactory results, for instance when the conclusion is some equality between terms. The present work extends selection to context axioms and to comparison predicates. Here we propose new heuristics increasing the number of automatically discharged VCs. The following methodology made it possible to identify relevant heuristics and thereby to define an efficient hypothesis selection algorithm:

1. Starting from an unproved VC, try to validate it under the Coq [2] proof assistant;
2. Determine the minimal subset of needed hypotheses;
3. Search how these hypotheses are linked to the conclusion;
4. Search some criteria to reject the largest possible amount of hypotheses that are not needed;
5. Experiment the time needed by different provers to automatically discharge this VC reduced by hypothesis rejection.

The plan of the article is as follows. Section 2 presents the general structure of a verification condition. Section 3 presents a running example. Section 4 shows how we store dependencies in graphs. The selection of hypotheses is then presented in Section 5. These last two sections are the first contribution. The second contribution is the implementation of this strategy as a module of Caduceus/Why [16] and its application to an industrial C example (Section 6). This case study is a part of the Oslo [4] secure bootloader annotated with a safety property. Section 7 discusses related work, concludes and presents future work.

2 Verification Conditions

A verification condition (VC) is a first-order logic formula whose validity implies that a piece of annotated source code satisfies some soundness property. This section describes the general structure of VCs generated by Caduceus/Why and a preprocessing which rewrites VCs into a normal form considered in the rest of the paper. A VC is composed of a *context* and a *goal*. This structure is illustrated in Fig. 1.

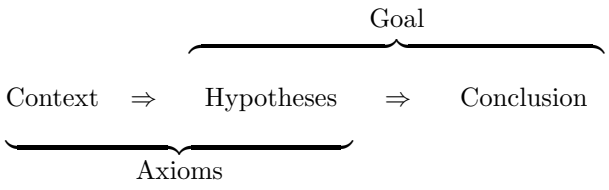

Figure 1: Structure of verification conditions

The context depends on the programming language. It is in fact a first-order axiomatization of the language features used in the program under verification. Typical features are memory access and update, based on an axiomatized memory model. For SMT solvers, the context is presented as a base theory, usually a combination of equality with uninterpreted function symbols and linear arithmetic, extended with a large set of specific axioms. For instance, a classical VC produced by Caduceus/Why has a context with more than 80 axioms.

The goal depends on the program and on the property under verification. When this property is an assertion about some program control point, the goal is generated by the weakest precondition (wp) calculus of Dijkstra [14] at that control point. The goal is considered as a *conclusion* implied by *hypotheses* that encode the program execution up to the control point.

Context and hypotheses are reduced to conjunctive normal form as two conjunctions $Ctx_1 \wedge \ldots \wedge Ctx_n$ and $H_1 \wedge \ldots \wedge H_m$ of clauses ($n, m \in \mathbb{N}$). Each of these clauses is called an *axiom*. Each axiom from the context is assumed to be universally closed. The conclusion C is assumed to be a clause. For SMT solvers, a VC is turned into the satisfiability question $Ctx_1 \wedge \ldots \wedge Ctx_n \wedge H_1 \wedge \ldots \wedge H_m \wedge \neg C$.

3 Running Example

The C program in Figure 2 is the starting point of the running example that illustrates the approach taken throughout the following sections. The left side column of this C program introduces matryoshka structures. The right side column presents an interface `g` and a function `f` that calls `g` and explicitly modifies the value stored in one of the innermost fields of the structure pointed by `a` (namely `a->y->v[1].x`).

Functions are annotated in the Caduceus/Why language [16] composed of

```
struct p {
 int x;
} p;

struct s {
 struct p v[2];
} s;

struct t {
 struct s *y;
} t;


/*@ requires \valid(c)
  @ assigns c->v[0].x */
void g(struct s *c);

/*@ requires \valid(a)
  @       && \valid(b)
  @       && \valid(a->y)
  @ as-
signs a->y->v[0..1].x */
void f(struct t *a,struct p *b){
 int i = b->x;
 g(a->y);
 a->y->v[1].x=i;
}
```

Figure 2: A running example in C

- preconditions (defined by the `requires` keyword); they ensure that each pointer given in parameter is `\valid` (i.e. is correctly allocated) when `f` or `g` is invoked;

- a list of data modified by side effects (defined by the `assigns` keyword); for instance `assigns a->y->v[0..1].x` means that function `f` does not modify other locations than `a->y->v[0].x` and `a->y->v[1].x`; this property can be established by considering the side effects of `f` body and of `g`.

This example is representative of programs with pointers for which the absence of threats is hard to check statically. Threats include null pointer dereferencing and out-of-bounds array access. For these programs, Caduceus/Why yields two kinds of VCs. Validity VCs express that pointers point to a regularly allocated memory block at each memory access. Effect VCs express that all the side effects of a function are in its annotated list of side effects. For instance, the statement `g(a->y)` constrains `a` to be valid for a legal access to its pointed memory block and `a->y` to be valid to satisfy the precondition of `g`.

We apply the Burstall-Bornat memory model [7, 5] where one 'array' variable (later called a memory) models each structure field. This modeling syntactically encodes the fact that two structure fields cannot be aliased. An important consequence is that whenever one field is updated, the corresponding array is the only one which is modified. Hence, we have for free that any other field is left unchanged. In the example, the fields `x`, `v` and `y` respectively yield the memories m_x, m_v and m_y.

Memories can be accessed only by the `acc` function. $\mathsf{acc}(m, p)$ returns the value stored in memory m at index p, where p is a pointer. A fresh memory can be generated by the `upd` function. $\mathsf{upd}(m, p, v)$ duplicates m except at pointer p where it sets the value v. Caduceus' VC generator yields predicates `valid` and `diff` which have the semantic of `\valid` and `assigns` respectively. Let m_1 and m_2 be two memories and l be a set of pointers. The predicate `diff` is such that $\mathsf{diff}(m_1, m_2, l)$ means that differences between m_1 and m_2 only concern the set

l. The predicate can be defined by

$$\mathsf{diff}(m_1, m_2, l) \Leftrightarrow \big(\forall p\,.\,\mathsf{valid}(p) \wedge \neg\mathsf{mem}(p, l) \Rightarrow \mathsf{acc}(m_1, p) = \mathsf{acc}(m_2, p)\big) \quad (1)$$

where p is a pointer and $\mathsf{mem}(p, l)$ means that p is a member of l. This formula is rewritten in CNF as the conjunction

$$Ctx_1 \wedge Ctx_2 \wedge Ctx_3 \wedge Ctx_4 \quad (2)$$

of the following four clauses:

$$\neg\mathsf{diff}(M_1, M_2, L) \vee \neg\mathsf{valid}(P) \vee \mathsf{mem}(P, L) \vee \mathsf{acc}(M_1, P) = \mathsf{acc}(M_2, P) \quad (Ctx_1)$$

$$\mathsf{valid}(p_0) \vee \mathsf{diff}(M_1, M_2, L) \quad (Ctx_2)$$

$$\neg\mathsf{mem}(p_0, L) \vee \mathsf{diff}(M_1, M_2, L) \quad (Ctx_3)$$

$$\mathsf{acc}(M_1, p_0) \neq \mathsf{acc}(M_2, p_0) \vee \mathsf{diff}(M_1, M_2, L) \quad (Ctx_4)$$

where capitalized variables are universally quantified and p_0 is a fresh constant resulting from the skolemization of p. These clauses are four of the 80 clausal axioms that compose the context (memory model) of Caduceus/Why, denoted Ctx in all that follows.

The effect VC for function f is the conjunction

$$Ctx \wedge H_1 \wedge H_2 \wedge H_3 \wedge H_4 \wedge H_5 \wedge H_6 \wedge \neg C \quad (3)$$

of the context Ctx and of the following ground formulas:

$$\mathsf{valid}(a) \quad (H_1)$$

$$\mathsf{valid}(b) \quad (H_2)$$

$$\mathsf{valid}(\mathsf{acc}(m_y, a)) \quad (H_3)$$

$$\mathsf{valid\_acc\_range}(m_v, 2) \quad (H_4)$$

$$\mathsf{separation1\_range}(m_v, 2) \quad (H_5)$$

$$\mathsf{diff}(m_x, m_x\_0, \mathsf{singleton}(\mathsf{shift}(\mathsf{acc}(m_v, \mathsf{acc}(m_y, a)), 0))) \quad (H_6)$$

$$\neg\,\mathsf{diff}(m_x, \mathsf{upd}(m_x\_0, \mathsf{shift}(\mathsf{acc}(m_v, \mathsf{acc}(m_y, a), 1)), \mathsf{acc}(m_x, b)), \mathsf{range}(\mathsf{singleton}(\mathsf{acc}(m_v, \mathsf{acc}(m_y, a))), 0, 1)) \quad (\neg C)$$

The meaning of these formulas is the following: The first three clauses correspond to the precondition of the `f` function. The next two clauses come from the definition of structure `s`: the predicate $\mathsf{valid\_acc\_range}(m_v, 2)$ means that any access to the memory m_v returns an array t such that pointers $t[0]$ and $t[1]$ are

valid; with the same notation separation1_range$(m_v, 2)$ means that $t[0] \neq t[1]$. The sixth clause (H_6) comes from the statement `g(a->y)`, more precisely from the annotation `assigns c->v[0].x` where `c` is substituted by `a->y`. The function singleton has the usual meaning and shift(t, i) allows to access to the index i in the array t. This hypothesis defines the access of a variable m_x_0 which is equal to the access in m_x except for the index shift(acc$(m_v, \text{acc}(m_y, a)), 0)$ corresponding to `a->y->v[0]`. The conclusion C is a diff predicate applied to two memories. The first memory is the memory before execution of `f` and the second memory is the memory after execution of `f`. The third parameter range(singleton(acc$(m_v, \text{acc}(m_y, a))), 0, 1)$ defines the set of pointers located at indices 0 and 1 in the array acc$(m_v, \text{acc}(m_y, a))$. In fact this is the representation of `a->y->v[0..1]`.

Even if this example is small, Simplify and haRVey are the sole SMT provers, among Simplify [12], Yices [15], Alt-Ergo [8], haRVey [24] and CVC-lite [3], which succeed in establishing the validity of this VC (3) in a few seconds.

However, an engineer would have deduced from the background theory that valid and valid_acc_range are not directly useful in the present case. Removing the hypothesis concerning valid_acc_range permits him to obtain the desired result.

The next section shows how dependencies are memorized in the problem of proving a goal in a SMT solver. This is the starting point of the approach of removing useless axioms.

4 Graph Based Memorizing of Dependency

Basically, a conclusion is a propositional combination of potentially quantified predicates built with some functional terms. Dependencies between axioms and the conclusion can then arise from predicates and terms. Terms in the goal may either come from the annotated program (from statements or assertions) or may result from a weakest precondition calculus applied on the program and its assertions. The term dependency just transcribes that parts of the goal (in particular, hypotheses and conclusion) share common terms. It is presented in Section 4.1. Two predicates are dependent if there exists a (deductive) path leading from one to the other. The predicate dependency is presented in Section 4.2. Finally, Section 4.3 presents a special dependency analysis for comparison predicates.

4.1 Term Dependency

An undirected graph G_c is constructed by syntactic analysis of term occurrences in each ground clausal hypothesis of a VC. The graph describes how these hypotheses relate terms together. The graph vertices are labeled with the constants occurring in the goal and with new constants resulting from a flattening process on ground clauses. This flattening repeatedly replaces a functional term $f(t_1, \ldots, t_n)$ in some ground clausal hypothesis with a fresh constant f_i where i is some unique integer. The resulting conjunction of clauses augmented with the new unitary clause $f(t_1, \ldots, t_n) = f\_i$ is equisatisfiable with the former conjunction. Finally, all these ground clausal hypotheses are "flat", meaning that all their parameters are constants. There is a graph edge between two (vertices labeled with) constants c_1 and c_2 when c_1 and c_2 both appear is

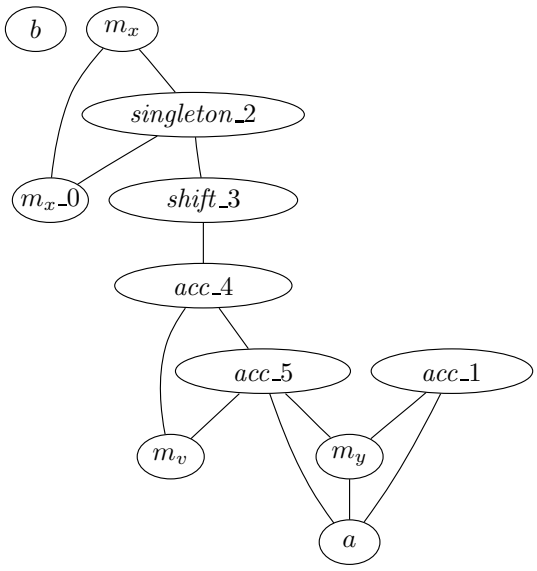

Figure 3: Dependency Graph of Verification Condition (3)

some flat hypothesis. Consequently, each flat hypothesis appears in the graph as a complete subgraph. Notice that flattening and edge drawing do not concern the VC conclusion: its constants will be used to initiate a graph traversal for selecting hypotheses. For this purpose, adding new constants in the conclusion by flattening has no interest.

Running example. The graph representing verification condition (3) is given in Fig. 3. The flattening of hypothesis (H_6) introduces the fresh constants *singleton*_2, *shift*_3, *acc*_4 and *acc*_5 and the flat hypotheses resulting from H_6 are represented by the complete subgraphs defined by the sets $\{m_x,\ m_x\_0,$ *singleton*_2$\}$, $\{$*singleton*_2, *shift*_3$\}$, $\{$*shift*_3, *acc*_4$\}$, $\{$*acc*_4, *acc*_5$, m_v\}$ and $\{$*acc*_5, $m_y, a\}$. Similarly, (H_3) introduces the fresh constant *acc*_1 and is represented by the set $\{$*acc*_1, $m_y, a\}$.

4.2 Predicate Dependency

A weighted directed graph is constructed to represent implication relations between predicates in an efficient way. Intuitively, each graph vertex represents a predicate name and an arc from a vertex p to a vertex q means that p may imply q. What follows details how to compute such a graph of predicates, named G_P. We first describe the general approach. Then we propose a special treatment for comparison operators.

Each graph vertex is labeled with a predicate symbol that appears in at least one literal of the theory. If a predicate p appears negated (as $\neg p$) in an axiom clause, it is represented by a vertex labeled with $\overline{p}$. For each clause Cl and for each pair $(p, q) \in Cl \times Cl$ of literals in this clause (considered as a set of literals), there is an arc in G_P depending on the positiveness of p and q. Modulo symmetry there are three distinct cases to consider. They are enumerated in

Table 1. To reduce the graph size, the contraposite of each implication is not represented as an arc in the graph but is considered when traversing it, as detailed in Section 5.1.

The intended meaning of an arc weight is that the lower the weight is, the higher is the probability to establish q from p. Therefore, the arc introduced for the pair (p, q) along Table 1 is labeled with the number $card(Cl) - 1$ of predicates in the clause Cl under consideration. For instance, a large clause with many negative literals, with $\neg p$ among them, and with many consequents, with q among them, is less useful for a deduction step leading to q than the smaller clause $\{\neg p, q\}$. Finally, two weighted arcs $p \xrightarrow{w_1} q$ and $p \xrightarrow{w_2} q$ are replaced with the weighted arc $p \xrightarrow{min(w_1,w_2)} q$.

	Pair	Arcs
Pure predicates	$(\neg p, q)$	$\{p \longrightarrow q\}$
	(p, q)	$\{\overline{p} \longrightarrow q\}$
	$(\neg p, \neg q)$	$\{p \longrightarrow \overline{q}\}$

Table 1: Translating Pair of Literals into Arcs.

4.3 Handling Comparison Predicates

In a former work [9], (in)equalities were abstracted in the step of memorizing dependency. This leads to unsatisfactory results when (in)equality is central in deduction steps, e.g. when the conclusion is some equality between terms.

One may handle each comparison predicate as all the other binary predicates and for instance memorize an equality with a node labeled with $=$. Nevertheless, since comparisons are ubiquitous in axioms, this solution would produce a graph where nodes labeled with $=$, $\neq$, $\leqslant$, $<$, $\geqslant$ or $>$ have a huge number of arcs. Consequently it allows to abusively link predicates, leading to losing the semantic of the graph.

The objective of the predicate graph is to memorize logical paths between predicates. It could be of great benefit to make it memorizing the transitivity property of comparison predicates. In doing so, the comparison operator itself is less important than its operands. Consequently, we generate specific nodes for comparison literals, annotated with the toplevel symbol of one of their operands.

Practically, each comparison $t_1 = t_2$, $t_1 \neq t_2$, $t_1 \leqslant t_2$, $t_1 < t_2$, $t_1 \geqslant t_2$ and $t_1 > t_2$ is represented by two nodes labeled with $\diamond_{f_1}$ and $\diamond_{f_2}$ where f_1 (resp. f_2) is the functional symbol at the top of t_1 (resp. t_2). These labels are called *post-fixed* comparison predicates. Again, arcs are constructed modulo symmetry, for each pair $(l, t_1 \circ t_2) \in Cl \times Cl$, where $\circ$ is a comparison symbol in the set $Comp = \{=, \neq, \leqslant, <, \geqslant, >\}$ and l is a literal. The construction rules are given in Table 2. In the first two lines, p is not a comparison predicate. In this case, an arc is constructed in the same way as for pure predicates, but with $\diamond_{f_1}$ or $\diamond_{f_2}$ instead of q. Two additional arcs relate $\diamond_{f_1}$ and $\diamond_{f_2}$. The last line defines how pair of comparisons are represented. Since $\diamond_{f_1}$ and $\diamond_{f_2}$ uniformly represent all the comparison predicates between t_1 and t_2, there is no distinction between $\diamond_{f_1}$ and $\overline{\diamond_{f_1}}$.

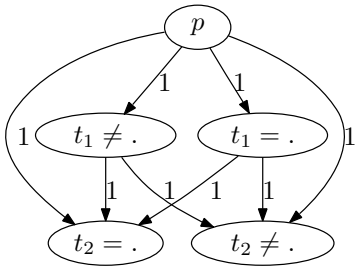

Figure 4: Abstract representation of $\{\neg p, t_1 \circ t_2\}$

The semantics of these nodes is detailed now. Given a comparison predicate $\circ \in \mathit{Comp}$, two functional terms t_1 and t_2, and a set L of l literals, the clause

$$L \cup \{t_1 \circ t_2\} \tag{4}$$

is equisatisfiable with $L \cup \{t_1 = c_1 \wedge t_2 = c_2 \wedge c_1 \circ c_2\}$ where c_1 and c_2 are two fresh variables. The weaker clause $L \cup \{t_1 = c_1, t_1 \neq c_1, t_2 = c_2, t_2 \neq c_2\}$ is memorized. It abstracts away relations between c_1 and c_2. Nodes and arcs of the predicate graph are computed following lines of Section 4.2. Notice weights are calculated from the original equation, not from the weaker clause. For instance, the graph of Fig. 4 represents the clause (4) through this abstraction where L is $\{\neg p\}$.

Pair	Arcs
$(\neg p, f_1(\ldots) \circ f_2(\ldots))$	$\{p \longrightarrow \diamond_{f_1}, p \longrightarrow \diamond_{f_2}, \diamond_{f_1} \longrightarrow \diamond_{f_2}, \diamond_{f_2} \longrightarrow \diamond_{f_1}\}$
$(p, f_1(\ldots) \circ f_2(\ldots))$	$\{\overline{p} \longrightarrow \diamond_{f_1}, \overline{p} \longrightarrow \diamond_{f_2}, \diamond_{f_1} \longrightarrow \diamond_{f_2}, \diamond_{f_2} \longrightarrow \diamond_{f_1}\}$
$(g_1(\ldots) \odot g_2(\ldots),$ $f_1(\ldots) \circ f_2(\ldots))$	$\{\diamond_{g_1} \longrightarrow \diamond_{f_1}, \diamond_{g_1} \longrightarrow \diamond_{f_2}, \diamond_{g_2} \longrightarrow \diamond_{f_1}, \diamond_{g_2} \longrightarrow \diamond_{f_2},$ $\diamond_{f_1} \longrightarrow \diamond_{f_2}, \diamond_{f_2} \longrightarrow \diamond_{f_1}, \diamond_{g_1} \longrightarrow \diamond_{g_2}, \diamond_{g_2} \longrightarrow \diamond_{g_1}\}$

where $\circ$ and $\odot$ are comparison operators in *Comp* and
p is a literal that is not a comparison.

Table 2: Translating Comparisons into Arcs.

Finally, both $t_1 = .$ and $t_1 \neq .$ are represented by $\diamond_{f_1}$. Consequently, $\diamond_{f_1}$ and $\overline{\diamond_{f_1}}$ are the same node, and then not represented twice.

Notice that, given $\circ \in \mathit{Comp}$, t_1 and t_2 two functional terms, and L a set of literals, another attempt could consist in translating the clause $L \cup \{t_1 \circ t_2\}$ into the three clauses $L \cup \{t_1 = c_1\}$, $L \cup \{t_2 = c_2\}$ and $L \cup \{c_1 \circ c_2\}$ where c_1 and c_2 are two fresh constants, whilst preserving satisfiability. Later each equality $t_i = c_i$ between a functional term and a constant and each comparison $c_1 \circ c_2$ between two constants could be represented by a node labeled with $\circ_{t_i}$ (as before but more precisely) or directly by a node labeled with $\circ$ respectively. However this careful approach that allows to consider comparison as other uninterpreted predicates leads to a combinatorial explosion of clauses. In what follows, we only retain the translation from comparison to *post-fixed* comparison $\diamond_{\cdot}$.

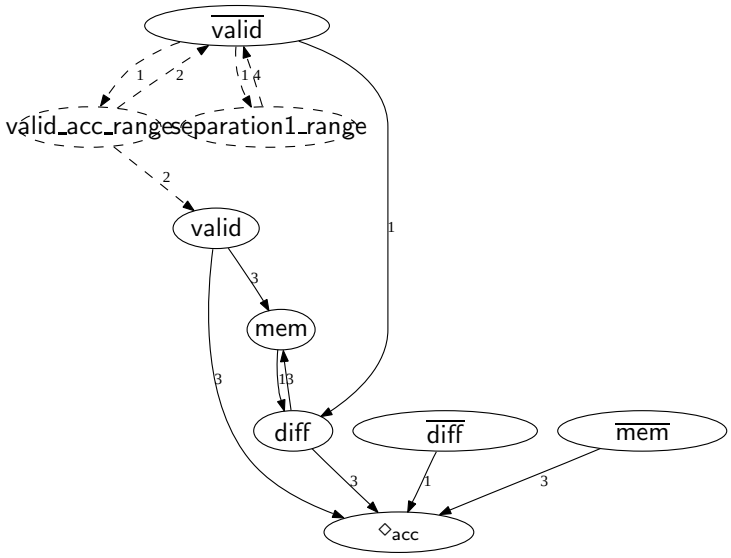

Figure 5: Dependency Graph of Axiom (1), with Post-Fixed Comparison Predicates and Some Vertices and Arcs From Other Axioms

Running example. Figure 5 represents the dependency graph of the conjunction of clausal axioms (2), with a post-fixed comparison predicate for the equality between two $\mathsf{acc}(\ldots)$ terms. The dashed vertices represent the other predicates from the verification condition (3). The dashed arcs represent some links between these predicates and those of (2), extracted from other axioms of the Caduceus/Why context. These links are thus an excerpt of the graph representing the memory model of Caduceus/Why. Notice that the arc weight from $\overline{\mathsf{valid}}$ to diff is 1 since $(\neg\mathsf{diff}, \neg\mathsf{valid})$ appears in the clause (Ctx_1) with 4 literals and $(\mathsf{valid}, \mathsf{diff})$ appears in the clause (Ctx_2) with 2 literals.

5 Axiom Selection

It remains to select *relevant* axioms. Intuitively, an axiom is relevant with respect to a conclusion if that conclusion cannot be established without this axiom. More formally, let $A_1 \wedge \ldots \wedge A_n \wedge \neg C$ be a satisfiability problem. The axiom A_1 is relevant w.r.t. C if $A_1 \wedge \ldots \wedge A_n \wedge \neg C$ is not satisfiable whereas $A_2 \wedge \ldots \wedge A_n \wedge \neg C$ is.

This section shows how to select relevant predicates (Section 5.1), relevant variables (Section 5.2) and explains how to combine these results to select relevant axioms (Section 5.3).

5.1 Relevant Predicates

In what follows, we do not distinguish a predicate symbol from its corresponding vertex in the graph of predicates G_P. A predicate symbol p is relevant w.r.t. a predicate symbol q if there is a path from p to q in G_P, or dually from $\overline{q}$

to $\overline{p}$. Intuitively, the weaker the path weight is, the higher is the probability of p to establish q. Relevant predicates extracted from G_P are stored into an increasing sequence $(\mathcal{L}_n)_{n\in\mathbb{N}}$ of sets gathering predicates helping to establish predicates of the conclusion. The natural number n is the maximal weight of paths considered in the graph of predicates.

We now present how $\mathcal{L}_n$ is computed. $\mathcal{L}_0$ gathers all predicates of the conclusion, assumed to be a single clause. For each predicate symbol p that is not in $\mathcal{L}_0$, a graph traversal computes the paths with the minimal weight w from p to some predicate in $\mathcal{L}_0$.

Furthermore, contraposition of each implication is considered: let p_1 and p_2 be two node labels, corresponding either to a positive or a negative literal. If the arc $p_1 \xrightarrow{w} p_2$ is taken into account, its couterpart $\overline{p_2} \longrightarrow \overline{p_1}$ is too, with the convention that $\overline{\overline{p}}$ is p. Let n be the minimal distance from $\mathcal{L}_0$ to the deepest reachable predicate. For $1 \leqslant i \leqslant n$, $\mathcal{L}_i$ is the set of vertices of G_P whose distance to $\mathcal{L}_0$ is less than or equal to i. $\mathcal{L}_\infty$ is the limit $\bigcup_{i\geq 0} \mathcal{L}_i$ augmented with the vertices from which $\mathcal{L}_0$ is not reachable.

Notice that if the VC is a propositional formula, the set of all predicates of the fixpoint is the set of *hypotheses with polarized connectivity* [18].

Running example. According to the graph given in Fig. 5, we have:

$$
\begin{array}{lcl}
\mathcal{L}_0 & = & \{\mathsf{diff}\} \\
\mathcal{L}_1 & = & \mathcal{L}_0 \cup \{\mathsf{mem}, \diamond_{\mathsf{acc}}, \overline{\mathsf{valid}}\} \\
\mathcal{L}_2 & = & \mathcal{L}_1 \cup \{\overline{\mathsf{diff}}\} \\
\mathcal{L}_3 & = & \mathcal{L}_2 \cup \{\mathsf{valid\_acc\_range}\} \\
\mathcal{L}_4 & = & \mathcal{L}_3 \cup \{\mathsf{valid}, \overline{\mathsf{mem}}\} \\
\mathcal{L}_5 & = & \mathcal{L}_4 \cup \{\mathsf{separation1\_range}, \overline{\mathsf{separation1\_range}}, \overline{\mathsf{valid\_acc\_range}}\} \\
\ldots & & \\
\mathcal{L}_\infty & = & \bigcup_{i\geq 0} \mathcal{L}_i \cup \ \textit{unreachable vertices}
\end{array}
$$

Notice that the spurious predicate $\mathsf{valid\_acc\_range}$ only appears at depth 3, allowing all the deductions concerning predicates obtained with a depth less or equal to 2 to be more easily discharged.

5.2 Relevant Constants

Nodes in the graph of constants G_c are identified with their labeling constant. Let n be the diameter of the graph of constants G_c. Starting from the set $\mathcal{C}_0$ of constants in the conclusion, a breadth-first search algorithm computes the sets $\mathcal{C}_i$ of constants in G_c that are reachable from $\mathcal{C}_0$ with at most i steps ($1 \leqslant i \leqslant n$). Finally, unreachable constants are added to the limit of the sequence $\left(\mathcal{C}_n\right)_{n\in\mathbb{N}}$ for completeness. Let $\mathcal{C}_\infty$ be the set so obtained.

To introduce more granularity in the calculus of reachable constants, we propose as a heuristic to insert nodes that are linked several times before nodes that are just linked once. Semantically it gives priority to constants which are closer to the conclusion. Notice that, in this case, the index i of $\mathcal{C}_i$ does not correspond to a path length anymore.

Running example. According to Fig. 3, the sequence of reachable constants sets associated to the verification condition (3) is:

$$\begin{array}{lcl} \mathcal{C}_0 & = & \{m_x, m_x\_0, m_v, m_y, a, b\}, \\ \mathcal{C}_1 & = & \mathcal{C}_0 \cup \{acc\_5\}, \\ \mathcal{C}_2 & = & \mathcal{C}_1 \cup \{acc\_1, singleton\_2\}, \\ \mathcal{C}_3 & = & \mathcal{C}_2 \cup \{acc\_4\}, \\ \mathcal{C}_4 & = & \mathcal{C}_3 \cup \{shift\_3\} \text{ and} \\ \mathcal{C}_\infty & = & \mathcal{C}_4. \end{array}$$

5.3 Selection of Relevant Axioms

In this section, we present the main principles of the axiom selection combining predicates selection and constant selection.

Suppose given the sequences $(\mathcal{L}_n)_{n\in\mathbb{N}}$, $\left(\mathcal{C}_n\right)_{n\in\mathbb{N}}$ respectively of relevant predicate sets and of relevant constant sets. Let i be the counter which represents the depth of predicate selection. Similarly, let j be the counter corresponding to the depth of constant selection.

In a first part we describe the hypotheses selection and in the second one we extend the approach to consider also axioms from the context.

Suppose given a clause Cl from a hypothesis. Let V be the set of constants of Cl augmented with constants resulting from flattening (see Section 4.1). Let P be the set of predicates of Cl. The clause Cl should be selected if it includes constants or predicates that are relevant according to the conclusion. Different criteria can be used to verify this according to its sets P and V. Possible choices are, in increasing order of selectiveness

1. the clause includes at least one relevant constant or one relevant predicate:

$$V \cap \mathcal{C}_j \neq \emptyset \ \vee \ P \cap \mathcal{L}_i \neq \emptyset$$

2. the clause includes more than a threshold t_v of relevant constants or more than a threshold t_p of relevant predicates:

$$card(V \cap \mathcal{C}_j)/card(\mathcal{C}_j) \geqslant t_v \ \wedge card(P \cap \mathcal{L}_i)/card(\mathcal{L}_i) \geqslant t_p$$

3. all the clause constants and clause predicates are relevant:

$$V \subseteq \mathcal{C}_j \ \wedge \ P \subseteq \mathcal{L}_i$$

Our experiments on these criteria have shown that a too weak criterion does not accomplish what it is designed for: too many clauses are selected for few iterations, making the prover quickly diverge. In the following, we only consider the strongest criterion.

Consider now the case of selecting relevant axioms from the context. The semantic of context axioms that defines algebraic structures is distinct from the semantic of hypotheses, that defines the execution of the program. Hence, selecting axioms cannot be uniformly expressed for these both cases.

Intuitively, an axiom of the context has to be selected if one of the predicate relations it defines is relevant for one hypothesis, i.e. the corresponding arc is used in the computation of $\mathcal{L}_i$. Practically, for each arc that is passed through

while generating $\mathfrak{L}_i$, we keep all the axioms of the context that have generated this arc, abstracting away weights.

Consider a formula resulting from the selection of axioms (from context or hypotheses) according to the strongest criterion. Discharging it into a prover with the following algorithm, and starting from $i = 0$ and $j = 0$, can yield three issues: satisfiable, unsatisfiable or timeout.

1. If the formula is declared to be unsatisfiable, the procedure ends. Adding more axioms cannot make the problem satisfiable.

2. If the formula is declared to be satisfiable, we may have omitted some axioms; we are then left to increment either i or j, i.e. to enlarge either the set of selected predicates or the set of selected constants.

   However, divergence appears when the generation of new literals by a set of axioms falls in a bottomless pit. Such a generation is controlled by the presence in the formula of predicates of incriminated axioms. Given a set of predicates and a set of constants, allowing the use of new predicates has a more critical impact than allowing the use of new constants.

   Therefore we recommend to first increment j, increasing $\mathcal{C}_j$ until eventually $\mathcal{C}_\infty$, before considering incrementing i. In this later case, j resets to 0.

3. If the formula is not discharged in less than a given time, after having iteratively incremented i and j, then the approach halts.

Running example. For $\mathfrak{L}_0$, no axiom is selected with $\mathcal{C}_0$, $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{C}_3$. However with $\mathcal{C}_4$, it yields the VC:

$$\begin{array}{l}\mathsf{diff}(m_x, m_x\_0, \mathsf{singleton}(\mathsf{shift}(\mathsf{acc}(m_v, \mathsf{acc}(m_y, a)), 0)))\wedge \\ \neg(\mathsf{diff}(m_x, \mathsf{upd}(m_x\_0, \mathsf{shift}(\mathsf{acc}(m_v, \mathsf{acc}(m_y, a), 1)), \mathsf{acc}(m_x, b)), \\ \qquad \mathsf{range}(\mathsf{singleton}(\mathsf{acc}(m_v, \mathsf{acc}(m_y, a))), 0, 1)))\end{array} \tag{5}$$

This VC does not contain the axiom with valid_acc_range, which causes provers to diverge. In addition to Simplify and haRVey that already discharged the original VC, Alt-Ergo and Yices run successfully on VC (5).

6 Experiments

The proposed approach is included in a global context of annotated C program certification. T. Hubert and C. Marche [19] proposed a separation analysis that strongly simplifies the verification conditions generated by a weakest precondition calculus, and thus greatly helps to prove programs with pointers. Their approach is supported by the Why tool. The pruning heuristics presented here are developed as a post-process of this tool.

Section 6.1 gives some details about algorithms developed following the lines of this paper. A case study for trusted computing is presented in Section 6.2. This case study raises new challenges associated to the certification of C programs annotated with a temporal logic formula.

```
Method parameters : VC, Prover, i_max, j_max, TO

  T_n = TO/((i_max + 1) * j_max + 1)
  Res := Prover(VC, timeout = T_n, without axiom selection)
  if Res = timeout then
    i := 0;
    j := 0;
    While (Res ≠ unsat) ∧ i ⩽ i_max ∧ j ⩽ j_max do
      Res := Prover(VC, timeout = T_n, axiom selection=(i, j))
      j := j + 1;
      if j > j_max then
        i := i + 1;
        j := 0;
  return Res;
```

Figure 6: General Algorithm Discharging a VC with Axiom Selection

6.1 Methodology

All the strategies presented in this work are implemented in OCaml as modules in the Why [16] tool in less than 2000 lines of code. Since these criteria are heuristics, their use is optional, and Why has command line arguments which allow a user to enable or disable their use. In the current version, several others heuristics have been developed, which we do not consider in the following as their impact on the performance of Why seems to be less pronounced than the one of the heuristics presented above. In order to use them, the arguments to include in the Why call are:

```
--prune-with-comp --prune-context --prune-vars-filter CNF
```

The first parameter includes comparison predicates in the predicate dependency graph. The second one asks for filtering also axioms from context (not only hypotheses). Finally, the third argument asks for rewriting hypotheses into CNF before filtering.

Internally, the last method described in Section 5.3 is implemented as shown in Fig. 6. The tool needs 5 parameters:

- a *VC* whose satisfiability has to be checked,
- a satisfiability solver *Prover*,
- two natural numbers i_{max} and j_{max} that are depth bounds for predicate graph and constant graph traversals, respectively, and
- the maximal amount of time *TO* allowed by the user to discharge a VC.

As announced in Section 5.3, three issues (*satisfiable*, *unsat* or *timeout*) can arise while discharging a VC. The last case denotes that the VC is not discharged in less than a given time, after which the prover is halted.

The global timeout TO is divided into $n = (i_{max}+1)*j_{max}+1$ equal timeouts T_n granted to each prover call. The experiment starts with a first attempt to discharge the VC without axiom selection. The algorithm stops if this first result is unsatisfiable or satisfiable: in the latter case, removing axioms cannot modify the result provided. The implemented decision procedure is complete. Next, *Prover* is called following an incremental constant-first selection.

Notice that the whole experiment is done on an Intel Xeon 3.20GHz with 2Gb of memory.

6.2 Trusted Computing Case Study

Some new challenges for axiom filtering are posed by the context of the PFC project[1] on Trusted Computing (TC). Section 6.2.1 presents the context of TC. Corresponding verification results are given in Section 6.2.2.

6.2.1 Motivations and Context

The main idea of the TC approach is to have some confidence about the execution context of a program. This confidence is obtained by construction, by using a *trusted chain*. A trusted chain is a chain of program executions where each program launched is previously registered with a trusted component (in our case, this component is a hardware chipset named TPM – Trusted Platform Module). The trusted component classically provides some services, such as *sealed memory* (e.g. for private cryptographic keys) and *remote attestation* (to identify which software is running on a remote computer). Since this device is passive and cannot monitor program execution, the main risk is identity spoofing. We then need to validate each trusted program according to trusted properties. In this context of TC, we focus on the Oslo [4] secure loader. This program is the first step of a trusted chain. It uses some hardware functionalities of recent CPUs[2] to initialize the chain and to launch the first program of the chain.

The main trusted chain properties are temporal, but can be rewritten into first order logic annotations in the C code, according to the approach proposed by Giorgetti and Groslambert in [17]. To summarize, the safety part of any temporal property can be rewritten, through its Büchi automaton representation, into annotations. Our specification of this code is the union of three specifications of preconditions, postconditions and invariants:

- a functional specification of each operation, described in terms of Oslo variables;
- a modelization of the hardware architecture, expressed in terms of Oslo variables and new variables introduced in order to describe the hardware status;
- a description of the Büchi automaton, modelized with new variables describing the current states.

[1] PFC (meaning trusted platforms – *Plateformes de Confiance* in french) is a project of the SYSTEM@TIC Paris Region French cluster.

[2] AMD-V or Intel-TET technologies.

Oslo program and specification	
Lines of code:	*approx. 1500 lines*
Lines of functional specification (program and TPM):	*approx. 1500 lines*
Number of VCs:	*approx. 7300*
Observed part of Oslo	
Lines of code observed:	*approx. 40 lines*
Lines of specification for the Büchi automaton:	*approx. 500 lines*
Number of VCs:	*771*

Table 3: Some Figures about the Oslo Program

Note that the last point depends on the others, and thus they have to be proved together. Hence, the main problem of this case study is the large amount of hypotheses. Table 3 gives some factual information about the studied part of Oslo. The web page `http://www.lri.fr/~stoulsn/tools/oslo/` gives more technical details about this benchmark.

6.2.2 Results of Oslo Verification

First of all, among the 771 generated VCs, 726 are directly discharged (i.e. without any axiom selection). Next, the approach developed in [9] allows to automatically prove 732 VCs. This result is obtained by using the algorithm described in Fig. 6 with the three provers Simplify, Alt-Ergo and Yices and with the algorithm parameters $i_{max} = 3$, $j_{max} = 6$ and $TO = 250$ (250 seconds come from the fact that 25 different prover calls are done in the worst case, with a timeout $T_n = 10$). These limits (i_{max} and j_{max}) are experimental and correspond to the timeout limit of the algorithm.

Among the remaining unproved VCs, some rely on quantified hypotheses. Others need comparison predicates that are not handled in the previous work [9] and have motivated the present extensions, namely CNF reduction, comparison handling and context reduction.

Figure 7 sums up these results. Finally, notice that the entire process needs 4.4 hours with the results presented in this work, while it needs 5.8 hours with the results from the former work.

7 Related Work and Conclusion

We have presented a new strategy to select relevant hypotheses in formulas coming from program verification. To do so, we have combined two separate dependency analyses based on graph computation and graph traversal. Moreover, we have given some heuristics to analyze the graphs with a sufficient granularity. Finally we have shown the relevance of this approach with a benchmark issued from a real industrial code.

Strategies to simplify the prover's task have been widely studied since automated provers exist [29], mainly to propose more efficient deductive systems [29, 28, 27]. The present work can be compared with the set of support (sos) selection strategy [29, 22]. This approach starts with asking the user to provide an initial sos: it is classically the negative of the conclusion and a subset

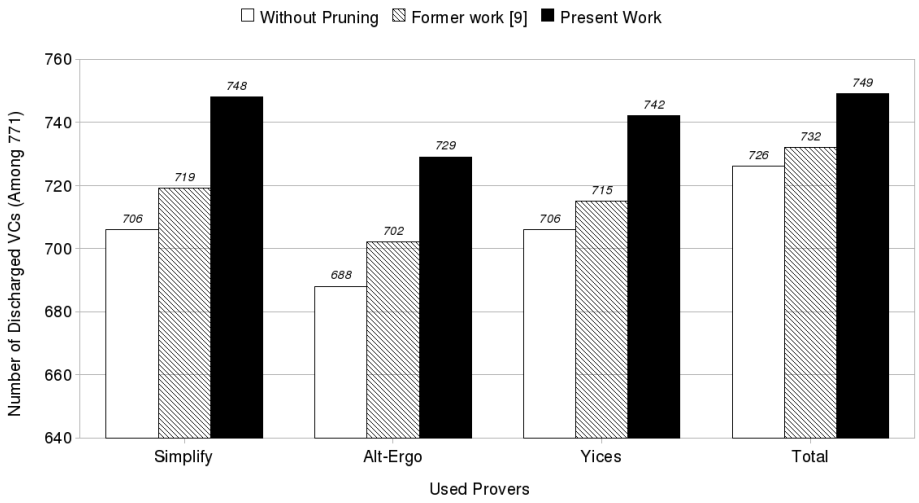

Figure 7: Comparison of Results on Oslo Between Proposed Methods

of hypotheses. It is then restricted to only apply inferences with at least one clause in the sos, consequences being added next into the sos. Our work can also be viewed as an automatic guess of the initial sos guided by the formula to prove. In this sense, it is close to [21] where initial relevant clauses are selected according to syntactical criteria, i.e. counting matching rates between symbols of any clause and symbols of clauses issued from the conclusion. By considering syntactical filtering on clauses issued from axioms and hypotheses, this latter work does not consider the relation between hypotheses, formalized by axioms of the theory: it provides a reduced forward proof. In contrast, by analyzing dependency graphs, we simulate natural deduction and are not far from backward proof search. By focusing on the predicative part of the proof obligation, our objectives are dual to those developed in [18]: this work concerns boolean verification conditions with any boolean structure whereas we treat predicative formulas whose symbols are axiomatized in a quantified theory. Even in a large set of context axioms, most of the time, each proof obligation only requires a tiny portion of this context. In [25, 10] a strategy to select relevant context axioms is presented, but it needs a preliminary manual task classifying axioms. Our predicate graph computation makes this axiom classification automatic. Recent advances have been made in the direction of semantic selection of axioms [26, 23]. Briefly speaking, at each iteration, the selection of each axiom depends on the fact that a computed valuation is a model or not of the axiom. By comparison, our syntactical axiom selection is more efficient, indeed linear in the size of the input formula.

In future work, we expect to avoid the decomposition of axioms in conjunctive normal form. This decomposition presents the advantage to make it possible to select only relevant parts of axioms, but it is costly and the resulting clauses often contain the same subformula. Hence the prover sometimes has to treat this formula several times and worst to instantiate it twice with the same value of its quantifiers. It seems interesting to avoid this decomposition

and to prune general axioms. The pruning would be done by preserving their general form and just removing irrelevant branches. We have already performed some manual experiments and observed a time gain of a factor of two on some verification conditions.

References

[1] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.

[2] B. Barras, S. Boutin, C. Cornes, J. Courant, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, P. Loiseleur, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual Version 6.2*. INRIA-Rocquencourt-CNRS-Université Paris Sud- ENS Lyon, May 1998.

[3] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Computer Aided Verification, 16th International Conference, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*. Springer, 2004.

[4] Bernhard Kauer. OSLO: Improving the security of Trusted Computing. In *16th USENIX Security Symposium, August 6-10, 2007, Boston, MA, USA*, 2007.

[5] Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

[6] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. Technical Report NIII-R0309, Dept. of Computer Science, University of Nijmegen, 2003.

[7] Rod Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.

[8] Sylvain Conchon and Evelyne Contejean. The Ergo automatic theorem prover. `http://ergo.lri.fr/`.

[9] Jean-François Couchot and Thierry Hubert. A Graph-based Strategy for the Selection of Hypotheses. In *FTP 2007 - International Workshop on First-Order Theorem Proving*, Liverpool, UK, September 2007.

[10] David Deharbe and Silvio Ranise. Satisfiability Solving for Software Verification. available at `http://www.loria.fr/~ranise/pubs/sttt-submitted.pdf`, 2006.

[11] Ewen Denney, Bernd Fischer, and Johann Schumann. An empirical evaluation of automated theorem provers in software certification. *International Journal on Artificial Intelligence Tools*, 15(1):81–108, 2006.

[12] David Detlefs, Greg Nelson, and James B. Saxe. The Simplify decision procedure (part of ESC/Java). `http://research.compaq.com/SRC/esc/simplify/`.

[13] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center, December 1998. See also `http://research.compaq.com/SRC/esc/`.

[14] Edsger W. Dijkstra. *A discipline of programming*. Series in Automatic Computation. Prentice Hall Int., 1976.

[15] Bruno Dutertre and Leonardo de Moura. The YICES SMT Solver. available at `http://yices.csl.sri.com/tool-paper.pdf`, 2006.

[16] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.

[17] Alain Giorgetti and Julien Groslambert. JAG: JML Annotation Generation for Verifying Temporal Properties. In Luciano Baresi and Reiko Heckel, editors, *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006*, volume 3922 of *Lecture Notes in Computer Science*, pages 373–376. Springer, 2006.

[18] E. Pascal Gribomont. Simplification of boolean verification conditions. *Theoretical Computer Science*, 239(1):165–185, 2000.

[19] Thierry Hubert and Claude Marché. Separation analysis for deductive verification. In *Heap Analysis and Verification (HAV'07)*, Braga, Portugal, March 2007. `http://www.lri.fr/~marche/hubert07hav.pdf`.

[20] K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005.

[21] Jia Meng and L.C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. In *ESCoR: Empirically Successful Computerized Reasoning*, 2006.

[22] David A. Plaisted and Adnan H. Yahya. A relevance restriction strategy for automated deduction. *Artificial Intelligence*, 144(1-2):59–93, 2003.

[23] Petr Pudlak. Semantic selection of premisses for automated theorem proving. In G. Sutcliffe, J. Urban, and S. Schulz, editors, *CEUR Workshop Proceedings*, volume 257, pages 27–44, 2007.

[24] Silvio Ranise and David Deharbe. Light-weight theorem proving for debugging and verifying units of code. In *Proc. SEFM'03*, Canberra, Australia, September 2003. IEEE Computer Society Press. `http://www.loria.fr/equipes/cassis/softwares/haRVey/`.

[25] Wolfgang Reif and Gerhard Schellhorn. Theorem proving in large theories. In Maria Paola Bonacina and Ulrich Furbach, editors, *Int. Workshop on First-Order Theorem Proving, FTP'97*, pages 119–124. Johannes Kepler Universität, Linz (Austria), 1997.

[26] Geoff Sutcliffe and Yury Puzis. Srass - a semantic relevance axiom selection system. In Springer, editor, *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, volume 4603 of *Lecture Notes in Computer Science*, pages 295–310, 2007.

[27] Larry Wos. Conquering the meredith single axiom. *Journal of Automated Reasoning*, 27(2):175–199, 2001.

[28] Larry Wos and Gail W. Pieper. The hot list strategy. *Journal of Automated Reasoning*, 22(1):1–44, 1999.

[29] Larry Wos, George A. Robinson, and Daniel F. Carson. Efficiency and completeness of the set of support strategy in theorem proving. *J. ACM*, 12(4):536–541, 1965.

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 Orsay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399