Graph-based Reduction of Program Verification Conditions

Jean-François Couchot — Alain Giorgetti — Nicolas Stouls

N° 6702
Octobre 2008

Thème SYM
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 (Réseau National des Technologies Logicielles), and by the SYSTÉM@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
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 3
2 Verification Conditions 5
3 Running Example 5
4 Graph Based Memorizing of Dependency 8
  4.1 Term Dependency 8
  4.2 Predicate Dependency 9
  4.3 Handling Comparison Predicates 10
5 Axiom Selection 12
  5.1 Relevant Predicates 12
  5.2 Relevant Constants 13
  5.3 Selection of Relevant Axioms 14
6 Experiments 15
  6.1 Methodology 16
  6.2 Trusted Computing Case Study 17
    6.2.1 Motivations and Context 17
    6.2.2 Results of Oslo Verification 18
7 Related Work and Conclusion 18

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.

\[\text{Goal} \quad \Rightarrow \quad \text{Hypotheses} \quad \Rightarrow \quad \text{Conclusion} \]

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 \(Ctx_1 \land \ldots \land Ctx_n\) and \(H_1 \land \ldots \land H_m\) 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 \(Ctx_1 \land \ldots \land Ctx_n \land H_1 \land \ldots \land H_m \land \neg C\).

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

/*@ requires \valid(c) @*/
void g(struct s *c);

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. 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. 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 diff(m_1, m_2, l) means that differences between m_1 and m_2 only concern the set...
The predicate can be defined by

\[
diff(m_1, m_2, l) \Leftrightarrow (\forall p. \text{valid}(p) \land \neg \text{mem}(p, l) \Rightarrow \text{acc}(m_1, p) = \text{acc}(m_2, p))
\]  

(1)

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

\[
\text{Ctx}_1 \land \text{Ctx}_2 \land \text{Ctx}_3 \land \text{Ctx}_4
\]

(2)

of the following four clauses:

\[
\neg \text{diff}(M_1, M_2, L) \lor \neg \text{valid}(P) \lor \text{mem}(P, L) \lor \text{acc}(M_1, P) = \text{acc}(M_2, P) \quad (\text{Ctx}_1)
\]

\[
\text{valid}(p_0) \land \text{diff}(M_1, M_2, L) \quad (\text{Ctx}_2)
\]

\[
\neg \text{mem}(p_0, L) \lor \text{diff}(M_1, M_2, L) \quad (\text{Ctx}_3)
\]

\[
\text{acc}(M_1, p_0) \neq \text{acc}(M_2, p_0) \lor \text{diff}(M_1, M_2, L) \quad (\text{Ctx}_4)
\]

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 \( \text{Ctx} \) in all that follows.

The effect VC for function \( f \) is the conjunction

\[
\text{Ctx} \land H_1 \land H_2 \land H_3 \land H_4 \land H_5 \land H_6 \land \neg C
\]

(3)

of the context \( \text{Ctx} \) and of the following ground formulas:

\[
\text{valid}(a) \quad (H_1)
\]

\[
\text{valid}(b) \quad (H_2)
\]

\[
\text{valid}(\text{acc}(m_y, a)) \quad (H_3)
\]

\[
\text{valid\_acc\_range}(m_v, 2) \quad (H_4)
\]

\[
\text{separation1\_range}(m_v, 2) \quad (H_5)
\]

\[
\text{diff}(m_x, m_x, \text{0}, \text{singleton}(\text{shift}(\text{acc}(m_v, \text{acc}(m_y, a)), 0))) \quad (H_6)
\]

\[
\neg \text{diff}(m_x, \text{upd}(m_x, \text{0}, \text{shift}(\text{acc}(m_v, \text{acc}(m_y, a)), 1)), \text{acc}(m_x, b)), \text{range}(\text{singleton}(\text{acc}(m_v, \text{acc}(m_y, a)), 0, 1)) \quad (\neg C)
\]

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 \( \text{valid\_acc\_range}(m_v, 2) \) 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 \( \text{separation1\_range}(m_v, 2) \) means that \( t[0] \neq t[1] \).

The sixth clause (\( H_6 \)) comes from the statement \( g(a \rightarrow y) \), more precisely from the annotation \( \text{assigns } c \rightarrow v[0].x \) where \( c \) is substituted by \( a \rightarrow y \). The function \( \text{singleton} \) has the usual meaning and \( \text{shift}(t, i) \) allows to access to the index \( i \) in the array \( t \). This hypothesis defines the access of a variable \( m_x \) which is equal to the access in \( m_x \) except for the index \( \text{shift}((\text{acc}(m_v, \text{acc}(m_y, a))), 0) \) corresponding to \( a \rightarrow y \rightarrow v[0] \). The conclusion \( C \) is a \( \text{diff} \) 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 \( \text{range}(\text{singleton}(\text{acc}(m_v, \text{acc}(m_y, a))), 0, 1) \) defines the set of pointers located at indices 0 and 1 in the array \( \text{acc}(m_v, \text{acc}(m_y, a)) \). In fact this is the representation of \( a \rightarrow y \rightarrow v[0..1] \).

Even if this example is small, Simplify and haR V ey are the sole SMT provers, among Simplify \[12\], Yices \[15\], Alt-Ergo \[8\], haR V ey \[24\] 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 \( \text{valid} \) and \( \text{valid\_acc\_range} \) are not directly useful in the present case. Removing the hypothesis concerning \( \text{valid\_acc\_range} \) 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
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 \(\text{singleton}_2, \text{shift}_3, \text{acc}_4\) and \(\text{acc}_5\) and the flat hypotheses resulting from \(H_6\) are represented by the complete subgraphs defined by the sets \(\{m_x, m_x0, \text{singleton}_2\}\), \(\{\text{singleton}_2, \text{shift}_3\}\), \(\{\text{shift}_3, \text{acc}_4\}\), \(\{\text{acc}_4, \text{acc}_5, m_v\}\) and \(\{\text{acc}_5, m_y, a\}\). Similarly, \((H_3)\) introduces the fresh constant \(\text{acc}_1\) and is represented by the set \(\{\text{acc}_1, m_y, a\}\).

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 \(\neg p\). 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
To reduce the graph size, the contrapositive 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 \( \text{card}(Cl) - 1 \) 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 \( p \overset{w_1}{\rightarrow} q \) and \( p \overset{w_2}{\rightarrow} q \) are replaced with the weighted arc \( p \overset{\text{min}(w_1, w_2)}{\rightarrow} q \).

<table>
<thead>
<tr>
<th>Pair</th>
<th>Arcs</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \neg p, q )</td>
<td>( {p \rightarrow q} )</td>
</tr>
<tr>
<td>( p, q )</td>
<td>( {\neg p \rightarrow q} )</td>
</tr>
<tr>
<td>( \neg p, \neg q )</td>
<td>( {p \rightarrow \neg q} )</td>
</tr>
</tbody>
</table>

Table 1: Translating Pair of Literals into Arcs.

### 4.3 Handling Comparison Predicates

In a former work \cite{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 =, \( \neq \), \( \leq \), \( < \), \( \geq \) 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 \( t_1 = t_2 \), \( t_1 \neq t_2 \), \( t_1 \leq t_2 \), \( t_1 < t_2 \), \( t_1 \geq t_2 \) 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 \( \text{Comp} = \{=, \neq, \leq, \prec, \triangleright, >\} \) 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 \( \diamond f_2 \).
The semantics of these nodes is detailed now. Given a comparison predicate \( \circ \in \text{Comp} \), two functional terms \( t_1 \) and \( t_2 \), and a set \( L \) of two literals, the clause

\[
L \cup \{ t_1 \circ t_2 \}
\]

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 \( \{ \neg p \} \).

![Figure 4: Abstract representation of \( \{ \neg p, t_1 \circ t_2 \} \)\n
<table>
<thead>
<tr>
<th>Pair</th>
<th>Arcs</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \neg p, f_1(\ldots) \circ f_2(\ldots) )</td>
<td>( p \rightarrow f_1, p \rightarrow f_2, f_1 \rightarrow f_2, f_1 \rightarrow f_1 )</td>
</tr>
<tr>
<td>( p, f_1(\ldots) \circ f_2(\ldots) )</td>
<td>( f_1 \rightarrow f_1, f_1 \rightarrow f_2, f_1 \rightarrow f_2, f_1 \rightarrow f_1 )</td>
</tr>
<tr>
<td>( g_1(\ldots) \circ g_2(\ldots) ), ( f_1(\ldots) \circ f_2(\ldots) )</td>
<td>( g_1 \rightarrow g_1, g_1 \rightarrow g_2, g_1 \rightarrow g_2, g_1 \rightarrow g_2, g_1 \rightarrow g_1 )</td>
</tr>
</tbody>
</table>

where \( \circ \) and \( \cdot \) are comparison operators in \( \text{Comp} \) 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 \( \circ f_1 \). Consequently, \( \circ f_1 \) and \( \overline{f_1} \) are the same node, and then not represented twice.

Notice that, given \( \circ \in \text{Comp} \), \( t_1 \) 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 \( \circ \).
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 \texttt{acc}(...) 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 \texttt{valid} to \texttt{diff} is 1 since (¬\texttt{diff}, ¬\texttt{valid}) appears in the clause (Ctx_1) with 4 literals and (\texttt{valid}, \texttt{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 \(A_1 \land \ldots \land A_n \land \neg C\) be a satisfiability problem. The axiom \(A_1\) is relevant w.r.t. \(C\) if \(A_1 \land \ldots \land A_n \land \neg C\) is not satisfiable whereas \(A_2 \land \ldots \land A_n \land \neg C\) 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 \(q\) to \(p\).
to $\mathcal{P}$. 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 counterpart $p_2 \xrightarrow{w} p_1$ is too, with the convention that $p$ is $p$. Let $n$ be the minimal distance from $\mathcal{L}_0$ to the deepest reachable predicate. For $1 \leq i \leq n$, $\mathcal{L}_i$ 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:

\[
\begin{align*}
\mathcal{L}_0 &= \{\text{diff}\} \\
\mathcal{L}_1 &= \mathcal{L}_0 \cup \{\text{mem}, \circ \text{acc}, \underline{\text{valid}}\} \\
\mathcal{L}_2 &= \mathcal{L}_1 \cup \{\text{diff}\} \\
\mathcal{L}_3 &= \mathcal{L}_2 \cup \{\underline{\text{valid}}, \underline{\text{acc}}, \underline{\text{range}}\} \\
\mathcal{L}_4 &= \mathcal{L}_3 \cup \{\underline{\text{valid}}, \underline{\text{mem}}\} \\
\mathcal{L}_5 &= \mathcal{L}_4 \cup \{\underline{\text{separation1}}, \underline{\text{range}}, \underline{\text{separation1}}, \underline{\text{range}}, \underline{\text{valid}}, \underline{\text{acc}}, \underline{\text{range}}\} \\
\cdots \\
\mathcal{L}_\infty &= \bigcup_{i \geq 0} \mathcal{L}_i \cup \text{unreachable vertices}
\end{align*}
\]

Notice that the spurious predicate $\underline{\text{valid}} \underline{\text{acc}} \underline{\text{range}}$ 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 ($1 \leq i \leq n$). Finally, unreachable constants are added to the limit of the sequence $(\mathcal{C}_n)_{n \in \mathbb{N}}$ 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:

\[ C_0 = \{ m_x, m_x \cdot 0, m_y, a, b \}, \]
\[ C_1 = C_0 \cup \{ acc, \bar{r} \}, \]
\[ C_2 = C_1 \cup \{ acc, 1, singleton \}, \]
\[ C_3 = C_2 \cup \{ acc, 4 \}, \]
\[ C_4 = C_3 \cup \{ shift, \bar{3} \} \]
\[ C_\infty = C_4. \]

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 \((L_n)_{n \in \mathbb{N}}, (C_n)_{n \in \mathbb{N}}\) 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 \(C_l\) from a hypothesis. Let \(V\) be the set of constants of \(C_l\) augmented with constants resulting from flattening (see Section 4.1). Let \(P\) be the set of predicates of \(C_l\). The clause \(C_l\) 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:
   \[ V \cap C_j \neq \emptyset \lor P \cap L_i \neq \emptyset \]

2. the clause includes more than a threshold \(t_v\) of relevant constants or more than a threshold \(t_p\) of relevant predicates:
   \[ \frac{\text{card}(V \cap C_j)}{\text{card}(C_j)} \geq t_v \land \frac{\text{card}(P \cap L_i)}{\text{card}(L_i)} \geq t_p \]

3. all the clause constants and clause predicates are relevant:
   \[ V \subseteq C_j \land P \subseteq L_i \]

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 \(L_i\). Practically, for each arc that is passed through...
while generating $L_i$, 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 $C_j$ until eventually $C_\infty$, 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 $L_0$, no axiom is selected with $C_0$, $C_1$, $C_2$, $C_3$. However with $C_4$, it yields the VC:

$$diff(m_x, m_x \bot, \text{singleton}(\text{shift}(\text{acc}(m_v, \text{acc}(m_y, a)), 0))) \land$$

$$\neg(diff(m_x, \text{upd}(m_x \bot, \text{shift}(\text{acc}(m_v, \text{acc}(m_y, a)), 1)), \text{acc}(m_x, b)), \text{range}(\text{singleton}(\text{acc}(m_v, \text{acc}(m_y, a))), 0, 1)))$$

This VC does not contain the axiom with valid $\text{acc-range}$, 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: $VC, Prover, i_{\text{max}}, j_{\text{max}}, TO$

\[
T_n = TO / ((i_{\text{max}} + 1) \times j_{\text{max}} + 1)
\]

\[
\text{Res} := \text{Prover}(VC, \text{timeout} = T_n, \text{without axiom selection})
\]

\[
\text{if Res = timeout then}
\]

\[i := 0;\]
\[j := 0;\]
\[
\text{While (Res } \neq \text{ unsat) } \land \ i \leq i_{\text{max}} \land \ j \leq j_{\text{max}} \text{ do}
\]

\[\text{Res} := \text{Prover}(VC, \text{timeout} = T_n, \text{axiom selection} = (i, j))\]

\[j := j + 1;\]
\[\text{if } j > j_{\text{max}} \text{ then}
\]
\[i := i + 1;\]
\[j := 0;\]
\[
\text{return Res;}
\]

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:

\[--\text{prune-with-comp} --\text{prune-context} --\text{prune-vars-filter} \text{ CNF}\]

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 $i_{\text{max}}$ and $j_{\text{max}}$ 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 $T_O$ is divided into $n = (i_{\text{max}}+1) \cdot j_{\text{max}}+1$ 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[^3] 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
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


