Tracing the Origins of Verification Conditions

Ranan Fraer

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

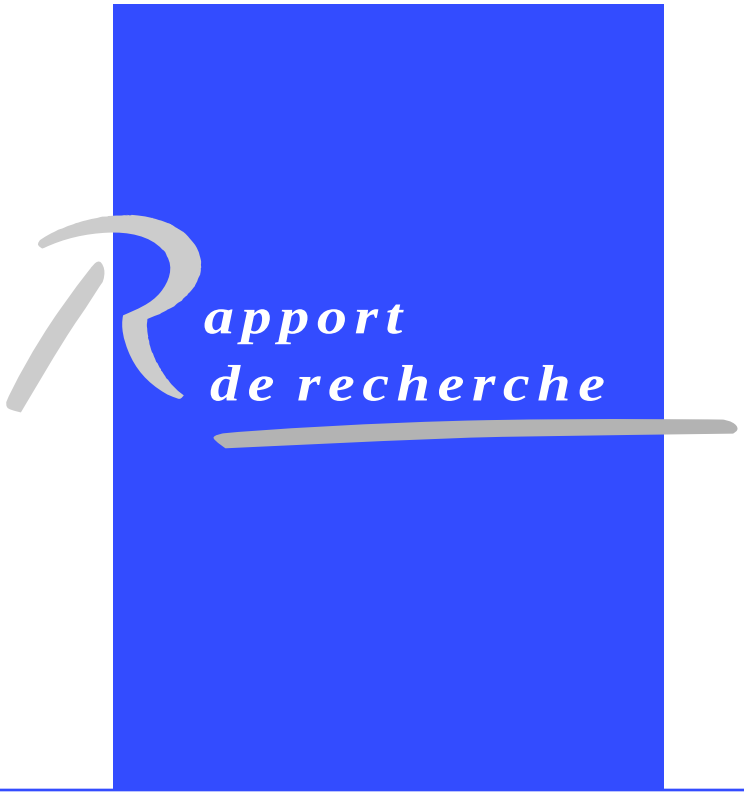
Tracing the Origins of Verification Conditions

Ranan Fraer

N˚ 2840

March 1996

THÈME 2



ISSN 0249-6399

Tracing the Origins of Verification Conditions

Ranan Fraer

Thème 2 — Génie logiciel
et calcul symbolique
Projet Croap

Rapport de recherche n ° 2840 — March 1996 — 17 pages

Abstract: The typical program verification system is a batch tool that accepts as input a program annotated with Floyd-Hoare assertions, performs syntactic and semantic analysis on it, and generates a list of verification conditions that is subsequently submitted to a theorem prover. When a verification condition cannot be proved, this may be due to an error in the program or an inconsistency in the annotations. Unfortunately, it is very difficult to relate a failing proof attempt to a particular piece of code or assertion. We propose a solution to this problem using the technique of origin tracking.

Key-words: program verification, origin tracking, programming environment

(Résumé : tsvp)

to appear in the proceedings of the *International Conference on Algebraic Methodology and Software Technology AMAST*, Springer-Verlag Lecture Notes in Computer Science, Munich, July 1996.

Ranan.Fraer@sophia.inria.fr

Unité de recherche INRIA Sophia-Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : (33) 93 65 77 77 – Télécopie : (33) 93 65 77 65

Tracer les Origines des Conditions de Vérification

Résumé : Le système typique de vérification de programmes est un outil peu intéractif qui accepte en entrée un programme annoté avec des assertions Floyd-Hoare, analyse ce programme syntaxiquement et sémantiquement, et génère une liste de conditions de vérification qui est ensuite soumise à un démonstrateur de théorèmes. Si une condition de vérification ne peut pas être prouvée, ceci peut être dû à une erreur dans le programme, ou à une inconsistance dans les annotations. Malheureusement, il est très difficile de relier un échec dans la preuve à un endroit dans le programme ou dans les assertions. Nous proposons une solution à ce problème en utilisant la technique du suivi des origines.

Mots-clé : vérification de programmes, suivi des origines, environnement de programmation

1 Introduction

Since the late sixties, when Floyd and Hoare [Flo66, Hoa69] set up the basis of a method for proving programs correct, several program verification systems have appeared. Usually the main component of such a system is a verification condition generator (VCG) that takes as input an imperative program and a specification (under the form of pre/post-conditions and loop invariants) and outputs a list of logical conditions. These conditions are supposed to be proved manually or mechanically (using a theorem prover), and their satisfiability ensures the correctness of the program. As examples of tools that work this way we can cite the Stanford Pascal Verifier [ILL75], Gypsy [Goo85], EVES [KPS+92] or Penelope [GMP90]. Verification condition generators are also used in formal methods based on stepwise refinement like VDM [Jon86] or B [Abr95], as each refinement step has to be validated by proving the corresponding set of verification conditions.

However, if one verification condition cannot be proved this means that either the program does not satisfy its specification, or the specification does not state correctly the intended meaning of the program. In both cases the user is supposed to modify the program or the specification and the system does not give any hint on where this modification should occur. Even when trying to prove a true condition, stated as an ordinary first order logic formula, it is quite difficult to understand what one is trying to prove, and in which way this condition is related to a possible execution of the program. In other words, the absence of links between the source program and the verification conditions is a serious lack to the task of formal verification. To compensate, users do some kind of mental processing for retrieving the origin of terms that appear in a condition. For instance, by recognizing in a condition the negation of the test of an `if` statement, they deduce that this condition is generated by the `else` branch of this statement.

This paper proposes a way of mechanizing this process by instrumenting a verification condition generator with origin tracking facilities. This technique, proposed by Bertot in [Ber91, Ber92, Ber93], has beeen initially applied in implementing source-level debugging of programming languages by establishing a relation between the current instruction and a position in the source program. Given a natural semantics description of the language, it is possible to integrate this relation in the semantics and the integration is shown to be systematic and semi-automatizable. Related work on origin tracking in the framework of algebraic specifications is reported by Van Deursen, Klint and Tip in [DKT93].

As origin tracking has been proved useful for interpreters, it is sensible to apply it to verification conditions generators. Indeed, both kinds of tools are similar in that they perform syntactic and semantic analysis on the input program and generate some output, be it the current state of an execution machine, or verification conditions. If debugging is considered essential in understanding program execution, there is no counterpart in program verification systems that could help understanding verification conditions. Our work represents an attempt to progress in this area.

We have applied these ideas in a verification condition generator built with the programming environment generator Centaur [Jac94]. We have benefited of the facilities of manipulating syntactic structures and of the built-in mechanisms of selection and highligh-

ting. Alternative approaches could use other programming environment generators like the Synthesizer Generator [RT88], or ASF+SDF [Kli93].

The paper is organized as follows. Section 2 presents the algorithm of verification condition generation. In the section 3 we exemplify the problem to solve on a specific program and its conditions. Sections 4 and 5 introduce origin functions as a data structure well suited to represent descendance information. In section 6 we describe the integration of origin functions in the verification conditions generator and the complexity of the new algorithm is analyzed in section 7. Section 8 explains the particularities of the Centaur implementation. Section 9 studies the extensibility of this work to real program verification systems. We conclude in section 10 by some remarks on the connections with program slicing.

2 Verification condition generators

We will concentrate on a small imperative programming language with assignments, conditionals and loops. Programs are annotated with loop invariants and optional assertions can be placed before instructions. A BNF syntax of the language is given below. Variable declarations, expressions and first-order assertions are defined in the usual way.

<program> ::= `decl` <decls> `in` {<assert>} <inst> {<assert>}

<inst> ::= `skip` | <id> `:=` <exp> | <annot_inst> ; <annot_inst> |
`if` <exp> `then` <annot_inst> `else` <annot_inst> |
`while` <exp> `inv` {<assert>} `do` <annot_inst>

<annot_inst> ::= {<assert>} <inst> | <inst>

The algorithm to generate verification conditions from an annotated program is based on Dijkstra's weakest-precondition calculus [Dij76]. Let $wp(I, Q)$ denote the *weakest precondition that should be satisfied before execution of I, for the postcondition Q to be true after execution of I*. The algorithm performs a backwards traversal of the instruction I starting from the postcondition Q, and computes the missing assertions at each intermediary point. When reaching a user provided assertion P, it generates a verification condition of the form $P \Rightarrow wp(I, Q)$, and it restarts with P as the new postcondition.

We introduce the relation[1] $Post \vdash I \rightarrow Pre, Conds$ with the meaning: *the postcondition $Post$ is true after a terminating execution of the instruction I if the precondition Pre is true before executing I, and if the list of verification conditions $Conds$ contains only valid formulas*. We restrict our presentation to partial correctness, but the same considerations apply to total correctness as well. In what follows, we use [] to denote the empty list, $[a, b]$ for a list with two elements a and b, and $L_1.L_2$ for the concatenation of lists L_1 and L_2.

[1] The convention followed here is to separate inputs and outputs of the relation by an arrow "$\Rightarrow$". The input term preceded by a turnstyle "$\vdash$" is also distinguished as the *subject* of the relation, while the other inputs form the evaluation context.

The algorithm is described using inference rules:

$$Q \vdash \texttt{skip} \rightarrow Q, []$$

$$\frac{x, E \vdash Q \rightarrow Q'}{Q \vdash x := E \rightarrow Q', []}$$

$$\frac{Q \vdash I_2 \rightarrow P_2, Conds_2 \qquad P_2 \vdash I_1 \rightarrow P_1, Conds_1}{Q \vdash I_1 \ ; \ I_2 \rightarrow P_1, Conds_1.Conds_2}$$

$$\frac{Q \vdash I_1 \rightarrow P_1, Conds_1 \qquad Q \vdash I_2 \rightarrow P_2, Conds_2}{Q \vdash \texttt{if}\ B\ \texttt{then}\ I_1\ \texttt{else}\ I_2 \rightarrow (B \Rightarrow P_1) \wedge (\neg B \Rightarrow P_2), Conds_1.Conds_2}$$

$$\frac{Inv \vdash S \rightarrow P, Conds}{Q \vdash \texttt{while}\ B\ \texttt{inv}\ \{Inv\}\ \texttt{do}\ S \rightarrow Inv, [Inv \wedge B \Rightarrow P, Inv \wedge \neg B \Rightarrow Q].Conds}$$

$$\frac{Q \vdash S \rightarrow P, Conds}{Q \vdash \{A\}\ S \rightarrow A, [A \Rightarrow P].Conds}$$

The relation $x, E \vdash Q \rightarrow Q'$ stands for $Q' = Q[E/x]$ where the substitution $Q[E/x]$ is formalized in the usual way avoiding the capture of free variables in quantifications. We present below two typical substitution rules:

$$\frac{x, E \vdash P \rightarrow P' \qquad x, E \vdash R \rightarrow R'}{x, E \vdash P \wedge R \rightarrow P' \wedge R'} \tag{1}$$

$$x, E \vdash x \rightarrow E \tag{2}$$

Finally, we need a rule for computing the conditions for the whole program. The notation $\vdash Prog \rightarrow Conds$ stands for: $Conds$ is the set of verification conditions generated from the annotated program $Prog$.

$$\frac{Q \vdash I \rightarrow P', Conds}{\vdash \texttt{decl}\ Decls\ \texttt{in}\ \{P\}\ I\ \{Q\} \rightarrow [P \Rightarrow P'].Conds} \tag{3}$$

3 An example

As an example, consider the following program computing the quotient q and the remainder r of the integer division of x by y. We have purposedly introduced a bug in the program: the test of the loop is $r > y$ instead of $r \geq y$. This way, if x is an exact multiple of y the program stops with r equal to y and not to 0.

In the figure 1 the program is shown on the left-hand side and the verification conditions on the right-hand side. The three conditions state in order that the invariant is satisfied at

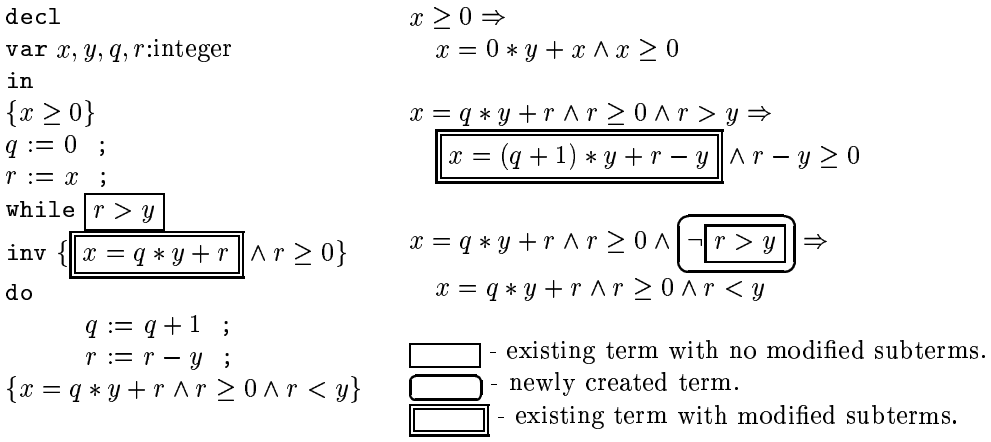


Figure 1: An integer division program and its verification conditions

the entry of the loop, the invariant is preserved by each loop iteration, and the postcondition is satisfied at the exit of the loop. The error appears in the last verification condition, that cannot be satisfied since $\neg(r > y)$ does not imply $r < y$. A system based on the technique proposed in this paper automatically highlights the origin of $r > y$ in the source program, when the user selects this expression in the verification condition.

Note also that the origin of $r > y$ is exactly the same expression in the program. This is not always the case. We distinguish two other cases:

- there are expressions like $\neg\ (r > y)$ that have no origin, although some of their subexpressions might have an origin. They represent new terms that have been constructed from already existing expressions during the generation process.
- more subtly, an expression like $x = (q+1) * y + r - y$ has its origin in the subexpression $x = q * y + r$ of the loop invariant, but some of its subexpressions, like $q + 1$ and $r - y$, have their origins in the right-hand sides of the assignments in the loop body.

We will propose a representation of origin informations that takes into account these differences by avoiding to store the origins of all subterms of a term that was not changed during the condition generation.

4 Origin functions

In order to understand how parts of the source program appear at different positions in the verification conditions, let us consider a very simple program `decl` D `in` $\{P\}\ x := E\ \{x > 0\}$