Métodos Formales de Desarrollo de Software

Grado en Ingeniería Informática

Paqui Lucio
Dpto de Lenguajes y Sistemas Informáticos.

eman ta zabal zazu

Universidad del País Vasco Euskal Herriko Unibertsitatea

1.- Introduction

Three steps

- From Turing to Hoare Logic
- From Hoare Logic to Verification Condition Generation (VCG)
- From VCG to Current Deductive Verification Tools

From Turing to Hoare Logic

- Turing 1949: *Checking a large routine*
  Programs can be mathematically described and analyzed.
- Naur 1966: *Proof of algorithms by general snapshots*
  The need for a proof arises because we want to convince ourselves that the algorithm we have written is correct.
- Floyd 1967: *Assigning meaning to programs*
  A "formally verified" program is one whose correctness can be proved mathematically.
- Hoare 1969: *An axiomatic basis for computer programming*
  A "correctness proof" is made w.r.t. a mathematical description (or specification) of what the program is intended to do (for all possible values of its input).

- Is program P correct? $\equiv$ Does P satisfy its specification S?
  - P is written in a programming language PL

    While-Programs
  - S is written in a specification language SL

    First-Order Formulas
  - Axiomatic semantics of PL: Set of rules for establishing that P satisfies S

    Hoare Formal System (a.k.a.Hoare Logic)

While-Programs

- Null: $\texttt{skip}$
- (Simoultaneous) Assignment: $\mathtt{x_1}, \ldots, \mathtt{x_n} := \mathtt{t_1}, \ldots, \mathtt{t_n}$
- Sequential Composition: $P_1; P_2$
- Conditional: $\texttt{if } b \texttt{ then } P_1 \texttt{ else } P_2$
- Iteration: $\texttt{while } b \texttt{ do } P$

- Example:

```
r := 0;
while (r+1)∗(r+1) ≤ x
      do
      r := r+1
```

First-Order Logic (FOL)

- First-order syntax:

$$\varphi ::= p(\bar{t}) \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \rightarrow \varphi_2 \mid \forall x(\varphi) \mid \exists x(\varphi)$$

$$\begin{aligned}&\forall n(\ (n > 2 \wedge even(n)) \rightarrow \\ &\qquad \exists x \exists y(prime(x) \wedge prime(y) \wedge n = x + y))\end{aligned}$$

- $\varphi[\bar{t}/\bar{x}]$ stands for the formula that results by (simultaneously) replace every *free* occurrence of each x_i in φ by t_i.

$$\exists z(x * y = 2 * z)[y+1, x-1/x, y] = \exists z((y+1) * (x-1) = 2 * z)$$

- A formula is valid if it is true in any model (i.p. for any value of its variables).

Axiomatic Semantics

- Hoare triple $\{\varphi\}P\{\psi\}$ says that if program P is started in (a state satisfying) φ, then P terminates (if it does) in a state that satisfies ψ.

$$\{m \leq n\}\ \mathtt{j := (m + n)/2}\ \{m \leq j \leq n\}$$

$$\{False\}\ \mathtt{skip}\ \{\forall n((n > 2 \wedge even(n)) \rightarrow \exists x \exists y(prime(x) \wedge prime(y) \wedge n = x + y))\}$$

Hoare Formal System

$$\frac{}{\{\psi[\overline{t}/\overline{x}]\}\ \overline{x} := \overline{t}\ \{\psi\}} \qquad \frac{\varphi \to \psi}{\{\varphi\}\texttt{skip}\{\psi\}} \qquad \frac{\varphi \to \varphi_1 \quad \{\varphi_1\}P\{\psi\}}{\{\varphi\}P\{\psi\}}$$

$$\frac{\{\varphi\}P_1\{\alpha\} \quad \{\alpha\}P_2\{\psi\}}{\{\varphi\}P_1;P_2\{\psi\}}$$

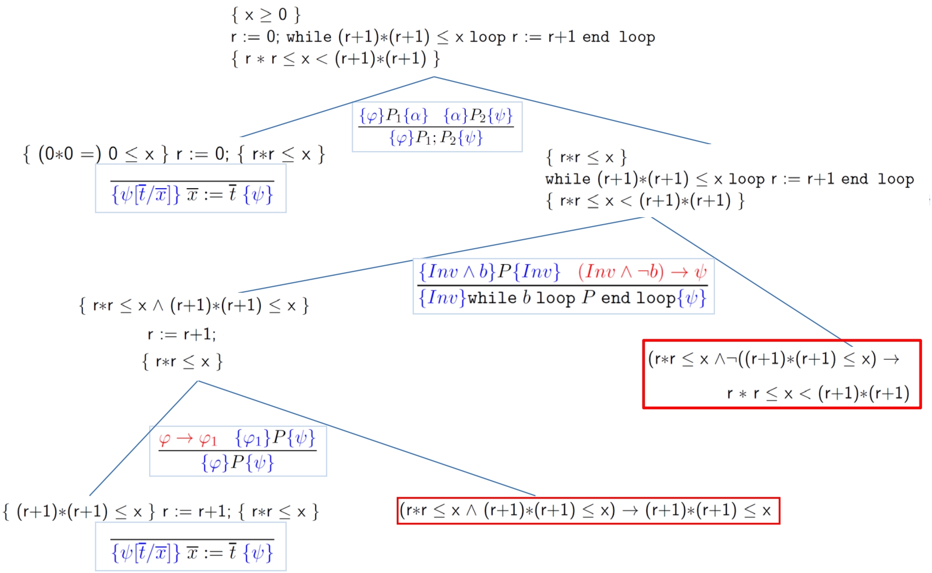
$$\frac{\{\varphi \wedge b\}P_1\{\psi\} \quad \{\varphi \wedge \neg b\}P_2\{\psi\}}{\{\varphi\}\texttt{if } b \texttt{ then } P_1 \texttt{ else } P_2\ \{\psi\}}$$

$$\frac{\{Inv \wedge b\}P\{Inv\} \quad (Inv \wedge \neg b) \to \psi}{\{Inv\}\texttt{while } b \texttt{ do } P\{\psi\}}$$

```
{$x \geq 0$}
r := 0;
while (r+1)*(r+1) $\leq$ x
        //invariant ???
        do
        r := r+1
{$\mathtt{r*r} \leq \mathtt{x} < (\mathtt{r}+1)*(\mathtt{r}+1)$}
    $\mathtt{r} \leq \sqrt{\mathtt{x}} < \mathtt{r}+1$
```

x	r	$(\mathsf{r}+1)*(\mathsf{r}+1) \leq \mathsf{x}$
13	0	$1*1 \leq 13$
13	1	$2*2 \leq 13$
13	2	$3*3 \leq 13$
13	3	$4*4 > 13$

invariant $\mathtt{r}*\mathtt{r} \leq \mathtt{x}$



$\{ x \geq 0 \}$
$r := 0;$ `while` $(r+1)*(r+1) \leq x$ `loop` $r := r+1$ `end loop`
$\{ r * r \leq x < (r+1)*(r+1) \}$

$$\frac{\{\varphi\}P_1\{\alpha\} \quad \{\alpha\}P_2\{\psi\}}{\{\varphi\}P_1;P_2\{\psi\}}$$

$\{ (0*0 =)\ 0 \leq x \}\ r := 0;\ \{ r*r \leq x \}$

$$\frac{}{\{\psi[\overline{t}/\overline{x}]\}\ \overline{x} := \overline{t}\ \{\psi\}}$$

$\{ r*r \leq x \}$
`while` $(r+1)*(r+1) \leq x$ `loop` $r := r+1$ `end loop`
$\{ r*r \leq x < (r+1)*(r+1) \}$

$$\frac{\{Inv \wedge b\}P\{Inv\} \quad (Inv \wedge \neg b) \rightarrow \psi}{\{Inv\}\texttt{while } b \texttt{ loop } P \texttt{ end loop}\{\psi\}}$$

$\{ r*r \leq x \wedge (r+1)*(r+1) \leq x \}$
$r := r+1;$
$\{ r*r \leq x \}$

$$\boxed{(r*r \leq x \wedge \neg((r+1)*(r+1) \leq x) \rightarrow r * r \leq x < (r+1)*(r+1)}$$

$$\frac{\varphi \rightarrow \varphi_1 \quad \{\varphi_1\}P\{\psi\}}{\{\varphi\}P\{\psi\}}$$

$\{ (r+1)*(r+1) \leq x \}\ r := r+1;\ \{ r*r \leq x \}$

$$\frac{}{\{\psi[\overline{t}/\overline{x}]\}\ \overline{x} := \overline{t}\ \{\psi\}}$$

$$\boxed{(r*r \leq x \wedge (r+1)*(r+1) \leq x) \rightarrow (r+1)*(r+1) \leq x}$$

From Hoare Logic to VCG

"Thus the practice of proving programs would seem to lead to solution of three of the most pressing problems in software and programming, namely, reliability, documentation, and compatibility. However, program proving, certainly at present, will be difficult even for programmers of high caliber; and may be applicable only to quite simple program designs."

C. Antony R. Hoare
An axiomatic basis for computer programming
1969

Dijkstra Weakest Precondition

The idea firstly appears in the paper *Guarded commands, nondeterminacy and formal derivation of programs*, E.W. Dijkstra, 1975.

- Hoare Logic:
  Given a precondition φ, a code fragment P and a postcondition ψ, is $\{\varphi\}P\{\psi\}$ true?
- Dijkstra Weakest Precondition:
  Given a code fragment P and postcondition ψ, find the unique formula $\mathsf{wp}(P,\psi)$ which is the *weakest* precondition for P and ψ.

  weakest*:* $\varphi \to \mathsf{wp}(P,\psi)$ *is valid for any* φ *such that* $\{\varphi\}P\{\psi\}$ *is true.*

Each experienced mathematician knows that achievements depend critically on the availability of suitable notations.
E.W. Dijkstra, *My hopes of computing science*, 1979

VCG computes the set of all the FOL-implications of a Hoare proof-tree.

```
VCG ( $\{ x \geq 0 \}$
        $r := 0;$
        while $(r+1)*(r+1) \leq x$ do $r := r+1$
        $\{ r * r \leq x < (r+1)*(r+1) \}$ )
```

$=$

$$\{ (r*r \leq x \wedge (r+1)*(r+1) \leq x) \rightarrow (r+1)*(r+1) \leq x ,$$

$$(r*r \leq x \wedge \neg((r+1)*(r+1) \leq x)) \rightarrow r * r \leq x < (r+1)*(r+1) \}$$

RootApprox.dfy

From VCG to Current Deductive Verification Tools

- Hoare's Verification Grand Challenge:
  - *This contribution ... revives an old challenge: the construction and application of a verifying compiler that guarantees correctness of a program before running it.*
  - *Correctness of computer programs is the fundamental concern of the theory of programming and of its application in large-scale software engineering.*

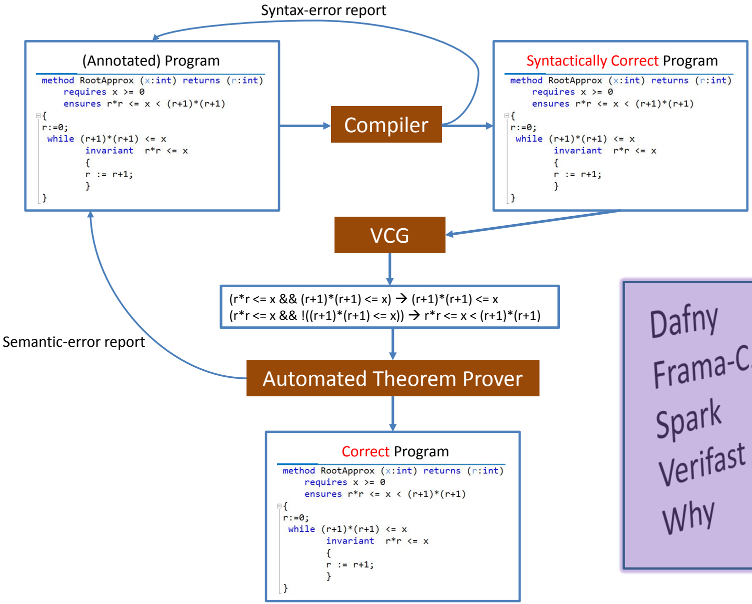
Tony Hoare
The Verifying Compiler: A Grand Challenge for Computing Research
2003

- Deductive Verification:
  - Logical reasoning (deduction) is used to prove properties.
  - Expressive (at least first-order) logic.
  - Contract-based specifications (standard approach)
- Arquitectures in deductive verification:
  1. On top of interactive proof assistants

     Isabelle/HOL, Coq, HOL Ligth, PVS, ...
  2. Automatic Program Verifiers
     1. Program logics for a specific target language

        ACL2, KeY, KIV, VeriFun, ...
     2. VCG + Automatic theorem provers (SMT-solver)

        Spark, Verifast, Dafny, Why, Frama-C, ...



Syntax-error report

(Annotated) Program

```
method RootApprox (x:int) returns (r:int)
    requires x >= 0
    ensures r*r <= x < (r+1)*(r+1)
{
r:=0;
 while (r+1)*(r+1) <= x
        invariant  r*r <= x
        {
        r := r+1;
        }
}
```

Compiler

Syntactically Correct Program

```
method RootApprox (x:int) returns (r:int)
    requires x >= 0
    ensures r*r <= x < (r+1)*(r+1)
{
r:=0;
 while (r+1)*(r+1) <= x
        invariant  r*r <= x
        {
        r := r+1;
        }
}
```

VCG

(r*r <= x && (r+1)*(r+1) <= x) → (r+1)*(r+1) <= x
(r*r <= x && !((r+1)*(r+1) <= x)) → r*r <= x < (r+1)*(r+1)

Semantic-error report

Automated Theorem Prover

Correct Program

```
method RootApprox (x:int) returns (r:int)
    requires x >= 0
    ensures r*r <= x < (r+1)*(r+1)
{
r:=0;
 while (r+1)*(r+1) <= x
        invariant  r*r <= x
        {
        r := r+1;
        }
}
```

Dafny
Frama-C.
Spark
Verifast
Why

Example: A more interesting invariant

Binary Search

Formal methods are mathematically-based techniques for the specification, development and verification of software aspects of digital systems. The mathematical basis of formal methods consists of formal logic, discrete mathematics and computer-readable languages. The use of formal methods is motivated by the expectation that, as in other engineering disciplines, performing appropriate mathematical analyses can contribute to establishing the correctness and robustness of a design.

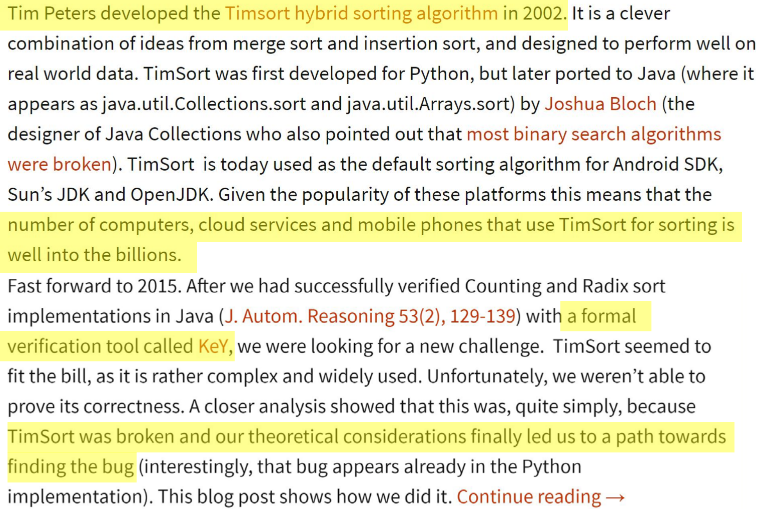
(*Formal Methods Supplement to DO-178C and DO-278A*)
Radio Technical Commission for Aeronautics
2011

> *"The future looks bright for the collaboration of verification and reasoning. Recent advances in both fields and increasingly tight interaction have already given rise to industrially relevant verification tools. We predict that this is only the beginning, and that within a decade tools based on verification technology will be as useful and widespread for software development as they are today in the hardware domain."*

Bernhard Beckert & Reiner Hähnle
Reasoning and Verification: State of the Art and Current Trends[1]
IEEE Intelligent System
2014

[1] This paper provides a representative selection of 27 verification systems.

Timsort is broken by S. de Gouw, F. de Boer, and J. Rot (2015)



Tim Peters developed the Timsort hybrid sorting algorithm in 2002. It is a clever combination of ideas from merge sort and insertion sort, and designed to perform well on real world data. TimSort was first developed for Python, but later ported to Java (where it appears as java.util.Collections.sort and java.util.Arrays.sort) by Joshua Bloch (the designer of Java Collections who also pointed out that most binary search algorithms were broken). TimSort is today used as the default sorting algorithm for Android SDK, Sun's JDK and OpenJDK. Given the popularity of these platforms this means that the number of computers, cloud services and mobile phones that use TimSort for sorting is well into the billions.

Fast forward to 2015. After we had successfully verified Counting and Radix sort implementations in Java (J. Autom. Reasoning 53(2), 129-139) with a formal verification tool called KeY, we were looking for a new challenge. TimSort seemed to fit the bill, as it is rather complex and widely used. Unfortunately, we weren't able to prove its correctness. A closer analysis showed that this was, quite simply, because TimSort was broken and our theoretical considerations finally led us to a path towards finding the bug (interestingly, that bug appears already in the Python implementation). This blog post shows how we did it. Continue reading →

- Dafny is an automatic program verifier (VCG + SMT-solver).
- Dafny is being developed by Microsoft Research.

  Dafny 1.9.9 (April 16, 2017) is the 15th stable release, since Oct 30, 2012.

- Dafny encourages using best-practice programming styles, in particular the design by contract approach.
- Dafny provides
  - Design-time feedback
  - Fluid interaction

  for accessible integrated verification.
- Dafny generates executable (.NET) code, omitting specification (ghost) constructs.