

# Construction and Verification of Software

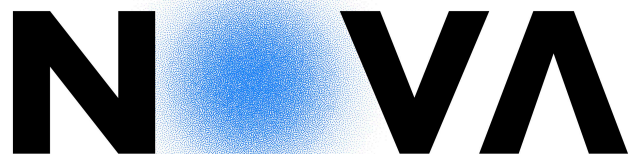
## 2021 - 2022

**MIEI - Integrated Master in Computer Science and Informatics**  
Consolidation block

**Lecture 2 - Specification and Verification**

**Bernardo Toninho** ([htoninho@fct.unl.pt](mailto:htoninho@fct.unl.pt))

based on previous editions by **João Seco** and **Luís Caires**



NOVA SCHOOL OF  
SCIENCE & TECHNOLOGY

Part I

Software Correctness

# Software Correctness: What and How

- **Key engineering concern:**  
Make sure that the software developed and constructed is “correct”.
- What does this mean?
  - Is it crash-free? (“runtime safety”)
  - Gives the right results? (“functional correctness”)
  - Does it operate effectively? (“resource conformance”)
  - Does it violate user privacy? (“security conformance”)
  - ...
- several process and methodological approaches to ensure and validate correctness exist (software engineering course)
- In this course, we cover some techniques to rigorously ensure and validate correctness **during software construction**

# Correctness against a specification

- Then what does “correct software” mean?
  - Always relative to some given (**our**) specification
- Correct means that software meets **our** specification
  - There is no such thing as the absolute “right specification”
  - But **the spec must not be wrong !**
  - Crafting good / checkable specifications can be challenging.
- It should be “easy” to check what the specification states
  - The spec **must be simple, much simpler than the code**
  - The spec should be **focused**
    - e.g., buffers are not being overrun
    - e.g., never transfer money without logging the source

# Checking Specs: Dynamic Verification

- By “dynamic verification” we mean that verification is **done at runtime**, while the program executes
- Some successful approaches:
  - **unit testing**
  - **coverage testing**
  - **regression testing**
  - **test generation**
  - **runtime monitoring**
- use runtime monitors to (continuously) check that code do not violate correctness properties
- violations causes exceptional behaviour or halt, so errors are detected after something wrong already occurred (think of a car crash, or a security leak)

# Checking Specs: Static Verification

---

- “Static verification” means verification at **compile time**
- Algorithmic reasoning about what programs do, by analyzing the source code, not by running the code
- Can ensure absence of all errors of a certain well-defined kind (e.g., “no null dereferences”)
- Can also address many complex correctness properties (e.g., functionality, absence of races, security, etc.)
- Does not introduce in performance overhead at runtime (e.g. generics work by erasure)
- Successful techniques:
  - **type checking**, as performed by the compiler
  - **extended checking**, static checking of assertions
  - **abstract interpretation**, simulates execution on a simpler decidable abstract model of runtime data

# Checking Specs: Static vs. Dynamic

---

- Dynamic Verification
  - Unsound
  - Runtime
  - No false positives (usually)
  - Execution overhead
  - Based on ad-hoc testing
- Static Verification
  - Sound
  - Compile time
  - Conservative (may have false positives)
  - Erasure of verification (no overhead)
  - Each analysis targets a specific of property
  - Some are more complex to design and use

# Checking Specs: Static Verification

---

- Specifications are the essential tool for abstraction and decomposition.
- For each program we need to know
  - in what conditions it can be used (requires/pre-conditions)
  - what are its effects (effects/ensures/post-conditions)
- **If our reasoning is sound**, the post condition can be assumed after the program's execution, provided that the pre-conditions were met at the beginning.
- We can only know as much as what is stated in the post-condition.



# Part II

## Specification and Verification

# Contract-based Verification

---

- Axiomatic approaches based on **Hoare Logic** (Pre- and post-conditions)
- If pre-condition holds **and** the program terminates **then** the post-condition holds.
- If all components are verified then all contracts are fulfilled in all cases.
- If a component does not fulfill a contract then no guaranties are given about the system's behavior.

# What may specs look like?

---

- A classical example is the use of “assertions”
  - You may have used assertions before (IP, POO, AED)?
- A simple and fine-grained spec is the “Hoare triple”:

$$\{ A \} P \{ B \}$$

- **A** and **B** are assertions (conditions on the program state)
- **P** is the piece of code we want to talk about
- The Hoare triple says:
  - If program **P** starts in a state satisfying **A**, then, if it terminates, the resulting state satisfies **B**.
  - **A** is called the “pre-condition”.
  - **B** is called the “post-condition”.

# Interface contracts in ADT specs

---

- ADT specifications (we will detail this later) involve method contracts, expressed as assertions

```
method P(... parameters ...)  
  requires PRE  
  ensures POST  
  modifies MOD  
  {  
    method code  
  }
```

- The method call  $P(\dots)$ , whenever started in a state that satisfies **PRE**, if it terminates, always ends in a state that satisfies **POST**, and only has effects on **MOD**

# Invariants in ADT specs

---

- ADT specifications (we will detail this later) may involve representation invariants and abstraction mappings also expressed as assertions

```
class C {  
    var v : T  
    invariant REPINV  
    invariant ABSMAP  
  
    ... methods ...  
}
```

- ADT C's implementation relies on a representation type T that satisfies the representation invariant REPINV and maps into the abstract type as specified by ABSMAP

# Stack Example : A glimpse of Dafny code

```
class Stack {  
  
  var elements:array<int>;  
  var count: int;  
  var MAX: int;  
  
  function StackInv(): bool  
  reads `count, `MAX, `elements  
  {  
    0 < MAX && 0 <= count <= MAX && elements.Length == MAX  
  }  
  
  constructor()  
  ensures StackInv()  
  {  
    MAX := 10;  
    elements := new int[10];  
    count := 0;  
  }  
}
```

# Stack Example : A glimpse of Dafny code

```
class Stack {
```

```
...
```

```
method push(x:int)
  requires StackInv() && notFull()
  ensures StackInv() && notEmpty()
  modifies elements, `count
{
  elements[count] := x;
  count := count + 1;
}
```

```
method pop() returns (x:int)
  requires StackInv() && notEmpty()
  ensures StackInv() && notFull()
  modifies elements, `count
{
  count := count - 1;
  x := elements[count];
}
```

```
function notFull():bool
reads `count, `MAX
{ count < MAX }
```

```
function notEmpty():bool
reads `count, `MAX
{ count > 0 }
```

# How are specs verified?

---

- Written in a logic used to prove properties of programs
- What kinds of properties are we interested in?
  - Safety properties (partial correctness):
    - If the program terminates (delivers an outcome), then the final state satisfies some property.
  - Liveness properties (total correctness)
    - The program terminates (at least under certain conditions)
- Hoare logic is the “mother of all program logics”: It provides a foundation for most program logics for imperative programming languages.
- Reason of HL success: compositional verification at the level of the programming language constructs.



# Dafny

“Dafny is an imperative object-based language with built-in specification constructs. The Dafny static program verifier can be used to verify the functional correctness of programs. The specifications include pre- and postconditions, frame specifications (read and write sets), and termination metrics”  
Leino, Koenig, 2010

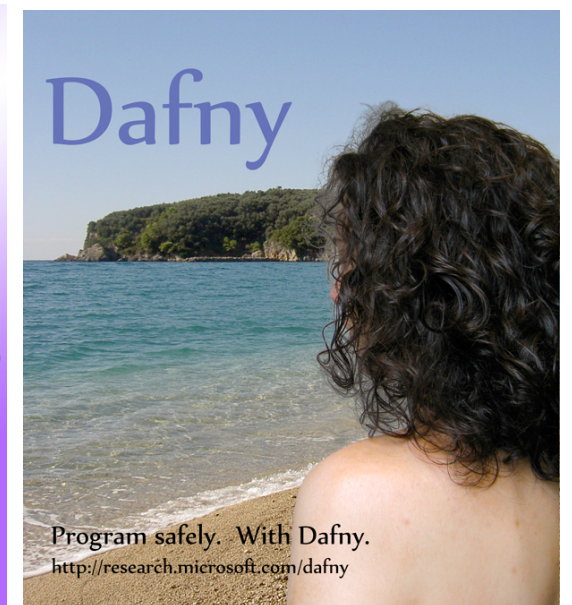
## Dafny: An Automatic Program Verifier for Functional Correctness

K. Rustan M. Leino  
Microsoft Research  
leino@microsoft.com


### Abstract

Traditionally, the full verification of a program's functional correctness has been obtained with pen and paper or with interactive proof assistants, whereas only reduced verification tasks, such as extended static checking, have enjoyed the automation offered by satisfiability-modulo-theories (SMT) solvers. More recently, powerful SMT solvers and well-designed program verifiers are starting to break that tradition, thus reducing the effort involved in doing full verification.

This paper gives a tour of the language and verifier Dafny, which has been used to verify the functional correctness of a number of challenging pointer-based programs. The paper describes the features incorporated in Dafny, illustrating their use by small examples and giving a taste of how they are coded for an SMT solver. As a larger case study, the paper shows the full functional specification of the Schorr-Waite algorithm in Dafny.




# Dafny @ GitHub



Search or jump to...

[Pull requests](#) [Issues](#) [Marketplace](#) [Explore](#)


 **dafny-lang / dafny** Public

[Watch](#) 76 [Fork](#) 188

[Code](#) [Issues](#) 348 [Pull requests](#) 41 [Discussions](#) [Actions](#) [Projects](#) [Wiki](#) [Security](#) [Insights](#)

[master](#) 50 branches 25 tags

[Go to file](#) [Add file](#) [Code](#)

 **robin-aws** Bump Boogie to 2.13.0 (#1892) ... ✓ 9f5e2f0 17 hours ago 🕒 4,738 commits

📁 .github	feat: New language constructs for the python backend (#1786)	14 days ago
📁 Binaries	chore: xUnit-based lit test runner (#680)	5 months ago
📁 Scripts	feat: Add <code>return s</code> , <code>if s</code> , and <code>block s</code> to the python compiler (#188...	3 days ago
📁 Source	Bump Boogie to 2.13.0 (#1892)	17 hours ago
📁 Test	feat: Add <code>return s</code> , <code>if s</code> , and <code>block s</code> to the python compiler (#188...	3 days ago
📁 Util	Remove trailing whitespace (#1078)	14 months ago
📁 docs	Fixed so that favicon appears both online and locally (#1885)	7 days ago
📁 third_party/Coco	fix: Use correct property to specify target framework in Coco.csproj ...	21 days ago
📄 .editorconfig	chore: Enable <code>dotnet_style_parentheses_in_other_binary_opera...</code>	3 months ago
📄 .gitattributes	Preserve line endings for dafny and dafny-server scripts	4 years ago
📄 .gitignore	feat: Prevent changes unrelated to a proof from changing the verific...	3 months ago
📄 .gitmodules	add dafny libraries as submodule (#1446)	5 months ago

### About


Dafny is a verification-aware programming language

[dafny-lang.github.io/dafny/](#)

- [Readme](#)
- [View license](#)
- [Code of conduct](#)
- 1.6k stars
- 76 watching
- 188 forks

### Releases


 22

-  **Dafny 3.4.2** Latest  
17 days ago
- + 21 releases

### Packages

# Dafny in VS Code

Extension: Dafny ×



**Dafny** v2.2.0  
dafny-lang | 1,228 | ★★★★★  
Dafny for Visual Studio Code  
Disable ▾ Uninstall ▾ ⚙️  
This extension is enabled globally.

[Details](#) [Feature Contributions](#) [Changelog](#) [Runtime Status](#)

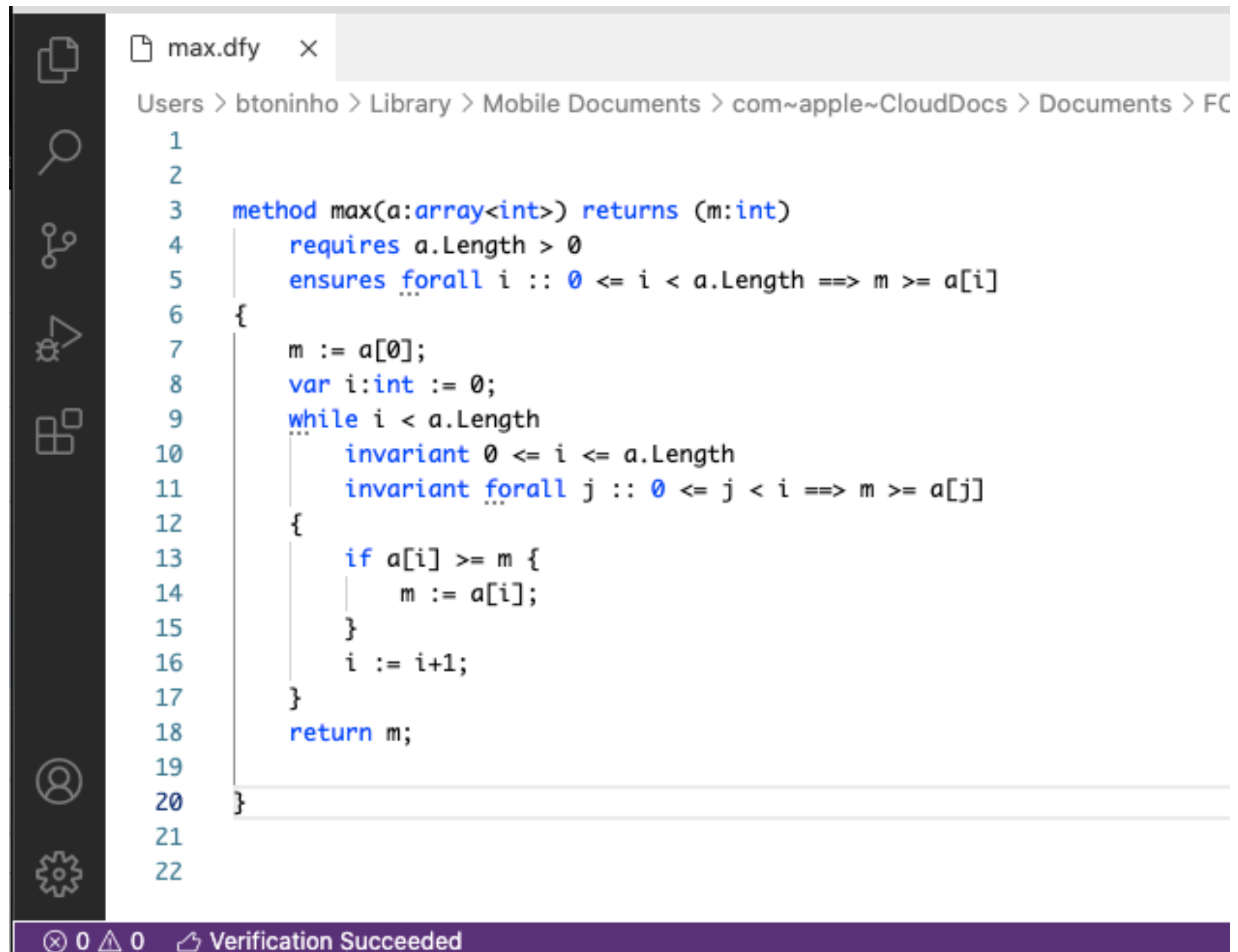
## Dafny for Visual Studio Code

This extension adds *Dafny 3* support to Visual Studio Code. If you require *Dafny 2* support, consider using the [legacy extension](#). This VSCode plugin requires the Dafny language server (shipped with the Dafny release since v3.1.0). The plugin will install it automatically upon first use.

### Features

- Compatible to **Dafny 3.3.0**.
- **Compile and Run** `.dfy` files.
- **Verification** as one types.
- **Syntax highlighting** thanks to [sublime-dafny](#). See file `LICENSE_sublime-dafny.rst` for license.
- Display **CounterExample** for failing proof.
- **IntelliSense** to suggest symbols.
- **GoToDefinition** to quickly navigate.
- **Hover Information** for symbols.

# Dafny in VS Code



```
1
2
3  method max(a:array<int>) returns (m:int)
4      requires a.Length > 0
5      ensures forall i :: 0 <= i < a.Length ==> m >= a[i]
6  {
7      m := a[0];
8      var i:int := 0;
9      while i < a.Length
10         invariant 0 <= i <= a.Length
11         invariant forall j :: 0 <= j < i ==> m >= a[j]
12     {
13         if a[i] >= m {
14             m := a[i];
15         }
16         i := i+1;
17     }
18     return m;
19 }
20
21
22
```

⊗ 0 △ 0 Verification Succeeded



# Dafny Documentation

This site contains links to Dafny documentation.

[Project site for releases, issues, installation instructions, and source code](#)

- Quick start material:
  - Dafny [Quick Reference](#)
  - [Getting started tutorial](#), focusing mostly on simple imperative programs
  - [Cheatsheet](#): basic Dafny syntax on two pages
- Detailed documents for programmers
  - [Dafny Reference Manual](#)
  - Language reference for the [Dafny type system](#), which also describes available expressions for each type
  - [Style Guide for Dafny programs](#)
- Dafny Tutorials
  - [Introduction to Dafny](#)
  - [Value Types](#)
  - [Sets](#)
  - [Sequences](#)
  - [Lemmas and Induction](#)
  - [Modules](#)
  - [Termination](#)

# Basic Program Specs (Hoare Logic)

---



C.A. R. HOARE  
United Kingdom – **1980**

For his fundamental contributions to the definition and design  
of programming languages.

# Hoare Logic (1969)

## An Axiomatic Basis for Computer Programming

C. A. R. HOARE

*The Queen's University of Belfast,\* Northern Ireland*

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.

KEY WORDS AND PHRASES: axiomatic method, theory of programming, proofs of programs, formal language definition, programming language design, machine-independent programming, program documentation

CR CATEGORY: 4.0, 4.21, 4.22, 5.20, 5.21, 5.23, 5.24

of axioms it is possible to deduce such simple theorems as:

$$x = x + y \times 0$$

$$y \leq r \supset r + y \times q = (r - y) + y \times (1 + q)$$

The proof of the second of these is:

$$\text{A5 } (r - y) + y \times (1 + q)$$

$$= (r - y) + (y \times 1 + y \times q)$$

$$\text{A9}$$

$$= (r - y) + (y + y \times q)$$

$$\text{A3}$$

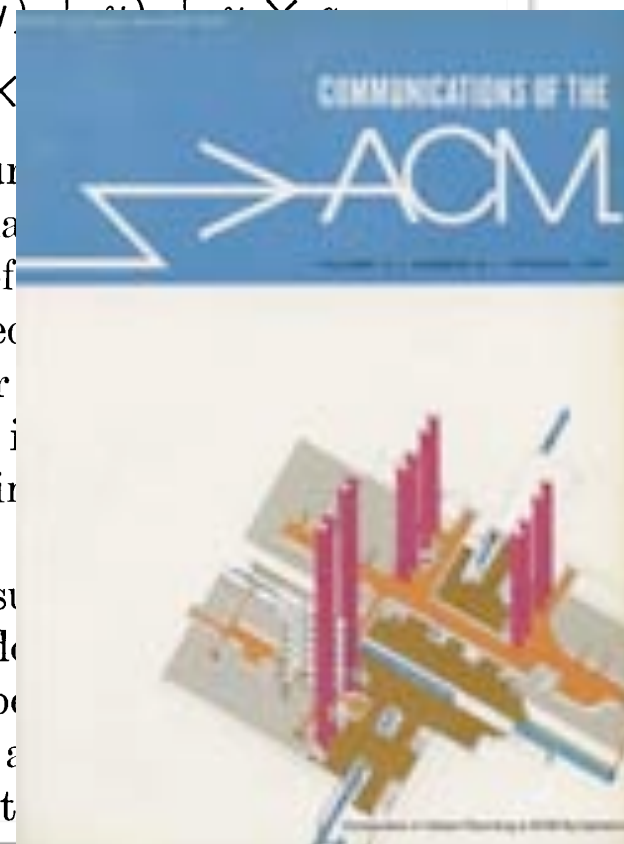
$$= ((r - y) + y) + y \times q$$

$$\text{A6}$$

$$= r + y \times q$$

The axioms A1 to A9 are, of course, true of the usual infinite set of integers in mathematics; they are also true of the finite sets of integers manipulated by computers provided the integers are confined to *nonnegative* numbers. Their validity is independent of the size of the set; furthermore, it is independent of the choice of technique applied in the proof; for example:

(1) Strict interpretation: the result of an overflow operation does not exist; when overflowing program never completes its operation; in this case, the equalities of A1 to A9 are true only if both sides exist or fail to exist



# Simple Programming Language

---

$E ::=$  Expressions

$num$

$x$

$E + E \mid \dots$

$E < E \mid \dots$

$E \text{ and } E \dots$

Integer

Variable

Integer operators

Relational operators

Boolean operators

$P ::=$

skip

$x := E$

$P; P$

if  $E$  then  $P$  else  $P$

while  $E$  do  $P$

Programs

No op

Assignment

Sequential Composition

Conditional

Iteration



# States and State Transformers

- An **imperative** program is a **state transformer**. It transforms an **initial** state into a **target** state.
- What is a program state? An assignment of values to state variables:

$$\sigma = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$$

- An imperative program transforms states into states

$$P \triangleq x := y + x; z := z - x$$

- If  $P$  is executed in state  $\sigma$  it yields state  $\sigma'$  where

$$\sigma' = \{x \mapsto 3, y \mapsto 2, z \mapsto 0\}$$

- We may say that  $P$  transforms  $\sigma$  in  $\sigma'$
- $P$  is only defined on states  $\sigma$  where  $\text{vars}(P) \subseteq \text{dom}(\sigma)$

# States and Assertions

---

- A (safety) property is a set of (safe) states
- Essentially an assertion is a boolean expression that only depends on observing program (state) variables
- An assertion is just a **pure observation**, it is either true or false, its evaluation **does not change** the state.
- In general, one may use all the expressiveness of (first order) logic in assertions (e.g. quantifiers, etc...).
- The assertion language is part of the specification language, separate from the programming language.
- In some cases, assertions may be expressed in the programming language (e.g. a subset of Dafny).

# Part IV

## A bit of History

# Some bits of history ... (extra)

---

Kick off:

- **“Checking a large routine”**

# Turing

Kick off:

- **“Checking a large routine”**

“How can one check a routine in the sense of making sure that it is right? In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.”

*Alan Turing, 24th June 1949*



Friday, 24th June.

Checking a large routine. by Dr. A. Turing.

How can one check a routine in the sense of making sure that it is right?

In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.

Consider the analogy of checking an addition. If it is given as:

1374
5906
6719
4537
7768
—
26104

one must check the whole at one sitting, because of the carries.

But if the totals for the various columns are given, as below:

1374
5906

# Assertions

Second boost:

## – Floyd's Assertion Method

*Robert Floyd's, "Assigning Meanings to Programs," opened the field of program verification. His basic idea was to attach so-called "tags" in the form of logical assertions to individual program statements or branches that would define the effects of the program based on a formal semantic definition of the programming language.*

*R. Floyd, MFCS, June 1967*

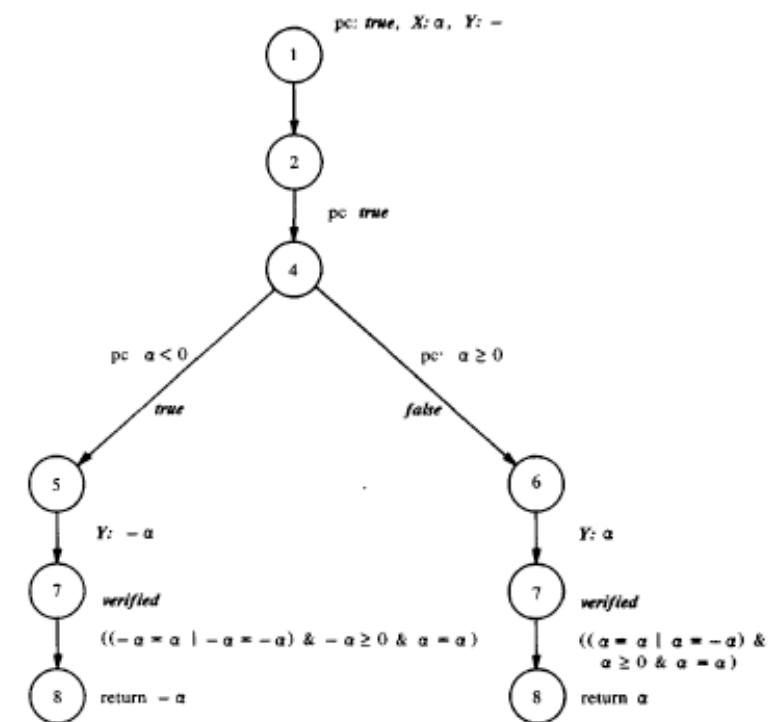


FIGURE 3. Symbolic execution tree for procedure *ABSOLUTE*.

# Assertions

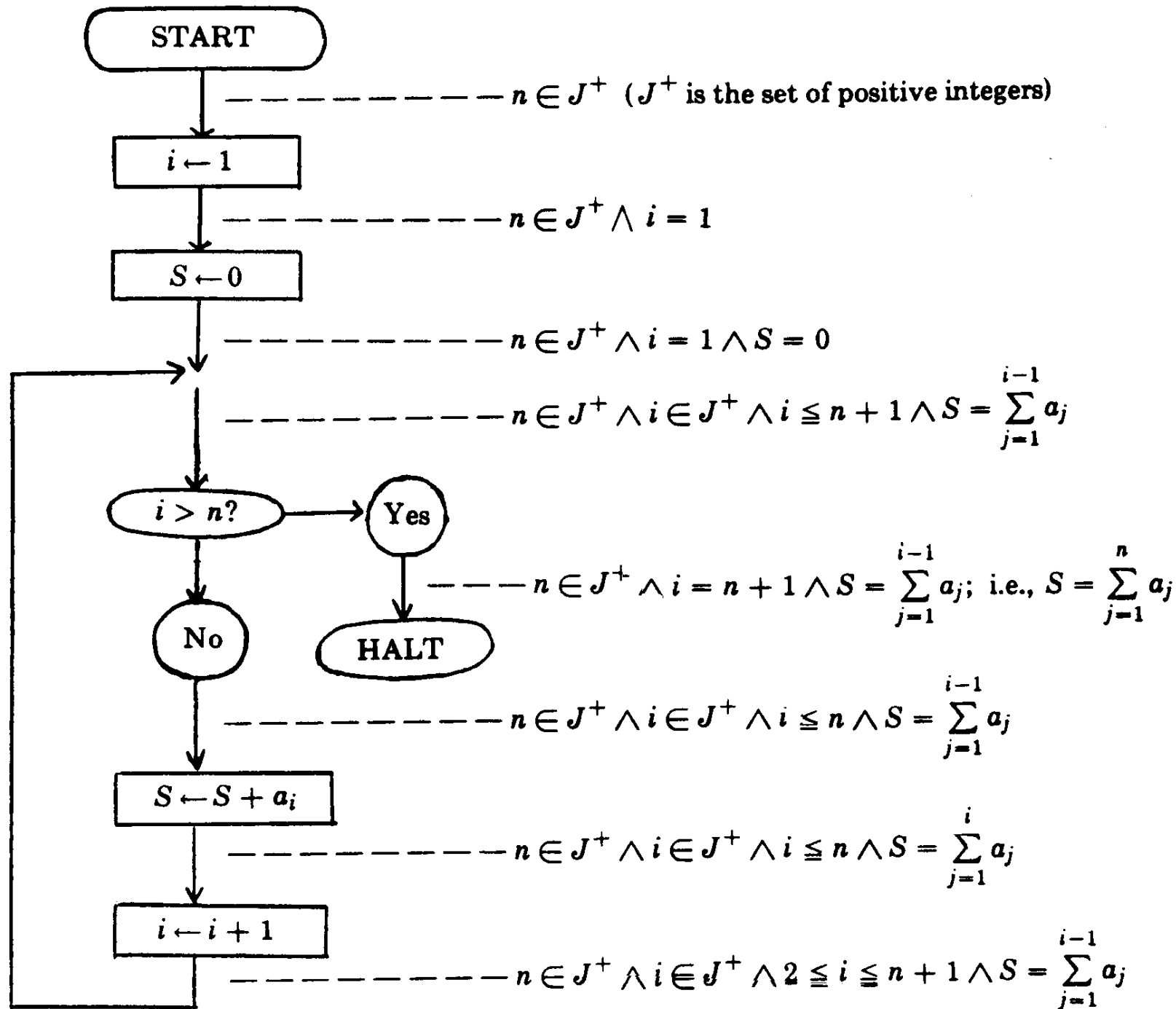


FIGURE 1. Flowchart of program to compute  $S = \sum_{j=1}^n a_j$  ( $n \geq 0$ )



# Language Based Program Specs

Lift Off:

## – Hoare Logic

*“Computer Programming is an exact science in that all the properties of a program and all consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning.”*

*Tony Hoare, CACM 1969*



### AXIOM 1: ASSIGNMENT AXIOM

$$\{p[t/x]\} x := t \{p\}.$$

### RULE 2: COMPOSITION RULE

$$\frac{\{p\} S_1 \{r\}, \{r\} S_2 \{q\}}{\{p\} S_1; S_2 \{q\}}.$$

### RULE 3: if-then-else RULE

$$\frac{\{p \wedge e\} S_1 \{q\}, \{p \wedge \neg e\} S_2 \{q\}}{\{p\} \text{ if } e \text{ then } S_1 \text{ else } S_2 \text{ fi } \{q\}}$$

### RULE 4: while RULE

$$\frac{\{p \wedge e\} S \{p\}}{\{p\} \text{ while } e \text{ do } S \text{ od } \{p \wedge \neg e\}}$$



# The Weakest Precondition

Programming  
Languages

T.A. Standish  
Editor

## Guarded Commands, Nondeterminacy and Formal Derivation of Programs

Edsger W. Dijkstra  
Burroughs Corporation

---

So-called “guarded commands” are introduced as a building block for alternative and repetitive constructs that allow nondeterministic program components for which at least the activity evoked, but possibly even the final state, is not necessarily uniquely determined by the initial state. For the formal derivation of programs expressed in terms of these constructs, a calculus will be shown.

**Key Words and Phrases:** programming languages, sequencing primitives, program semantics, programming language semantics, nondeterminacy, case-construction, repetition, termination, correctness proof, derivation of programs, programming methodology

**CR Categories:** 4.20, 4.22

### 1. Introduction

In Section 2, two statements, an alternative construct and a repetitive construct, are introduced, together with an intuitive (mechanistic) definition of their semantics. The basic building block for both of them is the so-called “guarded command,” a statement list prefixed by a boolean expression: only when this boolean expression is initially true, is the statement list eligible for execution. The potential nondeterminacy allows us to map otherwise (trivially) different programs on the same program text, a circumstance that seems largely responsible for the fact that programs can now be derived in a manner more systematic than before.

In Section 3, after a prelude defining the notation, a formal definition of the semantics of the two constructs is given, together with two theorems for each of the constructs (without proof).

In Section 4, it is shown how, based upon the above, a formal calculus for the derivation of programs can be founded. We would like to stress that we do not present “an algorithm” for the derivation of programs: we have used the term “a calculus” for a formal discipline—a set of rules—such that, if applied successfully: (1) it will have derived a correct program; and (2) it will tell us that we have reached such a goal. (We use the term as in “integral calculus.”)

# Closer to the present

Still relevant

## – Hoare Logic

*“The axiomatic method gives an objective criterion of the quality of a programming language, and the ease with which programmers could use it. The latest response comes from hardware designers, who are using axioms in anger to define the properties of modern multicore chips with weak memory consistency.”*

*Tony Hoare, CACM 2009*

Retrospective: An Axiomatic Basis for Computer Programming | October 2009 | Communications of the ACM

http://cacm.acm.org/magazines/2009/10/42360-re RSS Google

Gmail ISI OutSystems S...io 4.1 Help Google eracareers CLIP - Autenticação UROP - Und...es Program

ACM.ORG JOIN ACM ABOUT COMMUNICATION

COMMUNICATIONS OF THE ACM TRUSTED INSIGHTS FOR COMPUTING'S LEADING PROFESSIONALS

Home News Blogs Opinion Browse by Subject Magazine Archive Careers ACM Resou

Home » Magazine Archive » 2009 » No. 10 » Retrospective: An Axiomatic Basis for Computer Programming » Full Text

this article

- Abstract
- Full Text (HTML)
- Full Text (PDF)
- User Comments (0)
- In the Digital Edition
- In the Digital Library

article contents

- Introduction
- Retrospective (1969–1999)
- Progress (1999–2009)
- Prospective (2009–)
- The End
- Author
- Footnotes
- Figures


VIEWPOINTS

### Retrospective: An Axiomatic Basis for Computer Programming

C.A.R. Hoare revisits his past *Communications* article on the axiomatic approach to programming and uses it as a touchstone for the future.

C.A.R. Hoare

Communications of the ACM  
Vol. 52 No. 10, Pages 30-32  
10.1145/1562764.1562779



C.A.R. Hoare attending the NATO Software Engineering Techniques Conference in 1969.  
Credit: Robert M. McClure

This month marks the 40th anniversary of the publication of the first article I wrote as an academic.<sup>a</sup> I have been invited to give my personal view of the advances that have been made in the subject since then, and the further advances that remain to be made. Which of them did I expect, and which of them surprised me?

[back to top](#)

### Retrospective (1969–1999)

My first job (1960–1968) was in the computer industry; and my first major project was to lead a team that implemented an early compiler for ALGOL 60. Our compiler was directly structured on the syntax of the language, so elegantly and so rigorously formalized as a context-free language. But the semantics of the language was even more important, and that was left informal in the language definition. It occurred to me that an elegant



# Extended Static Checking

## Spec#

*Spec# is an extension of the object-oriented language C#. It extends the type system to include non-null types and checked exceptions. It provides method contracts in the form of pre- and postconditions as well as object invariants.*

*Barnett, Leino, Schulte, 2004*

### The Spec# Programming System: An Overview

Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte

Microsoft Research, Redmond, WA, USA

{mbarnett, leino, schulte}@microsoft.com

Manuscript KRML 136, 12 October 2004. To appear in CASSIS 2004 proceedings.

**Abstract.** The Spec# programming system is a new attempt at a more cost effective way to develop and maintain high-quality software. This paper describes the goals and architecture of the Spec# programming system, consisting of the object-oriented Spec# programming language, the Spec# compiler, and the Boogie static program verifier. The language includes constructs for writing specifications that capture programmer intentions about how methods and data are to be used, the compiler emits run-time checks to enforce these specifications, and the verifier can check the consistency between a program and its specifications.



# Dafny

## Dafny

*Dafny is an imperative object-based language with built-in specification constructs. The Dafny static program verifier can be used to verify the functional correctness of programs. The specifications include pre- and postconditions, frame specifications (read and write sets), and termination metrics*

*Leino, Koenig, 2010*

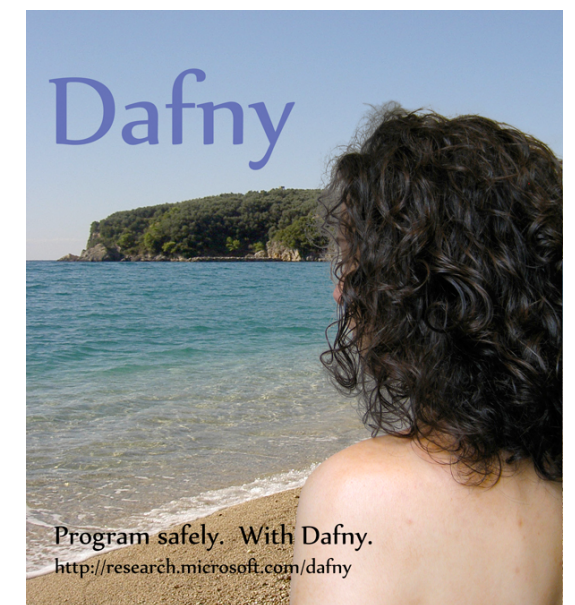
### Dafny: An Automatic Program Verifier for Functional Correctness

K. Rustan M. Leino  
Microsoft Research  
leino@microsoft.com

#### Abstract

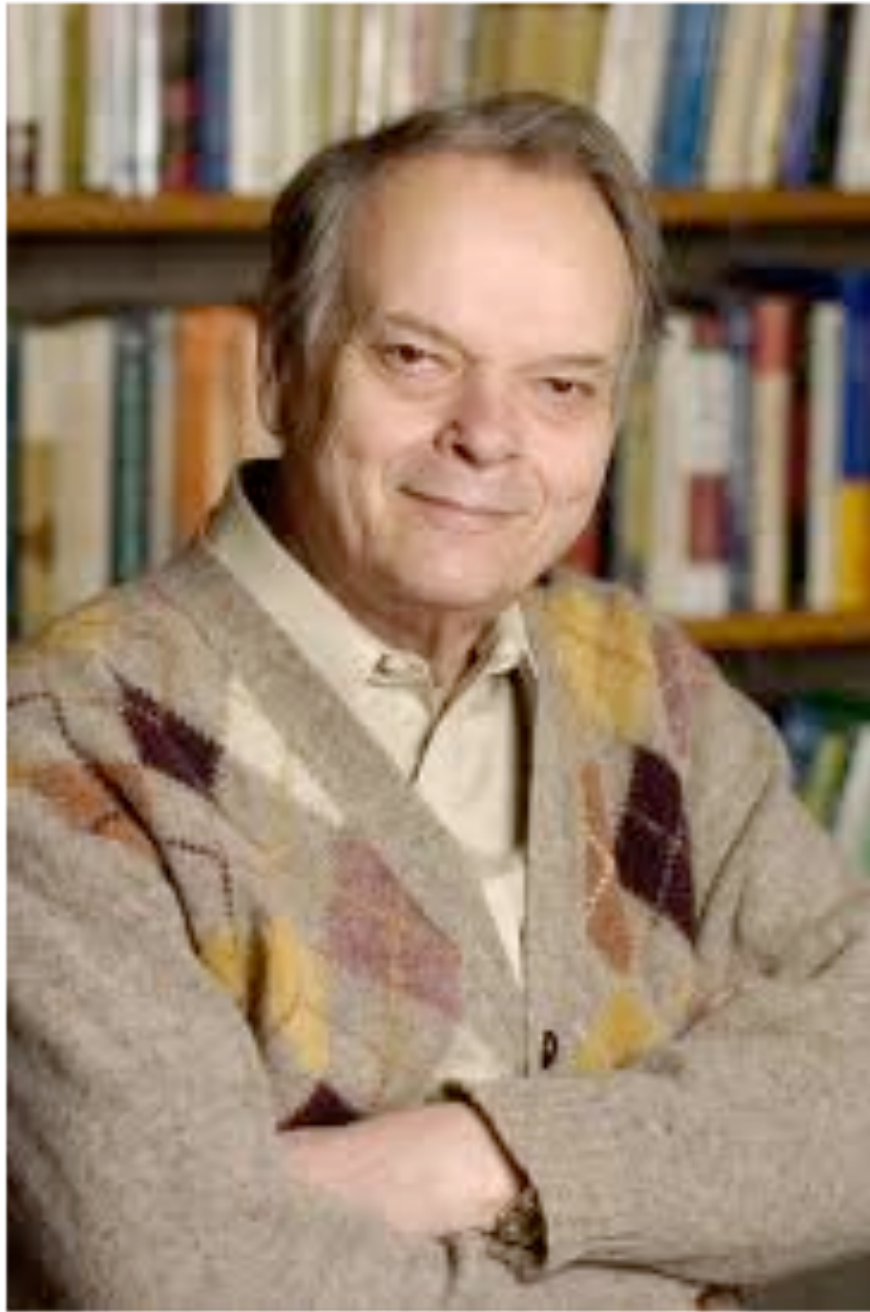
Traditionally, the full verification of a program's functional correctness has been obtained with pen and paper or with interactive proof assistants, whereas only reduced verification tasks, such as extended static checking, have enjoyed the automation offered by satisfiability-modulo-theories (SMT) solvers. More recently, powerful SMT solvers and well-designed program verifiers are starting to break that tradition, thus reducing the effort involved in doing full verification.

This paper gives a tour of the language and verifier Dafny, which has been used to verify the functional correctness of a number of challenging pointer-based programs. The paper describes the features incorporated in Dafny, illustrating their use by small examples and giving a taste of how they are coded for an SMT solver. As a larger case study, the paper shows the full functional specification of the Schorr-Waite algorithm in Dafny.





# Separation Logic



*John C. Reynolds*



$$\frac{s, h \models P * (P \multimap Q)}{s, h \models Q} \quad \frac{\{P\} C \{Q\}}{\{P * R\} C \{Q * R\}} \text{ mod}(C) \cap \text{fv}(R) = \emptyset$$

*Peter O'Hearn*

# Verifast

## Verifast

*VeriFast* is a verifier for single-threaded and multithreaded C and Java programs annotated with preconditions and postconditions written in separation logic.

Jacobs, Smans, Piessens, 2010

*NB: separation logic is a spec language for talking about programs that allocate memory and use references*

```
public void broadcast_message(String message) throws IOException
    //@ requires room(this) && message != null;
    //@ ensures room(this);
{
    //@ open room(this);
    //@ assert foreach(?members0, _);
    List membersList = this.members;
    Iterator iter = membersList.iterator();
    boolean hasNext = iter.hasNext();
    //@ length_nonnegative(members0);
    while (hasNext)
        /*@
        invariant
            foreach<Member>(?members, @member) && iter(iter, membersList, members, ?i)
            && hasNext == (i < length(members)) && 0 <= i && i <= length(members);
        @*/
    {
        Object o = iter.next();
        Member member = (Member)o;
        //@ mem_nth(i, members);
        //@ foreach_remove<Member>(member, members);
        //@ open member(member);
        Writer writer = member.writer;
        writer.write(message);
        writer.write("\r\n");
        writer.flush();
        //@ close member(member);
        //@ foreach_unremove<Member>(member, members);
        hasNext = iter.hasNext();
    }
    //@ iter_dispose(iter);
    //@ close room(this);
}
```



Part V

Hoare Logic

# Program Proofs in Hoare Logic

- A program proof in Hoare logic adds assertions between program statements, making sure that all Hoare triples are satisfied/valid.
- For example, consider the code snippet

```
if (x > y) {  
    z := x  
} else {  
    z := y  
}
```



# Program Proofs in Hoare Logic

- A Hoare Logic “proof” may look like

```
{ true }
```

```
if (x > y) {
```

```
    { (x > y) }
```

```
    z := x;
```

```
    { (x > y) && (z == x) }
```

```
}
```

```
else {
```

```
    { (x <= y) }
```

```
    z := y;
```

```
    { (x <= y) && (z == y) }
```

```
}
```

```
{ (x > y) && (z == x) || (x <= y) && (z == y) }
```

```
{ z == max(x, y) }
```

# In Dafny

```
function max(x:int, y:int):int { if x > y then x else y }
```

```
method maxImp(x:int, y:int) returns (z:int)
  ensures z == max(x,y)
{
  assert true;
  if (x>y) {
    assert x > y ;
    z := x;
    assert z > y && z == x;
    assert z >= y && z == x;
  } else {
    assert x <= y;
    z := y ;
    assert z >= x && z == y;
  }
  assert (z >= y && z == x) || (z >= x && z == y);
  assert z == max(x,y);
}
```

# Interlude

# Verification of Functions

# Functions as Specifications

- We often use functions in the specification of (imperative) methods.
- Functions must be **pure** (i.e, no state).
- Functions must **provably terminate** (aka total).
- Pure total functions  $\cong$  Mathematical functions.

```
function max(x:int, y:int):int { if x > y then x else y }
```

```
method maxImp(x:int, y:int) returns (z:int)  
  ensures z == max(x,y)  
  {...}
```

- Dafny proves, for all  $x$  and  $y$ ,  $z$  is equal to  $\text{max}(x, y)$
- What is the relationship between **max** and “math” max?

# Functions as Specifications

---

- We can prove that **max** is actually equivalent to mathematical max (**adequacy**).
- Since Dafny functions are both **pure** and **total**, we can reason about them using relatively simple techniques:
  - Calculation by evaluation (e.g.  $\text{max}(2, 3) \hookrightarrow 3$ )
  - Mathematical induction
  - Structural induction
- Reasoning about functions without totality is much harder (but doable).
- Reasoning about functions that manipulate state is as hard as reasoning about imperative code.

# Function Evaluation

- Identify function calls with their definitions, substituting args for formal parameters (**referential transparency**).
- Same as how it works in mathematics:
  - If  $f(x) = x^2 - x$  then  $f(5)$  is the same as 20.
  - $\max(2, 3)$  is the same as 3.
- In a general purpose lang., functions might not terminate and so we distinguish:
  - Evaluation ( $e \hookrightarrow v$ ) —  $e$  evaluates to value  $v$  (no “calculation” left).
  - Reduction ( $e \Longrightarrow e'$ ) —  $e$  computes to expr.  $e'$  (some “calculation” may remain).
- In our setting we need not make this distinction.

# Proving functional adequacy

- Assume that values (e.g., numbers, booleans) and ops. (e.g.,  $+$ ,  $-$ ,  $*$ , ...) map directly onto their mathematical counterparts.
- For non-recursive functions, proceed by **calculation**.
- For all  $n, m$  integers,  $\max(n, m) = n$  if  $n > m$ ;  $\max(n, m) = m$  otherwise.
- Calculation:  $\max(n, m) = \text{if } n > m \text{ then } n \text{ else } m$
- Case #1:  $n > m \hookrightarrow \text{true}$ ,  $\max(n, m) = n$  ✓
- Case #2:  $n > m \hookrightarrow \text{false}$ ,  $\max(n, m) = m$  ✓

# Proving functional adequacy

- For recursive functions, we use **induction**.
- Mathematical induction:
  - Proof technique for statements of the form  $\forall n . P(n)$ .
  - Prove base case  $P(0)$
  - Prove inductive case  $\forall k . P(k) \longrightarrow P(k + 1)$
  - We can construct an argument for any  $n$  by “iterating” the inductive case up from the base case.
- Many equivalent variations exist:
  - Base case as  $P(k)$  for some specific  $k$  ( $\forall n \geq k . P(n)$ )
  - **Strong induction**, inductive case as  $\forall k . (\forall n \leq k . P(n)) \longrightarrow P(k + 1)$
  - ...



# Proving functional adequacy

---

- Lets prove the following function is equivalent to +:

```
function slowAdd(a:nat, b:int) : int {  
  if (a==0) then b else 1+slowAdd(a-1,b)  
}
```


- $\forall n, m. \text{slowAdd}(n, m) = n + m$
- By induction on n

# Proving functional adequacy

---

- Lets prove the following function is equivalent to +:

```
function slowAdd(a:nat, b:int) : int {  
  if (a==0) then b else 1+slowAdd(a-1,b)  
}
```


- $\forall n, m. \text{slowAdd}(n, m) = n + m$
- By induction on n
  - Base case:  $\text{slowAdd}(0, m) \hookrightarrow m$  

# Proving functional adequacy

---

- Lets prove the following function is equivalent to +:

```
function slowAdd(a:nat, b:int) : int {  
  if (a==0) then b else 1+slowAdd(a-1,b)  
}
```


- $\forall n, m. \text{slowAdd}(n, m) = n + m$
- By induction on n
  - Base case:  $\text{slowAdd}(0, m) \hookrightarrow m$  
  - Inductive case:
    - Assume  $\forall m. \text{slowAdd}(k, m) \hookrightarrow k + m$ , for some k.
    - Prove  $\text{slowAdd}(k + 1, m) \hookrightarrow k + 1 + m$

# Proving functional adequacy

---

- Lets prove the following function is equivalent to +:

```
function slowAdd(a:nat, b:int) : int {  
  if (a==0) then b else 1+slowAdd(a-1,b)  
}
```

- $\forall n, m. \text{slowAdd}(n, m) = n + m$
- By induction on n
  - Base case:  $\text{slowAdd}(0, m) \hookrightarrow m$  
  - Inductive case:
    - Assume  $\forall m. \text{slowAdd}(k, m) \hookrightarrow k + m$ , for some k.
    - Prove  $\text{slowAdd}(k + 1, m) \hookrightarrow k + 1 + m$
    - $\text{slowAdd}(k + 1, m) = 1 + \text{slowAdd}(k, m)$

# Proving functional adequacy

- Lets prove the following function is equivalent to +:


```
function slowAdd(a:nat, b:int) : int {  
  if (a==0) then b else 1+slowAdd(a-1,b)  
}
```

- $\forall n, m. \text{slowAdd}(n, m) = n + m$
- By induction on n
  - Base case:  $\text{slowAdd}(0, m) \hookrightarrow m$  ✓
  - Inductive case:
    - Assume  $\forall m. \text{slowAdd}(k, m) \hookrightarrow k + m$ , for some k.
    - Prove  $\forall m'. \text{slowAdd}(k + 1, m') \hookrightarrow k + 1 + m'$
    - $\text{slowAdd}(k + 1, m') = 1 + \text{slowAdd}(k, m')$
    - By i.h.  $= 1 + k + m' = k + 1 + m'$  ✓

# Generalizing the inductive hypothesis

- Sometimes the “obvious” statement is not general enough:

```
function factAcc(n:nat, a:int) : int {  
  if (n==0) then a else factAcc(n-1,n*a)  
}
```

- $\forall n. \text{factAcc}(n,1) = n!$
- By induction on  $n$ 
  - Base case:  $\text{factAcc}(0,1) \hookrightarrow 1$  
  - Inductive case:
    - Assume  $\text{factAcc}(k,1) \hookrightarrow k!$ , for some  $k$ .
    - Prove  $\text{factAcc}(k+1,1) \hookrightarrow (k+1)!$
    - $\text{factAcc}(k+1,1) = \text{factAcc}(k, k+1)$
    - We are stuck...



# Generalizing the inductive hypothesis

```
function factAcc(n:nat, a:int) : int {  
  if (n==0) then a else factAcc(n-1,n*a)  
}
```

- Instead, prove  $\forall n, m. \text{factAcc}(n, m) = m \times n!$
- By induction on n
  - Base case:  $\forall m. \text{factAcc}(0, m) \hookrightarrow m = m \times 0!$
  - Inductive case:
    - Assume  $\forall m. \text{factAcc}(k, m) \hookrightarrow m \times k!$ , for some k.
    - Prove  $\forall m'. \text{factAcc}(k+1, m') \hookrightarrow m' \times (k+1)!$
    - $\text{factAcc}(k+1, m') = \text{factAcc}(k, m' \times (k+1))$
    - by i.h. ( $m = m' \times (k+1)$ ):  
 $= m' \times (k+1) \times k! = m' \times (k+1)! \quad \checkmark$

# Generalizing the inductive hypothesis

---

- Related to inventing “good” loop invariants (later!).
- Sometimes this isn't enough:

```
function fib(n:nat) : int {  
  if n==0 then 0  
  else if n==1 then 1  
  else fib(n-1)+fib(n-2)  
}
```

- Use **strong induction**... (Exercise!)

Part V (redux)

Hoare Logic

# Program Proofs in Hoare Logic

- A Hoare Logic “proof” may look like

```
{ true }  
if (x > y) {  
    { (x > y) }  
    z := x;  
    { (x > y) && (z == x) }  
}  
else {  
    { (x <= y) }  
    z := y;  
    { (x <= y) && (z == y) }  
}  
{ (x > y) && (z == x) || (x <= y) && (z == y) }  
{ z == max(x, y) }
```

# Example: Rule for Sequence

- A sequence defines a dependency on the effects of both program statements.

$$\frac{\{A\} P \{B\} \quad \{B\} Q \{C\}}{\{A\} P; Q \{C\}}$$

- If  $\{A\} P \{B\}$  and  $\{B\} Q \{C\}$  then  $\{A\} P; Q \{C\}$

# Rules of Hoare Logic (general form)

- The inference rules of Hoare logic are used to derive (valid) Hoare triples given some already derived Hoare triples

$$\frac{\{A_1\} P_1 \{B_1\} \dots \{A_n\} P_n \{B_n\}}{\{A\} C(P_1, \dots, P_n) \{B\}}$$

- What is nice here:
  - the program in the conclusion contains the subprograms  $P_1, \dots, P_n$  as components
  - we derive properties of the composite from the properties of its parts (compositionality)
  - pretty much the same as with a type system



# “Structural” Proof Rules

- Basic logic proof systems operate on assertions, e.g.

$$\frac{A \quad A \Rightarrow B}{B} \quad \frac{A \quad B}{A \wedge B} \quad \frac{A}{A \vee B} \quad \frac{B}{A \vee B}$$

- Hoare logic proof system operates on Hoare triples, e.g.

$$\frac{\{A\} P \{B\} \quad \{B\} Q \{C\}}{\{A\} P; Q \{C\}}$$

# One rule for each PL construct

AXIOM 1: ASSIGNMENT AXIOM

$$\{p[t/x]\} x := t \{p\}.$$

RULE 2: COMPOSITION RULE

$$\frac{\{p\} S_1 \{r\}, \{r\} S_2 \{q\}}{\{p\} S_1; S_2 \{q\}}.$$

RULE 3: **if-then-else** RULE

$$\frac{\{p \wedge e\} S_1 \{q\}, \{p \wedge \neg e\} S_2 \{q\}}{\{p\} \text{ if } e \text{ then } S_1 \text{ else } S_2 \text{ fi } \{q\}}$$

RULE 4: **while** RULE

$$\frac{\{p \wedge e\} S \{p\}}{\{p\} \text{ while } e \text{ do } S \text{ od } \{p \wedge \neg e\}}$$



- A cool idea:
  - Programmers can use the rules informally to mentally check their code
  - Tools exist that automate most of the process
  - Lets go through each rule, one by one...

# Simple Programming Language

RECAP

$E ::=$  Expressions

$num$

Integer

$x$

Variable

$E + E \mid \dots$

Integer operators

$E < E \mid \dots$

Relational operators

$E \text{ and } E \dots$

Boolean operators

$P ::=$

Programs

$\text{skip}$

No op

$x := E$

Assignment

$P; P$

Sequential Composition

$\text{if } E \text{ then } P \text{ else } P$

Conditional

$\text{while } E \text{ do } P$

Iteration

# Rule for Skip

---

$\{A\} \text{ skip } \{A\}$

# Rule for Skip

---

$\{A\}$  skip  $\{A\}$

```
if x < 0
  { x := -x; }
```

```
if x < 0
  { x := -x; }
else
  skip
```

## Rule for Sequence

- A sequence defines a dependency on the effects of both program statements.

$$\frac{\{A\} P \{B\} \quad \{B\} Q \{C\}}{\{A\} P; Q \{C\}}$$



# Rule for Conditional

---

$$\frac{\{A \wedge E\} P \{B\} \quad \{A \wedge \neg E\} Q \{B\}}{\{A\} \text{ if } E \text{ then } P \text{ else } Q \{B\}}$$

# Rule for Deduction

$$\frac{A' \implies A \quad \{A\} P \{B\} \quad B \implies B'}{\{A'\} P \{B'\}}$$

- $A \implies B$  means “A logically implies B”
- We prove  $A \implies B$  using the principles of first order logic, plus basic properties of the domain data types, e.g. properties of integers, arrays, etc.

# Rule for Assignment

---

$$\{A[E/x]\} x := E \{A\}$$

- $A[E/x]$  means:
  - Result of replacing all free occurrences of variable  $x$  in assertion  $A$  by the expression  $E$
- For this rule to be sound, we require  $E$  to be an expression without side effects (a pure expression)

# Rule for Assignment

---

$$\{A[E/x]\} x := E \{A\}$$

- We can think of  $A$  as a condition where “ $x$ ” appears in some places.  $A$  is a condition dependent on “ $x$ ”.
- The assignment  $x := E$  changes the value of  $x$  to  $E$ , but leaves everything else unchanged
- So everything that could be said of  $E$  in the precondition, can be said of  $x$  in the postcondition, since the value of  $x$  after the assignment is  $E$
- Example:  $\{x + 1 > 0\} x := x + 1 \{x > 0\}$

# Rule for Assignment

---

$$\{A[E/x]\} x := E \{A\}$$

- Example, let's check  $\{x > -1\} x := x + 1 \{x > 0\}$

$$\{(x+1 > 0)\} x := x+1 \{x > 0\}$$

by the  $:=$  Rule

that is,  $\{(x > 0)[x+1/x]\} x := (x+1) \{x > 0\}$

$$\{x > -1\} x := x + 1 \{x > 0\}$$

by deduction

# Rule for Assignment

---

$$\{A[E/x]\} x := E \{A\}$$

- Trick: if  $x$  does not appear in  $E$  or  $A$ .

We can always write  $\{A \ \&\& \ E == E\} x := E \{x == E\}$

So, if  $x$  does not occur in  $E$ ,  $A$  the triple

$$\{A\} x := E \{A \ \&\& \ x == E\}$$

is always valid



# Rule for Assignment

---

$$\{A[E/x]\} x := E \{A\}$$

- Exercises. Derive:
  - $\{y > 0\} x := y \{x > 0 \ \&\& \ y == x\}$
  - $\{x == y\} x := 2 * x \{y == x \text{ div } 2\}$
  - $\{P(y) \ \&\& \ Q(z)\}$  (here P and Q are any properties)  
 $x := y ; y := z ; z := x$   
 $\{P(z) \ \&\& \ Q(y)\}$

# Example

---

- Consider the program

$P \triangleq \text{if } (x > y) \text{ then } z := x \text{ else } z := y$

- We can (mechanically) check the triple

$\{ \text{true} \} P \{ z == \max(x, y) \}$

# Example

---

- Consider the program

$P \triangleq \text{if } (x > y) \text{ then } z := x \text{ else } z := y$

- We can (mechanically) check the triple

$\{ \text{true} \} P \{ z == \max(x, y) \}$

$\{ x == \max(x, y) \} z := x \{ z == \max(x, y) \}$

$\{ x > y \} z := x \{ z == \max(x, y) \}$

$\{ y == \max(x, y) \} z := y \{ z == \max(x, y) \}$

$\{ y \geq x \} z := y \{ z == \max(x, y) \}$

# Procedures and method calls

$E ::= \text{Expressions}$   
|  
...

$S ::= \text{Statements}$   
|  
...  
|  
 $x := m(E_1, \dots, E_n)$

Call + Assignment

$D ::= \text{Declarations}$   
|  
method  $m(x_1, \dots, x_n)$  returns  $(r)$   
requires  $Pre(x_1, \dots, x_n)$   
ensures  $Post(x_1, \dots, x_n, r)$   
 $\{S\}$

$P ::= \text{Program}$   
|  
 $\overline{D}$

# Procedures and method calls

- Declarations annotated with pre- and post-conditions.
- Method calls built into a form of assignment.
- A program  $P$  is a set of method declarations.
- Each method decl. is validated, assuming its pre-condition and establishing its post-condition:

$$\frac{\{Pre(x_1, \dots, x_n)\} \ S \ \{Post(x_1, \dots, x_n, r)\}}{\text{method } m(x_1, \dots, x_n) \text{ returns } (r) \\ \text{requires } Pre(x_1, \dots, x_n) \\ \text{ensures } Post(x_1, \dots, x_n, r) \ \{S\}}$$

# Procedures and method calls

- Method calls built into a form of assignment:

method  $m(x_1, \dots, x_n)$  returns  $(r)$   
requires  $Pre(x_1, \dots, x_n) \in P$   
ensures  $Post(x_1, \dots, x_n, r) \{S\}$

$$\frac{A \Rightarrow Pre(E_1, \dots, E_n) \quad Post(E_1, \dots, E_n, r) \Rightarrow B[r/x]}{\{A\} x := m(E_1, \dots, E_n) \{B\}}$$

# Procedures and method calls

method  $m(x_1, \dots, x_n)$  **returns**  $(r)$   
**requires**  $Pre(x_1, \dots, x_n)$   $\in P$   
**ensures**  $Post(x_1, \dots, x_n, r)$   $\{S\}$

$$\frac{A \Rightarrow Pre(E_1, \dots, E_n) \quad Post(E_1, \dots, E_n, r) \Rightarrow B[r/x]}{\{A\} x := m(E_1, \dots, E_n) \{B\}}$$

- Instantiated method pre-condition must follow from A
- Instantiated method post-condition must imply B
- Calls are **opaque!** We only know what's in the post-condition.
- Verification with method calls is **modular**.



# Procedures and method calls

```
method maxImp(x:int,y:int) returns (r:int)
  ensures r >= x && r >= y
{
  if x > y { r := x; } else { r := y; }
  return r;
}
```

```
method Main() {
  var a := -10;
  var b := 23;
  var c := maxImp(a,b);
  assert (c == b);
}
```

⊗ assertion violation Verifier

# Procedures and method calls

---

```
method maxImp(x:int,y:int) returns (r:int)
  ensures r >= x && r >= y
{
  if x > y { r := x; } else { r := y; }
  return r;
}
```

```
method Main() {
  var a := -10;
  var b := 23;
  var c := maxImp(a,b);
  assert (c >= b);
}
```



# Procedures and method calls

---

```
method maxImp(x:int,y:int) returns (r:int)
  ensures (x>y ==> r == x) && (x <= y ==> r == y)
{
  if x > y { r := x; } else { r := y; }
  return r;
}
```

```
method Main() {
  var a := -10;
  var b := 23;
  var c := maxImp(a,b);
  assert (c == b);
}
```



# Next Week:

---

- Hoare Logic (continuation)
- Loops and Loop invariants
- Verification of ADTs