Construction and Verification of Software

2019 - 2020

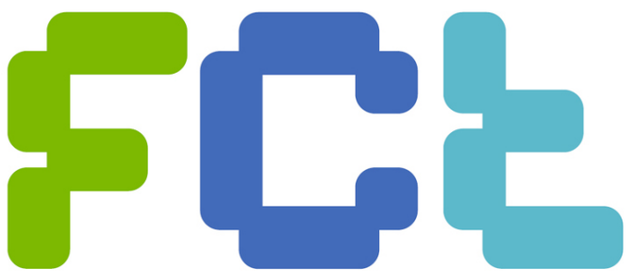
MIEI - Integrated Master in Computer Science and Informatics

Consolidation block

Lecture 2 - Specification and Verification

João Costa Seco (joao.seco@fct.unl.pt)

based on previous editions by **Luís Caires** (lcaires@fct.unl.pt)



FACULDADE DE CIÊNCIAS E TECNOLOGIA

UNIVERSIDADE NOVA DE LISBOA

Disclaimer

- This lecture is being transmitted remotely without special preparation to be e-learning material.
- This transmission is being in experimental mode using ZOOM and an untested infrastructure.
- Usage permission extends only the context of the CVS course at MIEI / FCT NOVA.
- Associated discussion forums are available:
  - Piazza: https://piazza.com/fct.unl.pt/spring2020/11159/home
  - Git: https://bitbucket.org/costaseco/cvs20

Part I

Software Correctness

Relevance of Software Correctness

RECAP

- Quality procedures must be enforced at all levels, in particular at the construction phase, where most of the issues are introduced and difficult to circumvent.
- **Questions for you now**:
  - What methods do you currently use to make sure your code is “bullet-proof” ?
  - How can you prove to yourself (and others) that your code is “bullet-proof” ?
  - What arguments do you use to convince yourself and others that your code works as expected and not goes wrong, with respect to functional correctness, security, or concurrency errors?
- You will **know better answers** at the end of this course.

Software Correctness: What and How

RECAP

- **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 is against a specification

RECAP

- Then what does “correct software” mean?
  - Always relative to some given (**our**) specs
- Correct means that software meets **our** specs
  - There is no such thing as the “right specification”
  - In practice, the spec is usually incomplete ...
  - But **the spec must not be wrong** !
  - It should be very easy to check what the spec states
  - The spec **must be simple, much simpler than code**
  - The spec should be **focused** (pick relevant cases)
    - e.g., buffers are not being overrun
    - e.g., never transfer money without logging the source

Checking Specs: Dynamic Verificatio

RECAP

- 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: Dynamic Verificatio

RECAP

- Some shortcomings of dynamic verification
  - always introduces a level of performance overhead
  - may show the existence of some errors, but does not ensure absence of errors (the code passed a test suite today, but may fail with some other clever test)
  - **Challenge**: how do you make sure that you are defining the “right” tests and “enough” tests
  - Will talk about testing methods later on in the course

Checking Specs: Static Verification

- "static verification" means verification at **compile time**
- relies on algorithmic reasoning about what programs do, by analysing 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 tackle 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)
- success stories:
  - **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
  - more flexible
  - execution overhead
  - based on ad-hoc testing
- Static Verification
  - sound
  - compile time
  - conservative
    (may have false positives)
  - erasure of verification
  - each analysis targets a kind 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)

- The post condition assertion can be assumed after the program's execution, provided that the pre-conditions were met at the beginning. That's the only assumptions that can be drawn from the post-condition.

Part II
Design by Contract
&
Defensive Programming

Axiomatic Based Verification

- Logic based verification
  - Axiomatic approaches based on Hoare Logic (Pre- and post conditions)
  - If all components are verified, all contracts are fulfilled in all cases.
  - If a component does not fulfil a contract then no guaranties are given about the systems behaviour.

Design by Contract

- Design by contract
  - Eiffel language (Bertrand Meyer)
  - Formal specification of pre-, post-conditions and invariants
  - Assume that all preconditions are met when invoking an operation and that all postconditions will be satisfied after the operation is executed.

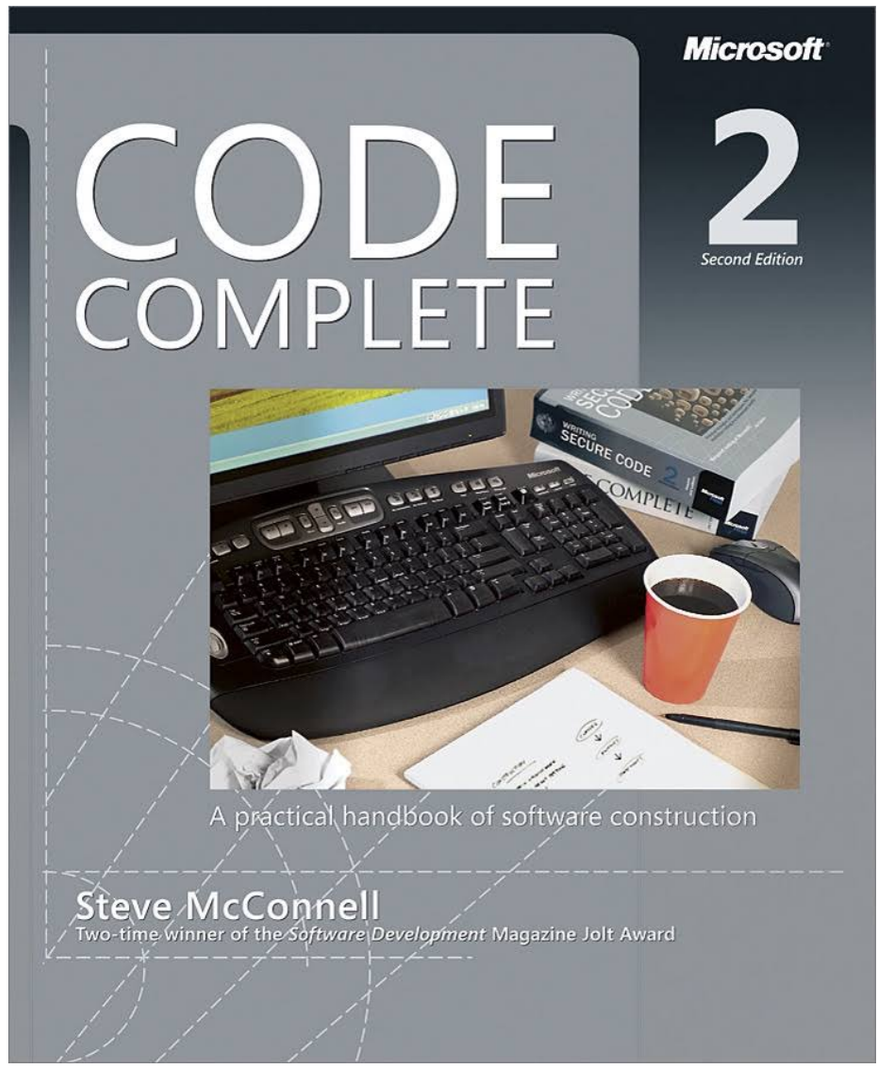
Defensive Programming

- Defensive programming
  - Prepare for all possible inputs and associated responses

Chapter 8

Defensive Programming

Contents



Part III

Specification and Verification

What may specs look like?

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

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

- 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-condition-assertion      % PRE
ensures post-condition-assertion      % POST
modifies global-state-changed         % MOD
{
   ... method code
}
```

- The method call P(...), 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 {
invariant invariant-assertion REPINV
invariant abstraction-map-assertions ABSMAP
{
   ... methods...
}
```

- ADT C 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 is used to prove properties of programs
- What kinds of properties are we interested in?
  - safety properties (partial correctness)
  - state that if the program terminates (delivers an outcome), then the final state satisfies some property
  - liveness properties (total correctness)
  - say that 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: verification at the level of the programming languages (not of programs, cf. Floyd)

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

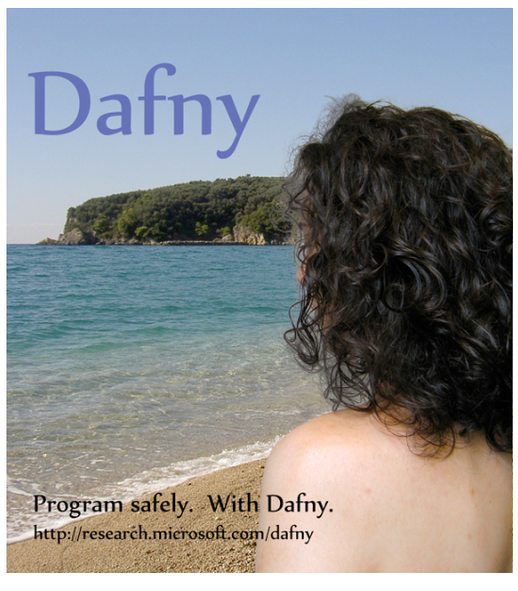
Dafny: An Automatic Program Verifier for Functional Correctness

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

Why GitHub? Enterprise Explore Marketplace Pricing Search / Sign in Sign up

dafny-lang / dafny

Watch 70 Star 1.1k Fork 109

Code Issues 123 Pull requests 7 Actions Projects 0 Wiki Security Insights

Dafny is a verification-aware programming language

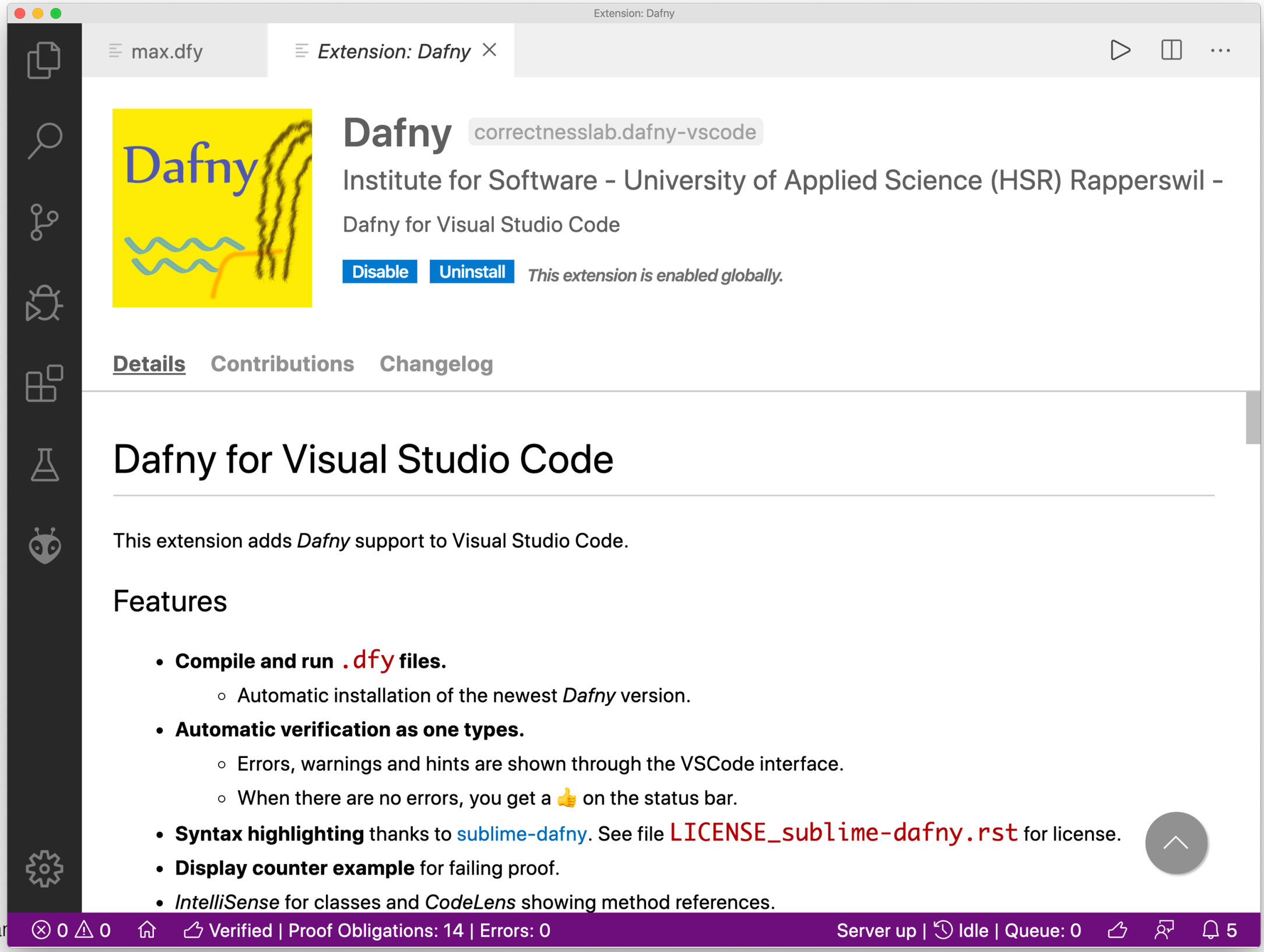
3,799 commits 21 branches 0 packages 8 releases 43 contributors View license

Branch: master New pull request Find file Clone or download

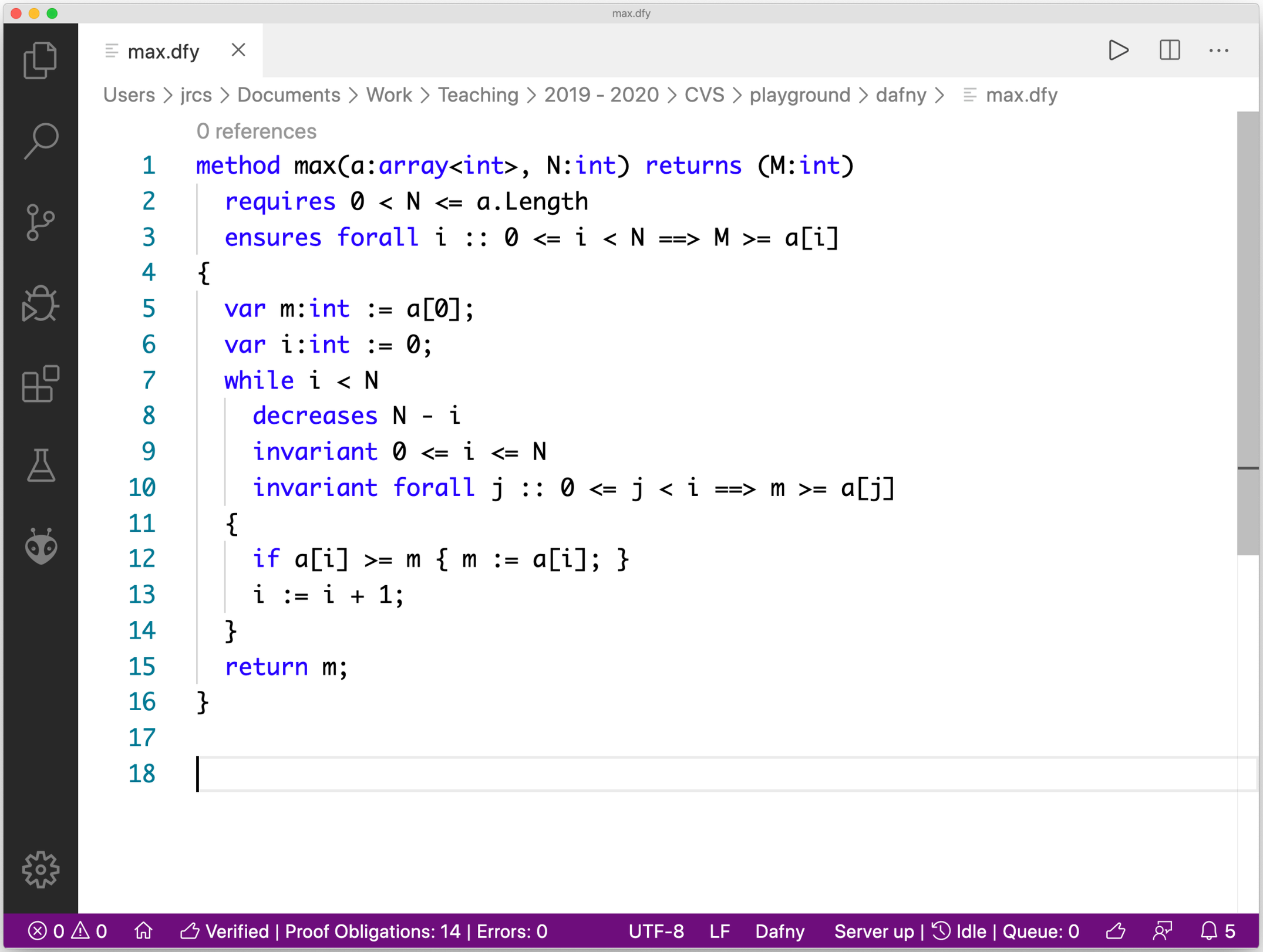
robin-aws fix: Don't call PropagateFailure() in assign-or-halt (#553) ... Latest commit 4ecab36 6 days ago

Binaries	feat: Add `expect` and `:- expect` (a.k.a. assign-or-halt) statements (...	6 days ago
Docs	Improve online tutorial	4 months ago
Source	fix: Don't call PropagateFailure() in assign-or-halt (#553)	6 days ago
Test	fix: Don't call PropagateFailure() in assign-or-halt (#553)	6 days ago
Util	Use pre-commit to trim trailing spaces (#394)	4 months ago
third_party/Coco	Updated the Coco build and .exe to use VS 2017 and a .NET version 4.5	2 years ago
.gitattributes	Preserve line endings for dafny and dafny-server scripts	2 years ago

Dafny@VSC


Extension: Dafny
max.dfy
Extension: Dafny
Dafny
correctnesslab.dafny-vscode
Institute for Software - University of Applied Science (HSR) Rapperswil -
Dafny for Visual Studio Code
Disable
Uninstall
This extension is enabled globally.
Details
Contributions
Changelog
Dafny for Visual Studio Code
This extension adds Dafny support to Visual Studio Code.
Features
Compile and run .dfy files.
Automatic installation of the newest Dafny version.
Automatic verification as one types.
Errors, warnings and hints are shown through the VSCode interface.
When there are no errors, you get a 👍 on the status bar.
Syntax highlighting thanks to sublime-dafny. See file LICENSE_sublime-dafny.rst for license.
Display counter example for failing proof.
IntelliSense for classes and CodeLens showing method references.
0 0
Verified | Proof Obligations: 14 | Errors: 0
Server up | Idle | Queue: 0
5

Dafny@VSC


max.dfy
max.dfy
Users > jrcs > Documents > Work > Teaching > 2019 - 2020 > CVS > playground > dafny > max.dfy
0 references
1 method max(a:array<int>, N:int) returns (M:int)
2 requires 0 < N <= a.Length
3 ensures forall i :: 0 <= i < N ==> M >= a[i]
4 {
5 var m:int := a[0];
6 var i:int := 0;
7 while i < N
8 decreases N - i
9 invariant 0 <= i <= N
10 invariant forall j :: 0 <= j < i ==> m >= a[j]
11 {
12 if a[i] >= m { m := a[i]; }
13 i := i + 1;
14 }
15 return m;
16 }
17
18
0 0 Verified | Proof Obligations: 14 | Errors: 0 UTF-8 LF Dafny Server up | Idle | Queue: 0 5

rise4fun @ Microsoft Research (MSR)

rise4fun

rise4fun.com

Reader

http://www....apapnce.pdf STM ERC metrics https://dsp.../cpd-10.pdf POPL 2013 RID TT-IST MIT-TLO TTxfer WoS Coq CLIP Heidegger Lectures DI-DEV SAIA Conselho Científico

rise4fun

your tool here

708285 programs analyzed

rise4fun

a community of software engineering tools

all tutorial automata concurrency design infrastructure languages security synthesis testing verification

new!

dafny
A language and program verifier for functional correctness

f*
A verification tool for higher-order stateful programs

visual c++
The Visual C++ compiler

z3rcf
Python interface for the Z3 Real Closed Fields package (and Theorem Prover)

microsoft

agl
Automatic Graph Layout

bek
A domain specific language for writing and analyzing common string functions.

boogie
Intermediate Verification Language

code contracts
Language agnostic modular program verification and repair with abstract interpretation.

counterdog
Theorem-prover for Counterfactual Datalog

dkal
Distributed Knowledge Authorization Language

esm
Empirical Software Engineering and Measurement Group

formula
Formal Modeling and Analysis

cry f#
Programming language combining functional, object-oriented and scripting programming.

f*
A verification tool for higher-order stateful programs

heapdbg
Runtime heap abstraction

koka
A function-oriented language with effect inference

pex
Automatic test generation using Dynamic Symbolic Execution for .NET

poirot
Poirot

https://rise4fun.com/dafny

A glimpse of Dafny programming

dafny Microsoft Research

Is this program correct?

```
1  class PSet
2  {
3    var s: set<int>;
4    var n: int;
5
6    function SetInv(): bool
7    reads this;
8    {
9      (forall x::x in s ==> x >= 0) && |s| == n
10   }
11
12 method initBag()
13 ensures SetInv();
14 modifies this;
15 {
16   s := {};
17   n := 0;
18 }
19
20 method add(x:int)
21   requires SetInv() && x >= 0;
22   modifies this;
```

tutorial

home video permalink

'►' shortcut: Alt+B

A glimpse of Dafny programming

Getting Started with Dafny: A Guide

Be sure to follow along with the code examples by clicking the "load in editor" link in the corner. See what the tool says, try to fix programs on your own, and experiment!

Introduction

Dafny is a language that is designed to make it easy to write correct code. This means correct in the sense of not having any runtime errors, but also correct in actually doing what the programmer intended it to do. To accomplish this, Dafny relies on high-level annotations to reason about and prove correctness of code. The effect of a piece of code can be given abstractly, using a natural, high-level expression of the desired behavior, which is easier and less error prone to write. Dafny then generates a proof that the code matches the annotations (assuming they are correct, of course!). Dafny lifts the burden of writing bug-free *code* into that of writing bug-free *annotations*. This is often easier than writing the code, because annotations are shorter and more direct. For example, the following fragment of annotation in Dafny says that every element of the array is strictly positive:

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:

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

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

The proof of the second of these is:

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

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

A9 $= (r - y) + (y + y \times q)$

A3 $= ((r - y) + y) + y \times q$

A6 $= r + y \times$

The axioms A1 to A9 are, of cou
tional infinite set of integers in ma
they are also true of the finite sets of
manipulated by computers provide
fined to *nonnegative* numbers. Their
of the size of the set; furthermore, it i
of the choice of technique applied i
flow"; for example:

(1) Strict interpretation: the resu
operation does not exist; when overfl
ing program never completes its op
this case, the equalities of A1 to A9 a
that both sides exist or fail to exist

Simple Programming Language

E	::=	Expressions	
		num	Integer
	|	x	Variable
	|	$E + E \mid ...$	Integer operators
	|	$E < E \mid ...$	Relational operators
	|	E `and` $E ...$	Boolean operators

P	::=		Programs
		`skip`	No op
	|	$x := E$	Assignment
	|	$P; P$	Sequential Composition
	|	`if` E `then` P `else` P	Conditional
	|	`while` E `do` P	Iteration

States and State Transformers

- A program is a state transformer,
  it transforms an initial state into a target state
- What is a program state? a state is 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 σ it yields state σ' where

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

- We may say that P transforms σ in σ'
- P is only defined on states σ where vars(P) ⊆ dom(σ)

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
- Thus, 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, not of the programming language
- But in some cases, assertions may be expressed in the programming language (Java / Dafny).

Part IV

A bit of History

Some bits of history … (extra)

Kick off:

– **“Checking a large routine”**

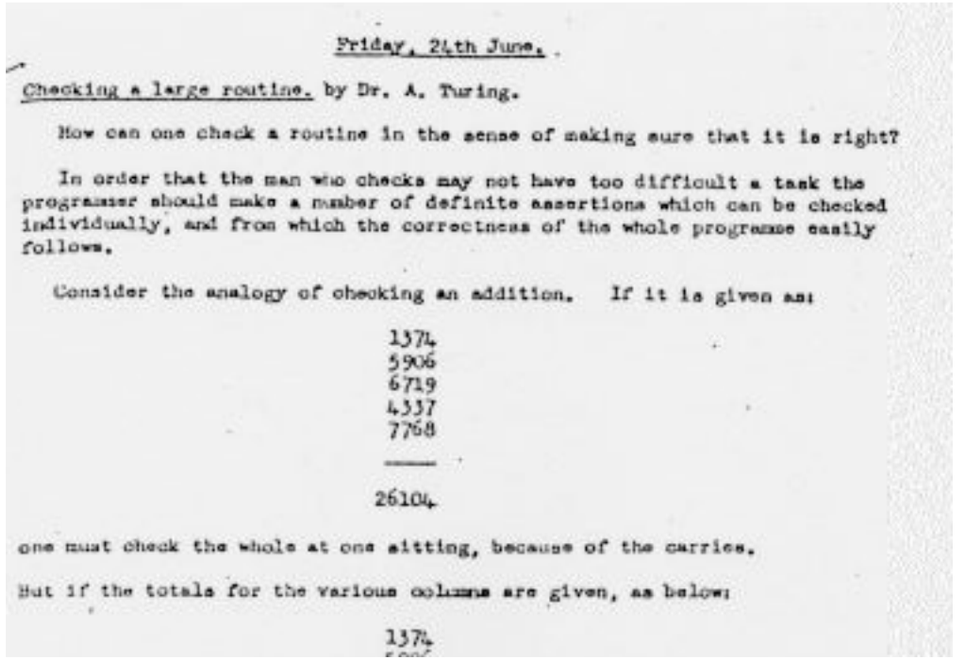
Turing

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
4337
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

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

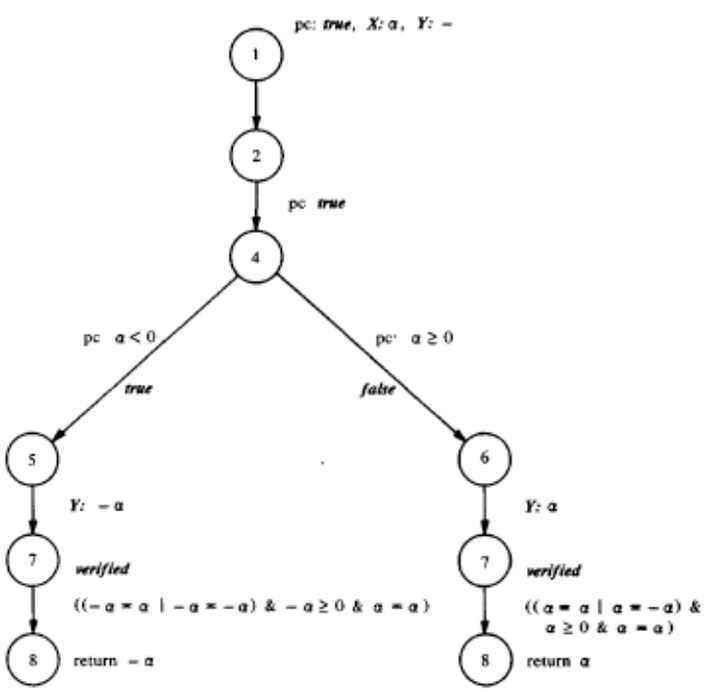


FIGURE 3. Symbolic execution tree for procedure *ABSOLUTE*.

Assertions

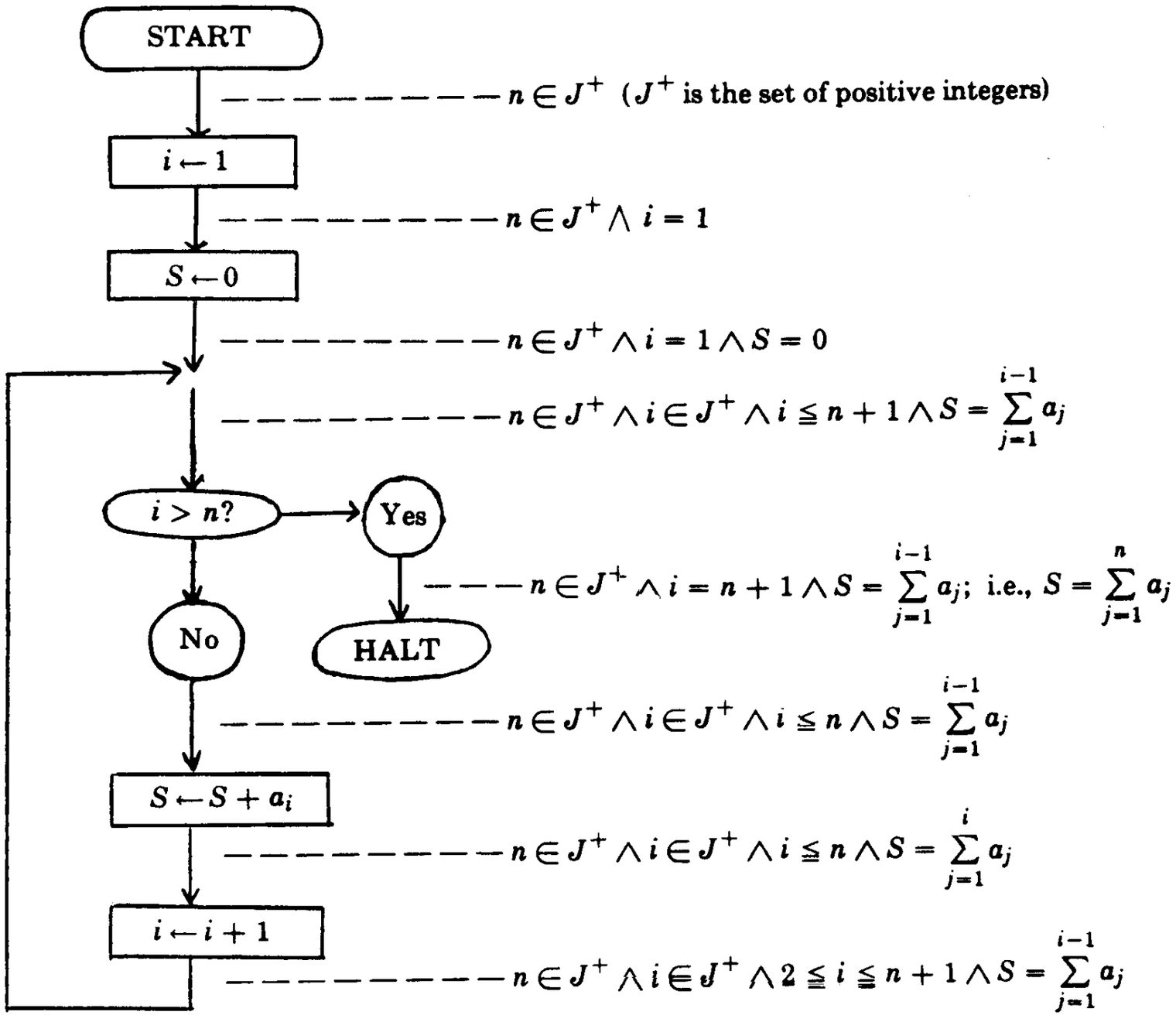


FIGURE 1. Flowchart of program to compute $S = \sum_{j=1}^{n} a_j$ ($n \geq 0$)

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

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

RULE 4: **while** RULE

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

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 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.")

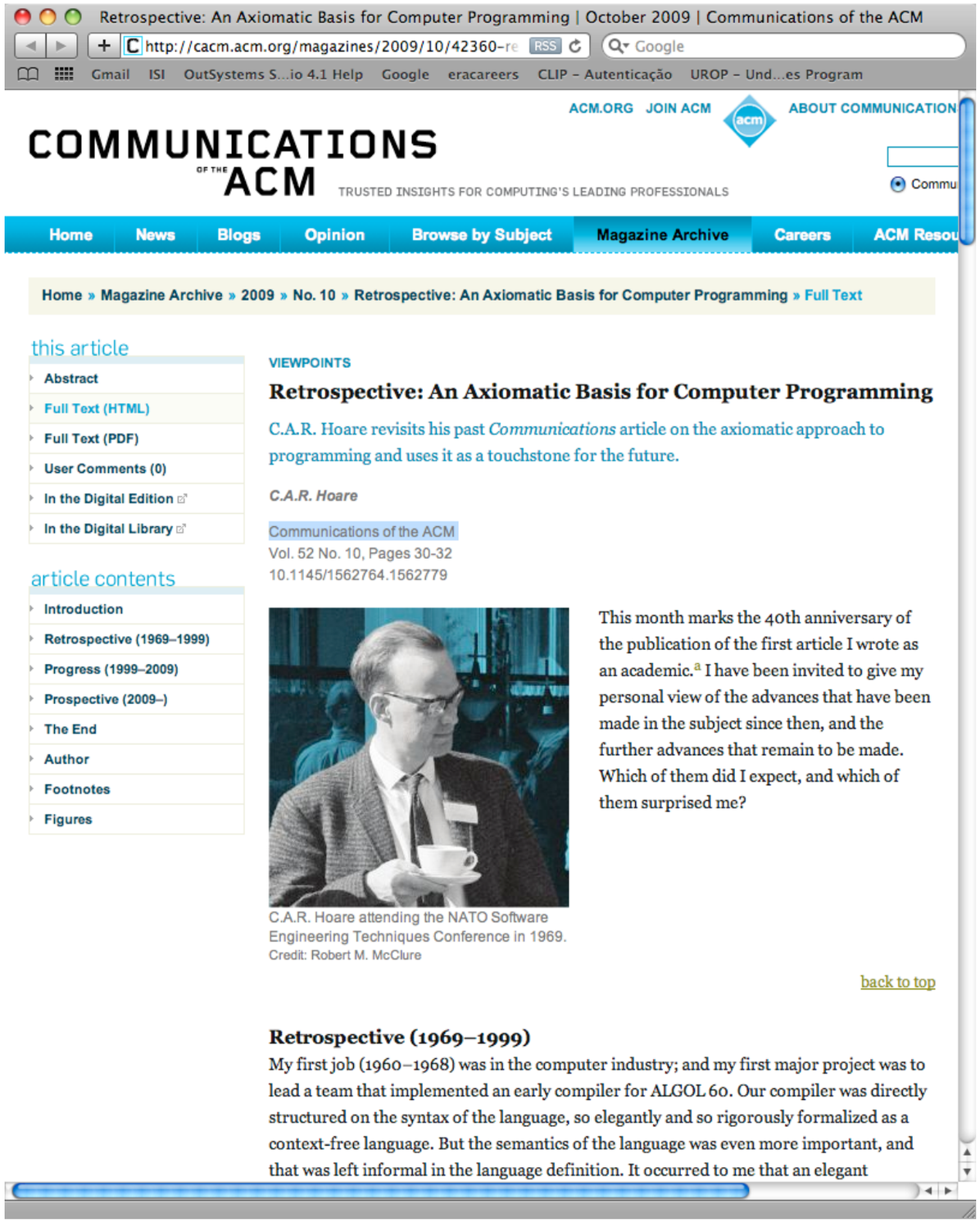
Hoare Logic Today

Still hot ...

– 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



Extended Static Checking

JML and Extended Static Checking for Java

ESC/Java2 is a programming tool that uses static analysis to verify the correctness of Java programs, using an extension of Hoare Logic called JML.

G.T. Leavens, 2000

Extended Static Checking for Java

Cormac Flanagan K. Rustan M. Leino* Mark Lillibridge
Greg Nelson James B. Saxe Raymie Stata

Compaq Systems Research Center 130 Lytton Ave. Palo Alto, CA 94301, USA

ABSTRACT

Software development and maintenance are costly endeavors. The cost can be reduced if more software defects are detected earlier in the development cycle. This paper introduces the Extended Static Checker for Java (ESC/Java), an experimental compile-time program checker that finds common programming errors. The checker is powered by verification-condition generation and automatic theorem-proving techniques. It provides programmers with a simple annotation language with which programmer design decisions can be expressed formally. ESC/Java examines the annotated software and warns of inconsistencies between the design decisions recorded in the annotations and the actual code, and also warns of potential runtime errors in the code. This paper gives an overview of the checker architecture and annotation language and describes our experience applying the checker to tens of thousands of lines of Java programs.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.4 [**Software Engineering**]: Program Verification

General Terms

Design, Documentation, Verification

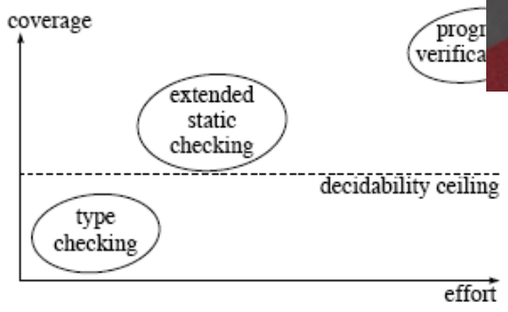


Figure 1: Static checkers plotted along the two di-

static check
formed with
cause ESC
tional static
automatic t
programs, w
many errors
ming langua
cast errors,
rors in conc

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

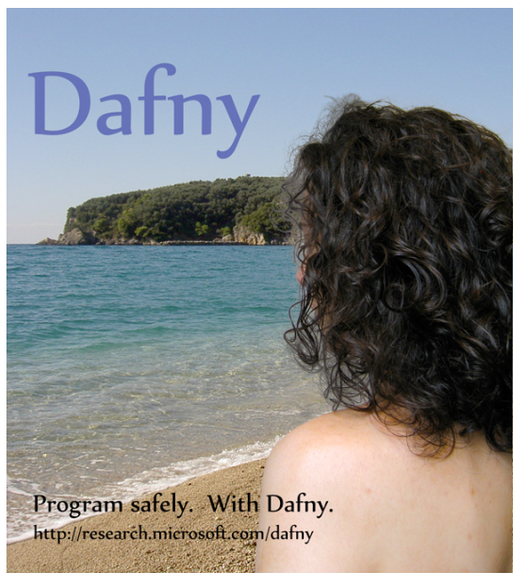
Dafny: An Automatic Program Verifier for Functional Correctness

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

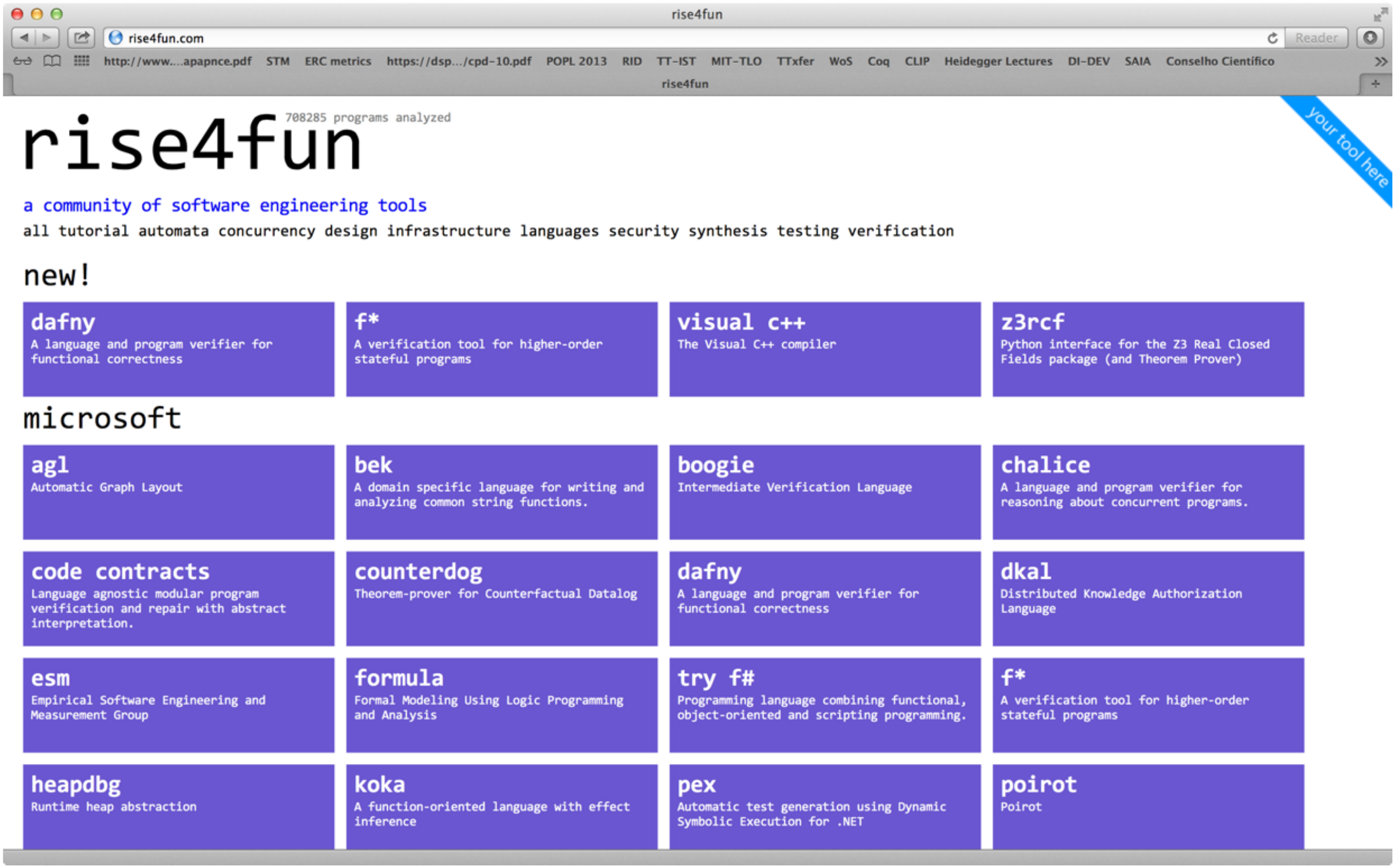
Abstract

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.

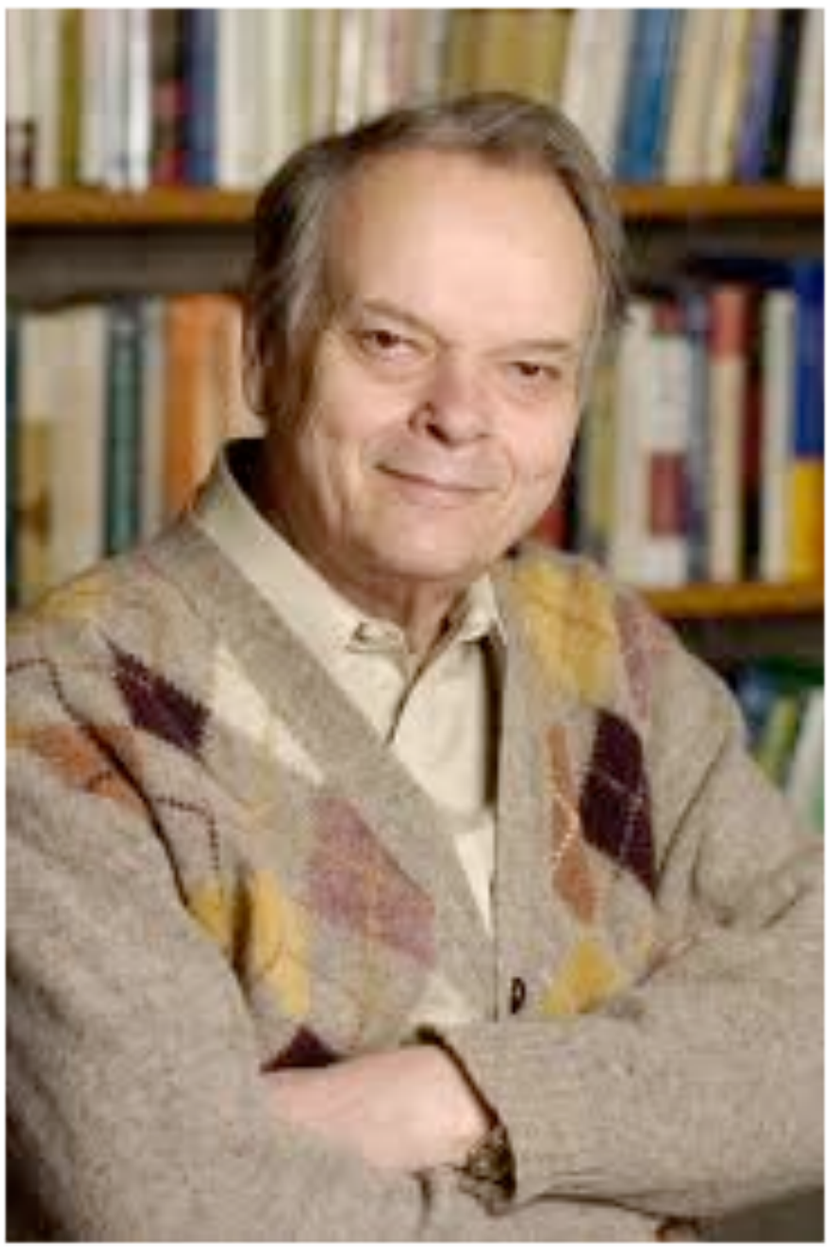
This paper gives a tour of the language and verifier Dafny, which has been used to verify the functional correctness of a number of challenging pointer-based programs. The paper describes the features incorporated in Dafny, illustrating their use by small examples and giving a taste of how they are coded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



rise4fun @ MSR



Separation Logic



John C. Reynolds

$$\frac{s, h \models P * (P \mathrel{-\!\!*} Q)}{s, h \models Q} \qquad \frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}\ \mathsf{mod}(C) \cap \mathsf{fv}(R) = \emptyset$$

Peter O'Hearn

Infer@Facebook

Infer

Docs Support Blog Twitter Facebook GitHub

A tool to detect bugs in Java and C/C++/Objective-C code before it ships

Infer is a static analysis tool - if you give Infer some Java or C/C++/Objective-C code it produces a list of potential bugs. Anyone can use Infer to intercept critical bugs before they have shipped to users, and help prevent crashes or poor performance.

GET STARTED | TRY INFER IN YOUR BROWSER

Star

Android and Java

Infer checks for null pointer exceptions, resource leaks, annotation reachability, missing lock guards, and concurrency race conditions in Android and Java code.

C, C++, and iOS/Objective-C

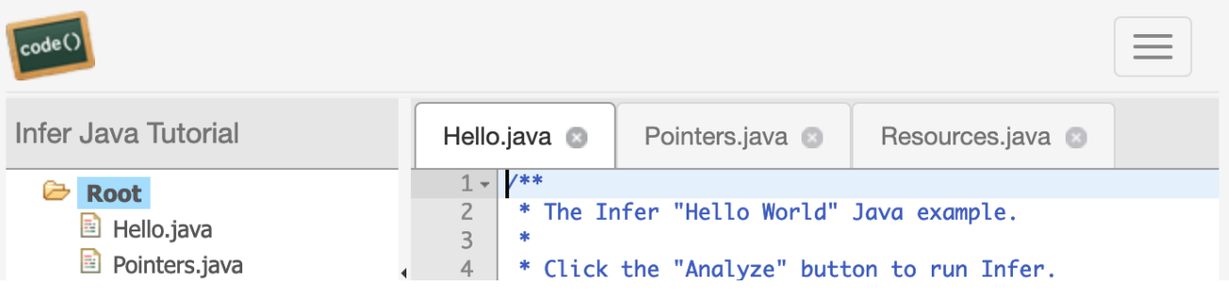
Infer checks for null pointer dereferences, memory leaks, coding conventions and unavailable API's.

Infer in Action

```
class Infer {

    String mayReturnNull(int i) {
        if (i > 0) {
            return "Hello, Infer!";
        }
        return null;
    }
```

Try Infer



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.
- 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) }
```

Using dafny

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

method maxMethodZ(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);
}
```

Example: Rule for Sequence

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

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

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

$$\frac{\{A_1\}\ P_1\ \{B_1\}\ \ldots\ \{A_n\}\ P_n\ \{B_n\}}{\{A\}\ C(P1, \ldots, P_n)\ \{B\}}$$

- What is nice here:
  - the program in the conclusion contains the subprograms $P_1, \ldots, P_n$ 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 propositions, e.g.

$$\frac{A \quad A \implies B}{B} \qquad \frac{A \quad B}{A \wedge B} \qquad \frac{A}{A \vee B} \qquad \frac{B}{A \vee B}$$

- 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

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

RULE 4: while RULE

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

- A really cool idea:
  - every programmer can use the Hoare rules informally to mentally check her code while coding
  - tools exist that automate most of the process
  - we now go through each rule, one by one

Simple Programming Language

RECAP

E	::=	Expressions	
		num	Integer
	|	x	Variable
	|	$E + E \mid ...$	Integer operators
	|	$E < E \mid ...$	Relational operators
	|	E `and` $E ...$	Boolean operators

P	::=		Programs
		`skip`	No op
	|	$x := E$	Assignment
	|	$P; P$	Sequential Composition
	|	`if` E `then` P `else` P	Conditional
	|	`while` E `do` P	Iteration

Rule for Skip

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

Rule for Skip

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

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

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

Rule for Sequence

RECAP

- 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

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

Rule for Deduction

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

- A ⇒ B means “A logically implies B”
- We prove A ⇒ 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:
  - the 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 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 ≜ if (x>y) then z := x else z := y

- We can (mechanically) check the triple

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

Example

- Consider the program

  P ≜ if (x>y) then z := x else z := y

- We can (mechanically) check the triple

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

Next Week:

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