Construction and Verification of Software 2014 - 2015

Luís Caires

lcaires@fct.unl.pt

Mestrado Integrado em Engenharia Informática

FCT UNL

this is a course about ...

techniques and methods for software construction

(techniques and tools)

Course Topics (Overview)

- Software Static Verification
  - understand the principles and know how to use assertion methods in practice to specify, reason about, and validate software, including tools
- Software Testing (Dynamic Verification)
  - understand the principles and methods for testing software, including some tools
- Concurrent Programming
  - understand how to write correct concurrent OO programs
- Secure Programming
  - understand how to write secure program code against specified information flow policies

Course Evaluation Rules

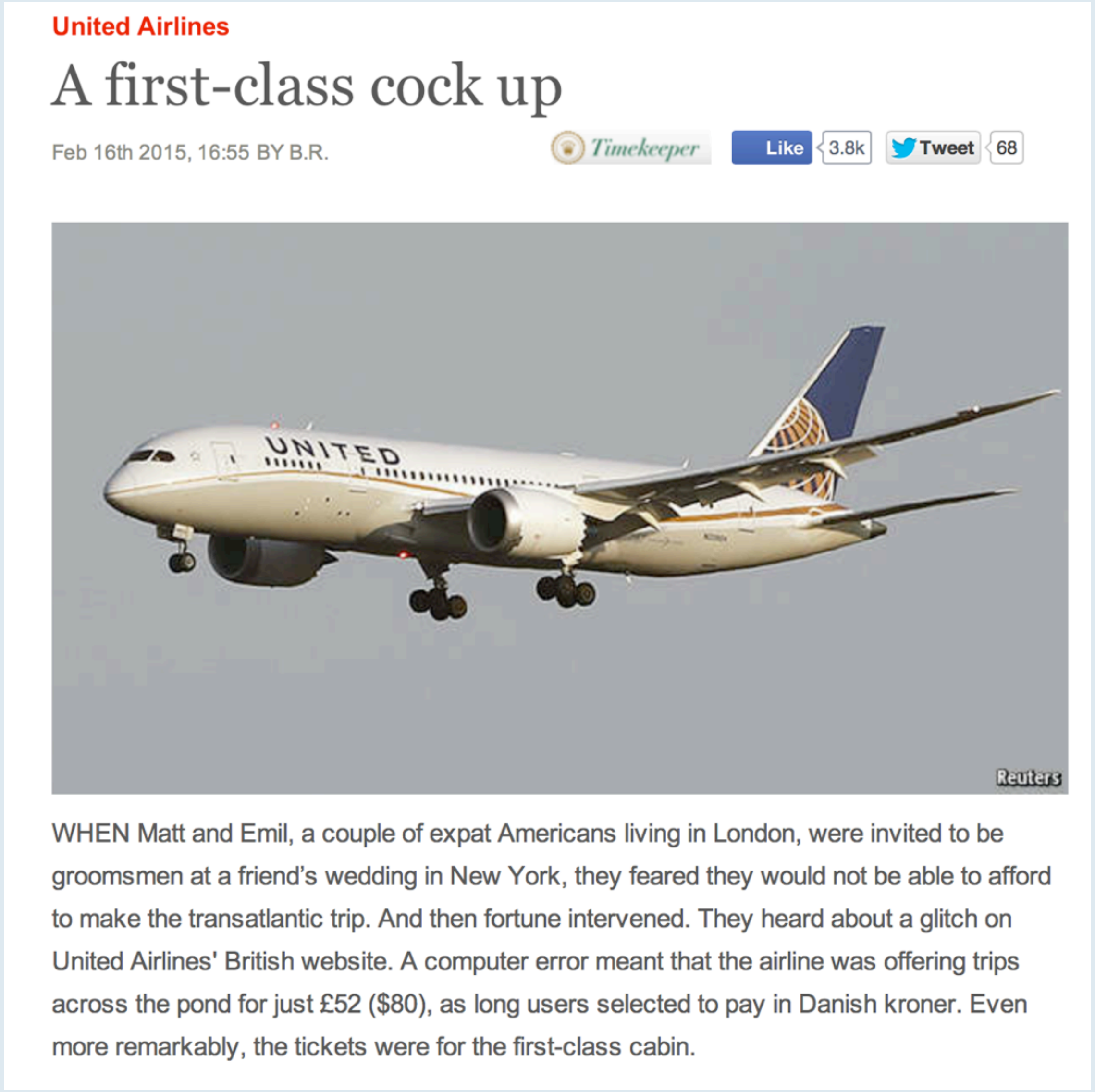
- Graded handouts (4)
  - Groups of 1-2 students
  - You will progressively design and implement a middle sized application and certify its correctness for a set of interesting properties using a combination of the techniques and tools learned in the course
  - Due dates : end of March, April, May, June
- Midterm test: 20.April.2015 (T1)
- Final test 1-2-3.Jun.2013 (P1-P2-P3)

Software Correctness

Relevance of Software Correctness

- The software industry is becoming increasingly competitive: although it seems that “anyone” in the IT field can develop software, to be successful a company must be extremely careful with the quality of its product
- Shipping incorrect, “buggy”, or instable software can kill the credibility of a developer (individual or company)
- Debugging and patching software is extremely expensive, both in terms of resources and reputation.
- Currently, applications may affect zillions of users, and impact of crashes, security flaws, data loss, or incorrect behavior is highly visible (e.g., think of cloud applications such as GMail or Facebook).

Too easy to make flawed software

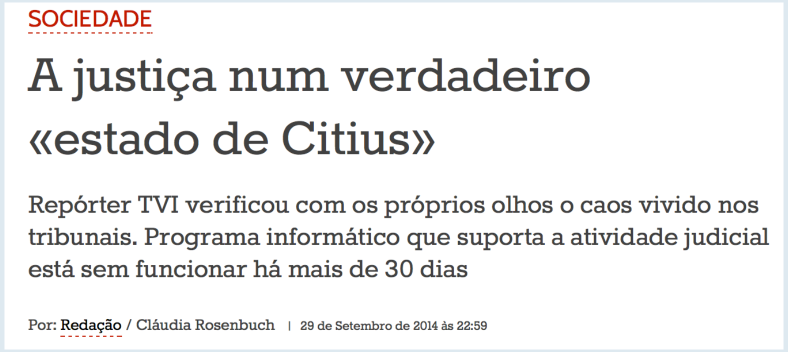


United Airlines

A first-class cock up

Feb 16th 2015, 16:55 BY B.R.

WHEN Matt and Emil, a couple of expat Americans living in London, were invited to be groomsmen at a friend's wedding in New York, they feared they would not be able to afford to make the transatlantic trip. And then fortune intervened. They heard about a glitch on United Airlines' British website. A computer error meant that the airline was offering trips across the pond for just £52 ($80), as long users selected to pay in Danish kroner. Even more remarkably, the tickets were for the first-class cabin.

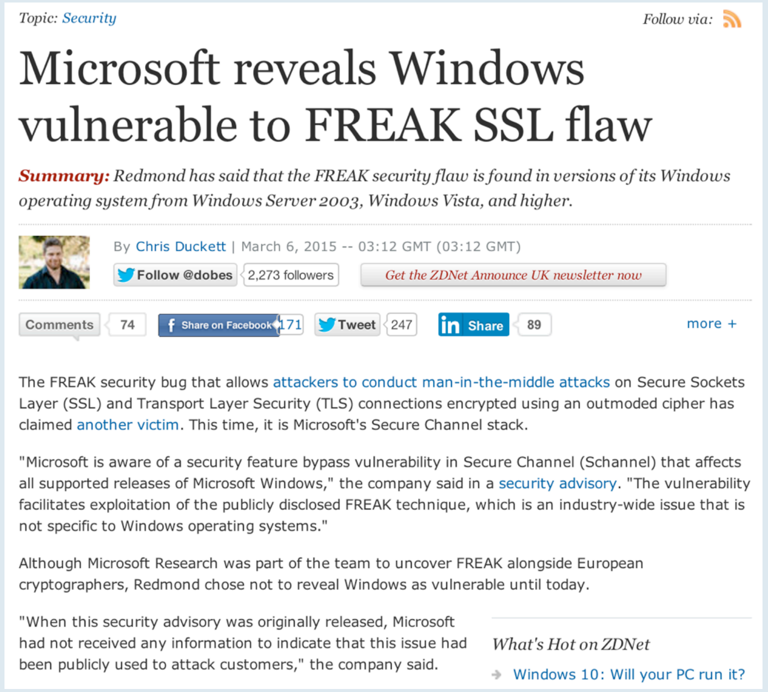


SOCIEDADE

A justiça num verdadeiro «estado de Citius»

Repórter TVI verificou com os próprios olhos o caos vivido nos tribunais. Programa informático que suporta a atividade judicial está sem funcionar há mais de 30 dias

Por: Redação / Cláudia Rosenbuch | 29 de Setembro de 2014 às 22:59



Topic: *Security* *Follow via:*

Microsoft reveals Windows vulnerable to FREAK SSL flaw

Summary: *Redmond has said that the FREAK security flaw is found in versions of its Windows operating system from Windows Server 2003, Windows Vista, and higher.*

By Chris Duckett | March 6, 2015 -- 03:12 GMT (03:12 GMT)

Follow @dobes 2,273 followers
Get the ZDNet Announce UK newsletter now
Comments 74
Share on Facebook 171
Tweet 247
Share 89
more +

The FREAK security bug that allows attackers to conduct man-in-the-middle attacks on Secure Sockets Layer (SSL) and Transport Layer Security (TLS) connections encrypted using an outmoded cipher has claimed another victim. This time, it is Microsoft's Secure Channel stack.

"Microsoft is aware of a security feature bypass vulnerability in Secure Channel (Schannel) that affects all supported releases of Microsoft Windows," the company said in a security advisory. "The vulnerability facilitates exploitation of the publicly disclosed FREAK technique, which is an industry-wide issue that is not specific to Windows operating systems."

Although Microsoft Research was part of the team to uncover FREAK alongside European cryptographers, Redmond chose not to reveal Windows as vulnerable until today.

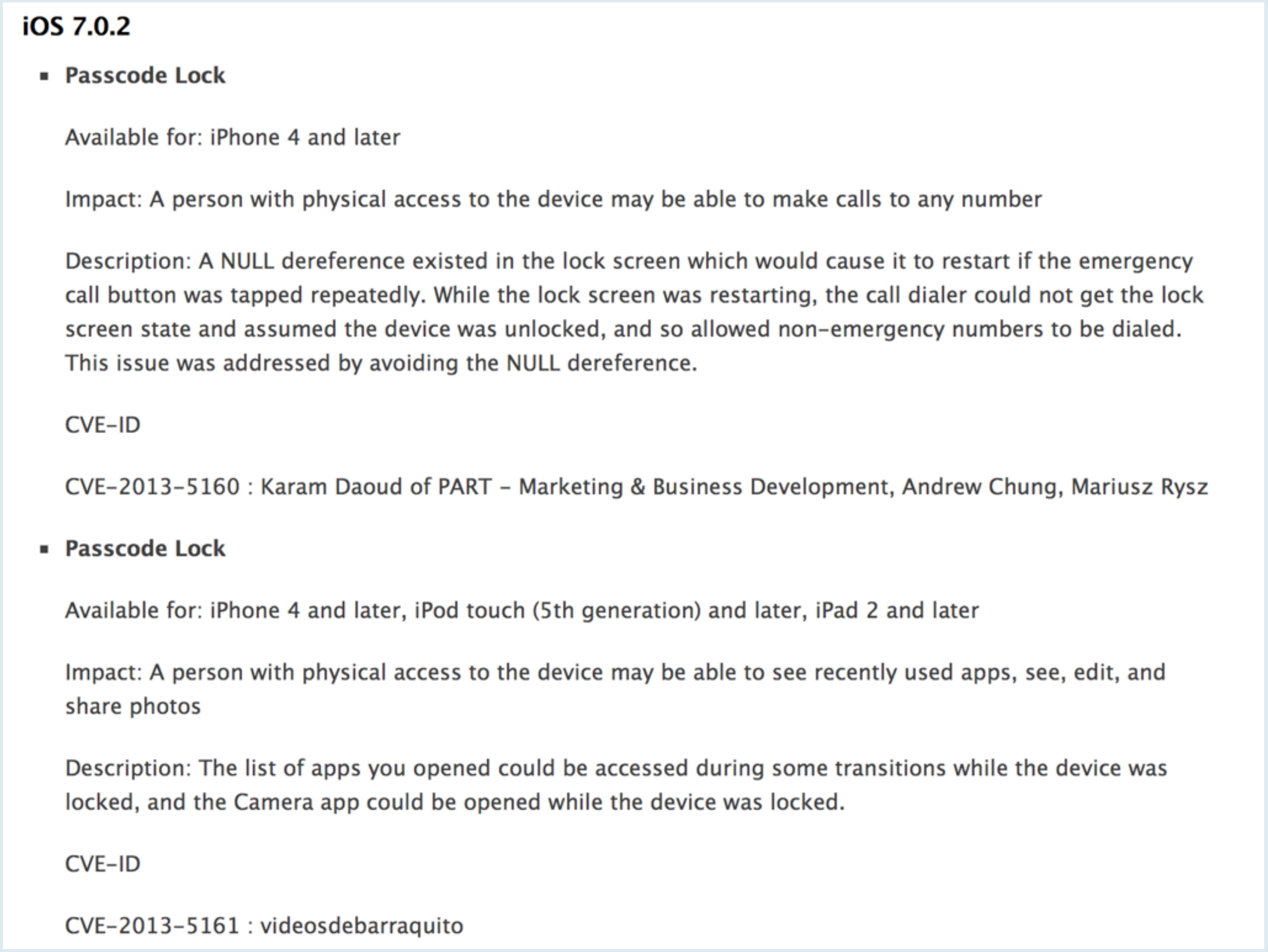
"When this security advisory was originally released, Microsoft had not received any information to indicate that this issue had been publicly used to attack customers," the company said.

What's Hot on ZDNet

→ Windows 10: Will your PC run it?

Bug Report from Apple (2013)

http://news.cnet.com/8301-1009_3-57603787-83/apple-promises-to-fix-ios-7-lock-screen-hack/



iOS 7.0.2

- **Passcode Lock**

  Available for: iPhone 4 and later

  Impact: A person with physical access to the device may be able to make calls to any number

  Description: A NULL dereference existed in the lock screen which would cause it to restart if the emergency call button was tapped repeatedly. While the lock screen was restarting, the call dialer could not get the lock screen state and assumed the device was unlocked, and so allowed non-emergency numbers to be dialed. This issue was addressed by avoiding the NULL dereference.

  CVE-ID

  CVE-2013-5160 : Karam Daoud of PART - Marketing & Business Development, Andrew Chung, Mariusz Rysz

- **Passcode Lock**

  Available for: iPhone 4 and later, iPod touch (5th generation) and later, iPad 2 and later

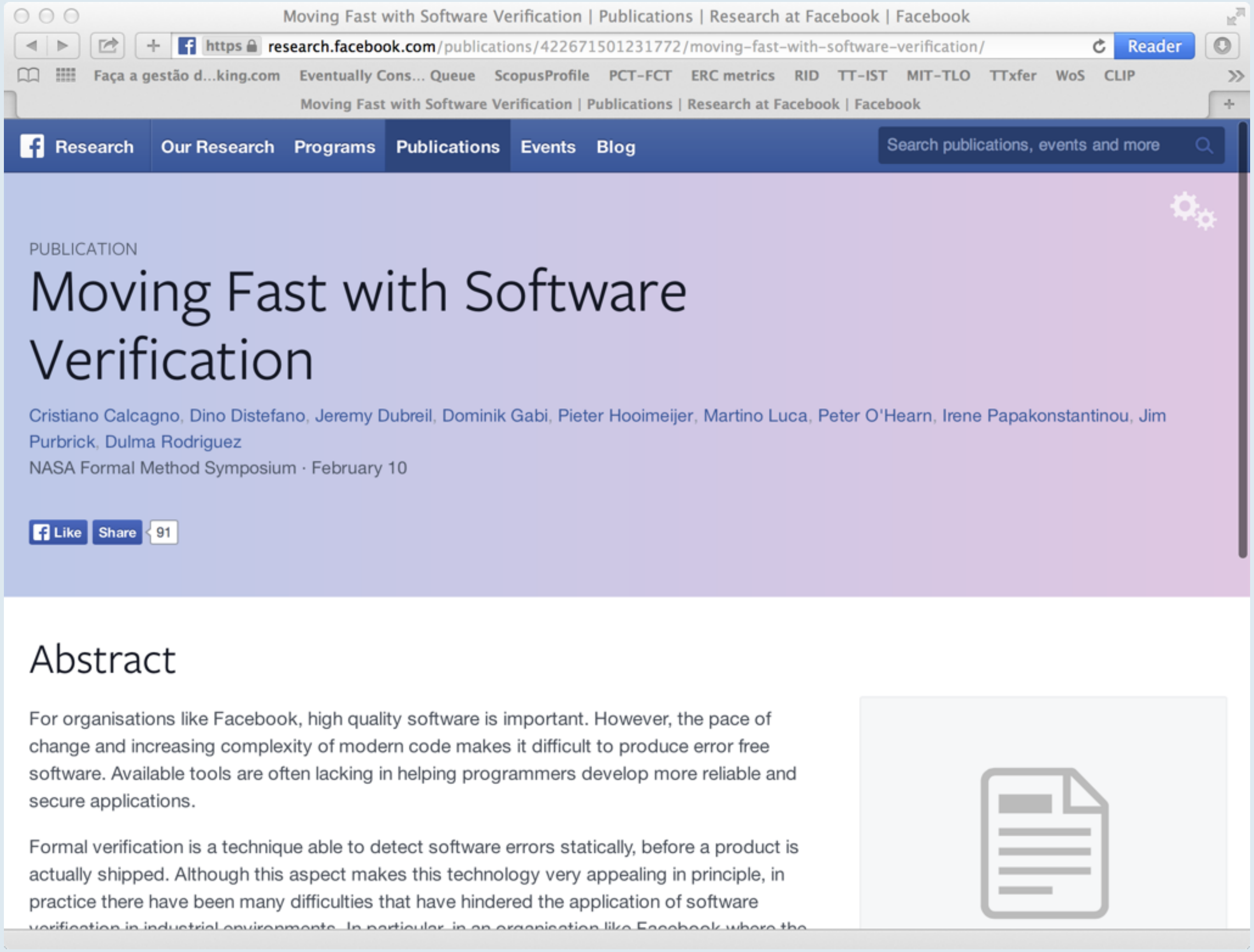
  Impact: A person with physical access to the device may be able to see recently used apps, see, edit, and share photos

  Description: The list of apps you opened could be accessed during some transitions while the device was locked, and the Camera app could be opened while the device was locked.

  CVE-ID

  CVE-2013-5161 : videosdebarraquito

Software Verification at Facebook


Moving Fast with Software Verification | Publications | Research at Facebook | Facebook
https research.facebook.com/publications/422671501231772/moving-fast-with-software-verification/
Reader
Faça a gestão d...king.com
Eventually Cons... Queue
ScopusProfile
PCT-FCT
ERC metrics
RID
TT-IST
MIT-TLO
TTxfer
WoS
CLIP
Research
Our Research
Programs
Publications
Events
Blog
Search publications, events and more
Like
Share
91

PUBLICATION

Moving Fast with Software Verification

Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, Dulma Rodriguez

NASA Formal Method Symposium · February 10

Abstract

For organisations like Facebook, high quality software is important. However, the pace of change and increasing complexity of modern code makes it difficult to produce error free software. Available tools are often lacking in helping programmers develop more reliable and secure applications.

Formal verification is a technique able to detect software errors statically, before a product is actually shipped. Although this aspect makes this technology very appealing in principle, in practice there have been many difficulties that have hindered the application of software

Relevance of Software Correctness

- Quality procedures must be enforced at all levels, in particular at the construction phase, where most of the issues are introduced and difficult to circunvent.
- **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?

Relevance of Software Correctness

- Quality procedures must be enforced at all levels, in particular at the construction phase, where most of the issues are introduced and difficult to circunvent.
- **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

Software Correctness: What and How

- **Key engineering concern**:
  – 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 methodological approaches to favour and validate correctness exist (software engineering course)
- In this course, we cover some techniques to rigorously ensure and validate correctness **during construction**

Software Correctness: What and How

- "runtime safety" is easier to define (no crashes, etc)
  - programming language type systems help on this
- other kinds of correctness are not so easy to define
- usually relative to special assumptions ...
  - what the system is supposed to do
    - play chess, manage bank accounts, ...
  - the available resources
    - bandwidth, memory, clock speed, ...
  - the security policies
    - only my friends can see my pics, ...
- To precisely define such assumptions, we need
  - rigorous **specification** languages
  - ways of validating that the system meets the spec

Specifications

- 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 ...
  - The spec should not be wrong !
  - It should be 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 source

What may specs look like?

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

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

- A and B are assertions (conditions on the program state)
- P is the program we want to check
- 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 non-local-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

Checking Specs: Dynamic Verification

- verification **done at runtime**
- success stories: **unit and coverage testing**
- runtime monitors to (continuously) check that code do not violate correctness properties
- violations causes exceptional behavior or halt, so errors are detected after something wrong already occurred (think of a car crash, or securiy leak)
- 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

Checking Specs: Static Verification

- verification done at **development / compile time**
- relies on reasoning about what programs do, by analyzing the source code
- does not incur in performance overhead, since the code is not actually executed
- can tackle many complex correctness properties (e.g., functionality, race absence, security, etc)
- success stories: type checking, as statically performed by a compiler, and extended static checking
- can ensure absence of all errors of a certain well defined kind, e.g., “no null dereferences”
- **challenge**: do you know how to give enough information to the verifier ? How “smart” is the verifier?

Some history ...

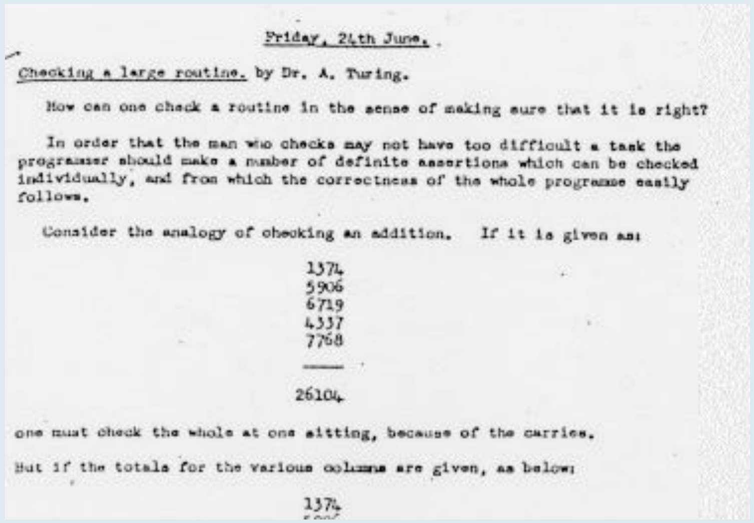
Turing

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

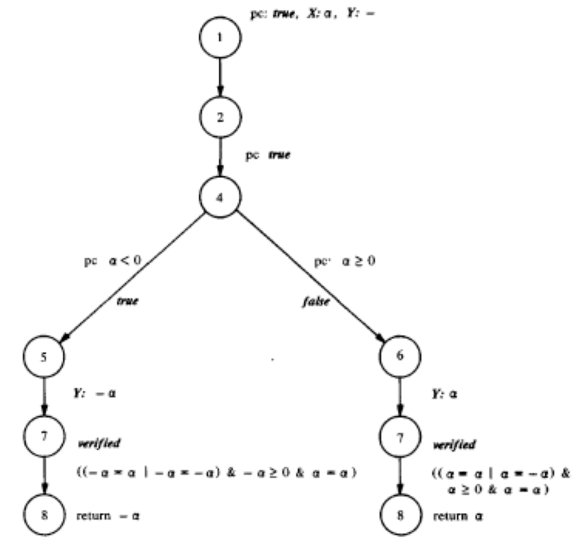


FIGURE 3. Symbolic execution tree for procedure *ABSOLUTE*.

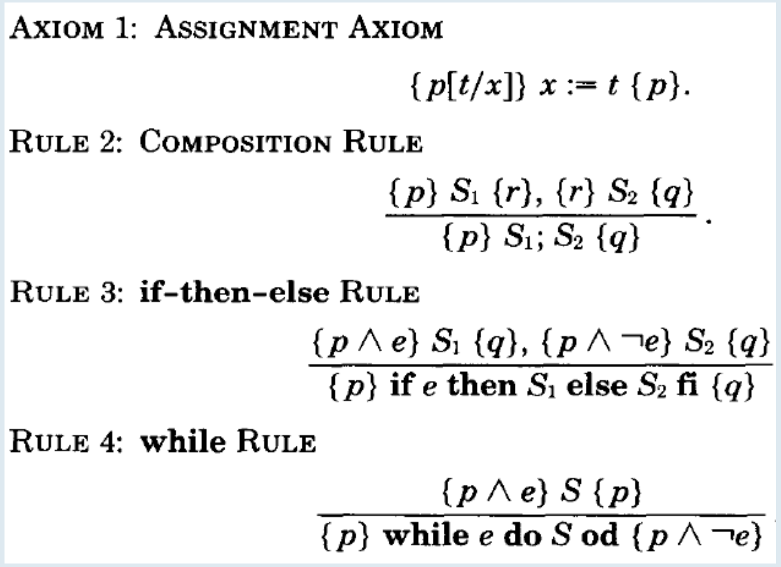
Program Specs

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

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

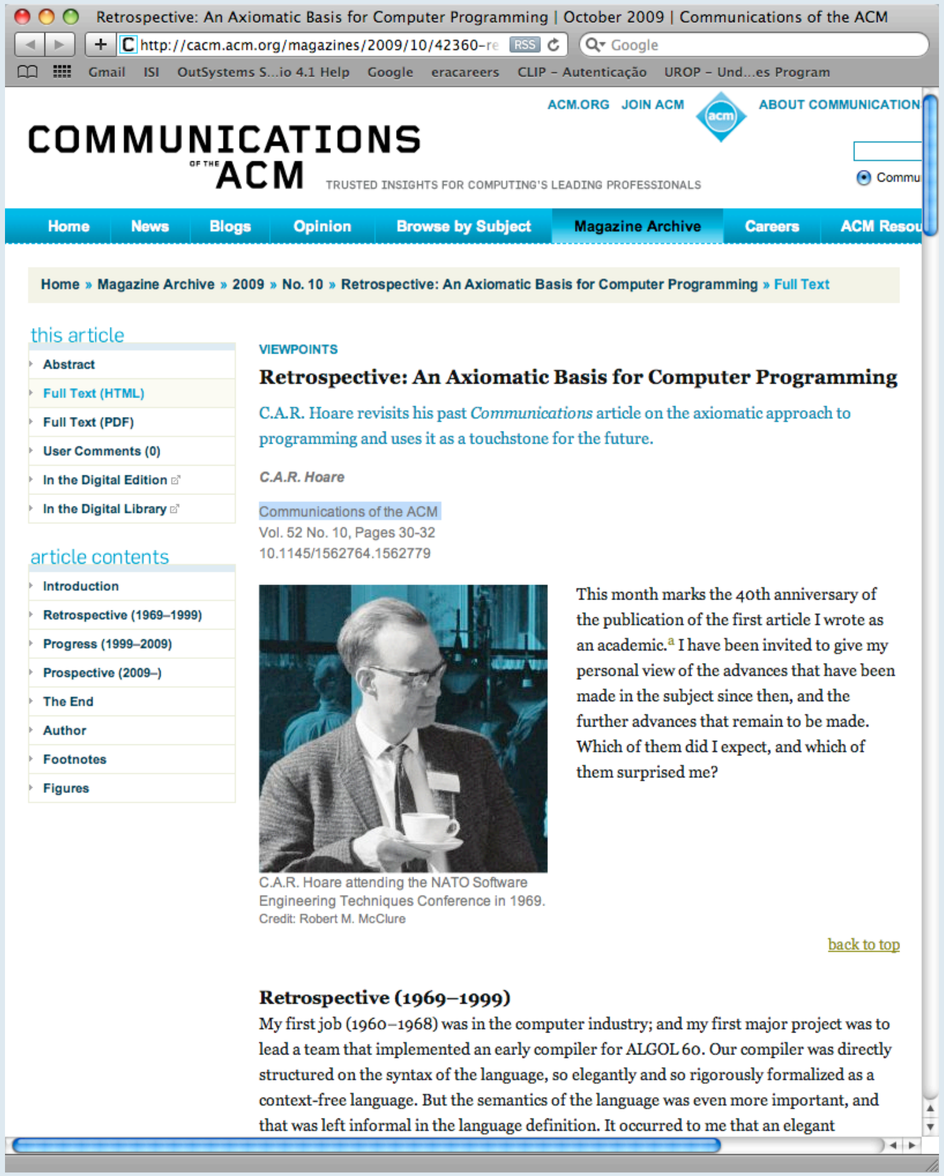
Hoare Logic Today

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


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

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

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

This month marks the 40th anniversary of the publication of the first article I wrote as an academic.[a] 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?

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

back to top

Retrospective (1969–1999)

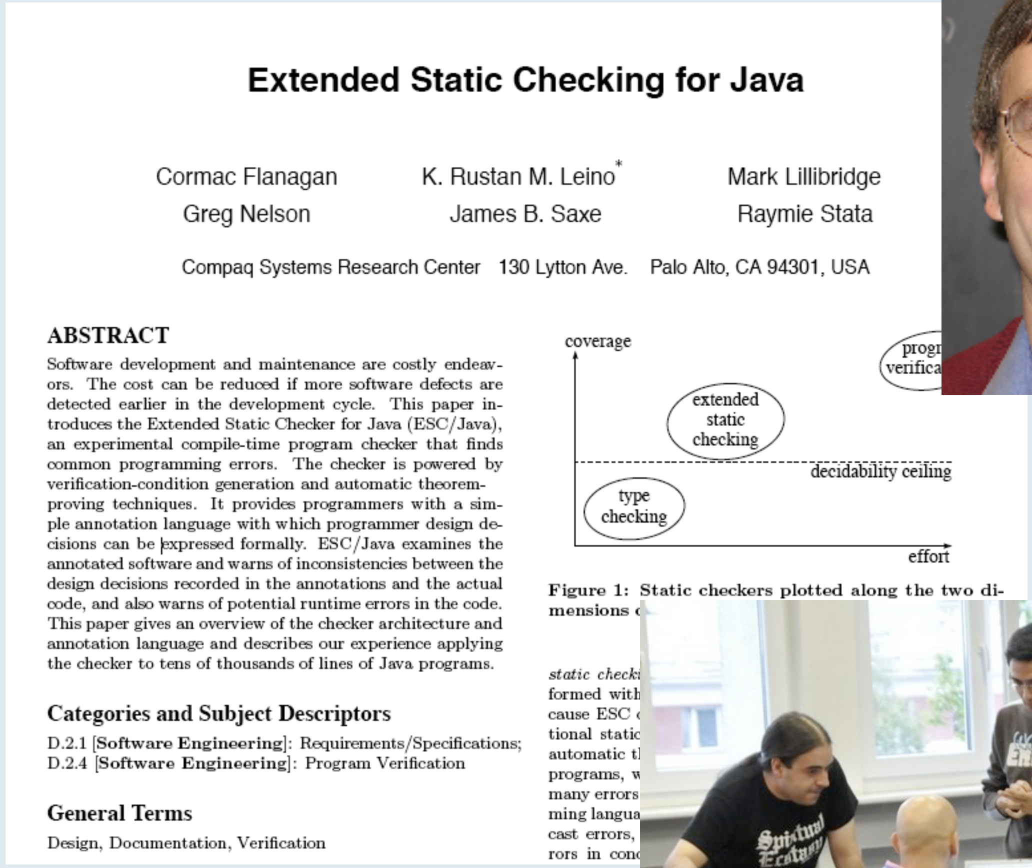
My first job (1960–1968) was in the computer industry; and my first major project was to lead a team that implemented an early compiler for ALGOL 60. Our compiler was directly structured on the syntax of the language, so elegantly and so rigorously formalized as a context-free language. But the semantics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

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 dimensions c

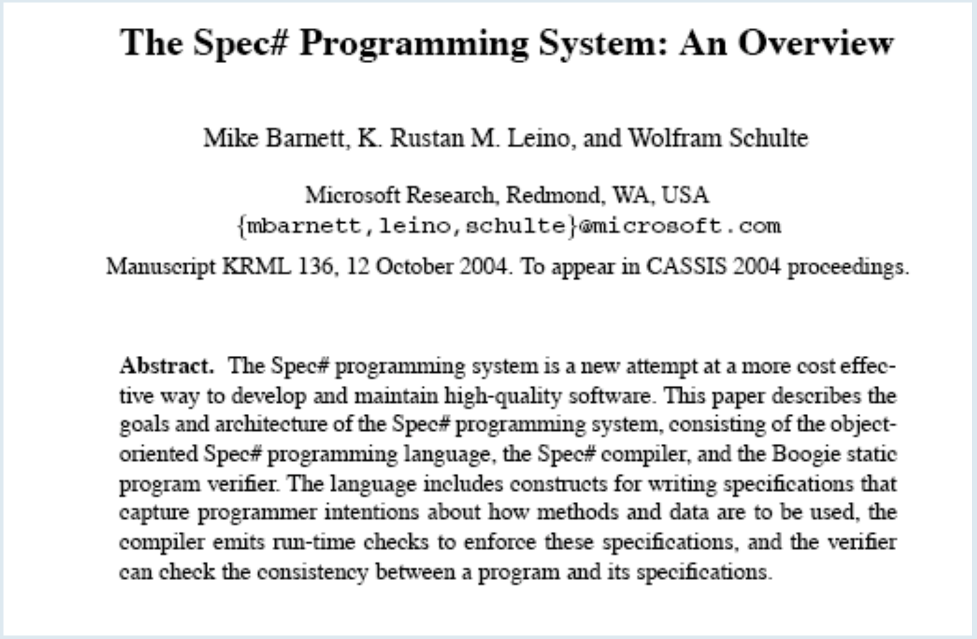
static check
formed with
cause ESC
tional static
automatic t
programs, w
many errors
ming langua
cast errors,
rors in con

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.

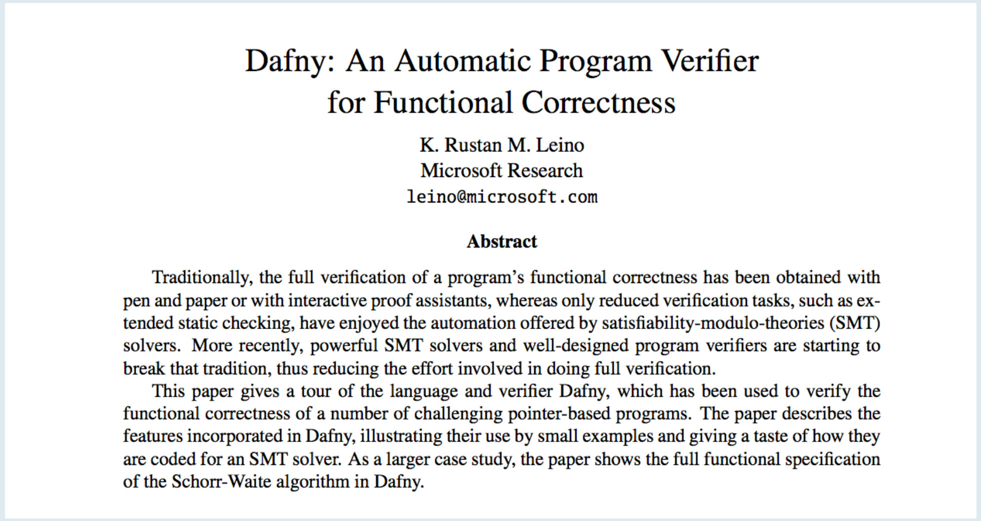
Abstract. The Spec# programming system is a new attempt at a more cost effective way to develop and maintain high-quality software. This paper describes the goals and architecture of the Spec# programming system, consisting of the object-oriented Spec# programming language, the Spec# compiler, and the 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



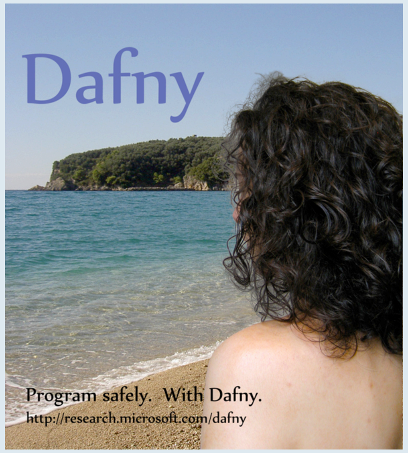
Dafny: An Automatic Program Verifier
for Functional Correctness

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

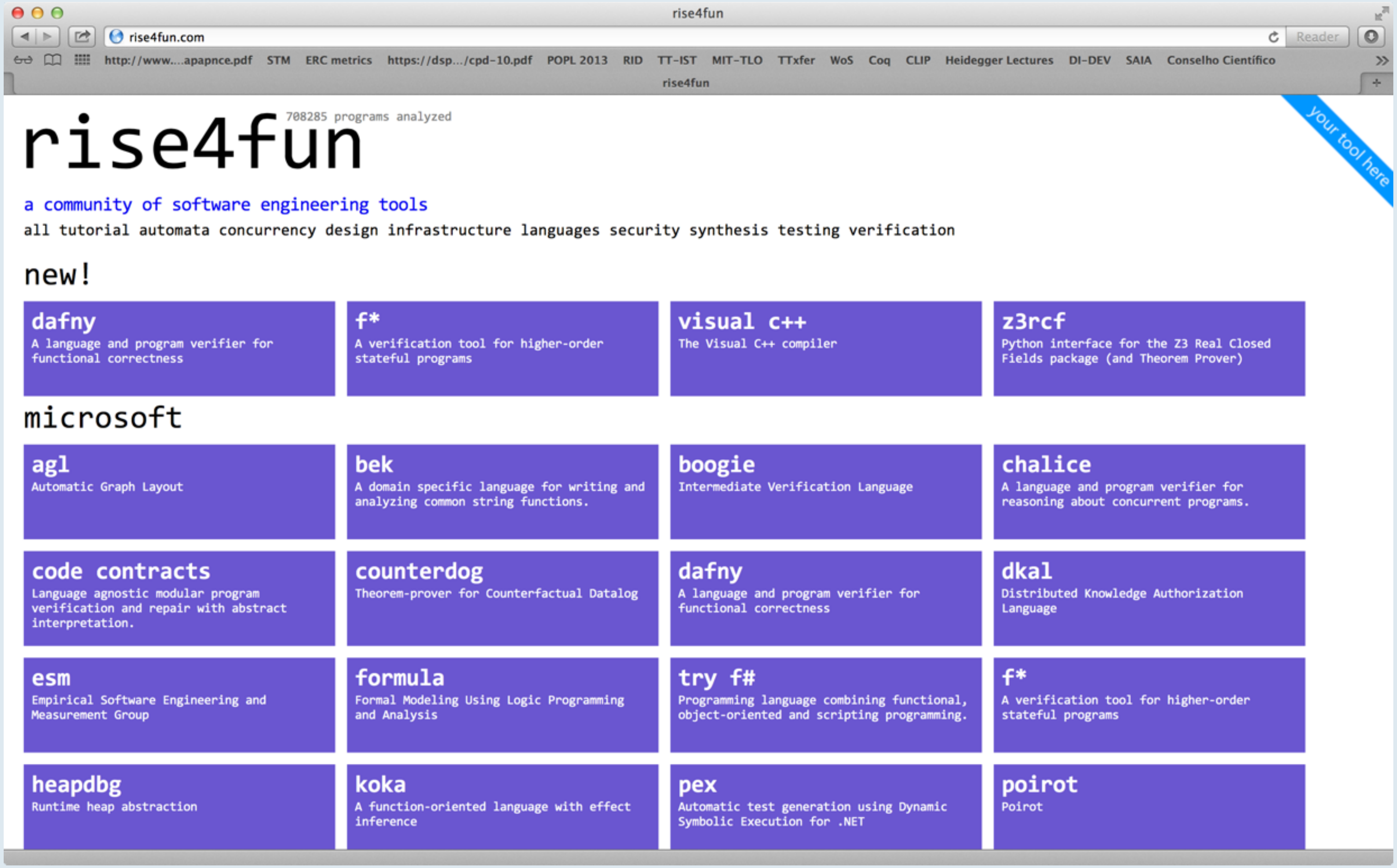
Abstract

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.


Dafny
Program safely. With Dafny.
http://research.microsoft.com/dafny

rise4fun @ 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

rise4fun 708285 programs analyzed

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

chalice A language and program verifier for reasoning about concurrent programs.

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

counterdog Theorem-prover for Counterfactual Datalog

dafny A language and program verifier for functional correctness

dkal Distributed Knowledge Authorization Language

esm Empirical Software Engineering and Measurement Group

formula Formal Modeling Using Logic Programming and Analysis

try 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

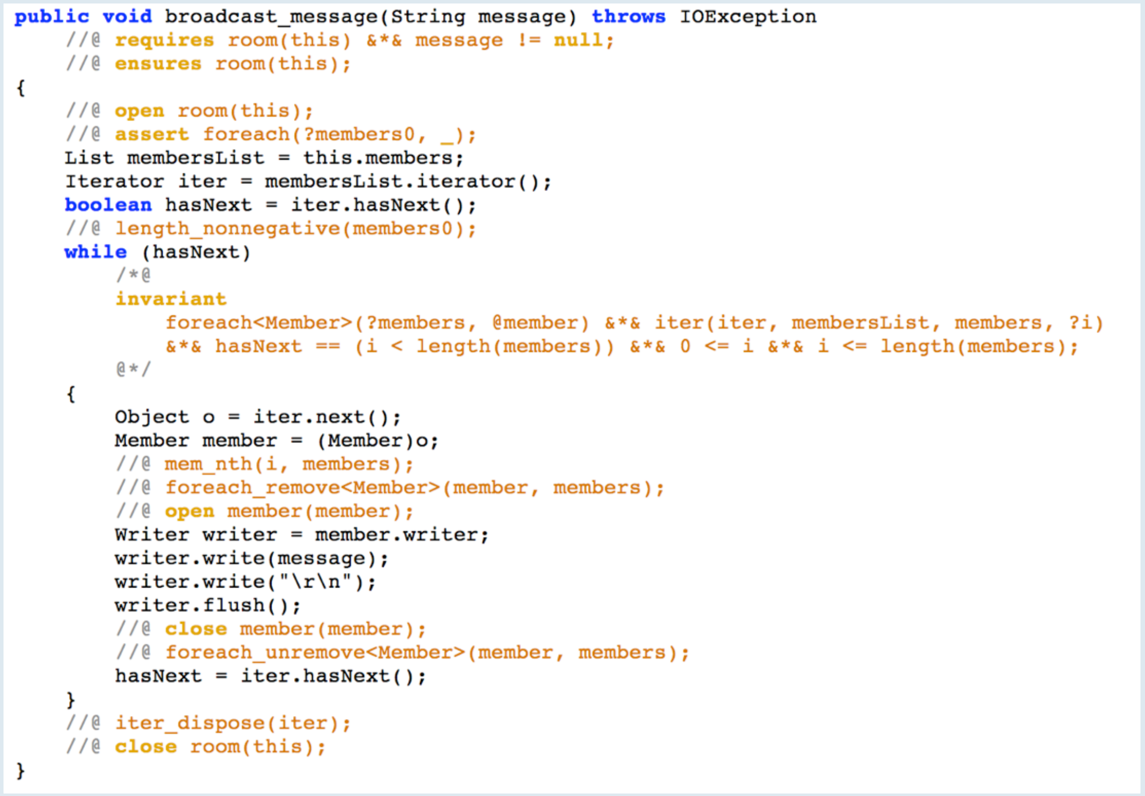
Verifast

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

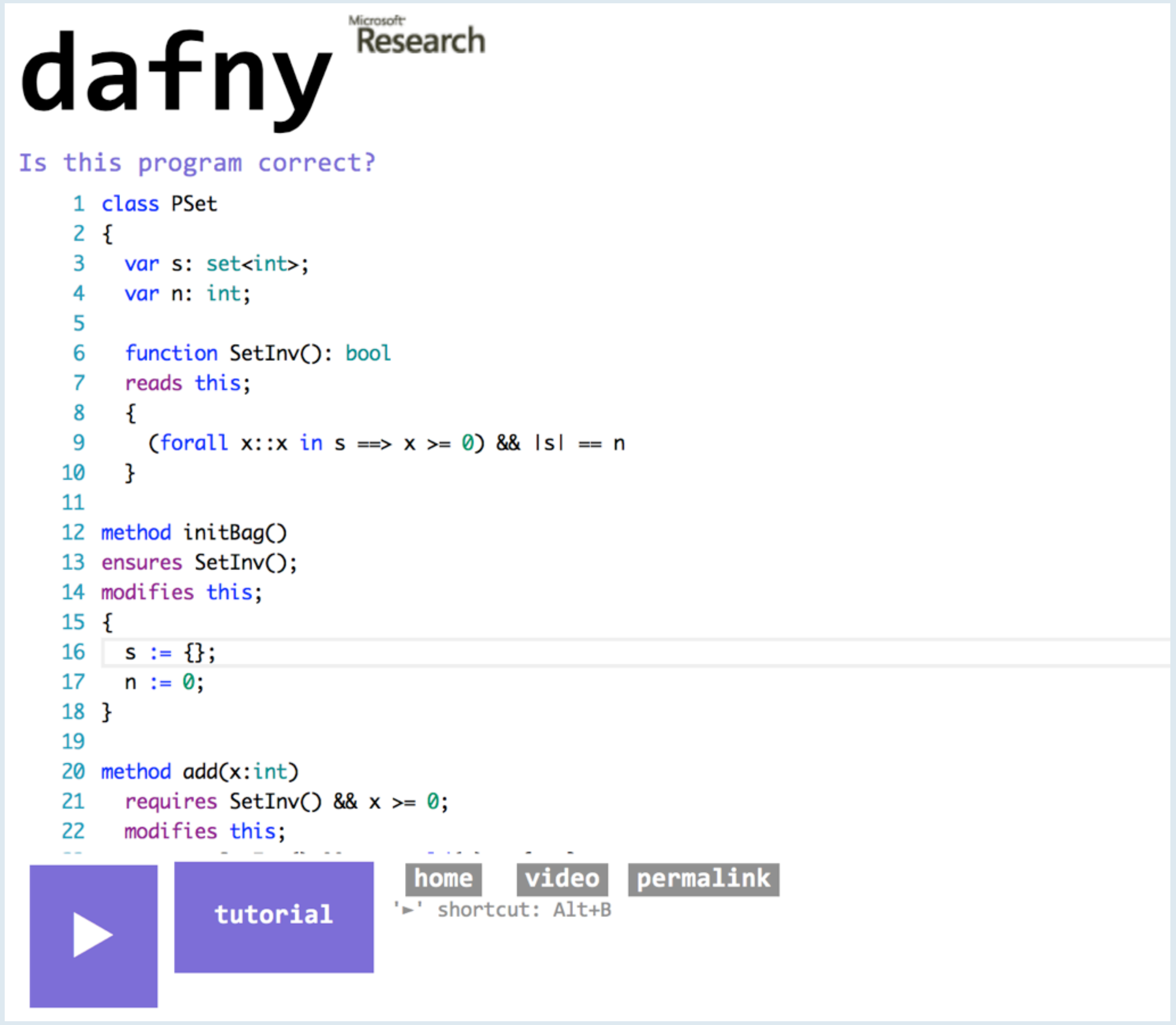
NB: separation logic is a spec language for talking about programs that allocate memory and use references



```
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);
}
```

Static Verification and Analysis of Software

Some Dafny programming first!



Is this program correct?

```
class PSet
{
  var s: set<int>;
  var n: int;

  function SetInv(): bool
  reads this;
  {
    (forall x::x in s ==> x >= 0) && |s| == n
  }

method initBag()
ensures SetInv();
modifies this;
{
  s := {};
  n := 0;
}

method add(x:int)
  requires SetInv() && x >= 0;
  modifies this;
```

Dafny warm up simple exercises

- consider the following method declaration:

```
method sum(a:array<int>, n:int) returns (s:int)
{
  ...
}
```

- The method sum is supposed to return (in s) the sum the values of the first n elements in array a. Write the code in Dafny.
- Compile your method in Dafny. Add the necessary contract assertions to compile your code without errors.
- Add to your code the postcondition

  ensures s >= 0

  Then, without changing your code, add to your code some method precondition that will enables your code to compile without errors.

Dafny warm up simple exercises

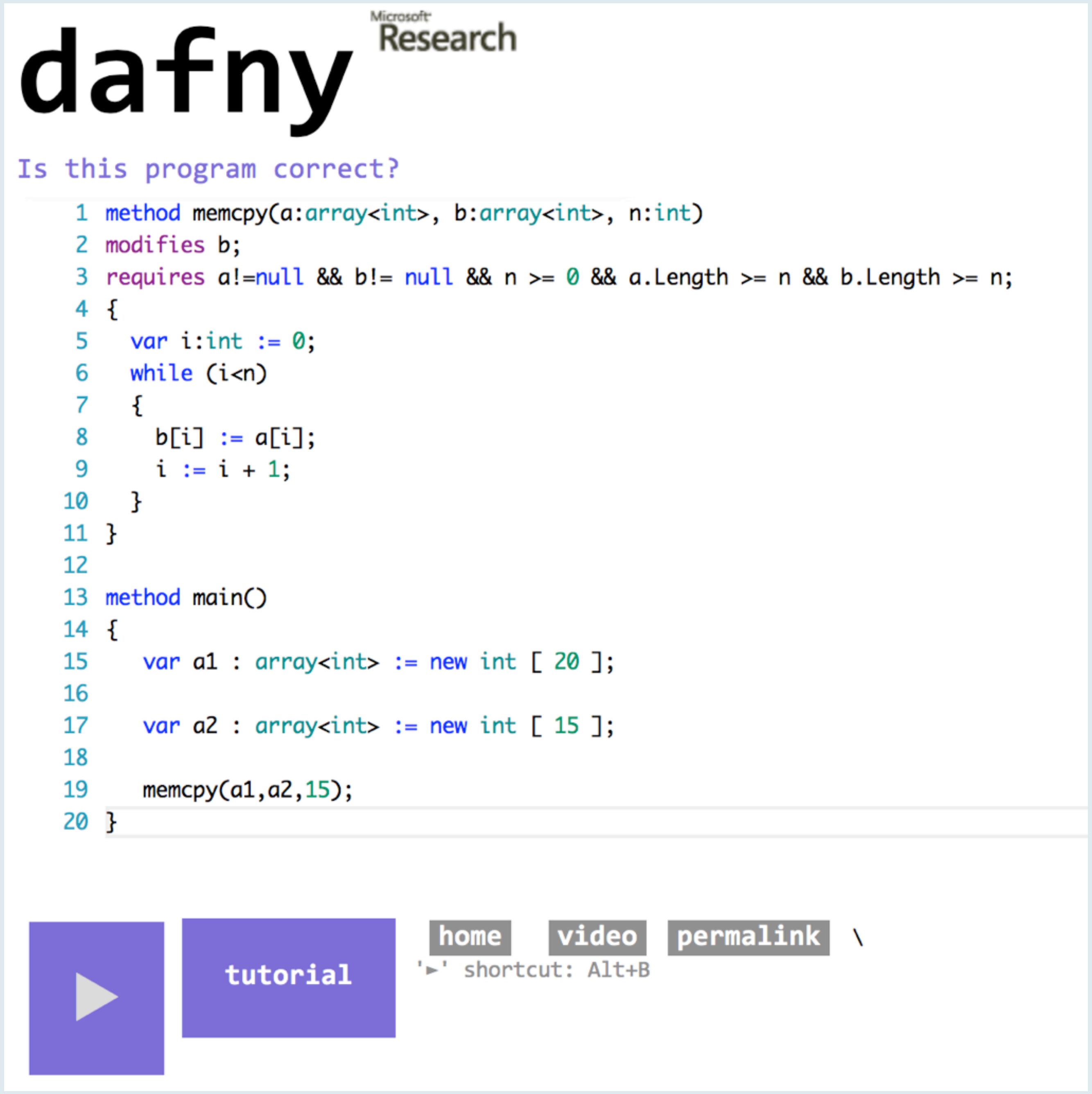
- consider the following method declaration:

```
method memcpy(a:array<int>, b:array<int>, n:int)
modifies b;
{
  ...
}
```

- The method memcpy is supposed to copy the first n values of array a to array b. Write the code in Dafny, with the necessary contract assertions to compile your code without errors.

- Write an appropriate post-condition that defines the behavior of the method.

- Check your code with Dafny. We need to define the invariant of the while loop.



Is this program correct?

```
method memcpy(a:array<int>, b:array<int>, n:int)
modifies b;
requires a!=null && b!= null && n >= 0 && a.Length >= n && b.Length >= n;
{
  var i:int := 0;
  while (i<n)
  {
    b[i] := a[i];
    i := i + 1;
  }
}

method main()
{
   var a1 : array<int> := new int [ 20 ];

   var a2 : array<int> := new int [ 15 ];

   memcpy(a1,a2,15);
}
```

Basic Program Specs (Hoare Logic)

Tony Hoare (MSR)

A.M. TURING AWARD

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

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

Hoare Logic

- Hoare logic is to talk about properties of programs
- What properties are we generally interested in?
  - 1. safety properties (partial correctness)
  - 2. prove that **if** the program terminates (delivers a an outcome), **then** the final state satisfies some property
  - 3. liveness properties (total correctness)
  - 4. state that the program actually terminates (at least, under certain well specified conditions)
- Hoare logic is the "mother of all program logics":
  - It helps us on 1 and 2, but not on 3 and 4 ...
- Reason for HL success: verification at the level of the programming languages (not of programs, cf. Floyd)
- Applicable to any imperative programming language

Simple Programming Language

E ::=		
	num	% integer
	x	% variable
	E + E ...	% integer operators
	E < E ...	% relational operators
	E and E ...	% boolean operators
P ::=		
	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

- An imperative code snippet essentially a state transformer, it transforms a initial state into a target state

- What is a **state**?

  state = assignment of **values** to **memory variables**

  e.g., $\sigma = \{ x \rightarrow 1, y \rightarrow 2, z \rightarrow 3 \}$

- A (imperative) program transforms states into states

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

  Then P executed in state σ yields state σ' where

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

  We may say that P **transforms** σ in σ'

  P is only defined on states σ where $\mathrm{vars}(P) \subseteq \mathrm{dom}(\sigma)$

States and Assertions

- A (correctness) property is a set of (good) states
- What is an **assertion** ?
  - A logical formula specifying a set of states (x != null)
- 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).

States and Assertions

Consider a state over variables x,...,z, myfile, yourfile

Examples of assertions:

- x > y
- x = 2*y
- s = max(x,y)
- forall i :: 0 <= i < n ==> store[i] >= 0
- CanWrite(myfile) && !CanWrite(yourfile)

Note that:

- An assertion need not specify a single state!
  - *E.g.*, even(x) and odd(y)
- An assertion may specify no state at all!
  - *E.g.*, x = x + 1, false, etc, ...

Hoare Triples

- Hoare logic is a deductive system for statically reasoning about programs
- The statements of Hoare Logic are called **Hoare triples**

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

{ A }	A is a state formula
{ B }	B is a state formula
P	P is a program

- Q: What does mean { A } P { B }, if it is valid?
- A: It means that program P, whenever started in a state that satisfies property A, **if it terminates**, always ends in a state that satisfies property B

Operation Contracts in ADTs

- 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 non-local-state-changed MOD
{
  { PRE } method code { POST }
}
```

- 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 ADTs

- 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

Rules of Hoare Logic

Proofs in Hoare Logic

- A proof in Hoare logic adds assertions between program statements, making sure that the program statements satisfy the corresponding Hoare triples.
- For example, consider the code snippet

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

- A Hoare Logic derivation 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

$$\frac{\{\,A\,\}\;P\;\{\,B\,\} \qquad \{\,B\,\}\;Q\;\{\,D\,\}}{\{\,A\,\}\;P;Q\;\{\,D\,\}}$$

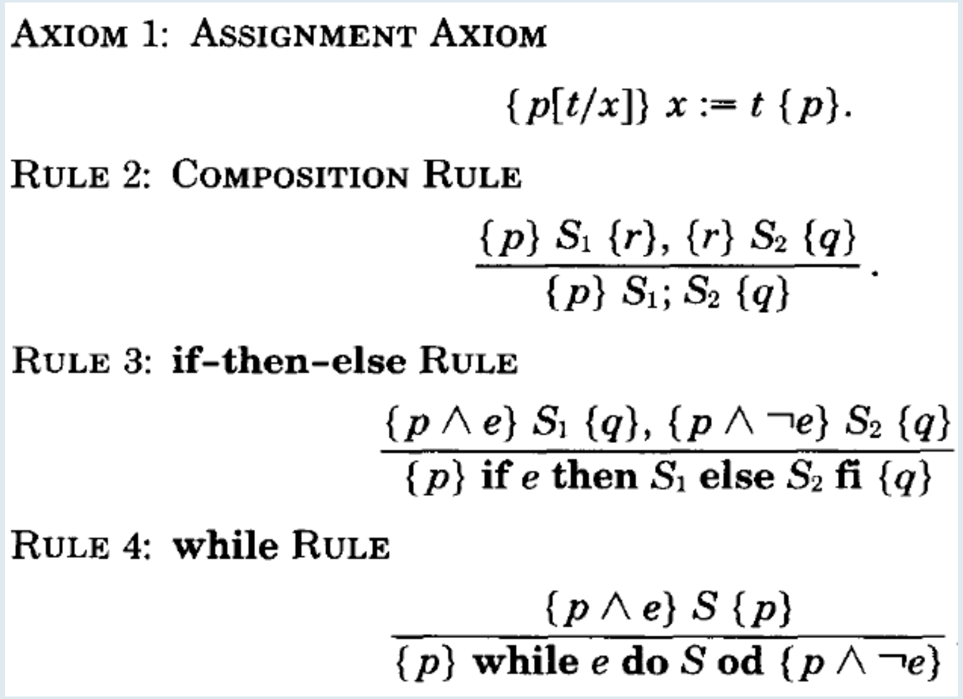
Rules of Hoare Logic (general form)

- The inference rules of Hoare logic may be presented as regular inference rules, deriving (valid) Hoare triples given some already derived Hoare triples

$$\frac{\{\ A_1\ \}\ P_1\ \{\ B_1\ \}\quad \ldots\ \{\ A_n\ \}\ P_n\ \{\ B_n\ \}}{\{\ C\ \}\ C(P_1, \ldots, P_n)\ \{\ D\ \}}$$

- 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

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

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

RULE 4: **while** RULE

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

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

Rule for Skip

{ A } **skip** { A }

Rule for Sequence

$$\frac{\{\,A\,\}\ P\ \{\,B\,\} \qquad \{\,B\,\}\ Q\ \{\,D\,\}}{\{\,A\,\}\ P;Q\ \{\,D\,\}}$$

Rule for Conditional

$$\frac{\{\ A\ \&\&\ E\ \}\ P\ \{\ B\ \} \qquad \{\ A\ \&\&\ !E\ \}\ Q\ \{\ B\ \}}{\{\ A\ \}\ \textbf{if}\ E\ \textbf{then}\ P\ \textbf{else}\ Q\ \{\ B\ \}}$$

Rule for Deduction

$$\frac{A' \Rightarrow A \quad \{ A \}\, P \,\{ B \} \quad B \Rightarrow B'}{\{ A' \}\, P \,\{ B' \}}$$

- $A \Rightarrow B$ means:
  - A logically implies B
  - We prove $A \Rightarrow B$ using the "usual" logical principles

Rule for Assignment

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

- A[x/E] 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

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

- We can think of A as a condition where “x” appears in some places. A is a condition constrained by “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
- Example: $\{ x + 1 > 0 \}\ x := x + 1\ \{ x > 0 \}$

Rule for Assignment

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

- Example:
  - $\{ (x+1 > 0) \}\ x := (x+1)\ \{ x > 0 \}$ the same as
  - $\{ (x > 0)[x / x+1] \}\ x := (x+1)\ \{ x > 0 \}$ by (Assignment)
  - $\{ (x + 1) > 0 \}\ x := (x+1)\ \{ x > 0 \}$ iff (by log equiv)
  - $\{ x > -1 \}\ x := x + 1\ \{ x > 0 \}$ iff (by log equiv)

Rule for Assignment

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

- Trick: if x does not occur in E, A
  - We can always write { A && E == E } x := E { x == E }
  - So, the following triple is always valid

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

if x does not occur in E, A

Rule for Assignment

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

- 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) }

Simple Example

- Consider the program

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

- We should be able to (mechanically) check that

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

Simple Example

- Consider the program

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

- Hint, check

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

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

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

Rule for Iteration

$$\frac{\{\, A \;\&\&\; E \,\}\ P\ \{\, A \,\}}{\{\, A \,\}\ \textbf{while}\ E\ \textbf{do}\ P\ \{\, A \;\&\&\; !E \,\}}$$

Rule for Iteration

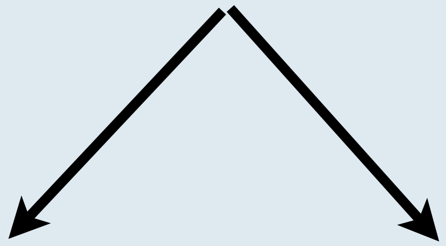
INV = Invariant Condition

$$\frac{\{\ INV\ \&\&\ E\}\ P\ \{\ INV\ \}}{\{\ INV\ \}\ \textbf{while}\ E\ \textbf{do}\ P\ \{\ INV\ \&\&\ !E\}}$$

- We cannot predict in general how many iterations will the while loop do (undecidability of the halting problem).
- We approximate execution by an invariant condition
- A loop invariant is a condition that always hold at loop entry and at loop exit.

Rule for Iteration

INV = Invariant Condition



$$\frac{\{\ \mathrm{INV}\ \&\&\ \mathrm{E}\}\ \mathrm{P}\ \{\ \mathrm{INV}\ \}}{\{\ \mathrm{INV}\ \}\ \textbf{while}\ \mathrm{E}\ \textbf{do}\ \mathrm{P}\ \{\ \mathrm{INV}\ \&\&\ !\mathrm{E}\}}$$

- If the invariant holds initially and is preserved by the loop body, it will hold when the loop terminates!
- It does not matter how many iterations will run
- Unlike for other rules of Hoare logic, finding the invariant requires human intelligence (you are a programmer :-)

Loop Invariants

- Consider the program

$P \triangleq s:=0\ ;\ i := 0;\ \textbf{while}\ (i<n)\ \textbf{do}\ \{\ i := i+1;\ s := s+i\ \}$

- What does P do?

$\{\ ??\ \}\ P\ \{\ ??\ \}$

Loop Invariants

- Consider the program

  P ≜ s:=0 ; i := 0; **while** (i<n) **do** { i := i+1; s := s+i }

- Here is a specification of program P

  { n >= 0 } P { s == $\Sigma(_{j=0..n},j)$ }

Loop Invariants

- Consider the program

  P ≜ s:=0 ; i := 0; **while** (i<n) **do** { i := i+1; s := s+i }

- We should be able to (mechanically) check that

  { n >= 0 } P { s == $\Sigma(_{j=0..n},j)$ }

Loop Invariants

```
{ n >= 0 }
s:=0 ;

i := 0;

while (i<n) do {

    i := i + 1;
    s := s + i;

}

{  s == Σ(j=0..n,j) }
```

Loop Invariants

```
{ n >= 0 }
s:=0 ;
{ n >=0 && s == 0 }
i := 0;

while (i<n) do {

    i := i + 1;
    s := s + i;

}

{ s == Σ(j=0..n,j) }
```

Loop Invariants

```
{ n >= 0 }
s:=0 ;
{ n >=0 && s == 0 }
i := 0;
{  s == $\Sigma(_{j=0..i},j)$ && 0 <= i <= n }
while (i<n) do {
    {  s == $\Sigma(_{j=0..i},j)$ && 0 <= i <= n }
    i := i + 1;
    s := s + i;
    {  s == $\Sigma(_{j=0..i},j)$ && 0 <= i <= n }
}
{  s == $\Sigma(_{j=0..i},j)$ && 0 <= i <= n  && i>=n} =>
{  s == $\Sigma(_{j=0..n},j)$ }
```

Invariant holds

Loop Invariants

```
{ n >= 0 }
s:=0 ;
{ n >=0 && s == 0 }
i := 0;
{  s == Σ(j=0..i,j) && 0 <= i <= n }
while (i<n) do {
{  s == Σ(j=0..i,j) && 0 <= i <= n }    <- Invariant holds
    {  s == Σ(j=0..i,j) && 0 <= i < n }  i := i + 1;
    {  s == Σ(j=0..i-1,j) && 0 <= i <= n } s := s + i;
    {  s == Σ(j=0..i,j) && 0 <= i <= n }    <- Invariant holds
}
{  s == Σ(j=0..i,j) && 0 <= i <= n  && i>=n}
{  s == Σ(j=0..n,j) }
```

Loop Invariants

```
{ n >= 0 }
s:=0 ;
{ n >=0 && s == 0 }
i := 0;                                                              Invariant
{ $s == \Sigma(_{j=0..i},j)$ && 0 <= i <= n }                       broken
while (i<n) do {
{ $s == \Sigma(_{j=0..i},j)$ && 0 <= i <= n }
    { $s == \Sigma(_{j=0..i},j)$ && 0 <= i < n }  i := i + 1;
    { $s == \Sigma(_{j=0..i-1},j)$ && 0 <= i <= n } s := s + i;
    { $s == \Sigma(_{j=0..i},j)$ && 0 <= i <= n }         Invariant
}                                                          restored
{ $s == \Sigma(_{j=0..i},j)$ && 0 <= i <= n  && i>=n}
{ $s == \Sigma(_{j=0..n},j)$ }
```

Loop Invariants

```
{ n >= 0 }
s:=0 ;

i := 0;

while (i<n) do {

    i := i + 1;
    s := s + i;

}

{ s == Σ(j=0..n,j) }
```

Hints for finding loop invariants

- **First**: carefully think about the post condition of the loop
  - Typically the post-condition talks about a property "accumulated" across a "range" (this is why you are using a loop, right ?)
  - e.g., maximum of all elements of an array
  - e.g., visited elements in a data structure
- **Second**: design a "generalized" version of the post-condition, in which the already visited part of the data is made explicit as a function of the "loop control variable"
- The loop body will temporarily break the invariant, but must restore it at the end of the body
- **Important**: make sure that the invariant together (&&) with the termination condition really implies your post-condition

Exercise

- max

```
// return the maximum of the values in array a[-]
// in positions i such that 0 <= i < numelems
// numelems > 0
// numelems cannot exceed the allocated array size
static int max(int a[], int numelems)
```

Exercise

- find

```
// if there is an index j ( 0 <= j < numelems )
// such that a[j] == v then return j
// otherwise return -1
// numelems cannot exceed the allocated array size
static int find(int v, int a[], int numelems)
```

Exercise

- reverse

```
// the method reverse should return a “copy” of array a
// but with the first n elements by reverse order

method reverse(a:array<int>, int n) returns (b:array<int>)
{

}

// write the code and fully check it with dafny
// define the weakest preconditions you can think of
// define the strongest postconditions you can think of
```

Exercise

- filled

```
// the method filled returns true if and only if
// the first n elements of array are all set to value v

method filled(a:array<int>, v:int, n:int) returns (s:bool)
{

}

// write the code and fully check it with dafny
// define the weakest preconditions you can think of
// define the strongest postconditions you can think of
```

Exercise

- strf

```
// the method checks if b appears within a, and returns the
// position if that is the case, or -1 if not
// n is the number of chars in a, m is the number of chars in b

method strf(a:array<char>, n:int, b:array<char>, m:int) returns (pos:int)
{

}

// write the code and fully check it with dafny
// define the weakest preconditions you can think of
// define the strongest postconditions you can think of
```

Exercise

- fibo

```
function fib(n : int) : int  // this is the recursive spec of fibonacci
requires n>=0;
{
  if (n==0) then 1 else
  if (n==1) then 1 else fib(n-1)+fib(n-2)
}

// the method fibo below should implement fib efficiently
// “bottom up” using a while loop

method fibo(n : int) returns (f : int)
requires n>=0;
ensures f == fib(n);
```

Exercise

```
// consider the following functions

function initarray(c:array<int>,nelems:int):bool
{ // array c is ok for nelems
  c!=null && 0<=nelems<=c.Length
}

function sorted(c:array<int>, nelems:int):bool
requires initarray(c,nelems);
reads c;
{ // first nelems elements are sorted
forall i:: (0<=i<nelems) ==> forall j::(i<j<nelems) ==> c[i]<=c[j]
}
```

Exercise

- insert

```
// the method inserts integer v in the sorted array a
// if a already contains v, the method does nothing

method insert(a:array<int>, nelems:int, v:int) returns (newsize:int)
modifies a;
requires initarray(a, nelems+1) && sorted(a, nelems);
ensures nelems <= newsize <= 1+nelems && sorted(a,newsize);
ensures exists p:: 0<=p<newsize && a[p] == v;
{
}

// write the code and fully check it with dafny
// define the weakest preconditions you can think of
// define the strongest postconditions you can think of
```

Exercise

- sort

```
// the method sort returns in b a sorted array
// first consider the following post-conditions
// and write the code for sort (use the selection sort algorithm)

method sort(a:array<int>, nelems:int, b:array<int>)
modifies b;
requires initarray(a,nelems) && initarray(b,nelems);
ensures sorted(b,nelems);
{
}

// to express the loop invariants, you may find it useful
// the function majors defined in the previous slide
```

Exercise

- sort

```
// the method sort returns in b a sorted array
// first consider the following post-conditions
// and write the code for sort (use the selection sort algorithm)

function majors(c:array<int>,i:int,nelems:int):bool
requires initarray(c,nelems);
reads c;
{
// first i elems of c are <= than the elems from i to nelems-1
forall k::0<=k<i ==> forall l::i<=l<nelems ==> (c[k] <= c[l])
}
```

Operation Contracts in ADTs

- 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 non-local-state-changed MOD
{
  { PRE } method code { POST }
}
```

- 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

Hints for finding loop invariants

- **First**: carefully think about the post condition of the loop
  - Typically the post-condition talks about a property “accumulated” across a “range” (this is why you are using a loop, right ?)
  - e.g., maximum of all elements of an array
  - e.g., visited elements in a data structure
- **Second**: design a “generalized” version of the post-condition, in which the already visited part of the data is made explicit as a function of the “loop control variable”
- The loop body will temporarily break the invariant, but must restore it at the end of the body
- **Important**: make sure that the invariant together (&&) with the termination condition really implies your post-condition

Invariants in ADTs

- 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...
}
```

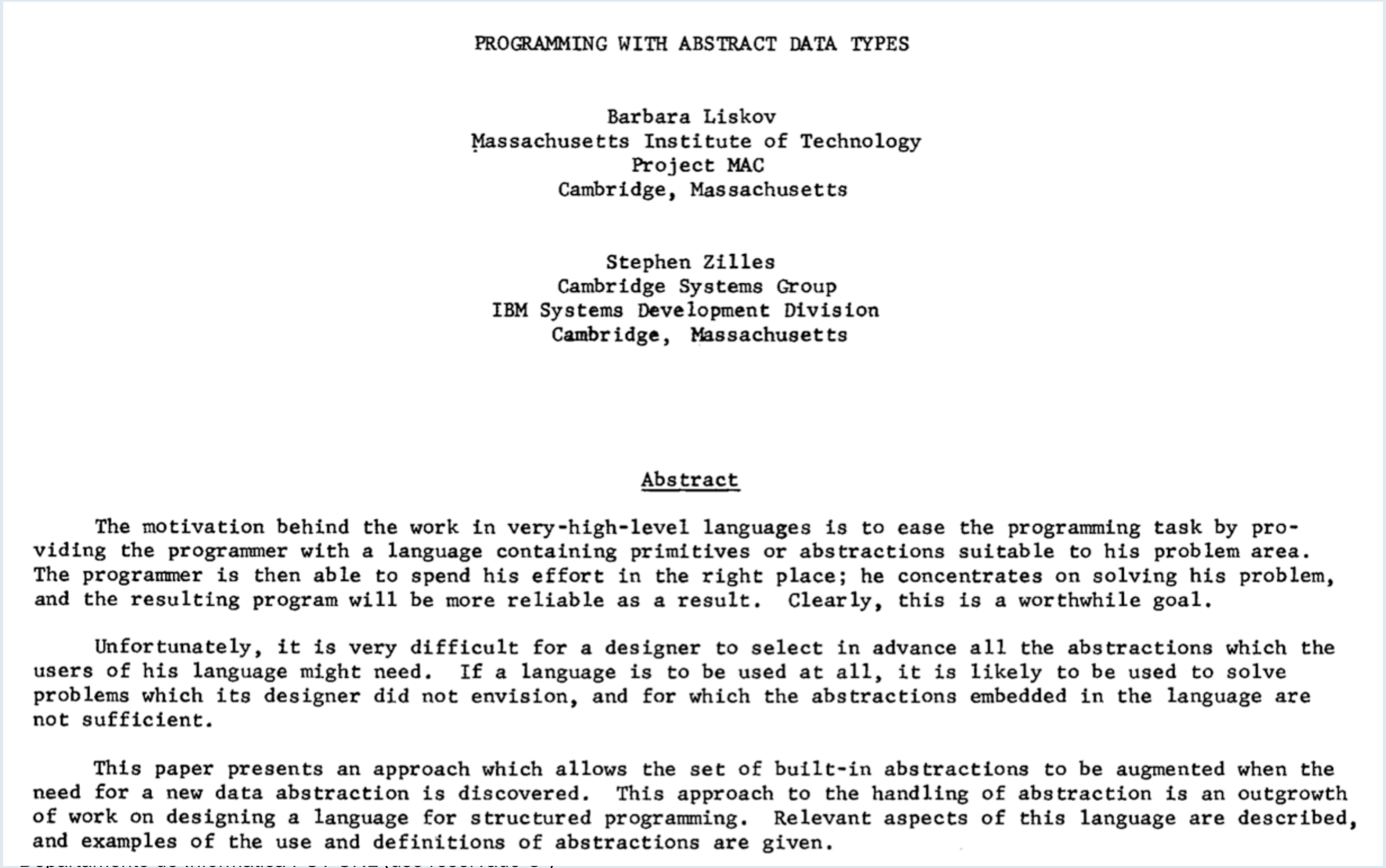
- ADT C implementation relies on a representation type T that satisfies the representation invariant REPINV and maps into the abstract type as specified by ABSMAP

Abstract Data Types
Classes and Objects

Abstract Data Types (Liskov, 78)

- ADTs: building blocks for software construction
  - Software System : set of ADTS
  - Promotes reuse, modifiability, and correctness

ADTs (Liskov & Zilles,78)



PROGRAMMING WITH ABSTRACT DATA TYPES

Barbara Liskov
Massachusetts Institute of Technology
Project MAC
Cambridge, Massachusetts

Stephen Zilles
Cambridge Systems Group
IBM Systems Development Division
Cambridge, Massachusetts

Abstract

The motivation behind the work in very-high-level languages is to ease the programming task by providing the programmer with a language containing primitives or abstractions suitable to his problem area. The programmer is then able to spend his effort in the right place; he concentrates on solving his problem, and the resulting program will be more reliable as a result. Clearly, this is a worthwhile goal.

Unfortunately, it is very difficult for a designer to select in advance all the abstractions which the users of his language might need. If a language is to be used at all, it is likely to be used to solve problems which its designer did not envision, and for which the abstractions embedded in the language are not sufficient.

This paper presents an approach which allows the set of built-in abstractions to be augmented when the need for a new data abstraction is discovered. This approach to the handling of abstraction is an outgrowth of work on designing a language for structured programming. Relevant aspects of this language are described, and examples of the use and definitions of abstractions are given.

Barbara Liskov (MIT)

A.M. TURING AWARD

BARBARA LISKOV
United States – **2008**

For contributions to practical and theoretical foundations of programming language and system design, especially related to data abstraction, fault tolerance, and distributed computing.

Barbara Liskov (MIT)

Barbara Liskov (MIT)

Turing Award winner

ADTs

Distributed
OO programming

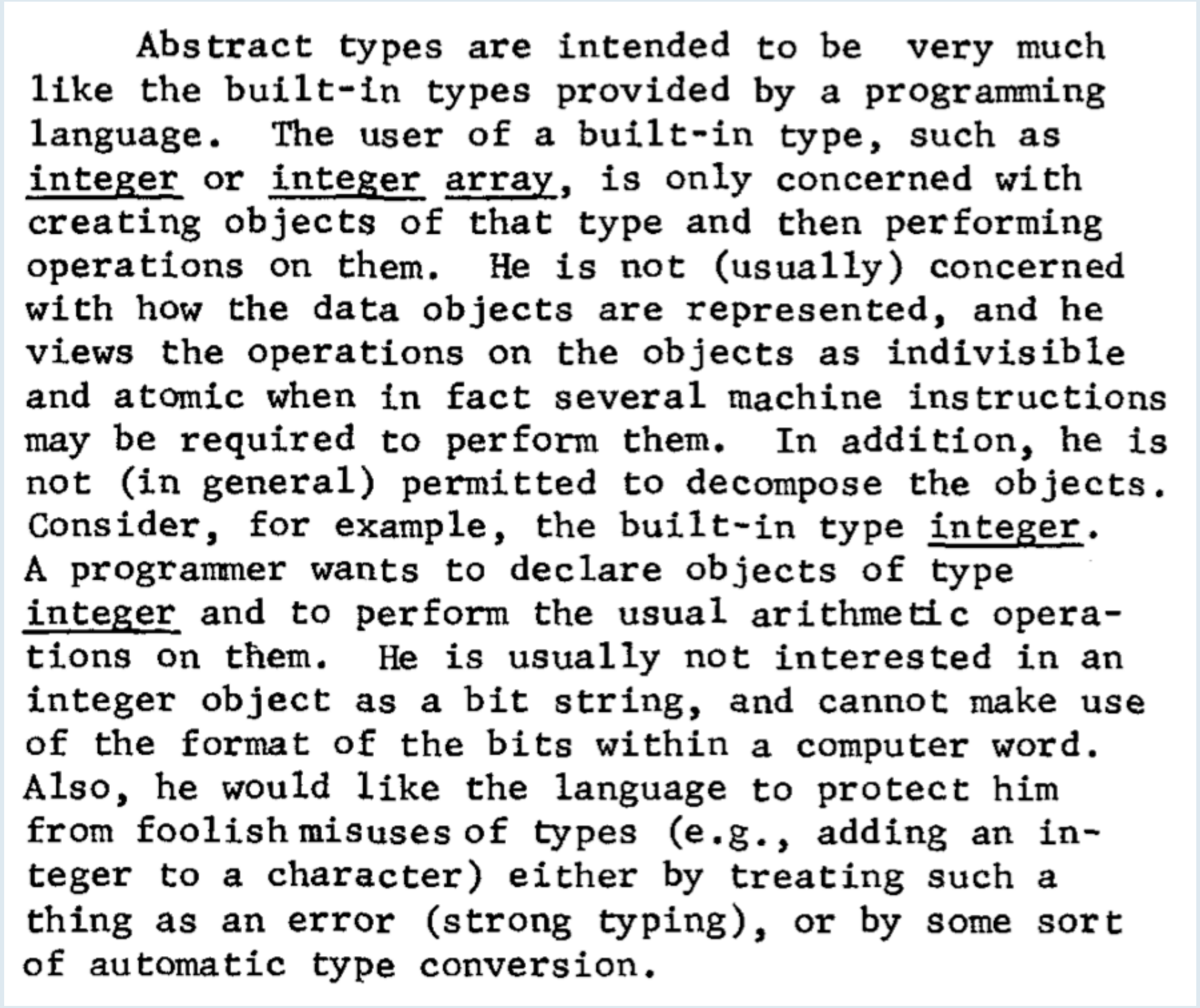
Behavioral
specifications

DI FCT UNL
Distinguished Lecture #1
3 October 2012 - 14:00

Abstract Data Type

- External View
  - A public opaque data type (that clients will use)
    Note: opaque means = behaves as a primitive type

Abstract Data Type



Abstract types are intended to be very much like the built-in types provided by a programming language. The user of a built-in type, such as integer or integer array, is only concerned with creating objects of that type and then performing operations on them. He is not (usually) concerned with how the data objects are represented, and he views the operations on the objects as indivisible and atomic when in fact several machine instructions may be required to perform them. In addition, he is not (in general) permitted to decompose the objects. Consider, for example, the built-in type integer. A programmer wants to declare objects of type integer and to perform the usual arithmetic operations on them. He is usually not interested in an integer object as a bit string, and cannot make use of the format of the bits within a computer word. Also, he would like the language to protect him from foolish misuses of types (e.g., adding an integer to a character) either by treating such a thing as an error (strong typing), or by some sort of automatic type conversion.

Abstract Data Type (External View)

- External View
  - A public opaque data type (that clients will use)
    Note: opaque means = behaves as a primitive type
  - A set of operations on this data type
  - Operations must neither reveal, nor allows a client to mess up the internal representation
  - pre and post conditions on these operations must be expressed in terms of the abstract type (the only type known to the client)
  - This is why ADTs promote reuse, modifiability, and correctness: the developer can change the implementation anytime, without breaking contracts

Abstract Data Type (Internal View)

- Internal View
  - A **representation** data type (hidden from clients)
  - A set of operations on the representation data type
- ***important remarks***
  - A programmer must define the operations in such a way that the representation state (invisible to clients) is kept consistent with the intended abstract state
  - Pre-conditions on the public operations, expressed on the abstract state, must map into pre-conditions expressed in terms of the representation state
  - The same for post-conditions
  - At all times the concrete state must represent a well defined abstract state (otherwise something is wrong!)

Example (PSet ADT)

```
class PSet {
// an abstract Pset aset

method new(sz:int) {}
// initializes aset ( e.g., Java constructor )

method add(v:int) {}
// adds v to aset if space available )

function size() : int
// returns number of elems in aset

function maxsize() : int
// returns max number of elems allowed in aset

}
```

Technical ingredients in ADT design

- The ***abstract state***
  - defines how client code sees the object
- The ***representation type***
  - chosen by the programmer to implement the ADT internals. The programmer is free to chose the implementation strategy (data-structures, algorithms). This is done at construction time.
- The ***concrete state***
  - in general, not all representation states are legal concrete states
  - a concrete state is a representation state that really represents some well-defined abstract state

Technical ingredients in ADT design

- The ***representation invariant***
  - the rep invariant is a condition that restricts the representation type to the set of concrete states
  - if the ADT representation falls outside the rep invariant, something is wrong (inconsistent rep state).
- The ***abstraction function***
  - maps every concrete states into some abstract states
- The ***operation pre- post- conditions***
  - expressed for the representation type
  - also expressed for the abstract type (for client code)

Example (PSet ADT)

```
class PSet {
// an abstract Pset aset

method new(sz:int) {}
// initializes aset ( e.g., Java constructor )

method add(v:int) {}
// adds v to aset if space available )

function size() : int
// returns number of elems in aset

function maxsize() : int
// returns max number of elems allowed in aset

}
```

Positive Set ADT

- Abstract State
  - a set of positive integers aset
  - aset: set<int> subject to
  - forall x :: x in aset ==> x >= 0

Positive Set ADT

- Representation type
  - an array of integers **store** with sufficient large size
  - an integer nelems counting the elements in **store**

Positive Set ADT

- Representation type
  - an array of integers **store** with sufficient large size
  - an integer nelems counting the elements in **store**
- Representation invariant
  - (store != null) &&
  - (0 <= nelems <= store.length) &&
  - (forall i :: 0 <= i < nelems ==> store[i] >= 0)

Positive Set ADT

- Representation type
  - an array of integers store with sufficient large size
  - an integer nelems counting the elements in store
- Representation invariant
  - (store != null) &&
  - (0 <= nelems <= store.length) &&
  - (forall i :: 0 <= i < nelems ==> store[i] >= 0)
- Abstraction mapping
  - $<\text{nelems}=n, \text{store}=[v_0,v_1,\ldots v_{\text{store.Length-1}}]> \longrightarrow \{v_0,\ldots,v_{n-1}\}$
  - more later ....

Example

```
class PSet {

var store: array<int>;
var nelems: int;

function RepInv():bool  // specify the representation invariant
reads this, store;
{
store != null && (0<= nelems <=store.Length) && (forall i :: 0<=i<nelems ==> store[i]>=0)
}

...

}
```

Example

```
class PSet {

var store: array<int>;
var nelems: int;

...
method newPSet(sz:int)
modifies this;
requires sz>=0;
ensures RepInv();  // The constructor must establish the invariant
{
  store := new int[sz];
  nelems := 0;
}

...

}
```

Example

```
class PSet {

var store: array<int>;
var nelems: int;

...
method add(v:int)
modifies this, store;
requires RepInv() && v >= 0; // all operations must require the representation invariant
ensures RepInv();  // all operations must ensure the representation invariant
{
  if(nelems<store.Length) {
   store[nelems] := v;
   nelems := nelems+1;
  }
}
...
}
```

Example

```
class PSet {

var store: array<int>;
var nelems: int;

...
function size() : int
requires RepInv();
reads this;
{
  nelems
}
...
}
```

Example

```
class PSet {

var store: array<int>;
var nelems: int;

...
function maxsize() : int
requires RepInv();
reads this, store;
{
  store.Length
}
...
}
```

Bank Account ADT

- Abstract State
  - the account balance bal
  - bal: int subject to
  - bal >= 0

Bank Account ADT

- Representation type
  - an integer bal
  - in this simple case the representation type is the same as the abstract type
  - the “meaning” is different
  - we do not want e.g., to multiply bank accounts :-)

Bank Account ADT

- Representation type
  - an integer bal
  - in this simple case the representation type is the same as the abstract type
  - the “meaning” of the rep and abs types are different
  - we do not want e.g., to multiply bank accounts :-)
- Representation invariant
  - (bal >= 0)
  - this time, pretty simple

Example (Account)

```
class Account {
var bal: int;

method Init()
modifies this;
{ bal := 0; }

function getBal(): int
reads this;
{ bal }

method deposit(v:int)
modifies this;
{ bal := bal + v; }

method withdraw(v:int)
modifies this;
{ if (bal>=v) { bal := bal - v; } }

}
```

Example (Account)

```
class Account {
var bal: int;

function RepInv():bool  // specify the representation invariant
reads this ;
{
   bal >= 0
}



}
```

Example (Account)

```
class Account {
var bal: int;

function RepInv():bool  // specify the representation invariant
reads this ;
{
   bal >= 0
}

method newAccount()
modifies this;
ensures RepInv();
{
  bal := 0;
}

...
}
```

Example (Account)

```
class Account {
var bal: int;

function getBal(): int
reads this;
requires RepInv();         // all operations must require the representation invariant
{
   bal
}

method deposit(v:int)
modifies this;
requires RepInv() && (v>=0);
ensures RepInv();          // all operations must ensure the representation invariant
{
  bal := bal + v;
}
...
}
```

Example (Account)

```
class Account {
var bal: int;

...

method withdraw(v:int)
modifies this;
requires RepInv() && (v>=0); // all operations must require the representation invariant
ensures RepInv();            // all operations must ensure the representation invariant
{
  if (bal>=v) { bal := bal - v; }
}

...

}
```

Example (Account)

```
class Account {
var bal: int;

...

method withdraw(v:int)
requires RepInv() && v <= getBal();
ensures  RepInv();
modifies this;
{
  bal := bal - v;
}

...

}
```

Soundness and Abstraction Map

- We have learned how to express the representation invariant and make sure that no unsound states are ever reached
- We have informally argued that the representation state in every case represents the right abstract state, but how to make sure?
- We now see how the correspondence between the representation state and the abstract state can be explicitly expressed in Dafny using ghost variables, specification operations, and abstraction map soundness check.

Soundness and Abstraction Map

- We go back to the (more interesting) PSet ADT
- Recall, we had for representation invariant:

```
function RepInv():bool
reads this, store;
{
  store != null &&
  (0 <= nelems <= store.Length) &&
  (forall i :: 0 <= i < nelems ==> store[i] >= 0)
}
```

Soundness and Abstraction Map

- We now represent the abstract state with a so-called ***ghost variable***.
- A ghost variable is only used in the specification and does not actually use memory at runtime
- Usages of ghost variables only occur in spec operations (not executed at runtime)

```
class PSet {

var store: array<int>; // the representation of the concrete state
var nelems: int;

ghost var aset: set<int>; // the abstract state

}
```

Soundness and Abstraction Map

- We define a Sound() boolean function, a predicate on the abstract and concrete state that specifies the precise relation ship between both:

```
function Sound():bool
reads this, store;
requires RepInv();
{
  forall x:: x in aset <==> (exists i :: 0<=i<nelems && store[i] == x)
}
```

- We now express in operations how the abstract state changes, and **make sure** that it is properly related with the (sound) representation state
- As a benefit we may now also express pre and post conditions in terms of the abstract state

Example (PSet)

```
class PSet {
var store: array<int>; // the representation of the concrete state
var nelems: int;

ghost var aset: set<int>; // the abstract state

function RepInv():bool  // specify the representation invariant
reads this, store;
{
  store != null && (0<= nelems <=store.Length) && (forall i :: 0<=i<nelems ==>
store[i]>=0)
}

function Sound():bool
reads this, store;
requires RepInv();
{
  (forall x:: x in aset <==> exists i :: 0<=i<nelems && store[i] == x)
}
...
}
```

Example (PSet)

```
class PSet {
var store: array<int>; // the representation of the concrete state
var nelems: int;

ghost var aset: set<int>; // the abstract state

...

method Init(s:int)
modifies this;
requires s>=0;
ensures RepInv() && Sound() && aset == {};
{
  store := new int[s];
  nelems := 0;
  aset := {}; // Specification operation, this works like a comment (checked by Dafny)
}

...
}
```

Example (PSet)

```
class PSet {
var store: array<int>; // the representation of the concrete state
var nelems: int;

ghost var aset: set<int>; // the abstract state

...

method add(v:int)
modifies this, store;
requires RepInv() && Sound() && v >= 0 && size()<maxsize();
ensures RepInv() && Sound() && v in aset;
{
   store[nelems] := v; // Implementation code (on the representation)
   nelems := nelems+1;
   aset := aset + {v}; // Specification operation (on the abstract state)
}

...
}
```

Key Points

- Software Design time
  - Abstract Data Type
  - What are the Abstract States / Concrete States?
  - What is the Representation Invariant ?
  - What is the Abstraction Mapping?
- Software Construction time
  - Make sure constructor establishes the Rep Inv
  - Make sure all operations preserve the Rep Inv
    - they may assume the Rep Inv
    - they may require extra pre-conditions (e.g. on op args)
    - they may enforce extra post-conditions
  - Use assertions to make sure your ADT is sound

Further Reading

- **Program Development in Java**, ***Barbara Liskov and John Guttag***, Addison Wesley, 2003, Chapter 5 “Data Abstraction” (other book chapters are also interesting).
- **Programming with abstract data types**, ***Barbara Liskov and Stephen Zilles***, ACM SIGPLAN symposium on Very high level languages, 1974 (read the introductory parts, the rest is already outdated, but the intro is a brilliant motivation to the idea of ADTs). You can access this here: http://dl.acm.org/citation.cfm?id=807045.

Dafny Exercises (1st+2nd Handout)

Instructions:

You have from April 9 (afternoon) to April 22 (23:59) to solve the handout.

You may work in groups of two or one (two is better)

You should send the teaching team your solutions by email to lcaires@fct.unl.pt and carla.ferreira@fct.unl.pt (always to both) by the deadline.

You may email us with questions, we will answer them all, and try to help you about the problem in every matter, except of course giving you the solution.

You may discuss the problem with your colleagues but you cannot of course use code from others or give code to others. Solutions will be randomly selected for discussion, and unability to demonstrate authorship of your work will trigger strict application of the DI FCT UNL ethics code.

Dafny Exercises (1st+2nd Handout)

- This exercise focuses on the development of a small but rigorously 100% bug free dictionary abstract data type (ADT). To make things simple, consider that the type of keys is the set of positive integers and the type of values is the type of integers.
- The ADT must provide the following operations

```
method assoc(k:int,v:int)
    // associates val v to key k in the dictionary
method find(k:int) returns (r:RES)
   // returns NONE if key k is not defined in the dict,
   // or SOME(v) if the dictionary
method delete(k:int)
  // removes any existing association of key k in the dictionary
```

Every dicionary entry should be represented by a record of type

```
datatype ENTRY = PACK(key: int, val: int)
```

The result of function find should be represented with type

```
datatype RES   = NONE | SOME(int)
```

Dafny Exercises (1st+2nd Handout)

You should take into consideration the following:

- The representation type of your ADT should be a mutable data structure (advice: start by using something simple - an array, an ordered array, or a closed hashtable. The work is already tricky using a simple unordered array, so do this first, to get confidence that you will make it.
- Express the representation invariant using an auxiliary boolean function **RepInv()**.

Levels of delivery:

- level 1: full development in Dafny, ensuring that the representation invariant is preserved by all operations. (worth 80%).
- level 2: full development in Dafny, ensuring that the representation invariant is preserved by all operations, and that all operations satisfy the post-conditions expressed in terms of the representation type. For this you are expected to specify the strongest post-conditions you can. (worth 90%).
- level 3: full development in Dafny, ensuring that the representation invariant is preserved by all operations, and that all operations satisfy the post-conditions specified, when expressed in terms of the abstract type; for this you will need to model the abstract state using a ghost variable and define a soundness abstraction mapping predicate **Sound()**. (worth 100%).

Interference, Separation Logic and Verifast for Java

Account ADT (Java)

```
public class Account {

int balance;
// RepInv() = balance >= 0;
public Account()
{
  balance = 0;
}

void deposit(int v)
{
 balance += v;
}

void withdraw(int v)
// requires v >=0;
{
 balance -= v;
}
}
```

Account ADT (Java)

```
public class Account {

int balance;

...

int getBalance()
{
  return balance;
}


static void main (String args[] )
{
      Account b1 = new Account();
      Account b2 = new Account();
      b1.deposit(10);
      // assert: b1.getBalance() == 10
}
}
```

Account ADT (Java)

- Consider the following code fragment and Hoare triple

```
{ v > 0 }
{
   int v1;
   v1 = a1.getBalance();
   if (v1 >= v) {
      a1.withdraw(v);
      a2.deposit(v);
   }
}
{ (old(a1.getBalance) >= v) ==> (a1.getBalance() < old(a1.getBalance())) }
```

- Is this Hoare triple valid?

Account ADT (Java)

- Consider the following code fragment and Hoare triple

```
{ v > 0 }
{
   int v1;
   v1 = a1.getBalance();
   if (v1 >= v) {
      a1.withdraw(v);
      a2.deposit(v);
   }
}
{ (old(a1.getBalance) >= v) ==> (a1.getBalance() < old(a1.getBalance())) }
```

- Only if **`a1`** and **`a2`** refer to different account objects!

Account ADT (Java)

- Tracking aliasing is **very challenging**, e.g.,

```
static void test(Account a1, Account a2, int v)
{
   int v1;
   v1 = a1.getBalance();
   if (v1 >= v) {
      a1.withdraw(v);
      a2.deposit(v);
   }
}

static void main (String args[] )
{
      Account b1 = new Account();
      Account b2 = new Account();
      b1.deposit(10);
      test(b1,b2,2);
      test(b1,b1,2);
}
```

Hoare Logic is unsound for aliasing

- Recall the basic Hoare Logic Rule:

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

- The soundness of this reasoning principle is rooted on the fact that no other variable aliases x. We have:

$$\{ y >= 0 \ \&\&\ x >= 0 \}\ x := -1\ \{ y >= 0 \ \&\&\ x < 0 \}$$

- But, if y and x are aliases, we would have, e.g.

$$\{ y >= 0 \ \&\&\ x >= 0 \}\ x := -1\ \{ y < 0 \ \&\&\ x < 0 \}$$

Aliasing and Interference

- Two programming language expressions are **aliases** if they refer to the same memory location
- Aliasing occurs in any programming language with pointers (e.g., C, C++) or references (Java, C#)
- **Two** program fragments **interfere** if the execution of **one** may change the effect or value of the **other**
- **Interference** in particularly important in the context of concurrent programs (we will see more on this later)
- But **interference** already occurs in **sequential** programs, due to aliasing.

Hoare Logic is unsound for aliasing

- In our Account ADT example:

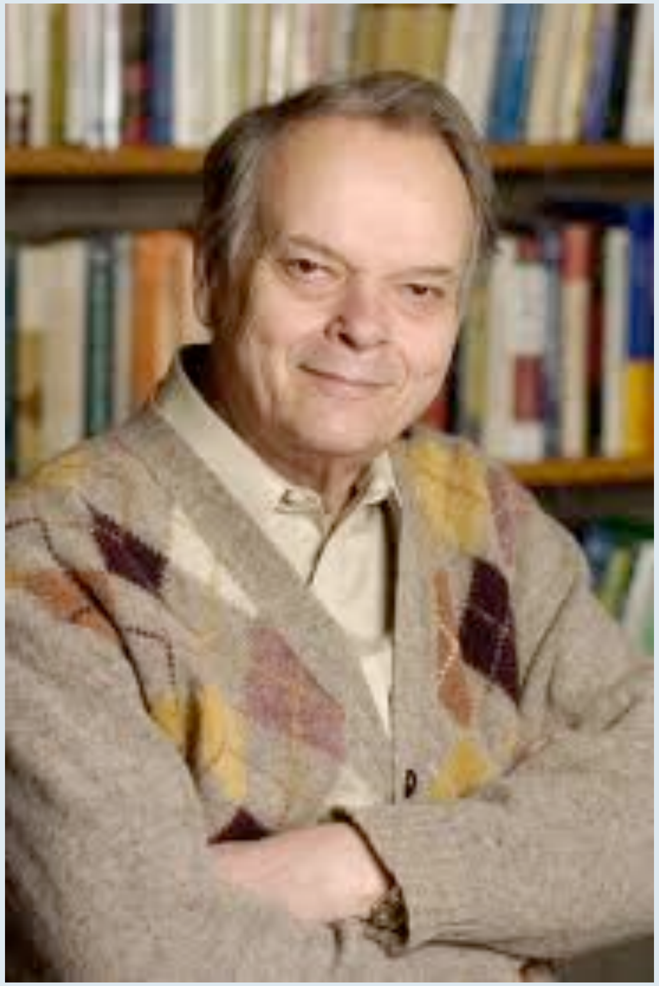
```
{ x.balance()==K && y.balance()>0 }
x.withdraw(K)
{ x.balance()==0 && y.balance()>0 }
```

- Again, if y and x are aliases, we would have, e.g.

```
{ x.balance()==K && y.balance()>0 }
x.withdraw(K)
{ x.balance()==0 && y.balance()==0 }
```

- To reason about programs with interference (aliasing or concurrency) a different approach is needed.

Separation Logic



John C. Reynolds

Peter O'Hearn

Separation Logic

- Separation logic is based in two key principles

  (1) Small footprint

  The precondition of a code fragment describes the part of the memory (heap) that the fragment needs to use.

  (2) Implicit framing

  No need to explicitly specify the properties of state that is changed / not changed by the program (**modifies**)

- It adds to Hoare Logic two key novel primitives

  - the **separating conjunction** operator A * B

  - the "precise" **memory access** assertion L |-> V

Separation Logic

Separation logic assertions used in our CVS course are described by the following grammar:

```
A ::= L |-> V       % memory access
    |  A * A        % separating conjunction
    |  emp          % empty heap
    |  B            % boolean condition (pure, not spatial)
    |  B ? A : A    % conditional
B ::= B && B | B || B | V == V |  V != V | ...
V ::= ...           % expression (pure)
L ::= x.Id          % class object field
```

Separation Logic

the **memory access** assertion L |-> V

Assertion L |-> V holds of the “piece” of the state that consists precisely of memory loc L holding V

the **empty** assertion **emp**

Assertion **emp** holds of the empty heap

the **separating conjunction** A * B

Assertion A * B holds of any “piece” of the state that can be **disjointly** decomposed in a “piece” that satisfies A and another piece that satisfies B.

NOTE: if B is “pure” then A * B <=> A && B

Digression: The Stack and the Heap

Stack

Stores local variables and method parameters

The so-called call-stack, also stores return addresses

Recover discipline: FIFO, release on block exit

Heap

Stores dynamically allocated objects (e.g. **new / malloc**)

Recover discipline:explicit release or garbage collection

Heap Model

A sequence of mem locations (L) and their contents (V)

Separation Logic

pure assertion (we remeber this from Dafny)

an assertion that does not depend on the state (e.g., a boolean expression not mentioning memory accesses).

precise assertion

an assertion that uniquely specifies a concrete part of the memory (unique footprint)

examples:

- no pure assertion is precise
- L |-> V is precise, for a unique value V
- exists o. (o.f |-> 3) is not precise.

Examples (SL / HL)

{ x |-> 2 } x := 4 { x |-> 4 } holds in SL

{ } x := 4 { x |-> 4 } does not hold in SL

{ } x := 4 { x == 4 } holds in HL

{ x |-> 2 * y |-> 3 } x := y { x |-> 3 * y |-> 3 } holds in SL

{ x |-> V * x |-> U } never holds in SL (as an assertion)

{ x == V && x == U } may hold in HL (as an assertion)

Basic Rules of SL

Assignment Rule (SL)

- Recall the basic Hoare Logic Rule:

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

- The assignment rule in separation logic is

{ x |-> V } x := E { x |-> E }

- Note the small footprint principle, the precondition refers exactly to the part of the memory used by the fragment

Frame Rule (SL)

- A key principle in SL is the **Frame Rule**

$$\frac{\{\, A \,\}\; P \;\{\, B \,\}}{\{\, A * C \,\}\; P \;\{\, B * C \}}$$

- This frame rule allows us to preserve info about the "rest of the world", and **locally reason** about the effects of a program that only manipulates a given piece of the state
- The given piece footprint is specified by precondition A

Lookup Rule (SL)

- The memory lookup rule in SL is

{ L |-> V } y := L { L |-> V && y == V}

- Here y is a stack variable, not a heap location L

{ L |-> V } L := L+1 { L |-> V+1}

{ L |-> V } y := L
{ L |-> V && y==V } L := y+1
{ L |-> y+1 && y==V}
{ L |-> V+1 }

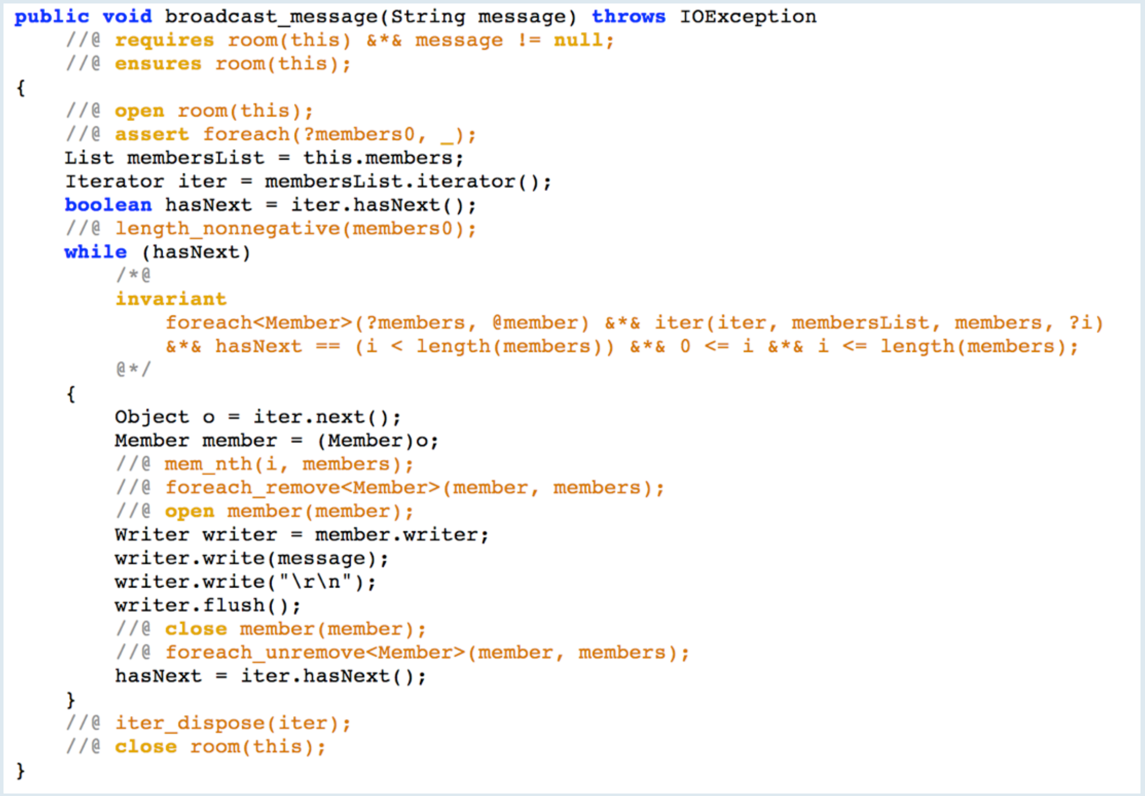
Verifast

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, 2010-14

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);
}
```

Account ADT (Java + Verifast)

```
public class Account {

int balance;

/*@
predicate AccountInv(int b) = this.balance |-> b &*& b >= 0;
@*/

public Account()
//@ requires true;
//@ ensures AccountInv(0);
{
  balance = 0;
}

...

}
```

Account ADT (Java + Verifast)

```
public class Account {

int balance;

...
void deposit(int v)
//@ requires AccountInv(?b) &*& v>=0;
//@ ensures AccountInv(b+v);
{
 balance += v;
}

void withdraw(int v)
//@ requires AccountInv(?b) &*& b >=v;
//@ ensures AccountInv(b-v);
{
 balance -= v;
}
...
}
```

Account ADT (Java + Verifast)

```
public class Account {

int balance;

...
void deposit(int v)
//@ requires AccountInv(?b) &*& v>=0;
//@ ensures AccountInv(b+v);
{
 //@ open AccountInv(_)
 balance += v;
 //@ close AccountInv(_)
}

}
```

- Verifast sometimes requires the programmer to explicitly open and close predicates, if assertions are not precise
- Not needed here, since AccountInv(_) is precise

Account ADT (Java + Verifast)

```
public class Account {

int balance;

...
int getBalance()
//@ requires AccountInv(?b);
//@ ensures AccountInv(b)  &*& result==b ;
{
  return balance;
}

...

}
```

Account ADT (Java + Verifast)