Synthesis, Analysis, and Verification

Lecture 01

Introduction, Overview, Logistics

http://lara.epfl.ch/w/sav15:top

Lectures:
Prof. **Viktor Kuncak**
Prof. **Ondrej Lhotak**
Exercises and Labs:
Etienne Kneuss
Mahsa Taziki

Today

- Introduction and overview of topics
  - Analysis and Verification
  - Synthesis
- Course organization and grading

SAV in One Slide

We study how to build software
analysis, verification, and synthesis
tools that automatically
answer questions about software systems.

We cover *theory* and *tool building* through
lectures, *exercises*, and *labs*.

Grade is based on

- **40%** mid-term exam, 22 April 2015
- **15%** assignments in labs and at home (1st quarter)
- **15%** discussing, reporting on research papers (2nd quarter)
- **30%** mini project, presented in the class (last week)

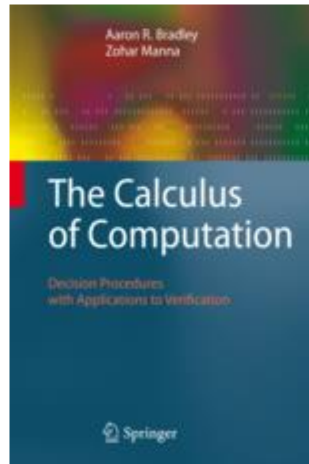
Good Textbook

A. Bradley, Z. Manna:

Calculus of Computation

- Decision Procedures with Applications to Verification

Springer, 2007



Steps in Developing Tools

Modeling: establish precise mathematical meaning for:
software, *environment*, and *questions* of interest

- discrete mathematics, mathematical logic, algebra

Formalization: formalize this meaning using appropriate representation of *programming languages* and *specification languages*

- program semantics, compilers, theory of formal languages, formal methods

Designing algorithms: derive algorithms that manipulate such formal objects - key technical step

- algorithms, dataflow analysis, abstract interpretation, decision procedures, constraint solving (e.g. SAT), theorem proving

Experimental evaluation: implement these algorithms and apply them to software systems

- developing and using tools and infrastructures, learning lessons to improve and repeat previous steps

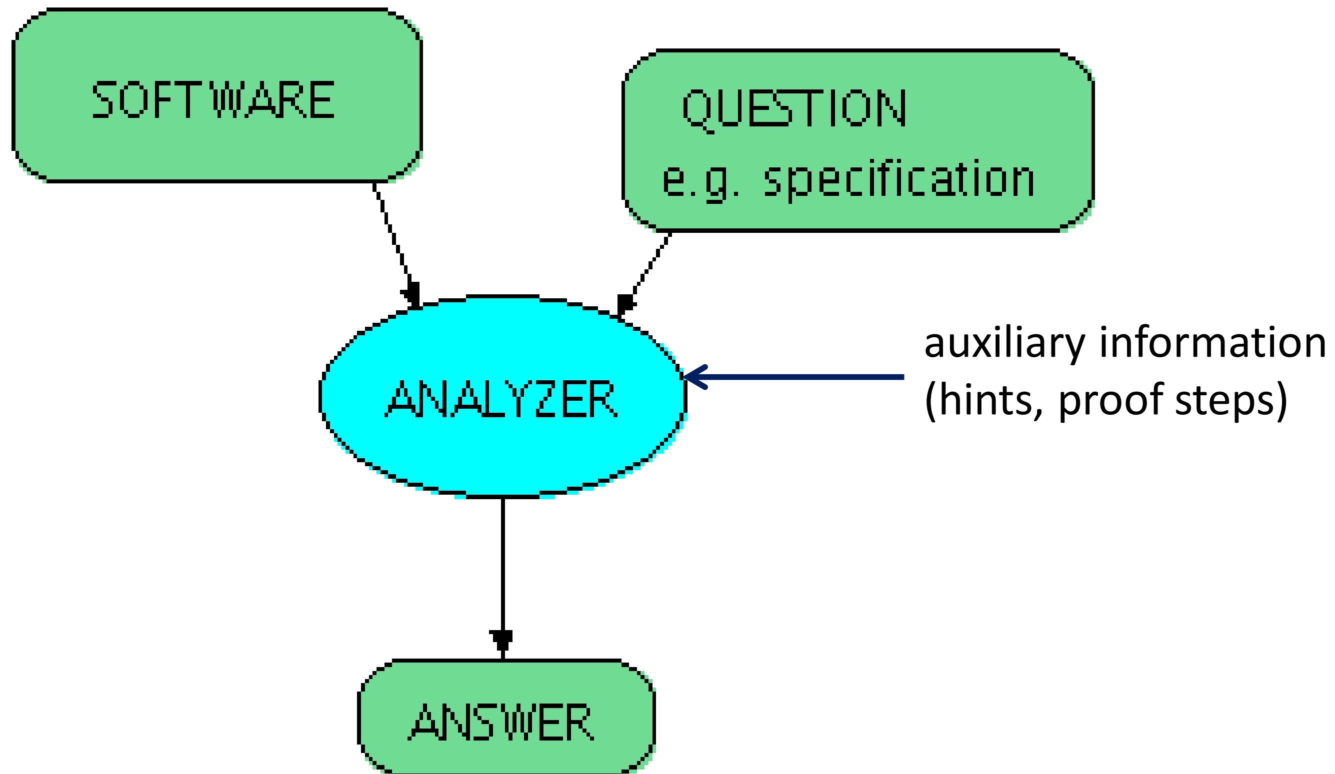
Comparison to other Sciences

Like science we model a part of reality (software systems and their environment) by introducing mathematical models. Models are by necessity *approximations* of reality, because 1) our partial knowledge of the world is partial and
2) too detailed models would become intractable for automated reasoning

Specific to SAV is the nature of software as the subject of study, which has several consequences:

- software is an engineering artifact: to an extent we can choose our reality through **programming language design** and **software methodology**
- software has complex **discrete**, **non-linear** structure: **millions of lines** of code, **gigabytes of bits** of state, one condition in if statement can radically change future execution path (non-continuous behavior)
- high standards of correctness: **interest in details** and exceptional behavior (bugs), not just in general trends of software behavior
- high standards along with large the size of software make manual analysis infeasible in most cases, and requires **automation**
- automation requires not just mathematical modeling, where we use everyday mathematical techniques, but also **formal modeling**, which requires us to specify the representation of systems and properties, making techniques from mathematical logic and model theory relevant
- automation means implementing **algorithms** for processing representation of software (e.g. source code) and representation of properties (e.g. formulas expressing desired properties), the study of these algorithms leads to questions of **decidability**, **computational complexity**, and **heuristics** that work in practice.

Analysis and Verification

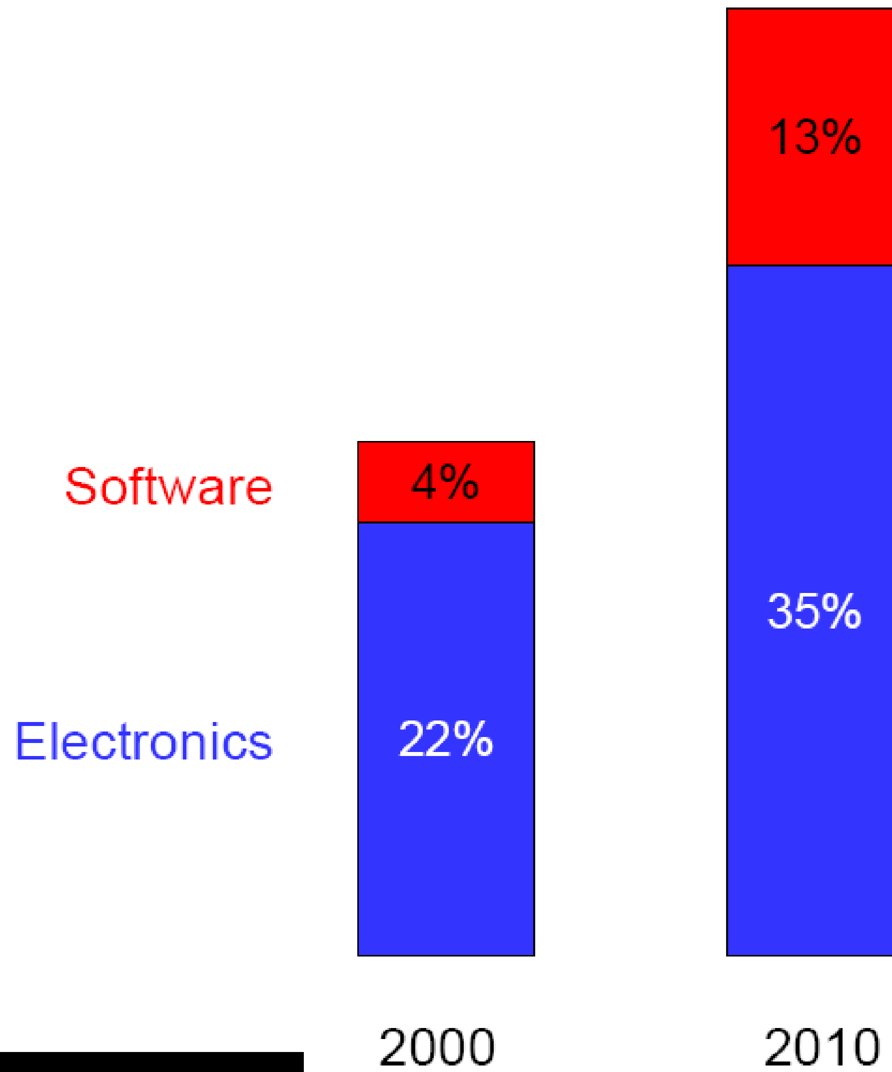


Questions of Interest

Example questions in analysis and verification (with sample links to tools or papers):

- Will the program crash?
- Does it compute the correct result?
- Does it leak private information?
- How long does it take to run?
- How much power does it consume?
- Will it turn off automated cruise control?

Production Cost of Automobiles



[MIT Tech Review]

Car Industry

December 4, 2006

The NHTSA said DaimlerChrysler is recalling 128,000 Pacifica sports utility vehicles because of a problem with the software governing the fuel pump and power train control. The defect could cause the engine to stall unexpectedly.

[Washington Post]

Car Industry

003 / 45 / 7844

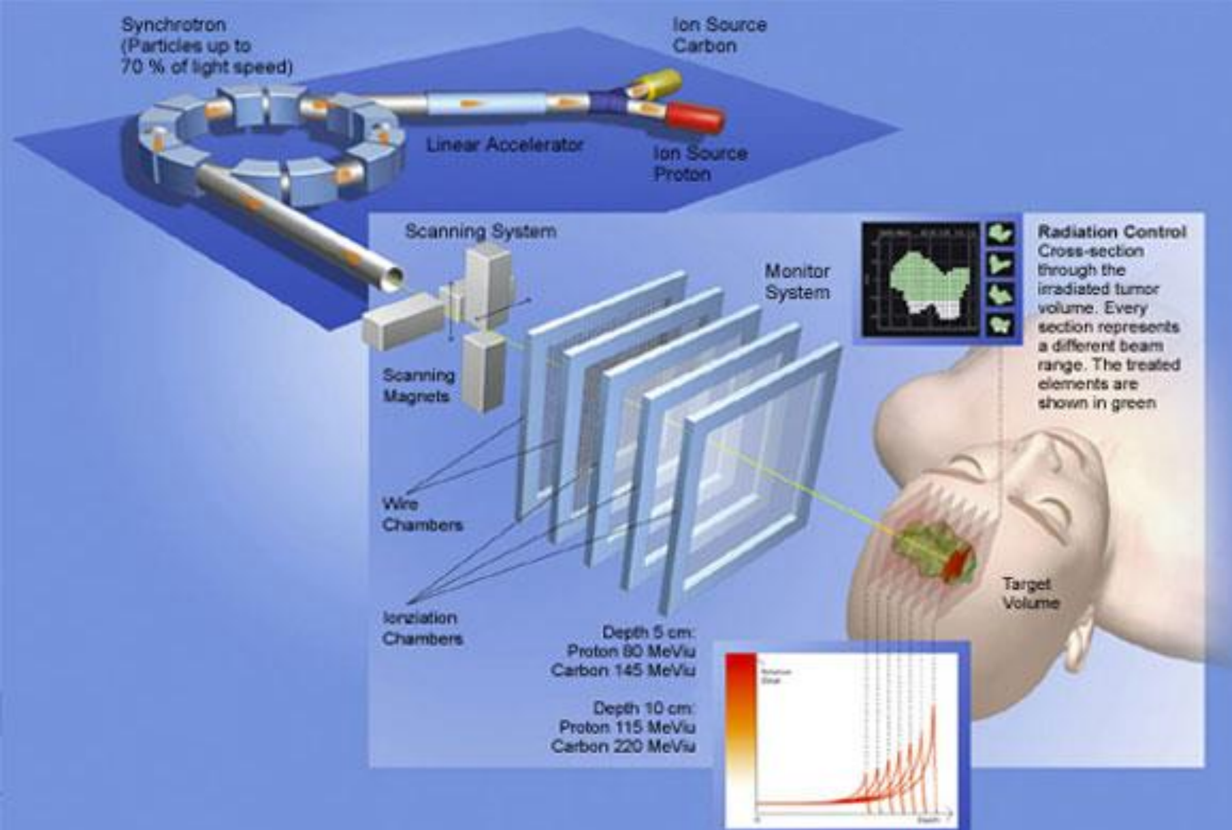
August 14, 2003

A programming error has been identified as the cause of the Northeast power blackout. The failure occurred when multiple computer systems trying to access the same information at once got the equivalent of busy signals.

[Associated Press]

Price tag: $ 6-10 billion

Essential Infrastructure: Northeast Blackout



Radio Therapy

Between June 1985 and January 1987, a computer-controlled radiation therapy machine, called the Therac-25, massively overdosed six people. These accidents have been described as the worst in the 35-year history of medical accelerators [6].

Nancy Leveson
Safeware: System Safety and Computers
Addison-Wesley, 1995

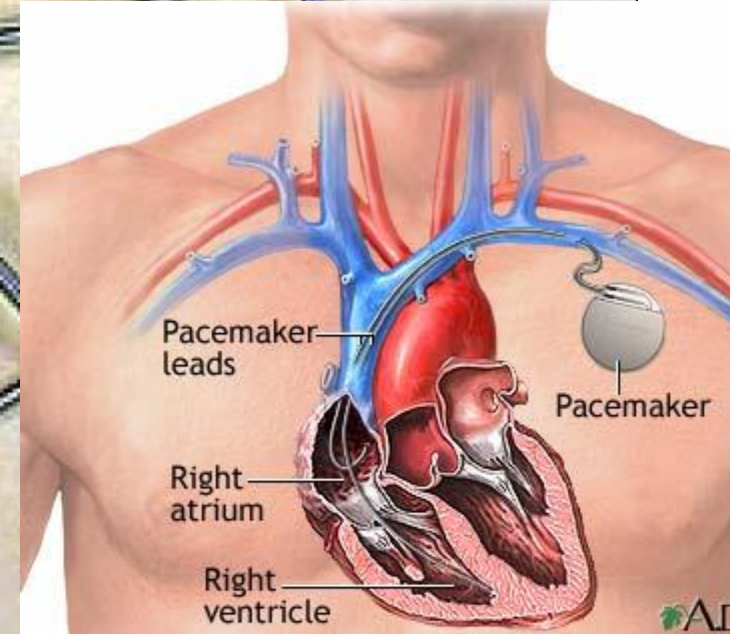
Life-Critical Medical Devices

December 2004

In 1 of every 12,000 settings, the software can cause an error in the programming resulting in the possibility of producing paced rates up to 185 beats/min. It is possible that one or both rate response sensors (i.e., breathing sensor and activity sensor) are switched on, but the timer reset for one or both sensors erroneously remains disabled. In this scenario, the clock timer and the rate response timers can trigger a pace. Of course, with three possible triggers now working independently this can result in high pacing rates.

Axios SR
VVIR/AAIR
unipolar/bipolar

[Journal of Pacing and Clinical Electrophysiology]



Life-Critical Medical Devices

French Guyana, June 4, 1996

t = 0 sec

t = 40 sec

$800 million software failure

Space Missions

Boeing could not assemble and integrate the fly-by-wire system until it solved problems with the databus and the flight management software. Solving these problems took more than a year longer than Boeing anticipated. In April, 1995, the FAA certified the 777 as safe.

Total development cost:	$ 3 billion
Software integration and validation cost:	one third of total

Air Transport

Success Stories

ASTREE Analyzer

"In Nov. 2003, ASTRÉE was able to prove completely automatically the absence of any RTE in the primary flight control software of the Airbus A340 fly-by-wire system, a program of 132,000 lines of C analyzed in 1h20 on a 2.8 GHz 32-bit PC using 300 Mb of memory (and 50mn on a 64-bit AMD Athlon™ 64 using 580 Mb of memory)."

- http://www.astree.ens.fr/

Now maintained by http://www.absint.com/

AbsInt

- 7 April 2005. AbsInt contributes to guaranteeing the safety of the A380, the world's largest passenger aircraft. The Analyzer is able to verify the proper response time of the control software of all components by computing the worst-case execution time (WCET) of all tasks in the flight control software. This analysis is performed on the ground as a critical part of the safety certification of the aircraft.

2014: Synopsis Buys Coverity

Synopsys, Inc. (Nasdaq:SNPS), a global leader providing software, IP and services used to accelerate innovation in chips and electronic systems, and Coverity, the leading provider of software quality, testing, and security tools, today signed a definitive agreement for Synopsys to acquire Coverity. Coverity products reduce the risk of quality and security defects, which can lead to the catastrophic failures that plague many of today's large software systems. …

Under the terms of the definitive agreement, Synopsys will pay approximately **$375 million**, or $350 million net of cash acquired.

…Since spinning out of a Stanford research project 10 years ago, Coverity has been developing revolutionary technology to find and fix defects in software code before it is released, improving software security.

Microsoft's Static Driver Verifier

Static Driver Verifier (SDV) is a thorough, compile-time, static verification tool designed for kernel-mode drivers. SDV finds serious errors that are unlikely to be encountered even in thorough testing. SDV systematically analyzes the source code of Windows drivers that are written in the C language. SDV uses a set of interface rules and a model of the operating system to determine whether the driver interacts properly with the Windows operating system.

…Development teams at Microsoft use SDV to improve the quality of the WDM, KMDF, and NDIS miniport drivers that ship with the operating system and the sample drivers that ship with the Windows Driver Kit (WDK).
SDV is included in the Windows Driver Kit (WDK) and supports all x86-based and x64-based build environments.

Interactive Theorem Provers

- A Mechanically Checked Proof of IEEE Compliance of a Register-Transfer-Level Specification of the AMD K7 Floating Point Multiplication, Division and Square Root Instructions, doine using ACL2 Prover
- Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. by Xavier Leroy

Recommended Reading

- Recent *Research Highlights* from the **Communications of the ACM**
  - A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World
  - Retrospective: An Axiomatic Basis for Computer Programming
  - Model Checking: Algorithmic Verification and Debugging
  - Software Model Checking Takes Off
  - Formal Verification of a Realistic Compiler

  http://video.epfl.ch/2656/1/10
  - seL4: Formal Verification of an Operating-System Kernel

WATCH: Prof. J Moore Lecture

Machines Reasoning about Machines

J Strother Moore, EPFL June 2011

http://slideshot.epfl.ch/play/suri_moore

Impact on Computer Science

Turing award is ACM's most prestigious award and equivalent to Nobel prize in Computing

In the next slides are some papers written by the award winners connected to the topics of this class

- A Basis for a Mathematical Theory of Computation by **John McCarthy**, 1961.

"It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last. The development of this relationship demands a concern for both applications and for mathematical elegance."

- Social processes and proofs of theorems and programs a controversial article by Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis
- Guarded Commands, Nondeterminacy and Formal Derivation of Programs by Edsger W. Dijkstra from 1975, and other Manuscripts
- Simple word problems in universal algebras by D. Knuth and P. Bendix (see Knuth-Bendix_completion_algorithm), used in automated reasoning

- Decidability of second-order theories and automata on infinite trees by **Michael O. Rabin** in 1965, proving decidability for one of the most expressive decidable logics
- Domains for Denotational Semantics by **Dana Scott**, 1982
- Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs by **John Backus**
- Assigning meanings to programs by R. W. **Floyd**, 1967
- The Ideal of Verified Software by **C.A.R. Hoare**
- Soundness and Completeness of an Axiom System for Program Verification by **Stephen A. Cook**
- An Axiomatic Definition of the Programming Language PASCAL by
  C. A. R. Hoare and **Niklaus Wirth**, 1973
- On the Computational Power of Pushdown Automata, by Alfred V. Aho, Jeffrey D. Ullman, **John E. Hopcroft** in 1970
- An Algorithm for Reduction of Operator Strength by **John Cocke**, Ken Kennedy in 1977

- A Metalanguage for Interactive Proof in LCF by Michael J. C. Gordon, **Robin Milner**, L. Morris, Malcolm C. Newey, Christopher P. Wadsworth, 1978
- Proof Rules for the Programming Language Euclid, by Ralph L. London, John V. Guttag, James J. Horning, **Butler W. Lampson**, James G. Mitchell, Gerald J. Popek, 1978
- Computational Complexity and Mathematical Proofs by **J. Hartmanis**
- Software reliability via run-time result-checking by **Manuel Blum**
- The Temporal Logic of Programs, by **Amir Pnueli** (see also the others of a few hundreds of publications)
- No Silver Bullet - Essence and Accidents of Software Engineering, by **Frederick P. Brooks Jr.**, 1987

- Formal Development with ABEL, by **Ole-Johan Dahl** and Olaf Owe
- Abstraction Mechanisms in the Beta Programming Language, by Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, **Kristen Nygaard**, 1983
- Formalization in program development, by **Peter Naur**, 1982
- Interprocedural Data Flow Analysis, by **Frances E. Allen**, 1974
- Counterexample-guided abstraction refinement for symbolic model checking by **Edmund Clarke**, Orna Grumberg, Somesh Jha, Yuan Lu, Helmut Veith, 2003
- Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications by Edmund M. Clarke, **E. Allen Emerson**, A. Prasad Sistla
- The Algorithmic Analysis of Hybrid Systems by Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, **Joseph Sifakis**, Sergio Yovine

How to prove programs

Proving Program Correctness

```
int f(int x, int y)
{
  if (y == 0) {
    return 0;
  } else {
   if (y % 2 == 0) {
     int z = f(x, y / 2);
     return (2 * z);
   } else {
     return (x + f(x, y - 1));
   }
  }
}
```

- What does ‘f’ compute?
- How can we prove it?

By translating Java code into math, we obtain the following mathematical definition of f:

$$f(x,y) = \begin{cases} 0, & \text{if } y = 0 \\ 2f(x, \lfloor \frac{y}{2} \rfloor), & \text{if } y > 0, \text{ and } y = 2k \text{ for some } k \\ x + f(x, y-1), & \text{if } y > 0, \text{ and } y = 2k+1 \text{ for some } k \end{cases}$$

By induction on y we then prove $f(x,y) = x \cdot y$.

- **Base case.** Let $y = 0$. Then $f(x,y) = 0 = x \cdot 0$
- **Inductive hypothesis.** Assume that the claim holds for all values less than y.
  - Goal: show that it holds for y where $y > 0$.
  - **Case 1:** $y = 2k$. Note $k < y$. By definition and I.H.

$$f(x,y) = f(x,2k) = 2f(x,k) = 2(xk) = x(2k) = xy$$

- - **Case 2:** $y = 2k+1$. Note $y - 1 < y$. By definition and I.H.

$$f(x,y) = f(x,2k+1) = x + f(x,2k) = x + x \cdot (2k) = x(2k+1) = xy$$

This completes the proof.

An simple imperative multiplication

```
int fi(int x, int y)
{
   int r = 0;
   int i = 0;
   while (i < y) {
    i = i + 1;
    r = r + x;
   }
   return r;
}
```

- What does ‘fi’ compute?
- How can we prove it?

Preconditions, Postconditions, Invariants

```
void p()
/*: requires Pre
    ensures Post */
{
  s1;
  while /*: invariant 𝒥 */  (e) {
   s2;
  }
  s3;
}
```

Loop Invariant

$\mathcal{I}$ is a loop invariant if the following three conditions hold:

- $\mathcal{I}$ **holds initially**: in all states satisfying *Pre*, when execution reaches loop entry, $\mathcal{I}$ holds
- $\mathcal{I}$ is **preserved**: if we assume $\mathcal{I}$ and loop condition *(e)*, we can prove that $\mathcal{I}$ will hold again after executing *s2*
- $\mathcal{I}$ is **strong enough**: if we assume $\mathcal{I}$ and the negation of loop condition *e,* we can prove that *Post* holds after *s3*

Explanation: because $\mathcal{I}$ holds initially, and it is preserved, by induction from **holds initially** and **preserved** follows that $\mathcal{I}$ will hold in every loop iteration. The **strong enough** condition ensures that when loop terminates, the rest of the program will satisfy the desired postcondition.

Back to our Program: what is Invariant, Precondition, Postcondition

```
int fi(int x, int y)
{
   int r = 0;
   int i = 0;
   while (i < y) {
    i = i + 1;
    r = r + x;
   }
   return r;
}
```

require $(y \geq 0)$

$(r = i * x \wedge i \leq y)$

$I \wedge \neg(i<y) \rightarrow r = x * y$

$i \geq y$

ensuring $(r = x * y)$

- What does 'fi' compute?
- **How can we prove it?**

Conditions We Prove in This Case

```
int fi(int x, int y)
{   require(y >= 0)
    int r = 0;
    int i = 0;
    while (invariant r = i*x  &&  i <= y)
     (i < y) {
      i = i + 1;
      r = r + x;
    }
    return r;
} ensuring (res => res==x*y)
```

x, y – assume read-only

Invariant holds initially:

$$(y \geq 0 \land r = 0 \land i = 0) \rightarrow (r = i \cdot x \land i \leq y)$$

invariant is preserved:

$$(y \geq 0 \land r = i \cdot x \land i \leq y \land i' = i + 1 \land r' = r + x \land i < y) \rightarrow (r' = i' \cdot x \land i' \leq y)$$

invariant implies postcondition:

$$(y \geq 0 \land r = i \cdot x \land i \leq y \land \neg(i < y)) \rightarrow r = x \cdot y$$

First Demo of http://leon.epfl.ch

Task:

Write tail recursive function that does fast multiplication and verify that it does multiplication.

Solution:

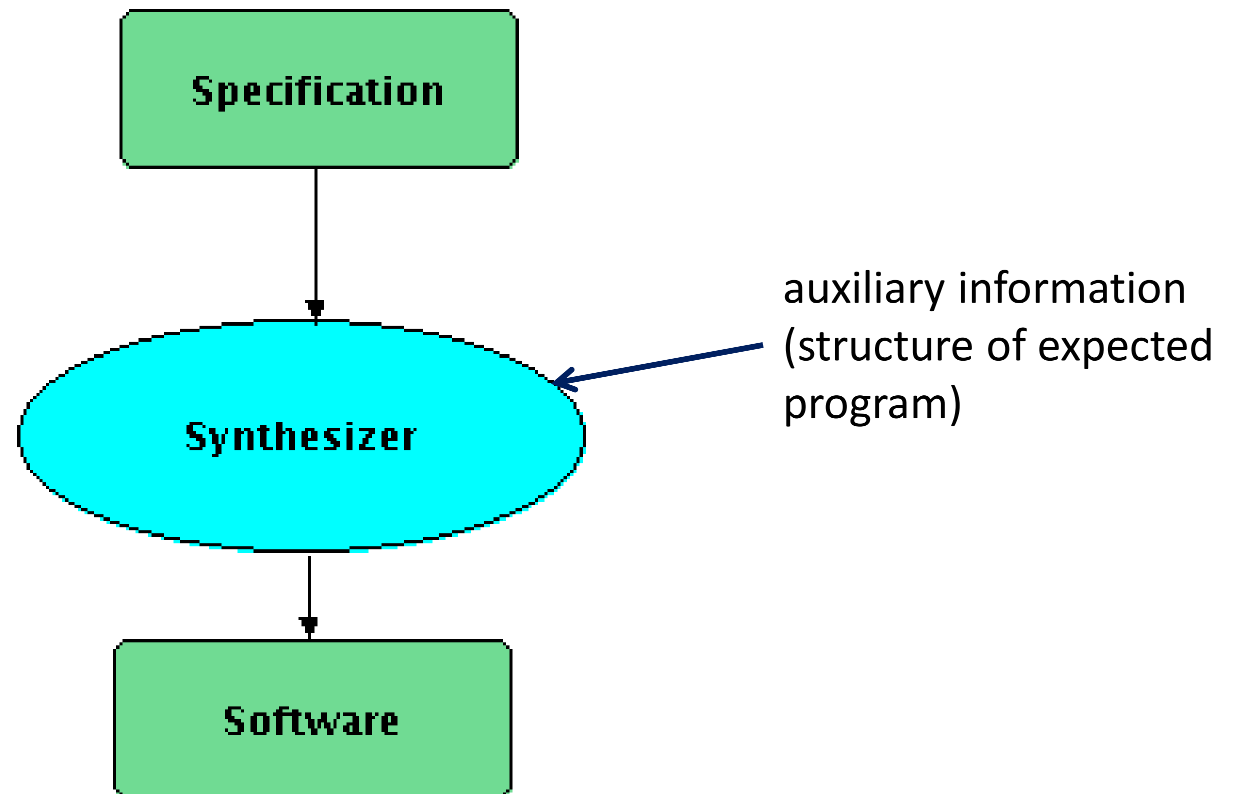
```
def fastmul(p: Int, a: Int, x: Int): Int = {
  require(x >= 0)
  if (x == 0) p
  else if (x % 2 == 0) fastmul(p, a*2, x/2)
  else fastmul(p + a, a*2, x/2)
 } ensuring (res => res == p + a*x)
```

How can we automate verification?

Important algorithmic questions:

- **verification condition generation**: compute formulas expressing program correctness
  - Hoare logic, weakest precondition, strongest postcondition
- theorem proving: prove verification conditions
  - proof search, counterexample search
  - decision procedures
- **loop invariant inference**
  - predicate abstraction
  - abstract interpretation and data-flow analysis
  - pointer analysis, typestate
- reasoning about numerical computation
- pre-condition and post-condition inference
- ranking error reports and warnings
- finding error causes from counterexample traces

Synthesis



Tasks of Interest (i: input, o: output)

both specification C and program p are given:

a) Check assertion while program p runs: $C(i,p(i))$

b) Verify whether program always meets the spec: $\forall i.\ C(i,p(i))$

only specification C is given:

c) Constraint programming: once **i** is known, find **o** to satisfy a given constraint: **find o such that $C(i,o)$**

d) Synthesis: solve C symbolically to obtain program p that is correct by construction, for all inputs: **find p such that $\forall i.C(i,p(i))$ i.e. $p \subseteq C$**

run-time	**compile-time**

Sorting Demo

http://leon.epfl.ch

Recursion Schemas + STE in Action

```
def delete(in1: List, v: Int) = choose {
  (out: List) => content(out) == content(in1) -- Set(v)
}

def delete(in1: List,  v: Int)  = {
  def rec(in: List, v: Int): List = in match {
    case Cons(h,t) =>
      val r = rec(t,v)
      if (h == v) {
CEGIS   r
      } else {
CEGIS   Cons(h, r)
      }
    case Nil =>
CEGIS     Nil
  } ensuring { content(_) == content(in1) -- Set(v) }
  rec(in1, v)
}
```

ADT Induction

EQ Split

Synthesizing Code from Free-Form Queries

```
public class Utils {
    public void backupFile(String fname) {
        String bname = fname+".bak";

    }
}
```

copy file fname to bname

- FileUtils.copyFile(new File(fname), new File(bname))
- FileUtils.copyFile(new File(bname), new File(fname))
- FileUtils.copyFileToDirectory(new File(fname), new File(bname))
- FileUtils.copyFileToDirectory(new File(bname), new File(fname))
- FileUtils.copyFile(<arg>, new File(fname))

Tihomir Gvero

Text → Parsing → Word Groups → Declaration Search → Declarations → Synthesis → Expressions

Online:
- Related WordMap → Parsing
- Word-Declaration Map → Declaration Search
- Unigram Declaration Model → Declaration Search
- PCFG Model → Synthesis

Offline:
- Map Builder → Related WordMap
- Declaration Parser → Word-Declaration Map
- Model Extractor → Unigram Declaration Model, PCFG Model
- WordNet → Map Builder
- API → Map Builder, Declaration Parser, Model Extractor
- Java Corpus from GitHub → Model Extractor

WATCH: Synthesis from Examples

Sumit Gulwani: *Automating String Processing in Spreadsheets using Input-Output Examples*

- Proceedings of the 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages
- http://dx.doi.org/10.1145/1925844.1926423
- **VIDEO:**

http://dl.acm.org/ft_gateway.cfm?id=1926423&ftid=978159&dwn=1&CFID=627723382&CFTOKEN=42173189