

Synthesis, Analysis, and Verification (SAV)

Lecture 01

<http://lara.epfl.ch/w/sav>

Lectures:

Viktor Kuncak

Exercises and Labs:

Eva Darulová

Giuliano Losa



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

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.

The grading is based on:

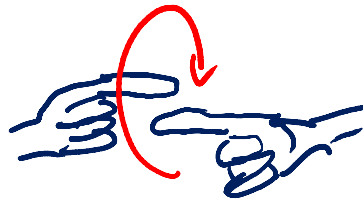
- short theoretical (bi)weekly homework
- fixed programming project, done in stages
- midterm (in the second half of the semester)
- personalized project, with writing code (or new proofs), presentation and report

Suggestion

- Attend all 3 weekly slots
- Always bring a laptop
- Ask questions

- Speed control gestures

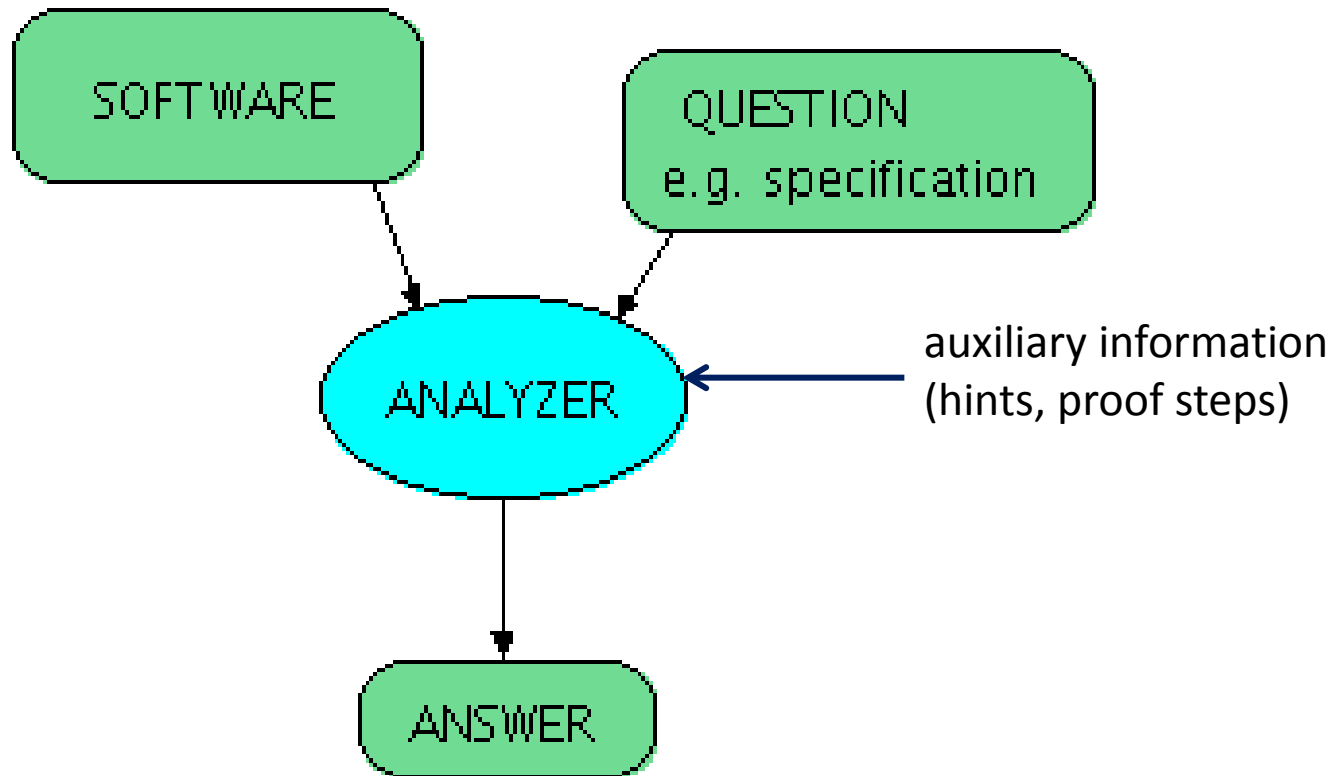
- Fast forward



- Slow down



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?](#)

Activities and Expertise Needed

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 analysis, 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

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.



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

August 2005



Gerardo Dominguez/zrh.airlinerpictures.net

As a Malaysia Airlines jetliner cruised from Perth, Australia, to Kuala Lumpur, Malaysia, one evening last August, it suddenly took on a mind of its own and zoomed 3,000 feet upward. The captain disconnected the autopilot and pointed the Boeing 777's nose down to avoid stalling, but was jerked into a steep dive. He throttled back sharply on both engines, trying to slow the plane.

Instead, the jet raced into another climb. The crew eventually regained control and manually flew their 177 passengers safely back to Australia.

Investigators quickly discovered the reason for the plane's roller-coaster ride 38,000 feet above the Indian Ocean. A defective software program had provided incorrect data about the aircraft's speed and acceleration, confusing flight computers.

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

September 14, 2004

Without warning, at about 5 p.m. PDT, air traffic controllers lost contact with about 400 airplanes they were tracking over the southwestern US. A backup system that was supposed to take over in such an event crashed within a minute after it was turned on.



French Guyana, June 4, 1996

$t = 0$ sec



Space Missions

$t = 40$ sec

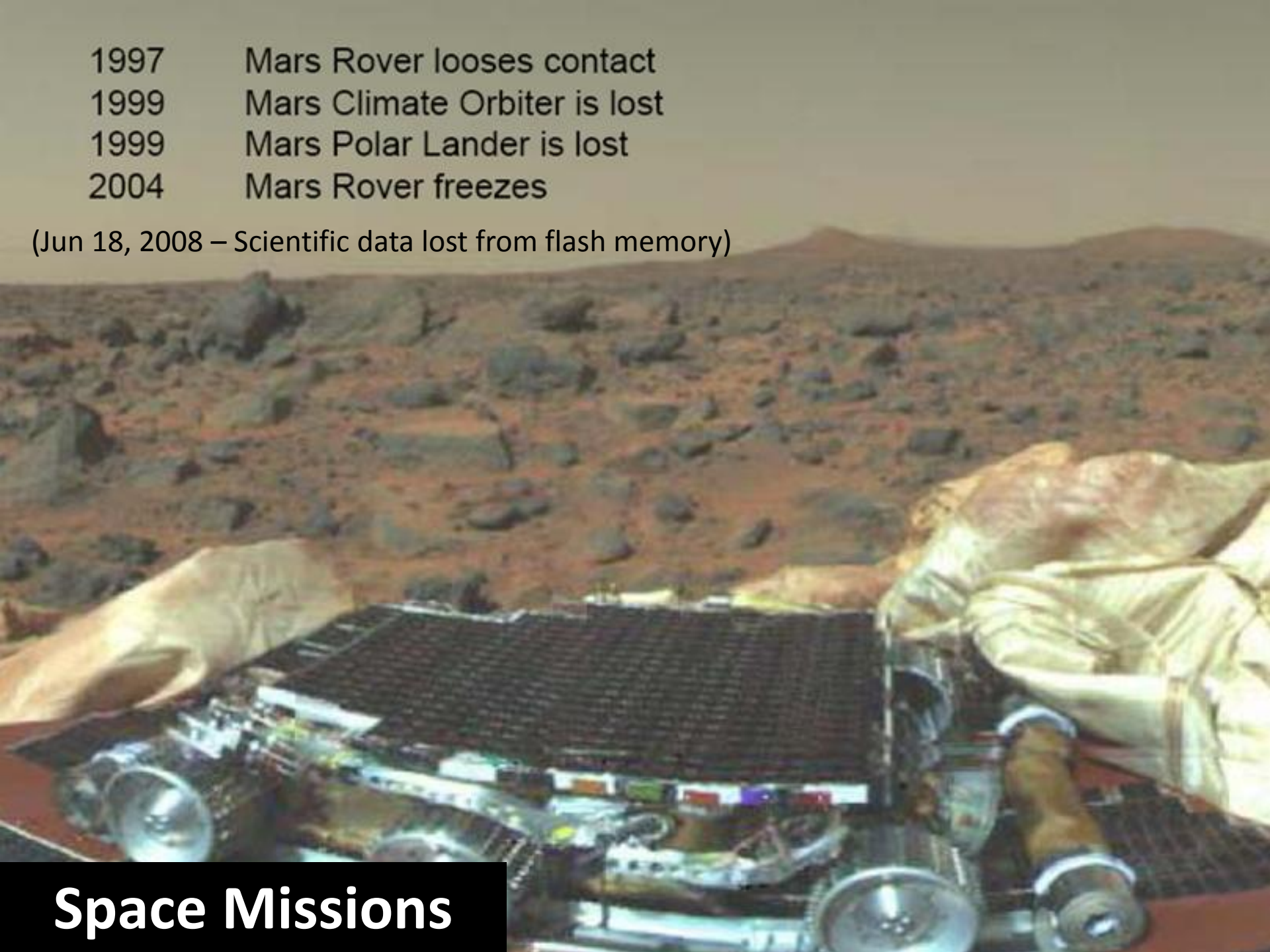
\$800 million software failure



1997 Mars Rover loses contact
1999 Mars Climate Orbiter is lost
1999 Mars Polar Lander is lost
2004 Mars Rover freezes

(Jun 18, 2008 – Scientific data lost from flash memory)

Space Missions



December 4, 2006

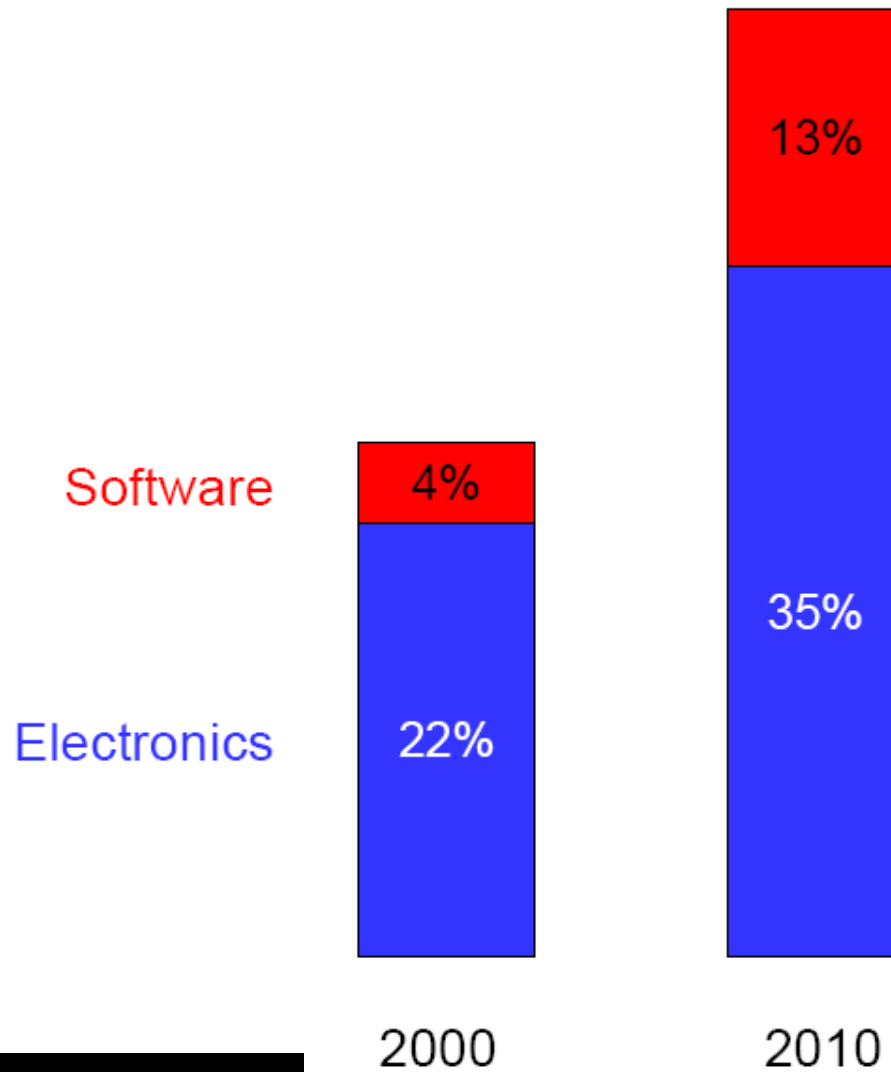
The NHTSA said DaimlerChrysler is recalling 128,000 Pacifica sports utility vehicles because of a problem with the software governing the fuel pump and power train control. The defect could cause the engine to stall unexpectedly.

[Washington Post]

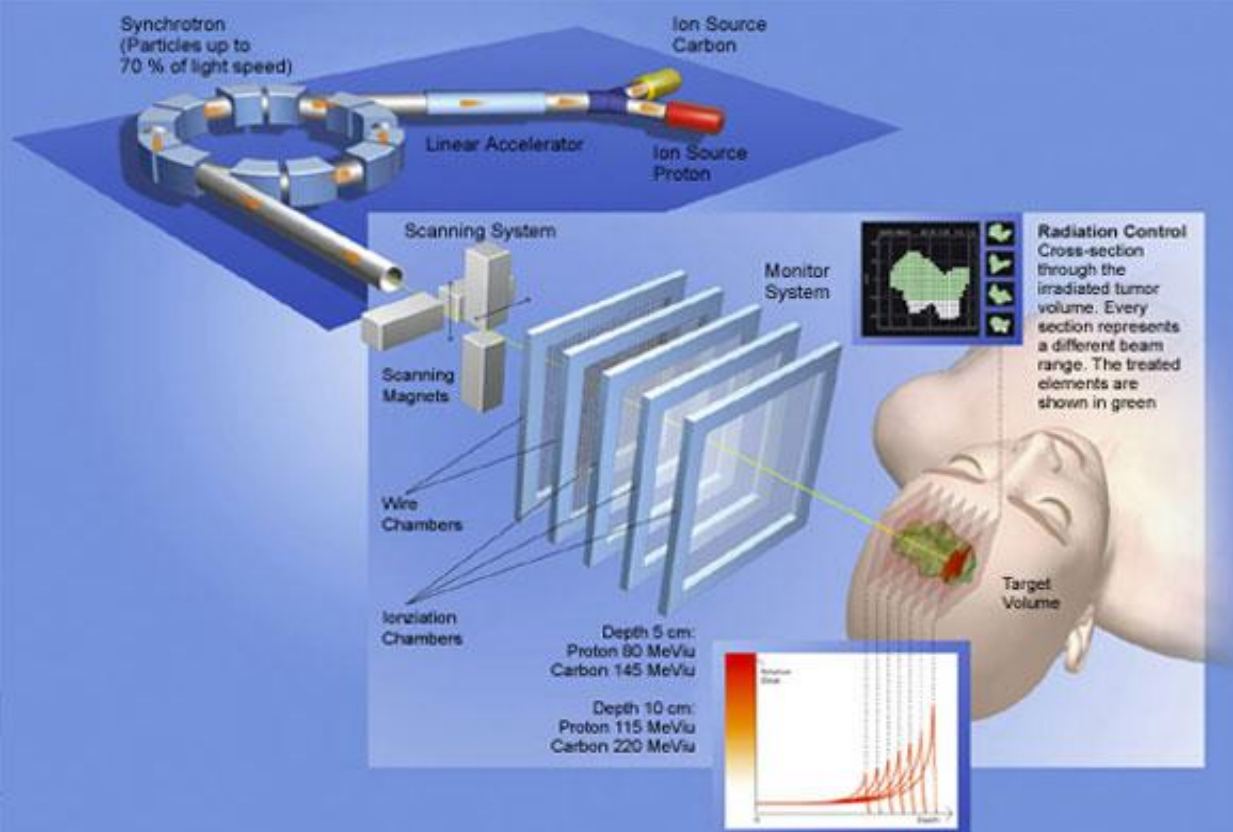


Car Industry

Production Cost of Automobiles



Radio Therapy

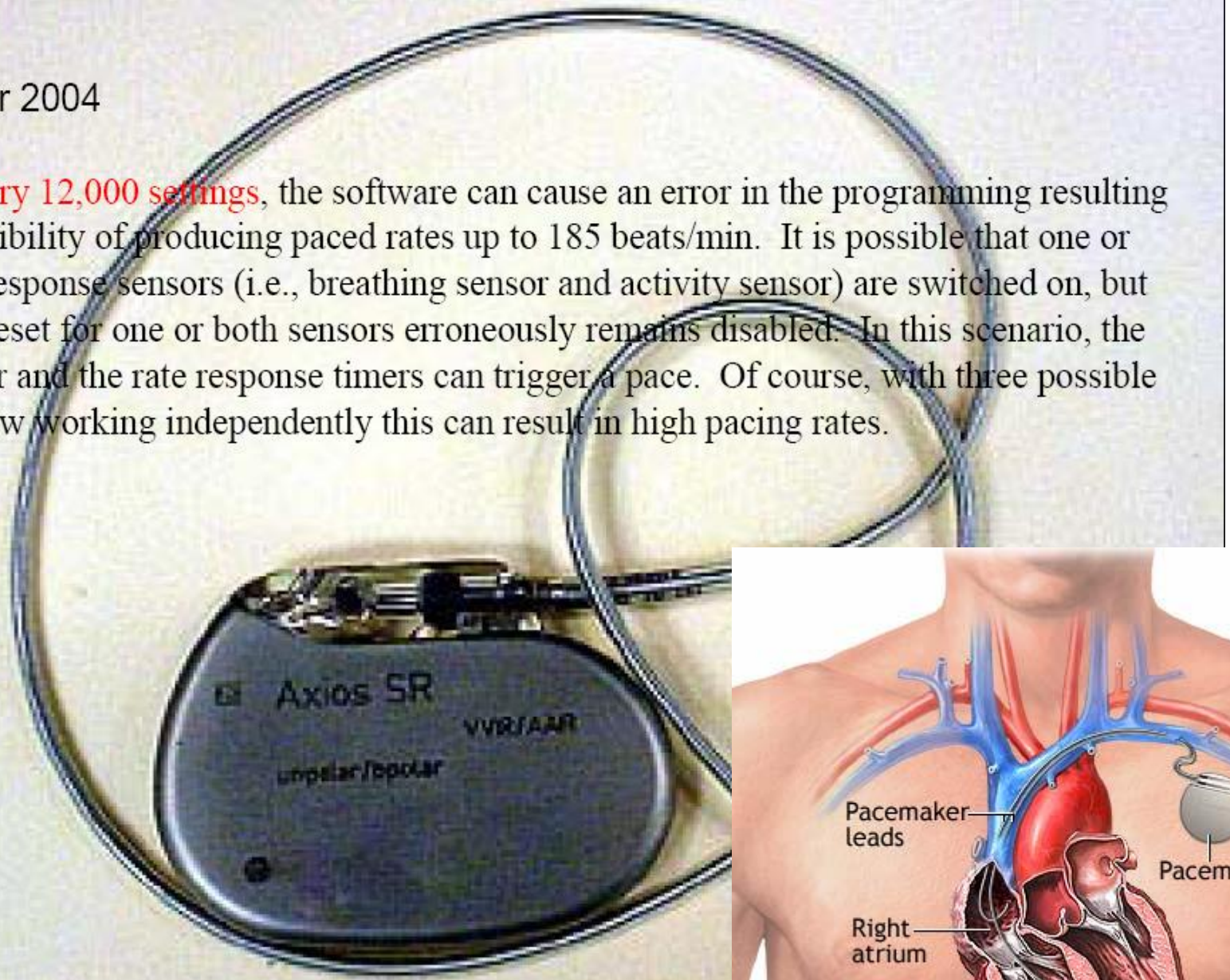


Between June 1985 and January 1987, a computer-controlled radiation therapy machine, called the Therac-25, massively overdosed six people. These accidents have been described as the worst in the 35-year history of medical accelerators [6].

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

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.



[Journal of Pacing and Clinical Electrophysiology]

Life-Critical Medical Devices

Zune 30 leapyear problem

- December 31, 2008
- “After doing some poking around in the [source code for the Zune’s clock driver](#) (available free from the Freescale website), I found the root cause of the now-infamous Zune 30 leapyear issue that struck everyone on New Year’s Eve. The Zune’s real-time clock stores the time in terms of days and seconds since January 1st, 1980. When the Zune’s clock is accessed, the driver turns the number of days into years/months/days and the number of seconds into hours/minutes/seconds. Likewise, when the clock is set, the driver does the opposite.
- The Zune frontend first accesses the clock toward the end of the boot sequence. Doing this triggers the code that reads the clock and converts it to a date and time...”
- “...The function keeps subtracting either 365 or 366 until it gets down to less than a year’s worth of days, which it then turns into the month and day of month. Thing is, in the case of the last day of a leap year, it keeps going until it hits 366. Thanks to the if (days > 366), it stops subtracting anything if the loop happens to be on a leap year. But 366 is too large to break out of the main loop, meaning that the Zune keeps looping forever and doesn’t do anything else.”

<http://www.zuneboards.com/forums/zune-news/38143-cause-zune-30-leapyear-problem-isolated.html>

More Information

<http://mtc.epfl.ch/~tah/Lectures/EPFL-Inaugural-Dec06.pdf>

<http://www.cse.lehigh.edu/~gtan/bug/softwarebug.html>

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/>

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.

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](#), done using [ACL2 Prover](#)
- [Formal certification of a compiler back-end, or: programming a compiler with a proof assistant.](#) by Xavier Leroy

Coverity Prevent

- SAN FRANCISCO - January 8, 2008 - Coverity®, Inc., the leader in improving software quality and security, today announced that as a result of its contract with US Department of Homeland Security (DHS), **potential security and quality defects** in 11 popular open source software projects were **identified and fixed**. The 11 projects are **Amanda, NTP, OpenPAM, OpenVPN, Overdose, Perl, PHP, Postfix, Python, Samba, and TCL**.

Microsoft's Static Driver Verifier

Static Driver Verifier (SDV) is a thorough, compile-time, static verification tool designed for kernel-mode drivers.

SDV is included in the [Windows Driver Kit \(WDK\)](#)

SDV systematically analyzes the source code of Windows drivers that are written in the C language.

SDV finds serious errors that are unlikely to be encountered even in thorough testing.

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.

Recognition by Turing Awards (illustrates the importance)

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 correct

Proving Program Correctness

```
def f(x : Int, y : Int) : Int
{
  if (y == 0)
    0
  } else {
    if (y % 2 == 0) {
      val z = f(x, y / 2);
      2*z
    } else {
      x + f(x, y - 1)
    }
  }
}
```

- What does 'f' compute?
- How can we prove it?

Proving Program Correctness

```
def f(x : Int, y : Int) : Int
{ require(y >= 0)
  if (y == 0)
    0
  else {
    if (y % 2 == 0) {
      val z = f(x, y / 2);
      2*z
    } else {
      x + f(x, y - 1)
    }
  }
}
ensuring (result => result == x * y)
```

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 imperative version

```
def fi(x : Int, y : Int) : Int
{
  val r : Int = 0
  val i : Int = 0
  while (i < y) {
    i = i + 1
    r = r + x
  }
  r
}
```

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

An imperative version

```
def fi(x : Int, y : Int) : Int
{ require (y >= 0)
  val r : Int = 0
  val k : Int = 0
  while invariant (r = x * k && k <= x)
    (k < y) {
    k = k + 1
    r = r + x
  }
  r
} ensuring (res => res == x * y)
```

Preconditions, Postconditions, Invariants

```
void p()  
/*: requires Pre  
   ensures Post */  
{  
  s1;  
  while /*: invariant  $\mathcal{I}$  */ (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.

Membership in Binary Search Tree

```
sealed abstract class BST {  
  def contains(key: Int): Boolean = (this : BST) match {  
    case Node(left: BST,value: Int, _) if key < value => left.contains(key)  
    case Node(_,value: Int, right: BST) if key > value => right.contains(key)  
    case Node(_,value: Int, _) if key == value => true  
    case e : Empty => false  
  }  
}  
  
case class Empty extends BST  
case class Node(val left: BST, val value: Int, val right: BST) extends BST
```

Leon verifier:

<http://lara.epfl.ch/leon/>

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, tpestate
- reasoning about numerical computation
- pre-condition and post-condition inference
- ranking error reports and warnings
- finding error causes from counterexample traces

Bubbling up an Element in Bubble Sort

```
int apartmentRents[];  
int grades[];  
...  
void bubbleUp(int[] a, int from)  
{  
    int i = from;  
    while (i < a.length) {  
  
  
  
  
  
  
    }  
}
```

Proving increasingly stronger properties:

- array indices are within bounds
- also that the element in `a[from]` is smaller than those stored after ‘from’
- also the property sufficient to prove correctness of bubble sort

Recommended Reading

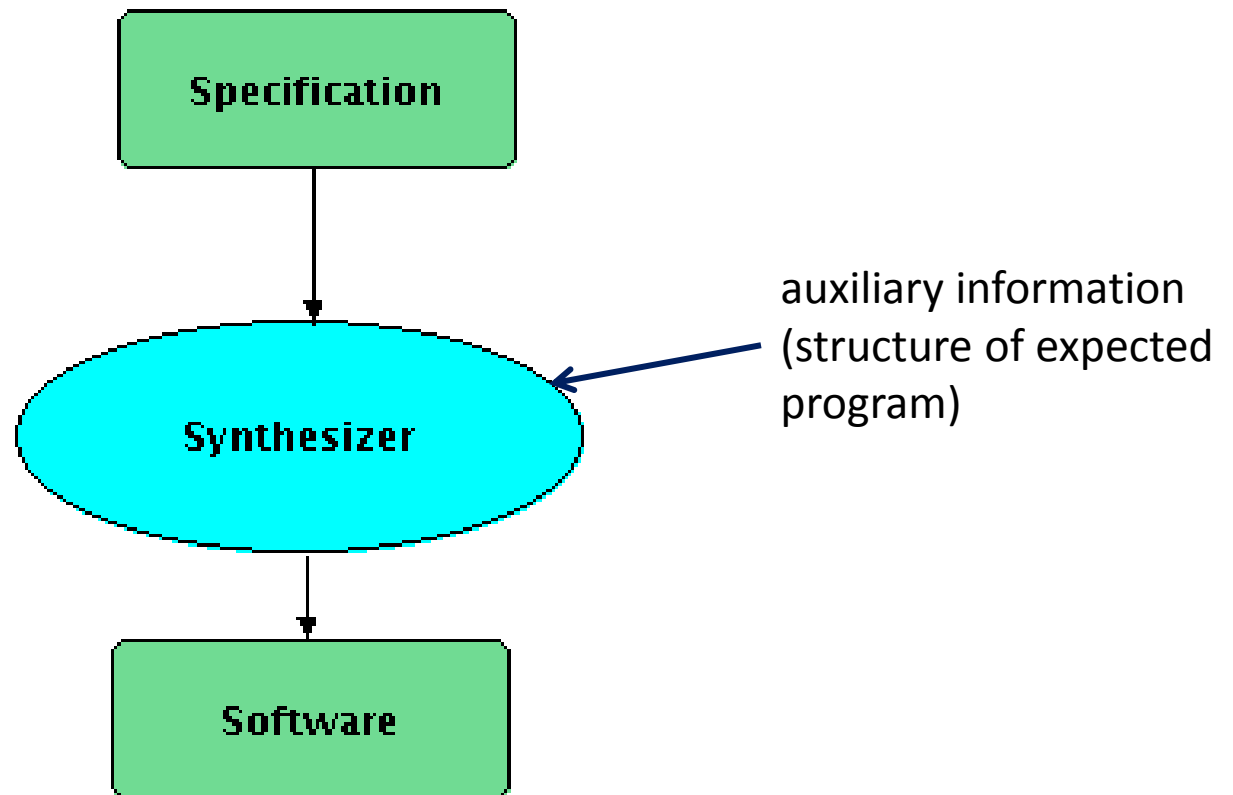
- Recent *Research Highlights* from the **Communications of the ACM**
 - [A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World](#)

A Great Video

Talk by Professor J Strother Moore

[http://slideshot.epfl.ch/play/suri moore](http://slideshot.epfl.ch/play/suri_moore)

Synthesis

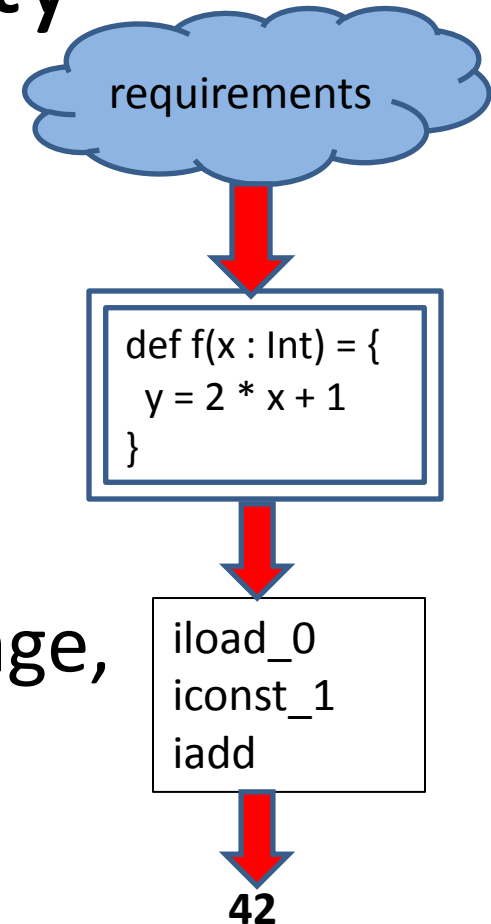


Programming Activity

Consider three related activities:

- Development within an IDE
(Eclipse, Visual Studio, emacs, vim)
- Compilation and static checking
(optimizing compiler for the language,
static analyzer, contract checker)
- Execution on a (virtual) machine

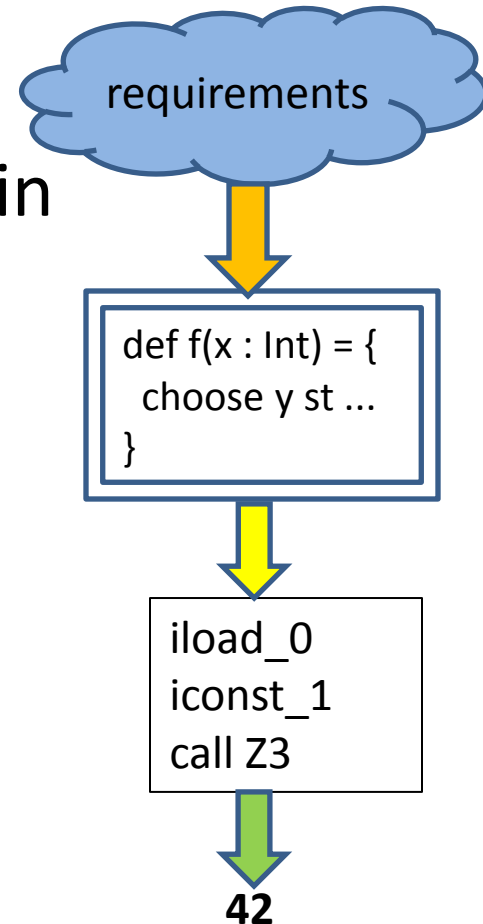
More compute power available for each of these
→ use it to improve programmer productivity



Synthesis at All Levels

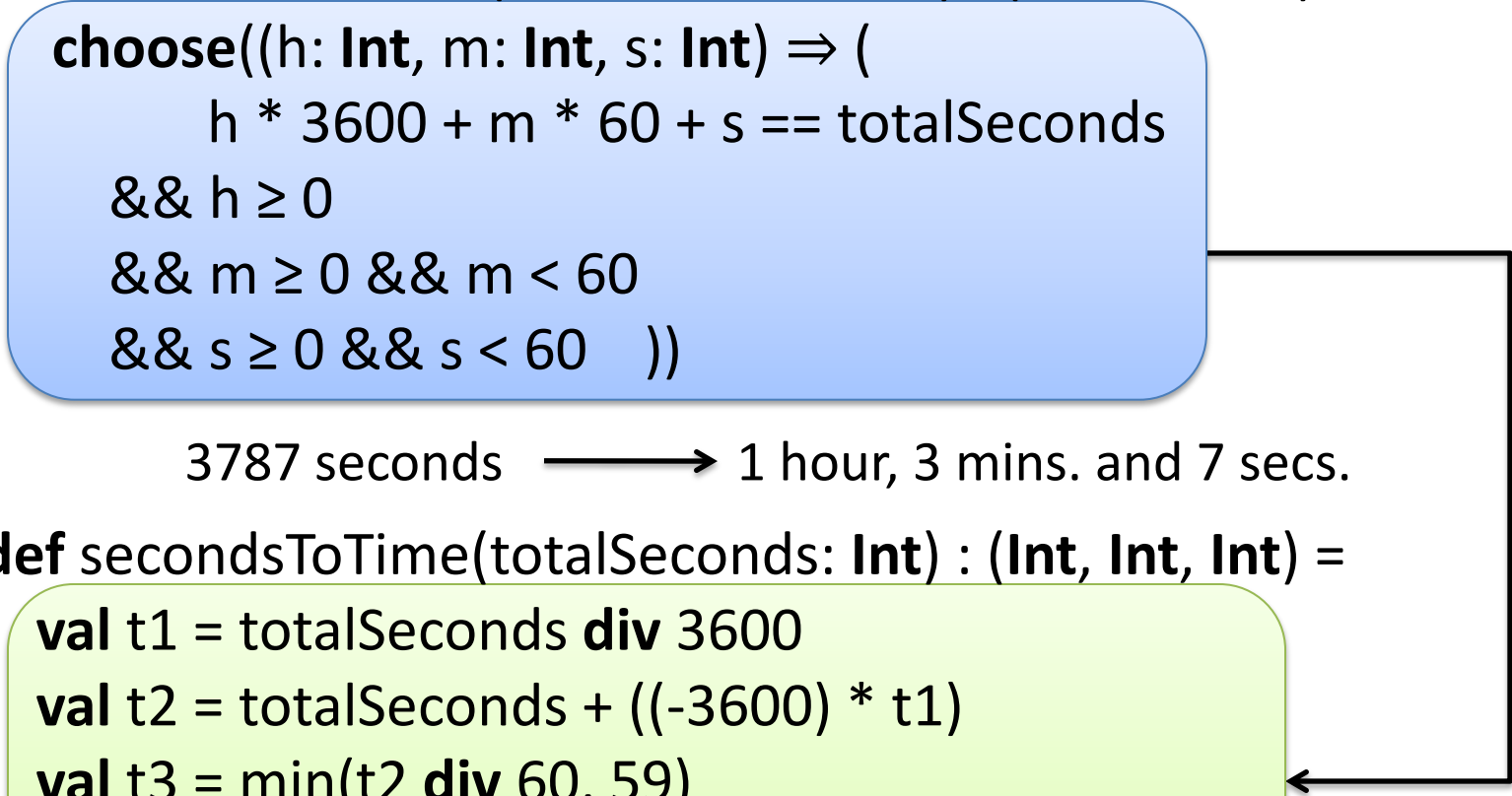
Opportunities for implicit programming in

- **Development** within an IDE
 - **isynth** tool
- ➔ • **Compilation**
 - **Comfusy** and **RegSy** tools
- **Execution**
 - **Scala^{Z3}** and **UDITA** tools



An example

```
def secondsToTime(totalSeconds: Int) : (Int, Int, Int) =  
  choose((h: Int, m: Int, s: Int)  $\Rightarrow$  (  
    h * 3600 + m * 60 + s == totalSeconds  
    && h  $\geq$  0  
    && m  $\geq$  0 && m < 60  
    && s  $\geq$  0 && s < 60  ))
```



3787 seconds \longrightarrow 1 hour, 3 mins. and 7 secs.

```
def secondsToTime(totalSeconds: Int) : (Int, Int, Int) =  
  val t1 = totalSeconds div 3600  
  val t2 = totalSeconds + ((-3600) * t1)  
  val t3 = min(t2 div 60, 59)  
  val t4 = totalSeconds + ((-3600) * t1) + (-60 * t3)  
  (t1, t3, t4)
```

Compile-time warnings

```
def secondsToTime(totalSeconds: Int) : (Int, Int, Int) =  
  choose((h: Int, m: Int, s: Int)  $\Rightarrow$  (  
    h * 3600 + m * 60 + s == totalSeconds  
    && h  $\geq$  0 && h < 24  
    && m  $\geq$  0 && m < 60  
    && s  $\geq$  0 && s < 60  
  ))
```

Warning: Synthesis predicate is not
satisfiable for variable assignment:
totalSeconds = 86400

Compile-time warnings

```
def secondsToTime(totalSeconds: Int) : (Int, Int, Int) =  
  choose((h: Int, m: Int, s: Int)  $\Rightarrow$  (  
    h * 3600 + m * 60 + s == totalSeconds  
    && h  $\geq$  0  
    && m  $\geq$  0 && m  $\leq$  60  
    && s  $\geq$  0 && s < 60  
  ))
```

Warning: Synthesis predicate has multiple solutions for variable assignment:

totalSeconds = 60

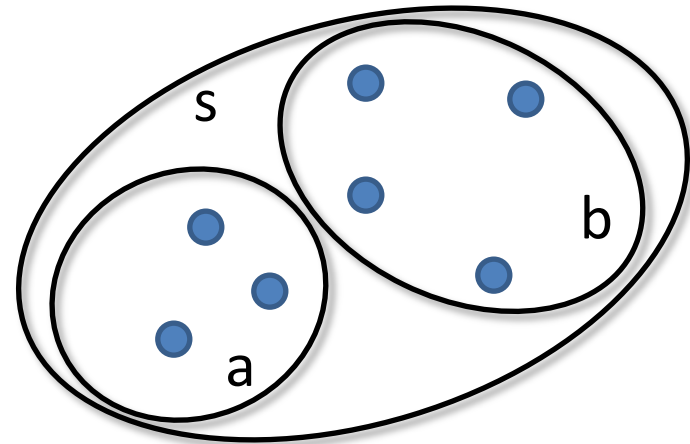
Solution 1: h = 0, m = 0, s = 60

Solution 2: h = 0, m = 1, s = 0

Synthesis for sets

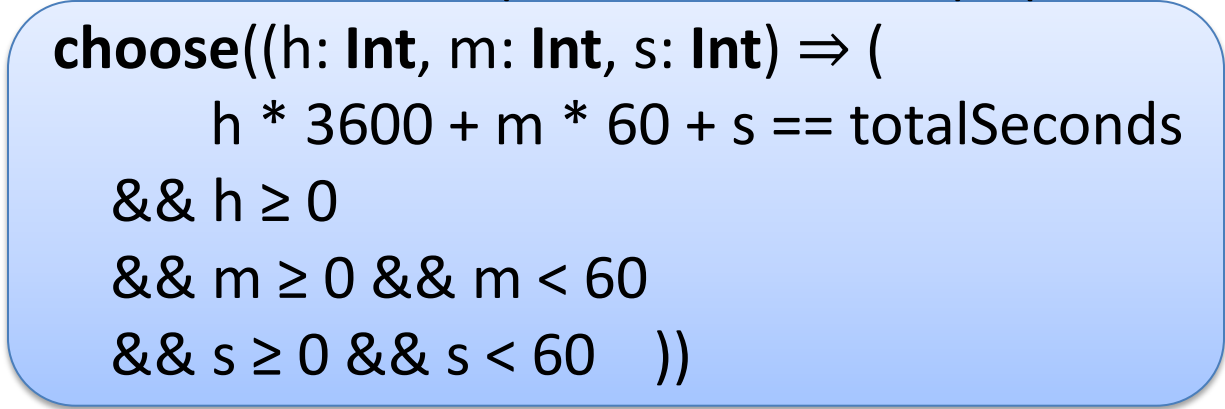
```
def splitBalanced[T](s: Set[T]) : (Set[T], Set[T]) =  
  choose((a: Set[T], b: Set[T])  $\Rightarrow$  (  
    a union b == s && a intersect b == empty  
    && a.size - b.size  $\leq$  1  
    && b.size - a.size  $\leq$  1  
  ))
```

```
def splitBalanced[T](s: Set[T]) : (Set[T], Set[T]) =  
  val k = ((s.size + 1)/2).floor  
  val t1 = k  
  val t2 = s.size - k  
  val s1 = take(t1, s)  
  val s2 = take(t2, s minus s1)  
  (s1, s2)
```



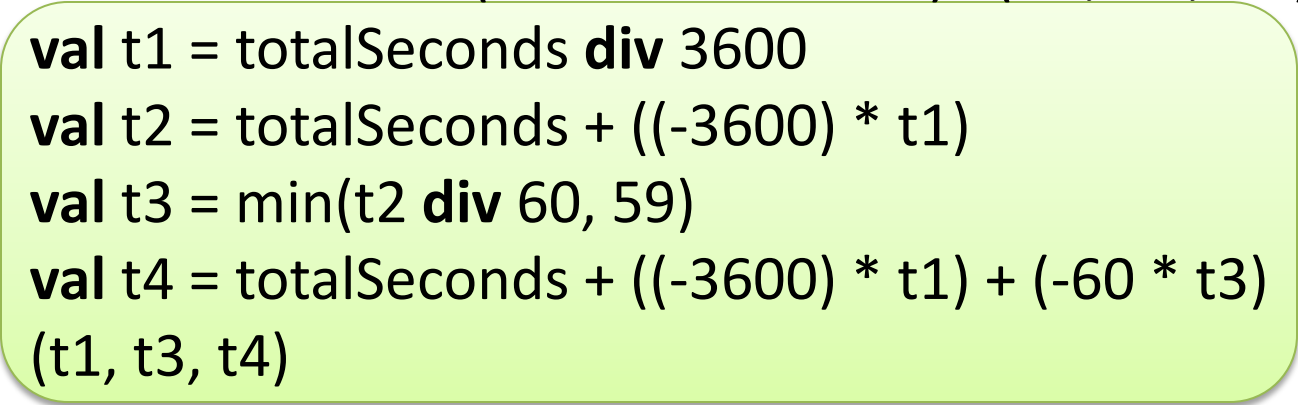
An example

```
def secondsToTime(totalSeconds: Int) : (Int, Int, Int) =  
  choose((h: Int, m: Int, s: Int)  $\Rightarrow$  (  
    h * 3600 + m * 60 + s == totalSeconds  
    && h  $\geq$  0  
    && m  $\geq$  0 && m < 60  
    && s  $\geq$  0 && s < 60  ))
```



3787 seconds \longrightarrow 1 hour, 3 mins. and 7 secs.

```
def secondsToTime(totalSeconds: Int) : (Int, Int, Int) =  
  val t1 = totalSeconds div 3600  
  val t2 = totalSeconds + ((-3600) * t1)  
  val t3 = min(t2 div 60, 59)  
  val t4 = totalSeconds + ((-3600) * t1) + (-60 * t3)  
  (t1, t3, t4)
```



choose((x, y) \Rightarrow 5 * x + 7 * y == a && x \leq y)

Use extended Euclid's algorithm to find particular solution to $5x + 7y = a$:

(5,7 are mutually prime, else we get divisibility pre.)

Express general solution of *equations* for x, y using a new variable z:

$$\begin{aligned}x &= 3a \\ y &= -2a\end{aligned}$$

$$\begin{aligned}x &= -7z + 3a \\ y &= 5z - 2a\end{aligned}$$

Rewrite *inequations* $x \leq y$ in terms of z:

$$\begin{aligned}5a &\leq 12z \\ \longrightarrow z &\geq \text{ceil}(5a/12)\end{aligned}$$

Obtain synthesized program:

```
val z = ceil(5*a/12)
val x = -7*z + 3*a
val y = 5*z + -2*a
```

For a = 31:

$$\begin{aligned}z &= \text{ceil}(5*31/12) = 13 \\ x &= -7*13 + 3*31 = 2 \\ y &= 5*13 - 2*31 = 3\end{aligned}$$

choose((x, y) \Rightarrow 5 * x + 7 * y == a && x \leq y && x \geq 0)

Express general solution of *equations*
for x, y using a new variable z:

$$x = -7z + 3a$$

$$y = 5z - 2a$$

Rewrite *inequations* $x \leq y$ in terms of z:

$$z \geq \text{ceil}(5a/12)$$

Rewrite $x \geq 0$:

$$z \leq \text{floor}(3a/7)$$

Precondition on a:

$$\text{ceil}(5a/12) \leq \text{floor}(3a/7)$$

(exact precondition)

Obtain synthesized program:

```
assert(ceil(5*a/12)  $\leq$  floor(3*a/7))  
val z = ceil(5*a/12)  
val x = -7*z + 3*a  
val y = 5*z + -2*a
```

With more inequalities
we may generate a for loop

Other Forms of Synthesis

Synthesis within IDEs

Compiling declarative constructs

Automata-Theoretic Synthesis

- reactive synthesis
- regular synthesis over unbounded domains

Synthesis of Synchronization Constructs

Quantitative Synthesis

Synthesis from examples

- **Sumit Gulwani**: *Automating String Processing in Spreadsheets using Input-Output Examples*
(video available in the ACM Digital Library)