Languages for Programming:

From Punched Cards to Wise Computing

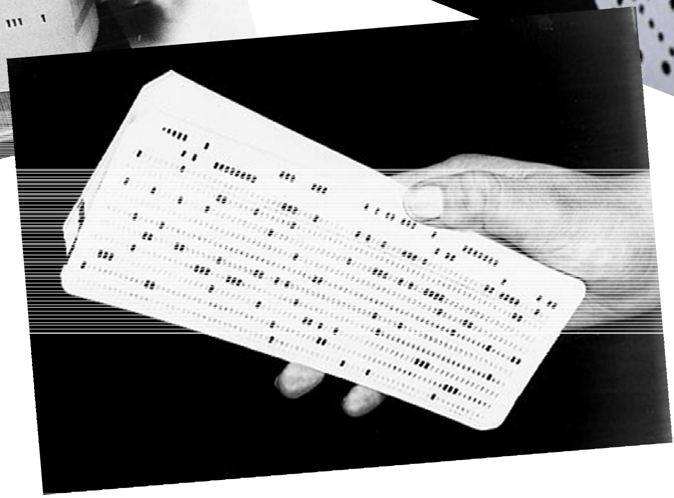
David Harel

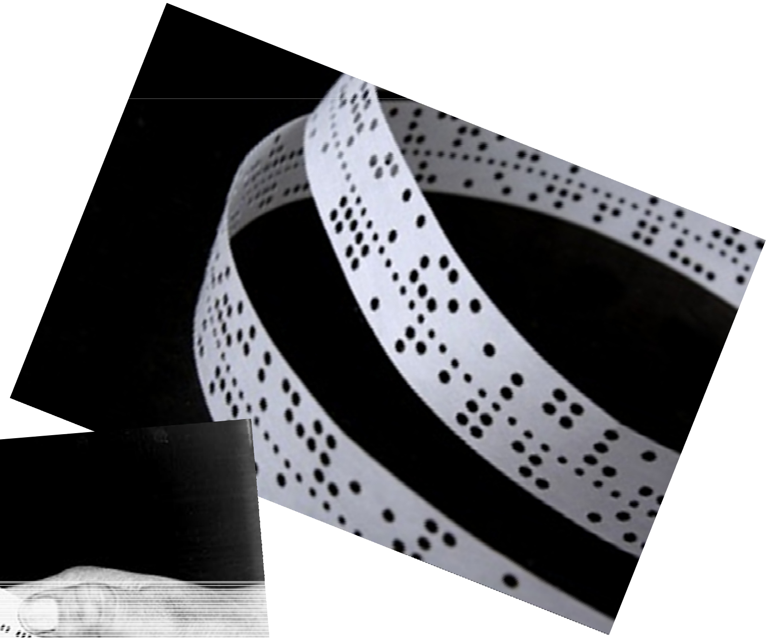
The Weizmann Institute

Languages for programming have to be endowed with formal syntax and semantics, which must unambiguously give rise to their intended functionality: full executability

First, a very brief history of general programming methods

Once upon a time, we used punched tape and punched cards...





Machine language

(1945)

```
C040: C0 4C 2B C0 AD 00 DC C9 8D
C048: 6F D0 E0 AD 83 C1 C9 05 2B
C050: F0 D9 EE 83 C1 A9 01 8D 87
C058: FD C8 AE 83 C1 BD 69 C1 FB
C060: AA A9 BA 9D 00 D0 A9 86 0E
C068: 9D 01 D0 A9 E3 8D FF 07 F9
C070: AE 83 C1 AD 15 D0 5D 6F C4
C078: C1 8D 15 D0 A9 01 8D FC E2
C080: C8 9D 75 C1 4C 2B C0 A2 F8
C088: 00 BD CF C4 9D 83 06 A9 AB
C090: 01 9D 83 DA E8 E0 21 D0 49
C098: F0 60 60 EE FA C8 AD FA A5
C0A0: C8 C9 02 D0 F5 A9 00 8D 33
C0A8: FA C8 AD FC C8 F0 25 AE A4
C0B0: 83 C1 BD 69 C1 AA DE 01 69
C0B8: D0 FE 00 D0 FE 00 D0 EE 18
C0C0: FB C8 AD FB C8 C9 06 D0 98
C0C8: 08 A9 00 8D FC C8 8D FB 57
C0D0: C8 4C 18 C1 AE 83 C1 BD 71
C0D8: 69 C1 AA DE 01 D0 DE 00 3E
C0E0: D0 DE 00 D0 EE FB C8 AD C2
C0E8: FB C8 C9 06 D0 2A A9 00 22
C0F0: 8D FB C8 8D FD C8 AE 83 C9
C0F8: C1 A9 01 9D 7B C1 A9 E0 CA
C100: 8D FF 07 AD 7C 05 8D 81 D2
C108: C1 20 84 C1 AD 20 89 8D 15
C110: F8 89 AD 21 89 8D F9 89 FB
C118: AE 83 C1 FE F8 07 BD F8 C1
C120: 07 C9 E6 D0 05 A9 E4 9D D9
C128: F8 07 60 06 A9 00 8D 2B F0
C130: C1 AE 2B C1 BD 7B C1 D0 59
C138: 0B EE 2B C1 AD 2B C1 C9 83
C140: 06 D0 EE 60 BD 69 C1 AA F9
C148: DE 00 D0 BD 00 D0 C9 18 68
C150: D0 E7 AE 2B C1 AD 15 D0 38
```

```
$MOD8253
DSEG
                ORG     20h
Var1            DS      1
STATE           BIT     Var1.0
OUTPUT          BIT     P1.0

CSEG
                ORG     0h
                AJMP    START

                ORG     0Bh
                AJMP    INTERRUPT

START           MOV     IE,#82h
                MOV     TMOD,#01
                MOV     TH0,#FEh
                MOV     TL0,#0Ch
                SETB    STATE
                SETB    TR0

LOOP            NOP
                SJMP    LOOP

INTERRUPT       CLR     TR0
                MOV     TH0,#FEh
                MOV     TL0,#0Ch
                SETB    TR0
                CPL     STATE
                MOV     C,STATE
                MOV     OUTPUT,C
                RETI
                END
```

Machine language
(1945)

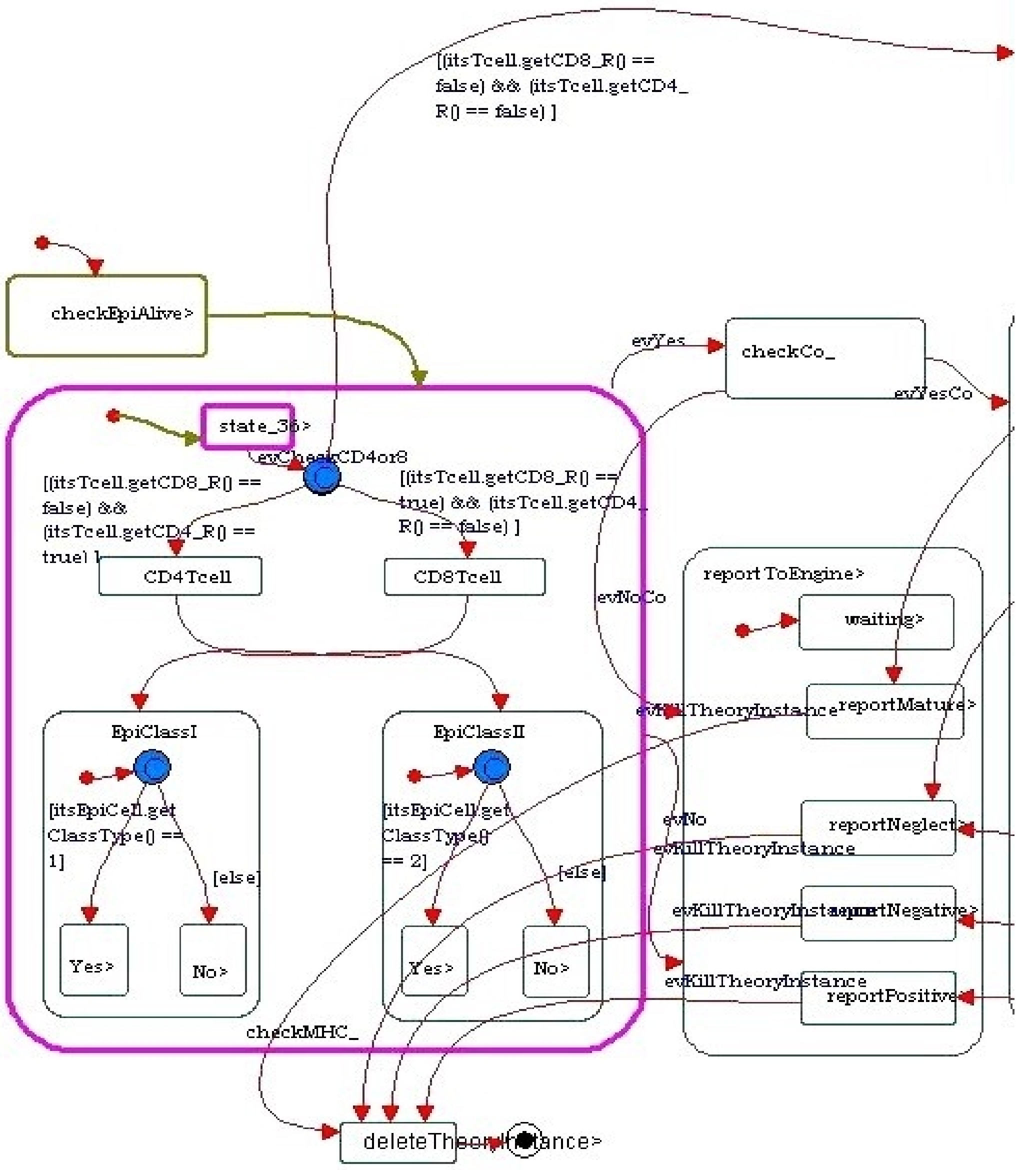
$\boxed{\text{Assembly language (1950)}}$

```
PlayerControl ()
{
    euler = Vector3::GetZero();
    speed = 0.2;

    turnSpeed = 10.0;
    maxTurnLean = 50.0;
    maxTilt = 50.0;

    sensitivity = 10.0;
    forwardForce = 1.0;
}


virtual void Start () {
    // Get an access to another script attached to the same GameObject
    missileLauncher = GetComponent<MissileLauncher>();
}

virtual void Update () {
    for (int touchIndex = 0; touchIndex < Input::GetTouchCount(); touch
    {
        Touch touch = Input::GetTouch(touchIndex);
        if (touch.phase == TouchPhase::Moved)
        {
            speed = touch.position.y / Screen::height;
            guiSpeedElement.position = Vector3 (0, speed, 0);
        }

        if (touch.phase == TouchPhase::Ended)
        {
            missileLauncher->Fire();
        }
    }
}

virtual void FixedUpdate () {
    rigidbody.AddRelativeForce(0, 0, speed * forwardForce);

    Vector3 accelerator = Input::GetAcceleration();
```

Machine language
(1945)

Assembly language
(1950)

High-level prog. langs.
(1970)



Machine language
(1945)

Assembly language
(1950)

High-level prog. langs.
(1970)

Modeling/graphical
langs.
(1985)

Machine language
(1945)

Assembly language
(1950)

High-level prog. langs.
(1970)

Modeling/graphical langs.
(1985)

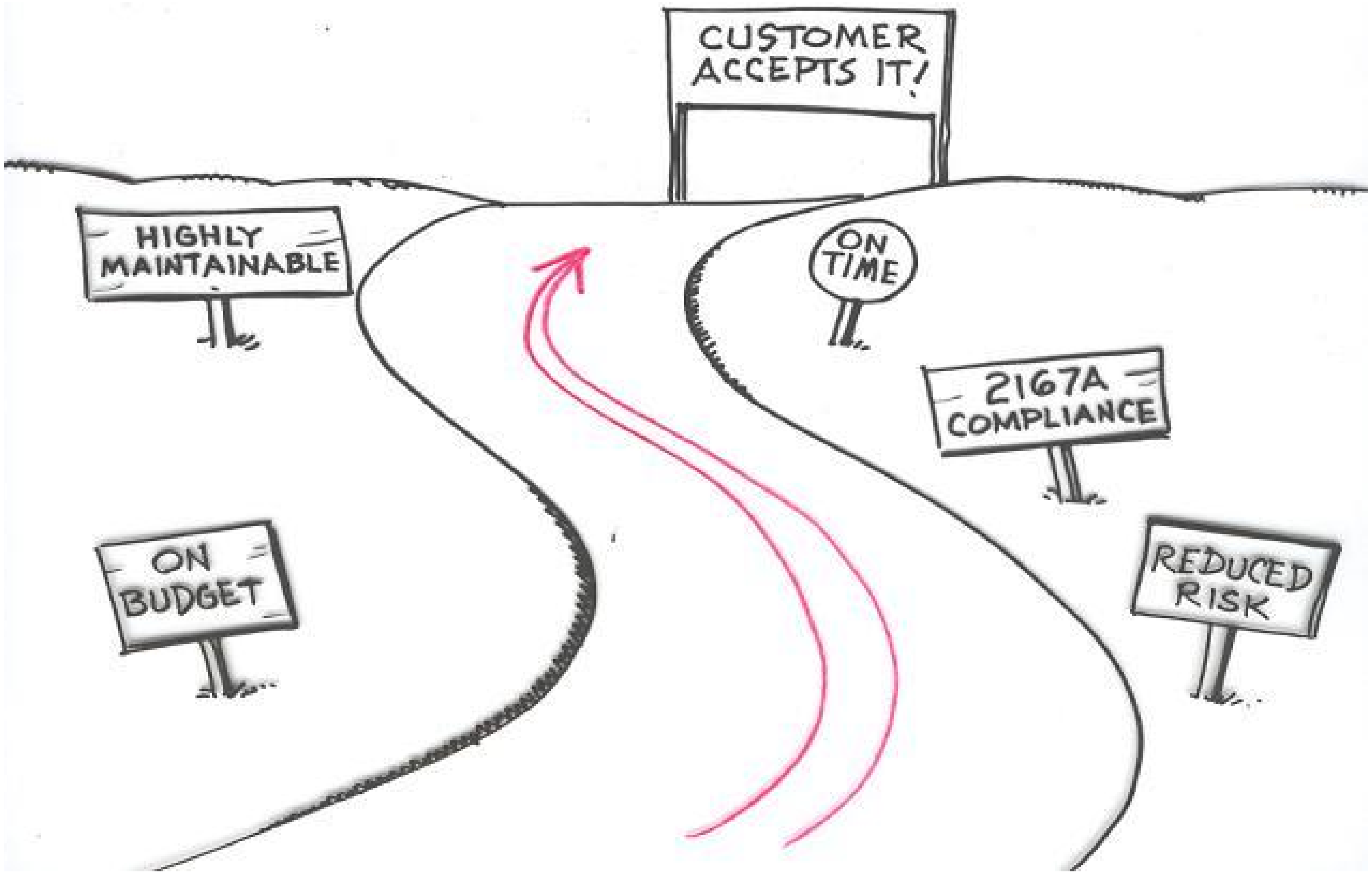
And what after that?

Let's concentrate on developing complex reactive systems

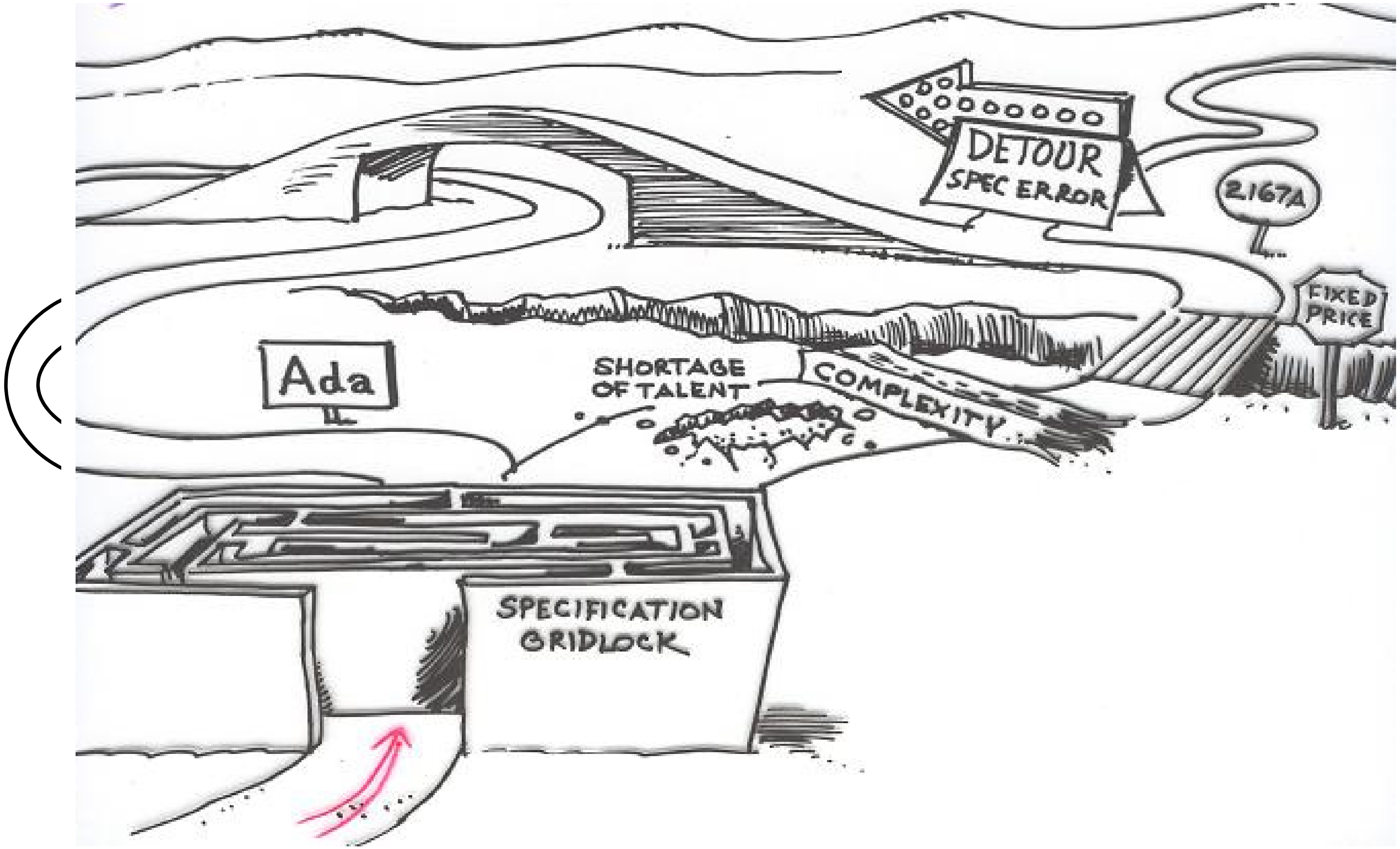
(term introduced with Pnueli 1985)

… which interact heavily with users or with other systems

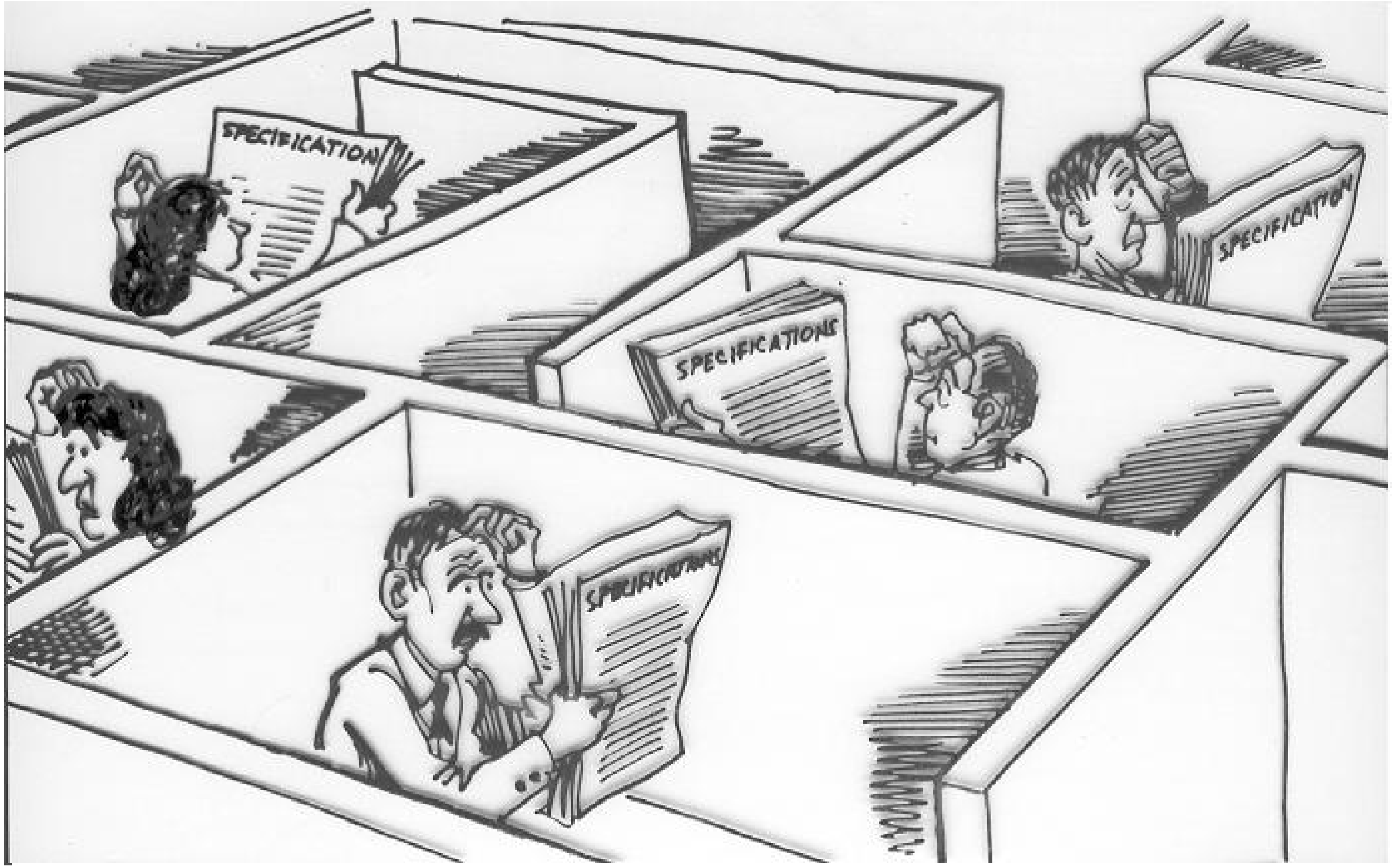
Speedway to success



The actual development process



Specification Gridlock: A Closer Look



Specification Gridlock: The root of the problem

Specifiers
- interpret requirements
- create specification

Behavior!

Implementors
- interpret specification
- create hardware & software

! ! !

Taken from a real spec!

Section 2.7.6: Security (~ page 10)

"If the system sends a signal hot then send a message to the operator."

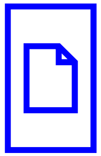
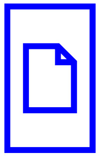
Section 9.3.4: Temperatures (~ page 150)

"If the system sends a signal hot and $T>60^0$, then send a message to the operator."

Summary of critical aspects (~ page 650)

"When the temperature is maximum, the system should display a message on the screen unless no operator is on the site except when $T<60^0$."

Statecharts (1984) were invented, at least in part, to help alleviate this problem

Actually, we "program" all the time, though not necessarily computers...

And we use scenarios, examples, implicit instructions, analogies, constraints, etc.

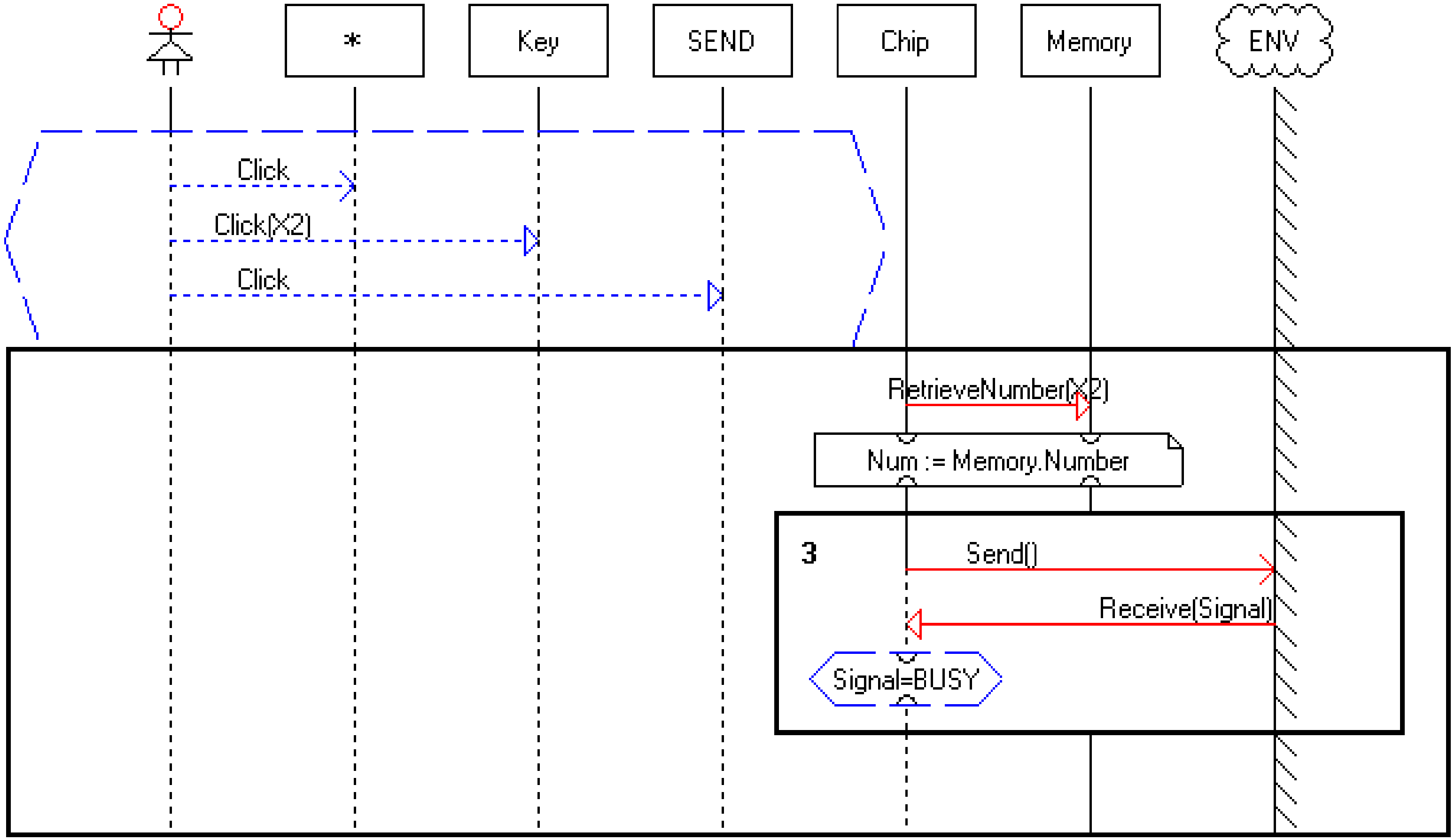
The recent scenario-based approach (1999 and on) brings programming a lot closer to the way humans prescribe and describe behavior

Multi-modal: includes mandatory, possible and forbidden behavior

A live sequence chart (LSC)

prechart
(if)

main chart
(then)



Have several non-graphical versions of this (e.g., Java, C++)

Approach called more generally
Scenario-Based (or Behavioral) Programming

How to most naturally construct LSCs?

I. Construct chart directly

II. “Play in” behavior from realistic graphical interface

III. Use Natural Language

Can start from scratch and go all the way to a full executable

IV. Use "Show & Tell"

Combine NL with play-in

Commercial break

New EdX online course

Liberating Programming:

System Development for Everyone

But,….. wouldn’t it be really nice if the process of programming a computer could be two-way, and the programing environment would be endowed with powerful human-like wisdom?

It would then become almost an equal partner, helpful and concerned, like human members of the system development team

Indeed, humans can do a lot more…

(health care robot; credit: A. Marron)

- **Notice irregularities, unexpected properties:**

  "The arm movement is not smooth!"

  "Hear that strange noise when it turns"

- **Detect missing requirements, assumptions:**

  "Will it understand the voice of a hoarse patient?"

  "Can it process voice commands with the TV on?"

- **Ask (& answer) hard "what if" and "why" questions:**

  "Will a loud command from the TV confuse it?"

  "Why is it just walking around ? Is it looking for something?"

- **Use broad knowledge and free association:**

"Recently a pacemaker was remotely hacked. Can this happen here?"

- **Exhibit creativity, unusual thinking (outside the box)**

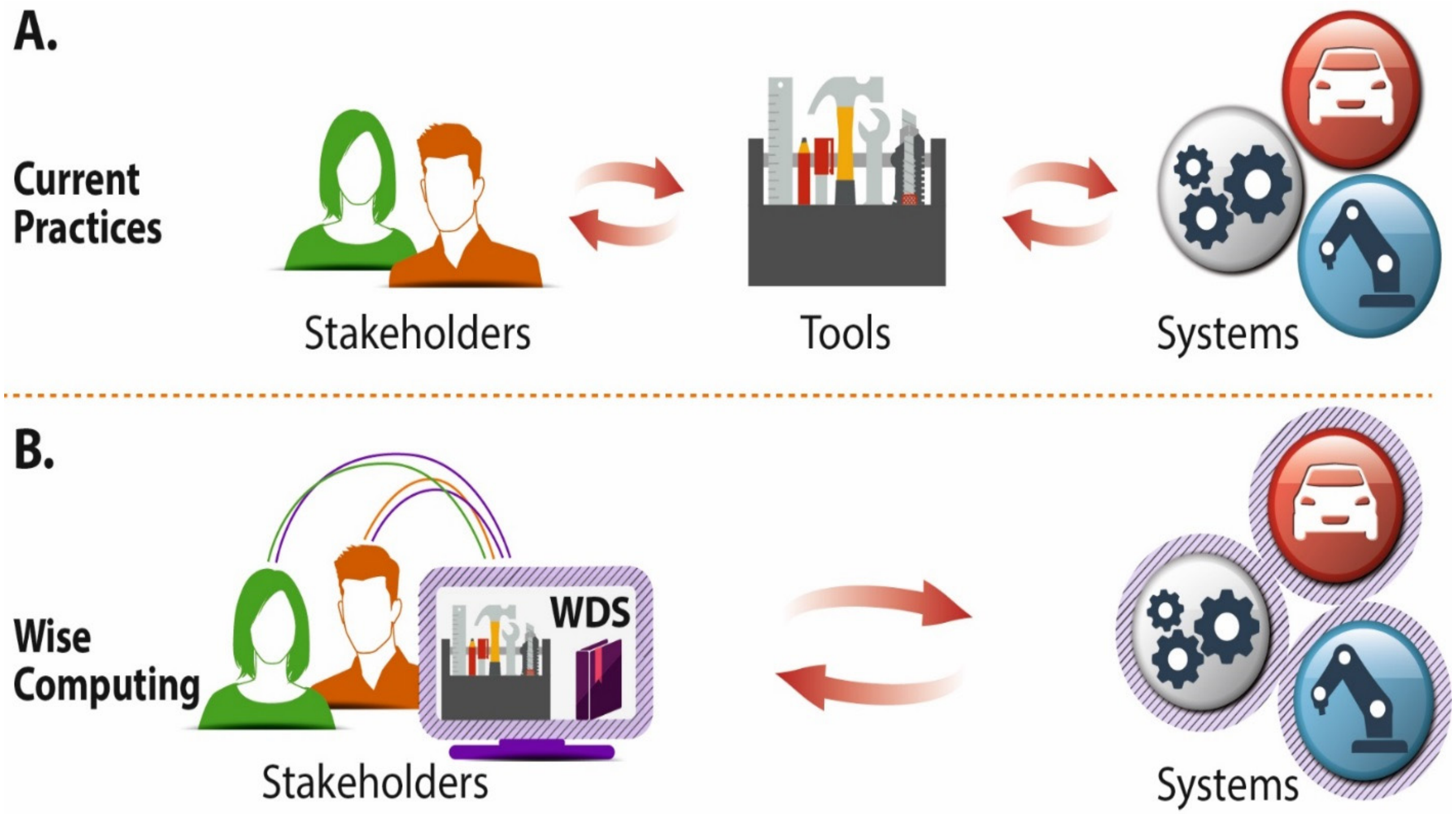
"Let's see what happens if I ask it to fetch something that's glued to the table…"

We call such a futuristic approach to programming "Wise Computing"

It entails all that, and lots more…

arXiv, Jan 2015; and *IEEE Computer*, Feb 2018

From a tool to a proactive partner



Main Research Directions:

Formalization

Analysis

Interaction

- **Common Formalism:** Statecharts and LSCs at its heart, but with much more, intended to capture all relevant knowledge.
- **Analysis Engine:** proactive, uses heavy-duty learning, verification, SMT solving, etc., mimics human skills.
- **Interaction Language & Engine:** two way, multiple abstraction levels, natural language, captures all level of communication with human team.

Two demos of proof-of-concept wise development suite

(mainly proactive analysis)

Concept and simple example: 12 min.

Cash coherence protocol: 18 min.

Main acks:

Amir Pnueli, Werner Damm, Rami Marelly,
Shahar Maoz, Assaf Marron, Smadar Szekely,
Gera Weiss, Michal Gordon, Guy Katz

Thank you for listening

Thank you for listening