From CAIA to RefPerSys
reflexive, introspective, meta- based AI systems

Basile Starynkevitch - starynkevitch.net/Basile
basile@starynkevitch.net - 92340 Bourg-La-Reine, France

RefPerSys - refpersys.org

Paris, March 6th, 2020 - seminar in honor of the late J.Pitrat

Opinions are only mine git commit 6956d6fffa7f2d4c

My employer (or funding agencies at work) would probably disagree with most of my opinions here. Any agreement with my employer's policies or positions is accidental.

Overview

These slides contain hyperlinks and are downloadable from
`refpersys.org/Starynkevitch-CAIA-RefPerSys-2020mar06.pdf`

What is AI?

- **A**rtificial **I**ntelligence (and AI winter - predicted by J.Pitrat)
  - **A**rtificial **G**eneral **I**ntelligence
  - symbolic **A**rtificial **I**ntelligence
  - machine learning **A**rtificial **I**ntelligence
  - **A**rtificial **I**ntelligence applications
- **A**dvanced **I**nformatics
- **A**bstract **I**nterpretation (a technique for static program analysis, by Cousot)
  $\rightarrow$ IMHO it could be the next "AI winter"
  I am impatiently waiting for Frama-C fully automated analysis of TensorFlow C++ code (its floating point precision issues), or of Caia C code ☺; the "robust AI" buzzword....

philosophical question

Is AI a science unrelated to computer science, or is it part of it?

(probable major disagreement between J.Pitrat and me)

Why AI systems are needed? hard problems (1/2)

Because mankind or nations or continents -and decision makers- face major problems which are not fully understood (incomplete list) :

- global warming, malnutrition and pollution
- the [co-] design and operation of complex systems (nuclear power plants -ITER-, autonomous transportation, smart grids, super grid, smart cities, communication networks, water distribution system, cobots)
- managing complex systems (e.g. the World Wide Web) or their implementation (fiber optics deployment in France) or organizations (Vélib) - avoiding boreout
- digital twins (e.g. of an automobile, of holiday travels) and distributed embedded systems (or edge computing)
- macro-economical policies of the Euro zone (why negative interest rates?)
- fighting global bio-viruses (e.g. coronavirus = closed software + chemistry)
- in France: retirement policies (why is it difficult to simulate?)

Why AI systems are needed? hard problems (2/2)

- in general: defining useful regulations (e.g. CO_2 global market, world-wide distribution of economical wealth, neuroeconomics)
- political problems: peace in Middle East?
- lack of motivated software developers (how to keep their motivations and productivity?) - see *Bullshit jobs* and *La comédie (in)humaine*
- going to Mars, terraforming it, colonizing it
- understanding the physical world (particle physics, quantum physics, exobiology, cosmology?) and improving it (climate-smart agriculture, world-wide better energy mix)
- understanding and repairing the human body (whole body digital twin? neurology? oncology? neurosurgery, obesity)
- understanding and improving the Internet (it could be very brittle?)
- understanding and improving the French law (online, but how much is it consistent?)
- improving communication and trust between human beings

Why AI systems need to be free software? (1/2)

The GNU FSF defines free software :

- The freedom to run the program as you wish, for any purpose (freedom 0).
- The freedom to study how the program works, and change it so it does your computing as you wish (freedom 1). Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help others (freedom 2).
- The freedom to distribute copies of your modified versions to others (freedom 3). By doing this you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this.

"**What is free software and why it is important to *society***"

The "source code" being defined as : "The source code must be the preferred form in which a programmer would modify the program"

Why AI systems need to be free software? (2/2)

See also Le fabuleux chantier : Rendre l'intelligence artificielle robustement bénéfique (The fabulous project: making artificial intelligence robustly beneficial)
Notice analogy between free software / open source and academical [mis-]practices

- publish or perish. Pitrat observed that there is not enough incentive to make real-sized AI *software systems* (not toy demos) and favoring experimental approaches.
- *The Simple Economics of Open Source*
- *Big other: surveillance capitalism and the prospects of an information civilization*

See `softwareheritage.org` and dream of mega knowledge bases

Pitrat's thesis

about intelligence

There is no *simple theory* or *simple model* of intelligence (be it artificial or natural).
$\Rightarrow$ any AI system has to be complex (with chaotic behavior) and evolving!

Pitrat's analogy: a bird can fly, an airplane also fly, they are designed differently.

- **experimentation is king**
- **theoretical limitations** (affecting humans too!) do not matter **in practice** :
  1. halting problem
  2. Rice's theorem
  3. Gödel's incompleteness theorems
  4. Church-Turing thesis
  5. Curry–Howard correspondence;
     see *Software, between mind and matter* (X.Leroy)
  6. heuristically and **usually** avoiding combinatorial explosion

consequences of Pitrat's thesis

AI systems have to be *globally* inconsistent (like humans are) and ***antifragile*** (Taleb)

- importance of "insight" : Eureka effect ("Aha! moment")
- intelligent systems cannot be fully modular or compositional to have an emerging intelligent behavior with self-organization
- theoretical quasi-impossibility to explain or modelize intelligence.
- AI systems are non-modular and will exhibit non-reproducible behavior
  $\Rightarrow$ randomized algorithms are practically essential to AI!
- **time is important in AI systems** (see `time(7)` on Linux)
- importance of self-awareness and introspection for intelligent systems (on Linux could be done with `libbacktrace`, conceptually related to the discoveries of continuations)

Since AI systems need to have some kind of randomness (see `random(4)` on Linux), **experimental reproducibility is *ethically* important**.

Engineering aspects of software (1/2)

We need a computer to run a software:

(photo by Matthieu Starynkevitch)

(we look inside the box and see things -cables- which are *not* inside but reflected)

Engineering aspects of software (2/2)

But J.Pitrat wrote (in *Artificial Beings*, §4 p67):

> *The black box is the opposite of consciousness, and we must always avoid it*

and later (*Artificial Beings* p259):

> CAIA *uses two programs that it has not written:* GCC *and* LINUX

emphasis is mine; I will dare doing some nitpicking

My black box runs CAIA ☺

D.Wheeler's `sloccount`

A **machine learning** approach to **estimate software development costs** by counting source lines of code (SLOC), and free (GPL) software measuring that, on **`dwheeler.com/sloccount/`**.
A good programmer is rumored to be able to write about 30K SLOC per year and this depends not much on the programming language used by him (if he is expert on that language).

Some figures (from Source lines of code wikipage) :

software	**source size**	
Debian 7	419 MSLOC	the old Debian 7 distribution
Linux 3.6	15.9 MSLOC	the Linux 3.6 kernel

Estimating software development costs is tricky: half of software projects are "failing"; refactoring a code base may involve replacing ten thousands lines by just one thousand of better lines.

But software code size remains a good measure for its development costs.

source code measurement of CAIA

From github.com/bstarynk/caia-pitrat, *all* its C lines being generated :

```
SLOC Directory SLOC-by-Language (Sorted)
506558 caia-pitrat ansic=506160,sh=281,perl=63,cpp=54

Totals grouped by language (dominant language first):
ansic: 506160 (99.92%)
sh: 281 (0.06%)
perl: 63 (0.01%)
cpp: 54 (0.01%)

Total Physical Source Lines of Code (SLOC) = 506,558
Development Effort Estimate, Person-Years (Person-Months) = 138.32 (1,659.86)
(Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Schedule Estimate, Years (Months) = 3.49 (41.84)
(Basic COCOMO model, Months = 2.5 * (person-months**0.38))
Estimated Average Number of Developers (Effort/Schedule) = 39.67
Total Estimated Cost to Develop = $ 18,685,394
(average salary = $56,286/year, overhead = 2.40).
SLOCCount, Copyright (C) 2001-2004 David A. Wheeler
```

other source code measures

software	**source size**	
CAIA	506KLOC	CAIA as before
Linux 3.6	15.9 MSLOC	the Linux 3.6 kernel on kernel.org
Xorg 1.12	406KLOC	The X11 window display server on www.x.org
GCC 9.2	5.605MLOC	The GCC compiler on gcc.gnu.org
binutils 2.34	2.104MLOC	GNU assembler and linker www.gnu.org/software/binutils/
bash 5.0	124KLOC	GNU Bourne Again SHell www.gnu.org/software/bash/
glibc 2.31	1.257MLOC	GNU C library www.gnu.org/software/libc/
libbacktrace	18.9KLOC	backtracking github.com/ianlancetaylor/libbacktrace
Qt5	23.799MLOC	a C++ GUI toolkit qt.io
GNU readline	36.6KLOC	a line editor library www.gnu.org/software/readline/

illusion about C compiler

J.Pitrat told me:

> *Only a small amount of the C language is used by* CAIA *so I don't depend that much on GCC*

But using `gcc -O2 -ftime-report` shows that about 80% of GCC optimization passes are used.

See some pages of my Bismon draft report for an explanation.
GCC optimizations are surprising.

Disabling optimizations would lower CAIA performance by perhaps 30%.
See also TinyCC.

The Joel Test: 12 Steps to Better Code

From dev.to/checkgit/the-joel-test-20-years-later-1kjk

1. Do you use Git, or some lesser source control system?
2. Can you build and release in one step?
3. Do you build and test before merging to master? `make bootstrap` in *GCC*
4. Do you have a bug database?
5. Do you fix bugs before writing new code?
6. Do you have an up-to-date schedule?
7. Do you write a spec before writing code?
8. Do programmers have quiet working conditions free of interruptions?
9. Do you use the best development tools money can buy?
10. Do you have human testers?
11. Do you have automated testing?

with code reviews (e.g. gcc-patches@gcc.gnu.org)

The Magical Number 7 ± 2

Miller's law (1956)

the number of objects an average person can hold in working memory is about seven

This empirically holds for software developers, computer scientists and human artificial intelligence researchers.

$\rightarrow$ a software whose cyclomatic complexity (Mc.Cabe 1976) is above 7 will probably remain undebuggable, and complex software are always buggy.

$\rightarrow$ avoid spaghetti code in human written programs. But that does not matter for machine generated code.

Miller's law in AI

$\rightarrow$ the types of functions in human written functional style programs remains reasonable. Pathological cases of type inference are practically rare.

So modular programming[1] or object-oriented programming is for humans. Hence Just-in-time compilation techniques (e.g. devirtualization of virtual method tables) are practically efficient.

Few functions get much more than 6 or 7 arguments

Miller's law in AI system

Is there one? Do AI systems have a short memory threshold?
Is the 7 threshold related to mathematical properties of semantic networks (diameter of reference graphs)

[1] Generated code could be non-modular!

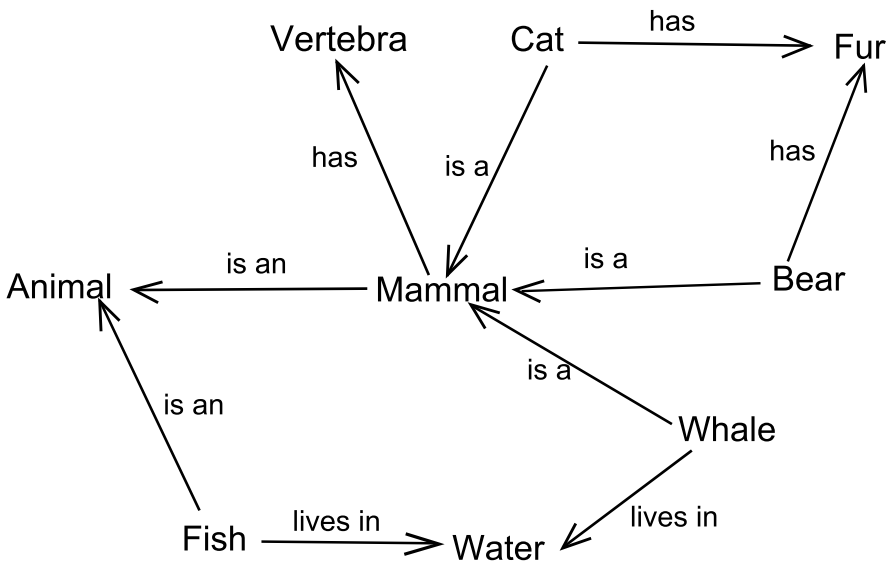
What should really matter in an AI system

- **explicit some trusted computer base**, that is all the things
  - hardware components (motherboard, keyboard, mouse, screen, Ethernet)
  - low-level software components (firmware: BIOS, UEFI, graphics card firmware)
  - operating system kernel (Linux kernel)
  - middleware software components (compiler, user interface, operating system, file systems, databases)

  the AI system depends on. Getting rid of every software component is practically futile (billions of source lines in a Linux desktop ; see also `linuxfromscratch.org` which is bootstrapped).
- **take advantage of most *existing* computing resources** (multi-core, cache locality, disk, MMU, Internet) **and *existing* software components**.
- **give *declarative* knowledge to your AI system about efficiently using *existing* open source software components** which are choosen by humans

AI software and computers "are" semantics networks

From wikipedia Semantic network (but imagine hundred of thousands of nodes and labelled arrows)



$\Rightarrow$ Flexible frame representation, including knowledge representation of various "expertises" about how to use existing "trusted" software components (cf DECODER project) and how to generate code.

Brook's law

From *The Mythical Man-Month* (F.Brooks, 1975):
Its central theme is that "adding manpower to a late software project makes it later".

while it takes one woman nine months to make one baby, "nine women can't make a baby in one month"

it takes some incompressible time to make a bootstrapped AI system.

and there is *No silver bullet*

Communication between software developers (or between artificial intelligence researchers) matters. *Open source* software is an effective communication channel.

Heisenbugs and heisenmetabugs. Also Cargo cult programming and rubber duck debugging (and usefulness of code reviews, even by junior programmers)

Hofstater's law

Hofstadter's law

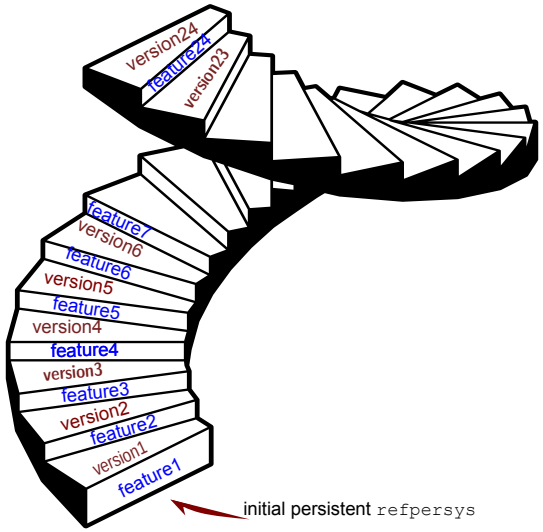
It always takes longer than you expect, even when you take into account Hofstadter's Law.

Very true of development of AI systems (or of preparing slides like these ones). See *Gödel, Escher, Bach: An Eternal Golden Braid*

Related: *I am a strange loop* and Ship of Theseus paradox (relevant to bootstrapped AI systems)

the staircase development model of bootstrapped AI

Illustrated with RefPerSys, applicable to CAIA:



*Each new feature -or small incremental change or a few of them (small `git commits`) - of RefPerSys enables us to build and **generate** the next version of RefPerSys, and a next feature is then added to that improved version, and so on repeatedly, etc....*

CAIA demo

`git clone https://github.com/bstarynk/caia-pitrat.git`. All its C code has been self-generated. You need 250Mbytes of disk space.
`cd caia-su-24feb2016`
Observe the buggy generated `dx.h` (missing `extern`).
`make` perhaps needed twice (15 minutes of CPU time).
`./caia`
Observe the ed like user interface. Try `L EDITE` perhaps twice.
Look into `_*` persistent data files. Too small to be friendly with most file systems. See figure on Ext2 wikipage. Read about file systems some Operating Systems textbook.
Use strace(1) to find out that the date is displayed by running the `date` command. But see time(7)
No C dynamic memory allocation using malloc(3). That was a design mistake.

some software I wrote for J.Pitrat

m

- `manydl.c` under `git clone` `https://github.com/bstarynk/misc-basile.git` generates random C code, compiles each of them as a plugin. Shows that a Linux program can dlopen(3) "simultaneously" many hundreds of thousands of plugins.
- `git clone` `https://github.com/bstarynsk/minil.git` deals with dynamically allocated frames (persisted in JSON format with `libjansson`) and `readline` command line interface with auto-completion. French comments, GPLv3+ (could be used by some intern as a starting point).

RefPerSys - early work in progress

A future successor to CAIA for Linux/x86-64 (GPLv3+, open science). A bootstrapped AI system. An acronym for **Ref**lexive **Per**sistent **Sys**tem.
See **refpersys.org** and code (C++ and persistent JSON data) on **https://gitlab.com/bstarynk/refpersys/**

SVG logo by Gaëtan Tapon (Paris, France)

With contributions from Abhishek Chakravarti and Nimesh Neema (India). Active additional contributors welcome.
Coded in C++ (some of which is generated). For Qt graphical interface.

immutable values of RefPerSys

Each value has a class. ObjVLisp model. Most values are immutable, except objects. They are dynamically allocated and garbage collected

- tagged 63 bits integers
- boxed doubles
- boxed UTF-8 strings
- mutable objects
- ascending sets of objects
- tuples of objects
- closures (the closed "variables" are values, the connective gives the function)
- immutable instances with components (values)
- boxed Qt pointers (to widgets)
- boxed JSON
- etc ...

mutable objects of RefPerSys

s Each object has :

- a fixed random "globally" unique object id (96 bits) externally represented by a oid string like `_02iWbXmFx8f04ldLRt`. So objects are comparable by their oid.
- a mutex for multi-thread locking
- a modification time (double)
- a mutable class (itself an object reference)
- a space (itself an object reference, or nil) for persistence
- a finite mapping (varying with time) for attribute entries. Each entry has a key which is an object, associated to a non-nil value. Every key is different.
- a varying vector for object components
- an optional mutable payload
- two atomic C++ function pointers (one for magic attribute getting, another for applying closures having that object as connective).

mutable payload in objects of RefPerSys

They hold any additional data not fitting elsewhere:

- class information (method dictionnary + superclass)
- symbol information
- string buffers
- mutable vectors of doubles
- mutable vectors of values
- mutable sets (e.g. `std::set`)
- mutable relations
- dictionnaries associating strings to values or object
- Qt windows
- file or process or database handles
- etc...

Objects and their payload may represent a semantics network

persistence in RefPerSys

The entire heap is dumped before exit and reloaded just after startup. Using JSON (so in a textual format friendly to `git`).

- some values are transient (Qt widgets, transient closures or instances).
- most values are persistent.
- objects are shared and persistent in their space.
- continuations (call stacks) are not persisted yet; using `libbacktrace`

graphical user interface in RefPerSys

Work in progress in march 2020.

- several Qt windows
- auto-completion
- syntax oriented editor (Centaur inspired)
- model view controller approach above flexible RefPerSysobject model

agenda machinery in RefPerSys

To be implemented:

- multi-threaded, multi-workers (e.g. a dozen of worker threads)
- several queues of tasklets
- each tasket is an object, conceptually a sort of bytecode VM
- each worker thread runs a tasklet for a few milliseconds. That run could modify the agenda by adding/removing/reorganizing tasklets in it
- the Qt GUI may add tasklets to the agenda
- the agenda should be partly persistent

future work

- generating most of C++ code (Quine), including persistence, Qt, GC
- perhaps interfacing `libgccjit` -runtime generation of code
- rule based approach
- metarules compiled to code
- richer frame model
- climbing the staircase with higher-level (more declarative) representations
- declarative knowledge to use machine learning libraries *TensorFlow*, *Ghudi*
- self-tuning approach à la MILEPOST GCC
- declarative knowledge about other major C++ open source libraries and generate C++ code calling them.

why generate C++? (1/2)

Mostly for social reasons: C++11 or C++17 is well known, with robust compilers. Also

- C++11 has powerful standard containers
- C++ is compatible with C, and `dlopen`-able on Linux: C++ dlopen mini HowTo
- I worked inside GCC, which accepts plugins coded in C++.
- C++ is multi-threaded
- C++17 don't have flexible array members yet but in practice on Linux/x86-64 we can use with care (as in C89) last fields which are arrays of dimension 0.
- C++ has closures and smart pointers.

why generate C++? (2/2)

Also

- existing C++ compilers optimize very well (but slowly).
- C++ on Linux is compatible with libgccjit and/or asmjit for just-in-time generation of code at runtime.
- many reputable interesting C++ (or C) libraries exist (JsonCpp, TensorFlow, Ghudi, Qt, POCO, GMPlib etc....)

But C++ is an ugly monster (n3337 has 1324 pages) which I don't know and it is compiled slowly. Be aware and scared of undefined behavior.

Other language implementations could have been considered: Rust, Go, SBCL, Bigloo, Ocaml, assembler

a difficult problem for AI systems

Finding seed funding and real life problems. Should require some AI (self-application of RefPerSys on that problem).

Generating more and more complex C++ programs (GCC or Firefox plugins; FastCGI programs; embedded Linux code?). Inspiration from obsolete GCC MELT experience (was a Lispy DSL translated to C++ for GCC plugins)

your help is needed

mailing list hosting for low traffic publicly archived `list@refpersys.org` (using MailMan or Sympa....) or archived IRC, or help on setting them up on refpersys.org

Contributing your time and efforts to grow RefPerSys

Contributing old hardware, books.

Sponsoring (travel, workshops) and **suggesting real-world challenging but sellable applications**.

Finding funding (HorizonEurope? freelancing activities?)

git commit `6956d6fffa7f2d4c`