Applications of Logic

Lijun Zhang

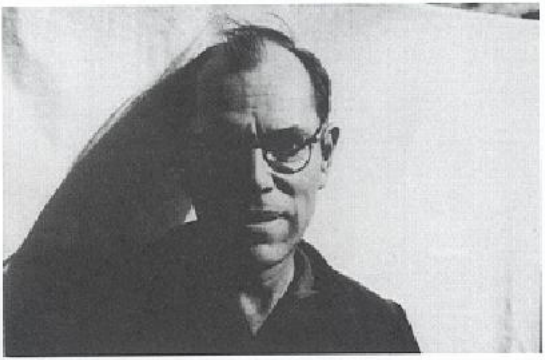
Institute of Software, Chinese Academy of Sciences

3rd April 2018

$a \quad o \quad e \quad i \quad u \quad \boxed{\hat{u}}$

- Who is $\underbrace{\text{Büchi}}$?
- What is Büchi automata?
- Why he introduced Büchi automata?

Julius Richard Büchi


J. Richard Büchi, 1983

- a Swiss logician and mathematician (1924–1984)
- received his diploma in mathematics and theoretical physics at ETH Zürich (Prof. Hopf)
- went to home (St. Gallen) for eight months to work on a problem of his own

Why he introduced Büchi automata?

- Büchi, J.R. (1962). ”On a decision method in restricted second order arithmetic”. Proc. International Congress on Logic, Method, and Philosophy of Science. 1960. Stanford: Stanford University Press: 1–12.

Logic of Finite Words

Definition (Logic of Finite Words)

View finite word $w = a_0, \ldots, a_{n-1}$ over alphabet Σ as a mathematical structure:

- Domain: $\boxed{0, \ldots, n-1}$
- Dyadic predicate: $\leq$
- Monadic predicates: $\{P_a : a \in \Sigma\}$

a b a a b

Monadic Second-Order Logic (MSO):

- Monadic atomic formulas: $P_a(x)$ $(a \in \Sigma)$
- Dyadic atomic formulas: $x < y$
- Set quantifiers: $\boxed{\exists P, \forall P}$
- Successor function: S Example: $(\exists x)((\forall y)(\neg(x < y)) \wedge P_a(x))$ - last letter is a.

Automata and Logic

Definition (Automata and Logic)

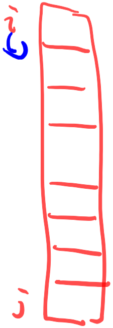
[Büchi, Elgot, Trakhtenbrot, 1957-8 (independently)]: $\boxed{\mathrm{MSO} \equiv \mathrm{NFA}}$
Both MSO and NFA define the class Reg.
Proof: Effective

- From NFA to MSO ($\underset{\sim}{A \to} \varphi_A$)
- From MSO to NFA ($\varphi \to A_\varphi$)

Program Termination Analysis

Does this program terminate?

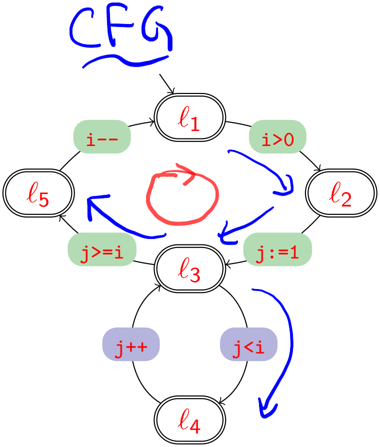
```
program sort(int i):
ℓ1: while (i>0):
ℓ2:   int j:=1
ℓ3:   while (j<i):
        // if (a[j]>a[i]):
        //   swap(a[j],a[i])
ℓ4:     j++
ℓ5:   i--
```



Program Termination Analysis

Does this program terminate?

```
program sort(int i):
ℓ1: while (i>0):
ℓ2:   int j:=1
ℓ3:   while (j<i):
        // if (a[j]>a[i]):
        //   swap(a[j],a[i])
ℓ4:     j++
ℓ5:   i--
```



Entscheidungsproblem

Definition (Entscheidungsproblem)

Entscheidungsproblem (The Decision Problem) [Hilbert-Ackermann, 1928]:
Decide if a given first-order sentence is valid (dually, satisfiable).
Church-Turing Theorem, 1936: The Decision Problem is unsolvable.
Turing, 1936:

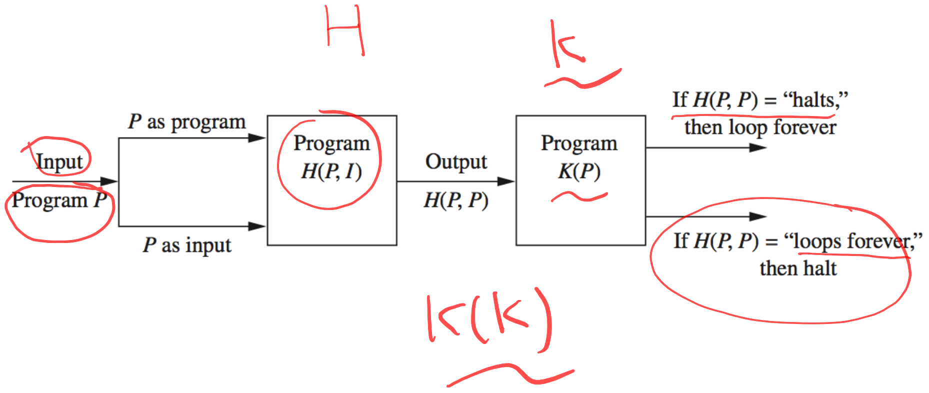
- Defined computability in terms of Turing machines (TMs)
- Proved that the termination problem for TMs is unsolvable (“this machine terminates iff it does not terminate”)
- Reduced termination to Entscheidungsproblem.

Halting Problem

It takes as input a computer program and input to the program and determines whether the program will eventually stop when run with this input.

- If the program halts, we have our answer.
- If it is still running after any fixed length of time has elapsed, we do not know whether it will never halt or we just did not wait long enough for it to terminate.

Undecidability of the Halting Problem



Termination

B. Cook, A. Podelski, and A. Rybalchenko, 2011, CACM: Proving Program Termination.

- “in contrast to popular belief, proving termination is not always impossible”
- The Terminator tool can prove termination or divergence of many Microsoft programs.
- Tool is not guaranteed to terminate! Explanation:
- Most real-life programs, if they terminate, do so for rather simple reasons.
- Programmers almost never conceive of very deep and sophisticated reasons for termination.

Andrey Rybalchenko, at 32, 2010: Innovators under 35, MIT Technology Review.

Andrey Rybalchenko

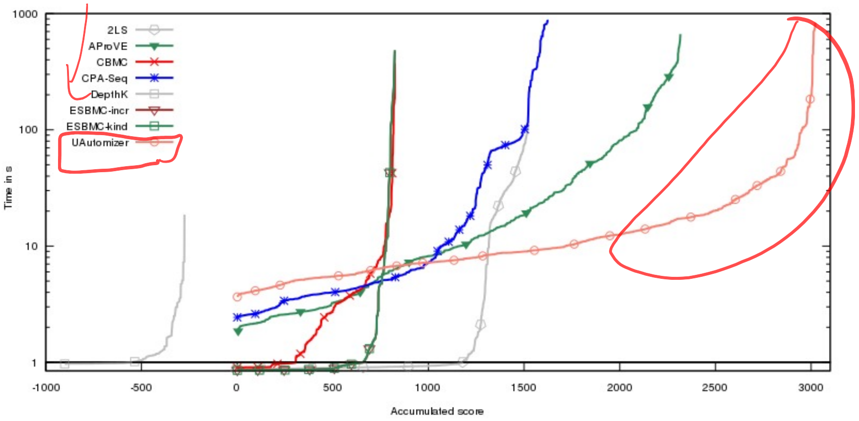
Computer scientist Andrey Rybalchenko has developed a new method for finding software bugs. Traditional automated testing systems detect when programs do "bad things" that lead to crashes, forcing the program to quit. By focusing on crashes, however, such testing often misses a significant class of bugs–those that allow the software to keep running but leave it unable to accept new input or do anything useful. In essence, Rybalchenko instead tries to identify when a program is doing "good things," such as making progress through loops or responding to other programs.

In a collaboration with Microsoft that began in 2006, Rybalchenko incorporated his methods into Terminator, a commercial program used to find bugs in the device drivers that mediate between an operating system and various pieces of hardware. Countless device drivers have been created by third-party developers, and they are often responsible for software failures that users blame on the OS. So detecting these bugs improves both actual and perceived OS reliability.

Terminator tools: starte-of-the-art

SV-COMP: Intl. Competition on Software Verification held at TACAS 2018

- the competition of software-verification tools, which will take place at TACAS
- no widely distributed benchmark suite of verification tasks
- most concepts are only validated in research prototypes.
- Goal of the competition: Provide a snapshot of the state-of-the-art in software verification to the community
- Increase the visibility and credits that tool developers
- Establish a set of benchmarks for software verification in the community.


1000
100
10
1
Time in s
-1000
-500
0
500
1000
1500
2000
2500
3000
Accumulated score
2LS
AProVE
CBMC
CPA-Seq
DepthK
ESBMC-incr
ESBMC-kind
UAutomizer

Terminator: starte-of-the-art tools

Büchi automaa

- UAutomizer: based on
- AProVE: based on reduction to term rewritting system
- Terminator: based on transition invariants
- T2, CPA-Seq: based on transition invariants

Program Behaviours

- Does the program terminates?
- Is the program safe (buffer overflow, zero pointer, deadlock, mutual exculsion?
- Is the protocol safe (same ip property in IEEE, ip4 protocol)?

Program Behaviours

Amir Pnueli (1941-2009)

- He studied mathematics at the Technion during 1958-1962
- He continued directly to PhD studies in the Weizmann Institute of Science in Israel
- During 1967 and 1968, postdoc at Stanford University and at IBM research center in Yorktown Heights, New York
- During a sabbatical at the University of Pennsylvania he was introduced to the work of the philosopher Arthur Prior

Arthur Prior: Past, Present, and Future in 1967

Arthur Prior (1914-1969)

Prior, born in New Zealand, introduced tense logic (Past, Present, and Future):

$$\varphi ::= a \mid \neg\varphi \mid \varphi \wedge \varphi \mid G\varphi \mid F\varphi \mid P\varphi \mid H\varphi$$

Consider the statement "I am hungry". It maybe true today, but false tomorrow.

Program Behaviours

Amir was the first to realize the potential implications of applying Prior's work to computer programs.
Crux: Need to specify ongoing behavior rather than input/output relation!

- Amir's 1977 seminal paper "The Temporal Logic of Programs"
- revolutionized the way computer programs are analyzed

"In mathematics, logic is static. It deals with connections among entities that exist in the same time frame. When one designs a dynamic computer system that has to react to ever changing conditions, ... one cannot design the system based on a static view. It is necessary to characterize and describe dynamic behaviors that connect entities, events, and reactions at different time points. Temporal Logic deals therefore with a dynamic view of the world that evolves over time."

Program Behaviours

Definition (The Temporal Logic of Programs)

Crux: Need to specify ongoing behavior rather than input/output relation!
"Temporal logic to the rescue" [Pnueli, 1977]:

- Linear temporal logic (LTL) as a logic for the specification of non-terminating programs
- Model checking via reduction to MSO

In 1996, Pnueli received the Turing Award for seminal work introducing temporal logic into computing science and for outstanding contributions to program and systems verification.

Mutual Exclusion

Definition (Examples)

- always not (CS1 and CS2): safety
- always (Request implies eventually Grant): liveness
- always (Request implies (Request until Grant)): liveness

Model Checking LTL Properties

- the most efficient algorithm for checking LTL formulae is based on

Büchi automata

Model Checking

MIT

Clarke and his student E. Allen Emerson saw an important possibility in temporal logic: it could be directly checked by machine.

- E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic, In: Proceedings of the Workshop on Logics of Programs, vol. 131 of LNCS, pages 52-71. Springer-Verlag, 1981.
- used to synthesize abstractions of concurrent programs
- model checking presented as a secondary result.
- Queille, J. P.; Sifakis, J. (1982), "Specification and verification of concurrent systems in CESAR", International Symposium on Programming
- Working independently, Jean-Pierre Queille and Joseph Sifakis developed similar ideas

Model Checking

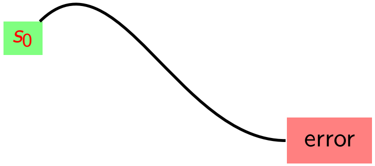
Turing Award 2007

"does a program behave as intended?"

- mathematical model M (e.g., Kripke structure, transition system)
- specification φ
- automatic proof or refutation of:

$$M \vDash \varphi$$

- Example: same ip, no arithmetic overflow...



Model Checking

Turing Award 2007

Model checking is an automatic technique that, given a finite state model of the system and a formal property, systematically checks whether such a property holds for (a given state in) that model.

- widely applicable: hardware, software, protocols
- potential "push-button" technology: software tools
- rapidly increasing industrial interest
- counterexamples for property violation
- sound and interesting mathematically foundations

The state space explosion

- application to practical systems was severely limited: the number of states to be explored.
- the number of states a memory location can assume is much
- From the literature, McMillan found an efficient encoding: BDD
- Symbolic model checker

Kenneth L. McMillan, Bell Labs, Cadence Berkeley Laboratories, Microsoft Research: CAV award for a series of fundamental contributions resulting in significant advances in scalability of model checking tools.

Futurebus+ Cache Coherence Protocol Clarke Bell Lab. et al. 1995

The first industrial scale case study using model checking

- Edmund M. Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David E. Long, Kenneth L. McMillan, Linda A. Ness
- Futurebus+: bus architecture for high-performance computers
- Cache coherence protocol: insure consistency of data in hierarchical systems
- 2300 lines of SMV code
- challenge: model construction, property specification (CTL)
- hierarchical, nondeterminism, abstraction
- state explosion: largest configuration verified has 3 bus segments, 8 processors 10^{30} states
- find potential erros in the protocol

Some major techniques against the explosion

- symbolic algorithms (open-source BDD manipulation libraries such as CUDD)
- bounded model checking algorithm: unroll the system for a fixed number of steps and do the checking
- bisimulation reduction: reduce the system to its bisimulation quotient
- partial order reduction: reduce the number of independent interleavings of concurrent processes that need to be considered
- abstraction: prove the property on the simplified system
- CEGAR: Counterexample guided abstraction refinement
- learning

Probabilistic model checking

Probability is the core part for several systems and situations:

- randomized algorithms (exploited in protocols)
- reliability, performance
- probabilistic programming
- optimization
- system biology

IEEE Zeroconf protocol

- network protocol for address assignment
- new devices joining the network get a unique IP address
- no user interaction needed

IEEE Zeroconf protocol

1. randomly choose one of the 65 024 addresses available in the private B-class 169.254.0.0/16
2. Loop: as long as the number of sent probes is less than n
3. broadcast the probe message "who is using the chosen address"?
4. got a reply? Go to 1
5. no reply within $r > 0$ time units:
    - if n probes have been sent: use the address
    - otherwise go to 2

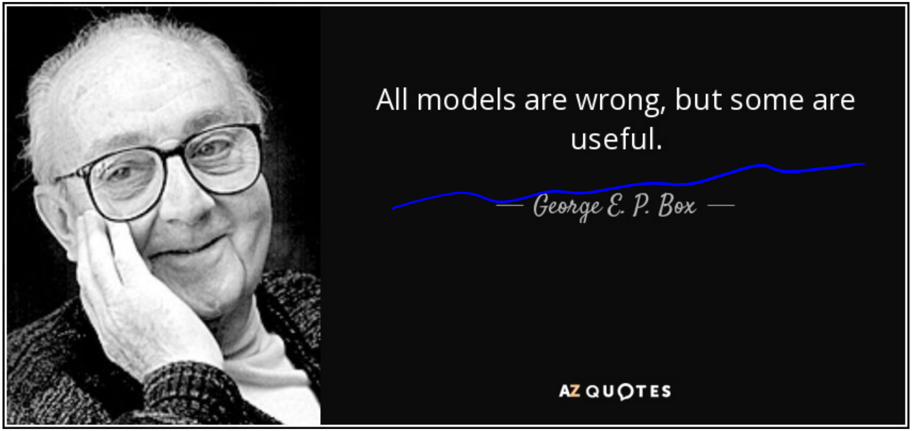
IEEE Zeroconf protocol

A simplified model for the Zeroconf protocol is:

$$\begin{tikzcd}
 & ok & & & & & err \\
{} \arrow[r] & s_0 \arrow[u, "1-q"'] \arrow[r, "q"] & s_1 \arrow[r, "p"] \arrow[l, bend left, "1-p"] & s_2 \arrow[r, "p"] \arrow[ll, bend left, "1-p"] & \cdots \arrow[r, "p"] \arrow[lll, bend left, "1-p"] & s_{n-1} \arrow[r, "p"] \arrow[llll, bend left, "1-p"] & s_n \arrow[u, "p"'] \arrow[lllll, bend left, "1-p"]
\end{tikzcd}$$

q: probability of choosing an address already in use,
$q = \frac{\#\text{devices}}{65024}$

p: probability of message loss


All models are wrong, but some are useful.

— George E. P. Box —

one can learn the model

Angluin-Style Exact Learning Framework

Angluin 1987

Büchi automata

Learning an automaton A using membership and equivalence query

Model Learning

Peled et al. Steffen et al. 2002

- SUL: System Under Learning
- Black box, active learning
- Assumption: we can bring it back to initial state
- Membership query is easy to answer
- Equivalence query: exploit conformance testing via test queries

Model Checking & Model Learning

Peled et al. 2002

- Goal: to check a system SUL satisfies a set of properties $\varphi_1, \ldots, \varphi_k$
- Learn M using model learning
- Equivalence query
  - M satisfies all φ_i: pass it through the conformance tester
  - otherwise: analyse counterexample (spurious, or real)

Compositional/AG verification

- Goal: to check a composed system $M \parallel M' \models \varphi$
- Divide & Conquer: find an abstraction A of M
- A preserves/abstracts M
- A should be <u>much smaller</u> than M
- check $A \parallel M' \models \varphi$ instead

Design learning algorithm to learn the abstraction A