Pratica e Teoria del Software Libero

Stefano Zacchiroli

Université Paris Diderot
Debian Project

28 Giugno 2013
Scuola Normale Superiore
Corso di Orientamento Universitario 2013
San Miniato (PI), Italy

Computer Science

> *. . . the core challenge for computing science is hence a conceptual one: what (abstract) mechanisms we can conceive without getting lost in complexities of our own making.* (Dijkstra, 1986)

Some sub-disciplines in computer science: (ACM, 2012)

- hardware
- computer system organization
- networks
- software and its engineering
- theory of computation
- mathematics of computing
- information systems
- security and privacy
- human-centered computing
- computing methodologies
- applied computing
- . . .

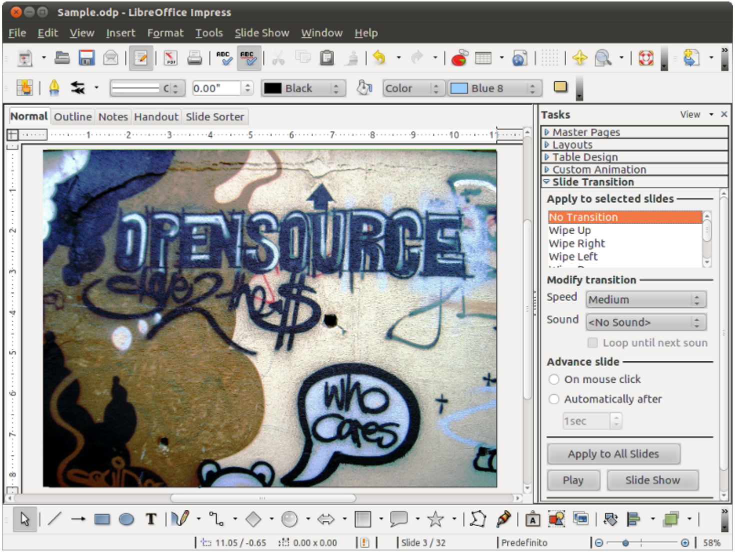
Computer Science

> *... the core challenge for computing science is hence a conceptual one: what (abstract) mechanisms we can conceive without getting lost in complexities of our own making.* *(Dijkstra, 1986)*

Some sub-disciplines in computer science: (ACM, 2012)

- hardware
- computer system organization
- networks
- software and its engineering
- theory of computation
- mathematics of computing
- information systems
- security and privacy
- human-centered computing
- computing methodologies
- applied computing
- ...

Software — for users



Software — for developers

for human consumption: source code

```
// beta distribution probability density function
double ScInterpreter::GetBetaDistPDF(double fX, double fA, double fB) {
    // special cases
    if (fA == 1.0) { // result b*(1-x)^(b-1)
        if (fB == 1.0)
            return 1.0;
        if (fB == 2.0)
            return -2.0*fX + 2.0;
        if (fX == 1.0 && fB < 1.0) {
            SetError(errIllegalArgument);
            return HUGE_VAL;
        }
        if (fX <= 0.01)
            return fB + fB * ::rtl::math::expm1((fB-1.0) * ::rtl::math::log1p(-fX));
        else
            return fB * pow(0.5-fX+0.5,fB-1.0);
    } if (fB == 1.0) // result a*x^(a-1) {
    ...
```

for machine consumption: binary program

```
400730: 53                push   %rbx
400731: e8 ca ff ff ff    callq  400700
400736: e8 e5 ff ff ff    callq  400720
40073b: 89 c3             mov    %eax,%ebx
40073d: 31 c0             xor    %eax,%eax
40073f: e8 ac ff ff ff    callq  4006f0
400744: 89 d8             mov    %ebx,%eax
400746: 5b                pop    %rbx
400747: c3                retq
400748: 31 ed             xor    %ebp,%ebp
40074a: 49 89 d1          mov    %rdx,%r9
40074d: 5e                pop    %rsi
40074e: 48 89 e2          mov    %rsp,%rdx
400751: 48 83 e4 f0       and    $0xfffffff0,%rsp
400755: 50                push   %rax
...
```

Free Software

free software (as in "free beer") software = software that has not (yet) to be payed

Free Software (as in "free speech") software = software that offers four freedoms to its users: (Stallman, 1986)

0. to run the program, for any purpose
1. to study how the program works, and change it
   (you need the source code for this)
2. to redistribute copies
3. to improve the program, and release improvements

(there are also obligations, which vary according to the license: BSD, GPL, LGPL, AGPL, . . .)

Why bother? — as citizens

Lester picked up a screwdriver. "You see this? It's a tool. You can pick it up and you can unscrew stuff or screw stuff in. You can use the handle for a hammer. You can use the blade to open paint cans. You can throw it away, loan it out, or paint it purple and frame it." He thumped the printer. "This [Disney in a Box] thing is a tool, too, but it's not your tool. It belongs to someone else — Disney. It isn't interested in listening to you or obeying you. It doesn't want to give you more control over your life." [. . .]

"If you don't control your life, you're miserable. Think of the people who don't get to run their own lives: prisoners, reform-school kids, mental patients. There's something inherently awful about living like that. Autonomy makes us happy."

— *Cory Doctorow, Makers*
http://craphound.com/makers/

Why bother? — as computer scientists

Free Software has *radically* changed the way software is:

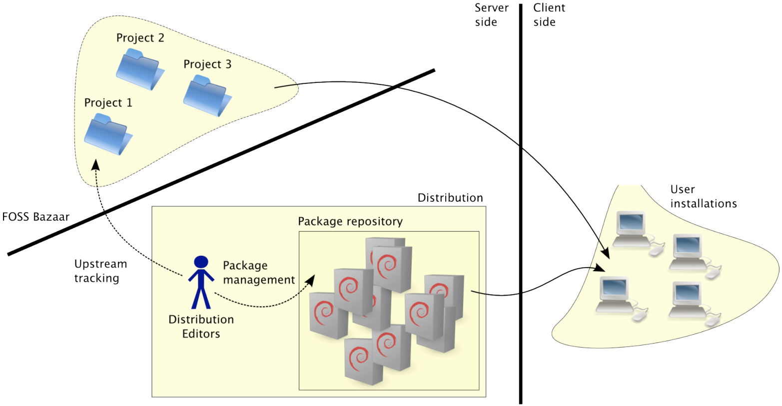
- developed
- tested
- proven
- conceived
- marketed
- sold
- maintained
- taught
- deployed
- ...

Free Software, raw

LibreOffice is cool, let's install it!

1. download `libreoffice_4.0.4.orig.tar.xz`
   - checksum mismatch, missing public key, etc.
2. ./configure
   - error: missing bar, baz, ...
3. foreach (bar, baz, ...) go to 1
   until (recursive) success
4. make
   - error: symbol not found
5. make install
   - error: cp: cannot create regular file /some/weird/path

Free Software, à point: distributions



- ease software life-cycle management
- key notion: the package abstraction (≈ "app")
- offer coherent software collections
- killer application: package managers (≈ "app stores")

Debian

- completely Free Software
- 35'000+ packages
  one of the largest (Free) software collection in existence
- 12 hardware architectures
- developed since 1993 by 1'000+ volunteers world-wide
- base for ≈140 other active distributions (47% of the total)

Recent highlights

- most popular GNU/Linux on the Web (32.7%) overall; including derivatives ≈2 Web server out of 10 are based on Debian
  — `w3techs.com`, March 2013
- powers: the International Station (NASA, 2013), Google's public cloud (Google, 2013), etc.

Packages, metadata, installation

package = { some files, some scripts, metadata }

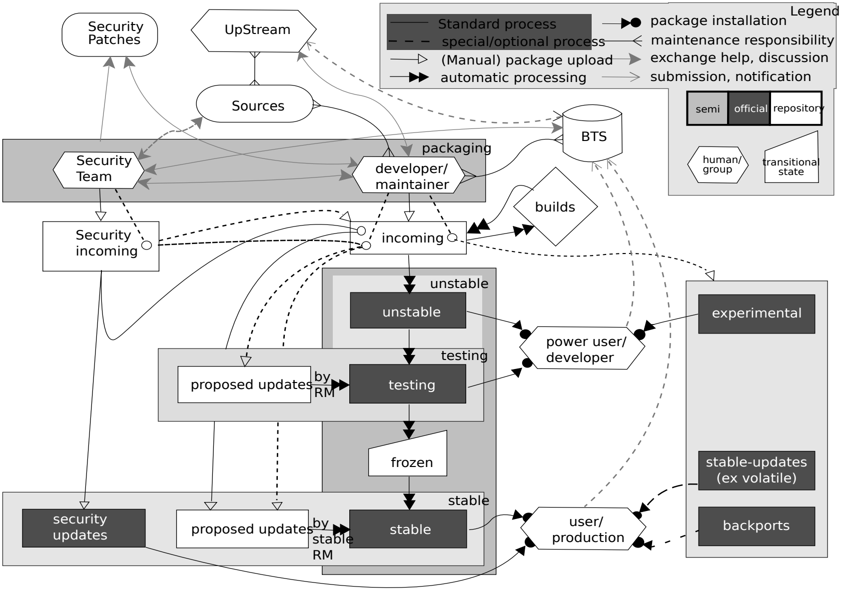
- identification
- inter-package relations
  - dependencies
  - conflicts
- feature declarations
- other (for humans)
  - package maintainer
  - textual descriptions
  - ...

Example (pacakge metadata)

```
Package: aterm
Version: 0.4.2-11
Section: x11
Installed-Size: 280
Maintainer: Göran Weinholt ...
Architecture: i386
Depends: libc6 (>= 2.3.2.ds1-4),
  libice6 | xlibs (> 4.1.0), ...
Conflicts: suidmanager (< 0.50)
Provides: x-terminal-emulator
...
```

A package is the elemental component of modern distribution systems (not FOSS-specific). A working system is deployed by installing a package set (2'000+ for modern FOSS distros)

The difficult life of distribution maintainers



The difficult life of distribution maintainers (cont.)

A distribution maintainer controls the evolution of a distribution by regulating the flow of new packages into / old packages out of it.

With 35'000+ packages, and 3'000+ new versions / month, we need efficient, (semi-)automatic tools to help answering Quality Assurance questions like:

- which packages are in the distro I am releasing?
- which packages block the installation of many other packages?
- what are the most depended upon packages?
- which non-installable packages can only be fixed by changing them (as opposed to changing other packages in the repo)?
- which future version changes will "break" the most packages in the distro?
- . . .

The difficult life of distribution maintainers (cont.)

A distribution maintainer controls the evolution of a distribution by regulating the flow of new packages into / old packages out of it.

With 35'000+ packages, and 3'000+ new versions / month, we need efficient, (semi-)automatic tools to help answering Quality Assurance questions like:

- which packages are not installable in the distro I am releasing?
- which packages block the installation of many other packages?
- what are the most depended upon packages?
- which non-installable packages can only be fixed by changing them (as opposed to changing other packages in the repo)?
- which future version changes will "break" the most packages in the distro?
- ...

The difficult life of distribution maintainers (cont.)

A distribution maintainer controls the evolution of a distribution by regulating the flow of new packages into / old packages out of it.

With 35'000+ packages, and 3'000+ new versions / month, we need efficient, (semi-)automatic tools to help answering Quality Assurance questions like:

- which packages are not installable in the distro I am releasing?
- which packages block the installation of many other packages?
- what are the most depended upon packages?
- which non-installable packages can only be fixed by changing them (as opposed to changing other packages in the repo)?
- which future version changes will "break" the most packages in the distro?
- ...

Developing a formal model for packages

- Before developing tools, or even only thinking about algorithms, we need a clear mathematical model of the problem!
- Once we have a model we can profit from existing knowledge (and existing tools) for the chosen formalisms.

so let's digress a bit. . .

Developing a formal model for packages

- Before developing tools, or even only thinking about algorithms, we need a clear mathematical model of the problem!
- Once we have a model we can profit from existing knowledge (and existing tools) for the chosen formalisms.

so let's digress a bit...

Propositional logic — syntax

Let $P = p, q, r, \ldots$ be a set of (atomic) propositions. Over an alphabet of symbols $= P \cup \{\neg,), \vee, \wedge, (, \rightarrow\}$, we want to define a language to express simple logical statements.

Definition (well-formed formula)

The set of well-formed formulae (WFF) of propositional logic is the *smallest set* such that:

- $P \subseteq$ WFF
- if $F \in$ WFF then $(\neg F) \in$ WFF
- if $F, G \in$ WFF then:
  - $(F \wedge G) \in$ WFF
  - $(F \vee G) \in$ WFF
  - $(F \rightarrow G) \in$ WFF

We omit parentheses according to conventional precedence rules, e.g.:

$$p \rightarrow q \wedge \neg r \vee s \quad \text{is the same of} \quad p \rightarrow ((q \wedge (\neg r)) \vee s)$$

Propositional logic — truth assignments

Definition (truth assignment)

A truth assignment is a function mapping propositions to either T (true) or F (false).

We can canonically represent a truth assignment as the set of propositions mapped to T.

Example

$\{p, q\} \in 2^P$ is the truth assignment in which p and q are mapped to T, and everything else $(r, s, t, \ldots)$ is mapped to F.

Propositional logic — semantics (cont.)

Definition (semantics of propositional logic)

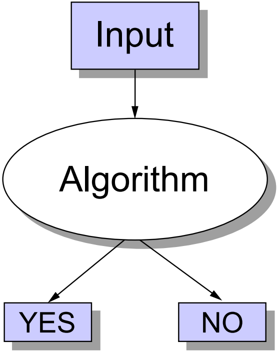
The semantics of propositional logic is a relation $\models$ between the set of all truth assignments 2^P and WFF, i.e. $\models \subseteq 2^P \times \text{WFF}$, defined inductively as the smallest set s.t.:

- $A \models p$ if $p \in A$
- $A \models \neg F$ if $A \not\models F$
- $A \models F \wedge G$ if $A \models F$ and $A \models G$
- $A \models F \vee G$ if either $A \models F$ or $A \models G$
- $A \models F \rightarrow G$ if when $A \models F$ it also holds $A \models G$

Example

- $\{p, q\} \models p \wedge q$ — read as "$\{p, q\}$ is a model of $p \wedge q$"
- $\{p\} \models p \vee q$ — "$\{p\}$ is a model of $p \vee q$"
- $\{q\} \not\models (p \vee q) \rightarrow p$ — "$\{q\}$ is not a model of $(p \vee q) \rightarrow p$"

Some decision problems in propositional logic



GFDL, `http://en.wikipedia.org/wiki/File:Decision_Problem.svg`

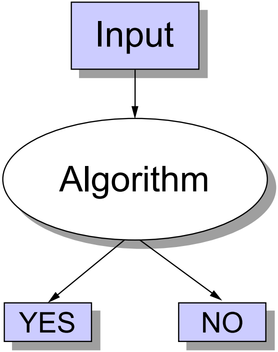
Evaluation: given as input $F \in WFF$ and $A \in 2^P$ tells whether $A \models F$ or not

- intuitively, this is easy: we "propagate" the truth value from sub-formulae to larger formulae, starting from propositions
- this is what digital electronic circuits do

Satisfiability (SAT): given as input $F \in WFF$ tells whether $\exists A$ such that $A \models F$

- how hard is this?

Some decision problems in propositional logic



GFDL, http://en.wikipedia.org/wiki/File:Decision_Problem.svg

Evaluation: given as input $F \in WFF$ and $A \in 2^P$ tells whether $A \models F$ or not

- intuitively, this is easy: we "propagate" the truth value from sub-formulae to larger formulae, starting from propositions
- this is what digital electronic circuits do

Satisfiability (SAT): given as input $F \in WFF$ tells whether $\exists A$ such that $A \models F$

- how hard is this?

On the complexity of decision problems

- some decision problems are impossible to solve (for a machine, in the general case), they are called undecidable problems, e.g.:

Halting problem

> *Given the description of an arbitrary program and a finite input, decide whether the program finishes running or will run forever* (Turing, 1936)

- other decision problems can be solved efficiently, i.e. given an input of size n, they can be solved using an amount of time which is at most polynomial in it (n, n^2, n^3, ...). These are called polynomial time (P) problems
  - e.g. evaluation of propositional logic formulae

On the complexity of decision problems

- some decision problems are impossible to solve (for a machine, in the general case), they are called undecidable problems, e.g.:

Halting problem

Given the description of an arbitrary program and a finite input, decide whether the program finishes running or will run forever *(Turing, 1936)*

- other decision problems can be solved efficiently, i.e. given an input of size n, they can be solved using an amount of time which is at most polynomial in it (n, n^2, n^3, ...). These are called polynomial time (P) problems
  - e.g. evaluation of propositional logic formulae

On the complexity of decision problems (cont.)

Some decision problems can be solved, but (at present) not efficiently, i.e. they can be solved using an amount of time which is exponential in the size of the input. These are called (non-deterministic polynomial) NP-complete problems

- the above apply to *finding* a solution; *verifying* that a solution is correct can be done efficiently

NP-complete problems enjoy an interesting property: if you can efficiently solve *one* of them, then *all* can be solved efficiently

- (and our digital world will crumble)

On the complexity of SAT

Example (NP-complete decision problem)

satisfiability of proposition logic (SAT) is NP-complete (Cook, 1971)

i.e. no known algorithm can efficiently solve all instances of SAT. In the worst case scenario they will be as efficient as:

```
sat(F) =                              // brute force approach
  P = atomic_propositions(F)
  A = truth_assignments(P)        // there are 2^|P| of these
  foreach a ∈ A do
    if eval(F, A) then           // this step is polynomial
      return T
  return F
```

SAT solvers are specialized software that give answers to SAT decision problems (as) efficiently (as possible).

Packages, repositories, installation

Let's go back to packages...

```
Package: aterm
Version: 0.4.2-11
Section: x11
Installed-Size: 280
Maintainer: Göran Weinholt ...
Architecture: i386
Depends: libc6 (>= 2.3.2.ds1-4),
  libice6 | xlibs (> 4.1.0), ...
Conflicts: suidmanager (< 0.50)
Provides: x-terminal-emulator
...
```

- a repository R is a set of packages where no two packages have the same name and version
- an installation $I \subseteq R$ is a set of packages s.t.:
  - abundance: $\forall p \in I$, the dependencies of p are satisfied
  - peace: $\forall p \in I$, the conflicts of p are *not* satisfied

formal details in (Di Cosmo et al. 2006)

Modeling packages (w/o versions)

- package = proposition
  - intuition: T for installed packages
  - e.g. `"Package: python"` becomes p
- dependencies: $p \rightarrow \phi$ where ϕ is a *positive* formula
  - e.g. `"Depends: foo, bar | qux"` becomes $p \rightarrow f \wedge (b \vee q)$
- conflicts between 2 packages: $\neg(p \wedge q)$
  - e.g. `"Conflicts: q"` becomes $\neg(p \wedge q)$
- package not available (but mentioned) in repository: $\neg p$

Intuition: installation = model

Theorem

p is installable in repository $R \iff T_R \wedge p$ is satisfiable

where T_r is obtained by applying the above translation to all packages in R and $\wedge$-ing together the obtained formulae

Installability as SAT — example (w/o versions)

Repository R

Package:	a	Package:	b
Depends:	b | d	Conflicts:	d
Package:	c	Package:	d
Depends:	d | e	Conflicts:	c

- $T_R = (a \rightarrow b \vee d) \wedge \neg(b \wedge d) \wedge (c \rightarrow d \vee e) \wedge \neg(c \wedge d) \wedge \neg e$
- package a is installable in $\iff$ $T_R \wedge a$ is satisfiable
- $\{a, d\} \models T_r \wedge a$ therefore package a is installable
  - corollary: {a, d} is an installation of R
  - there might be other installations containing a, e.g. {a, b}

Exercise

Prove that package c is not installable in repository R.

Modeling versions

To also capture package versions we need the following changes:

1. use atomic propositions to stand for pairs ⟨name, version⟩
2. before applying the translation, we perform an expansion phase on the repository:

   > replace every package name with version constraint (e.g. $p\ (\geq 3)$) by the disjunction ($\vee$) of all package versions that satisfy the constraint (e.g. $p_3 \vee p_4 \vee p_7$)

   This way we don't have to care about versions and comparisons in the logical model
3. add implicit conflicts between different versions of the same package[1]

[1] this is a common requirement in Debian and other packaging systems

Installability as SAT — complete example

Install libc6 version 2.3.2.ds1-22 in

```
Package: libc6
Version: 2.2.5-11.8

Package: libc6
Version: 2.3.5-3

Package: libc6
Version: 2.3.2.ds1-22
Depends: libdb1-compat

Package: libdb1-compat
Version: 2.1.3-8
Depends: libc6 (>=
2.3.5-1)

Package: libdb1-compat
Version: 2.1.3-7
Depends: libc6 (>=
2.2.5-13)
```

$\Rightarrow$

$$\begin{aligned}
&\texttt{libc6}_{2.3.2.ds1-22}\\
&\wedge\\
&\neg(\texttt{libc6}_{2.3.2.ds1-22} \wedge \texttt{libc6}_{2.2.5-11.8})\\
&\wedge\\
&\neg(\texttt{libc6}_{2.3.2.ds1-22} \wedge \texttt{libc6}_{2.3.5-3})\\
&\wedge\\
&\neg(\texttt{libc6}_{2.3.5-3} \wedge \texttt{libc6}_{2.2.5-11.8})\\
&\wedge\\
&\neg(\texttt{libdb1-compat}_{2.1.3-7} \wedge \texttt{libdb1-compat}_{2.1.3-8})\\
&\wedge\\
&\texttt{libc6}_{2.3.2.ds1-22} \rightarrow\\
&(\texttt{libdb1-compat}_{2.1.3-7} \vee \texttt{libdb1-compat}_{2.1.3-8})\\
&\wedge\\
&\texttt{libdb1-compat}_{2.1.3-7} \rightarrow\\
&(\texttt{libc6}_{2.3.2.ds1-22} \vee \texttt{libc6}_{2.3.5-3})\\
&\wedge\\
&\texttt{libdb1-compat}_{2.1.3-8} \rightarrow \texttt{libc6}_{2.3.5-3}
\end{aligned}$$

How hard are package installation problems?

We have translated package installability into SAT, therefore:

- we can now use a SAT solver to check installability (practice)
- we know installability is not harder than SAT (theory)

But is installability *easier* than SAT?

Theorem

Package (co-)installability is NP-complete.

Proof technique

Mapping of general SAT instances (in 3-SAT form) to package installation problems. (Di Cosmo et al. 2006)

How hard are package installation problems?

We have translated package installability into SAT, therefore:

- we can now use a SAT solver to check installability (practice)
- we know installability is not harder than SAT (theory)

But is installability *easier* than SAT?

Theorem

Package (co-)installability is NP-complete.

Proof technique

Mapping of general SAT instances (in 3-SAT form) to package installation problems. (Di Cosmo et al. 2006)

Practical complexity

Solving an NP-complete problem?

- Checking installability is NP-complete, but recent SAT solvers are able to handle easily current instances.
- In practice: explicit conflicts between packages are not very frequent (but they are crucial when they exist!)
- When checking installability wrt a single repository: only one version per package (except rare exceptions), hence no implicit conflicts.

A practical tool

- The `dose-distcheck` tool (Vouillon, 2006) checks installability of *all* packages wrt a repository of 35'000+ packages in a few seconds on commodity desktop hardware.
- Today integrated in the dose3 library (Abate, Zacchiroli et al. `http://www.mancoosi.org/software/`).

dose-debcheck — example


```
package: libgnuradio-dev
version: 3.2.2.dfsg-1
architecture: all
source: gnuradio (= 3.2.2.dfsg-1)
status: broken
reasons:
   -
    missing:
     pkg:
      package: libgruel0
      version: 3.2.2.dfsg-1+b1
      architecture: amd64
      unsat-dependency: libboost-thread1.40.0 (>= 1.40.0-1)
     depchains:
      -
       depchain:
        -
         package: libgnuradio-dev
         version: 3.2.2.dfsg-1
         architecture: all
         depends: libgnuradio (= 3.2.2.dfsg-1)
        -
         package: libgnuradio
         version: 3.2.2.dfsg-1
         architecture: all
         depends: libgnuradio-core0
        -
         package: libgnuradio-core0
         version: 3.2.2.dfsg-1+b1
         architecture: amd64
         depends: libgruel0 (= 3.2.2.dfsg-1+b1)
```

Usage in Debian

http://edos.debian.net/weather/

Usage in Debian (cont.)

- Verify installability of packages before uploading them to the archive
  - used daily by Emdebian
- Check that build-dependencies are satisfiable before attempting a package build
  - “life changing” for porters (quote)
- Generate test cases for finding errors occurring during package installation
  - file conflicts, by Ralf Treinen
  - `http://edos.debian.net/file-overwrites/`

Dijkstra
On a cultural gap
The Mathematical Intelligencer 8 (1986), 1:48–52 http://www.cs.utexas.edu/~EWD/transcriptions/EWD09xx/EWD924.html

Association for Computing Machinery
The 2012 ACM Computing Classification System
http://www.acm.org/about/class/2012

Turing
On computable numbers, with an application to the Entscheidungsproblem
http://www.turingarchive.org/browse.php/B/12, 1936

Cook
The complexity of theorem-proving procedures
ACM symposium on Theory of computing, 1971.

Di Cosmo et al.
Managing the complexity of large free and open source package-based software distributions.
ASE 2006: Automated Software Engineering.

Di Cosmo, Treinen, Zacchiroli
Formal Aspects of Free and Open Source Software Components
FMCO 2012, http://upsilon.cc/~zack/research/publications/fmco2012-foss-components.pdf

Looking back: our approach

- start from a CS discipline: software engineering
- observation of its social and technical ramifications
  - in particular: of Free Software
- analysis of some existing issue
  - in particular: QA in complex component-based systems
- development of a formal model that capture the relevant parts of the issue
- theory: upper and lower bounds to algorithmic complexity
- practice: tool development
- communication to promote the tools to the relevant public

Thanks!

Questions?

Stefano Zacchiroli
zack@upsilon.cc

http://upsilon.cc/zack
http://identi.ca/zack

about the slides:

available at	https://gitorious.org/zacchiro/talks/trees/master/2013/20130628-normale
copyright © 2010–2013	Stefano Zacchiroli
license	CC BY-SA 3.0 — Creative Commons Attribution-ShareAlike 3.0