Large-scale Modeling, Analysis, and Preservation of Free and Open Source Software

Stefano Zacchiroli

zack@upsilon.cc

IRIF, Université Paris Diderot

27 November 2017

Habilitation à Diriger des Recherches

Paris, France

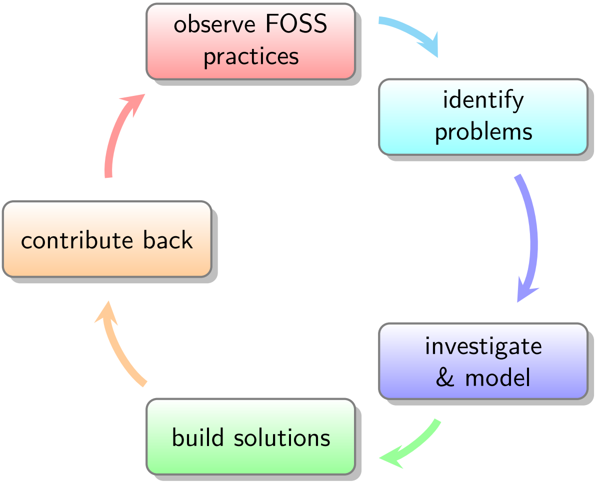
[Free] software is eating the world

- "*Every industrial company will become a software company*"

  — Jeff Immelt (General Electric CEO), 2013
- "*Every software company is an open source company*"

  — Mike Milinkovich, ICSE 2017 keynote

A virtuous cycle in empirical software engineering



Outline

1. Modeling FOSS package relationships
2. Beyond host boundaries
3. Back to the source (code)
4. Scaling to the entire software commons

Outline

1. Modeling FOSS package relationships

2. Beyond host boundaries

3. Back to the source (code)

4. Scaling to the entire software commons

Packages

package = files + configuration logic + metadata

```
Package: firefox-esr
Version: 52.5.0esr-1
Priority: optional
Section: web
Provides: gnome-www-browser, www-browser
Depends: libasound2 (>= 1.0.16), libatk1.0-0 (>= 1.12.4),
  libc6 (>= 2.17), libcairo-gobject2 (>= 1.10.0),
  libcairo2 (>= 1.10.0), libdbus-1-3 (>= 1.9.14),
  libdbus-glib-1-2 (>= 0.78), libevent-2.1-6 (>= 2.1.8),
  fontconfig, procps, [...]
Suggests: fonts-stix | otf-stix, fonts-lmodern, mozplugger, [...]
Conflicts: iceweasel (<< 45), j2re1.4, pango-graphite (<< 0.9.3)
Description: Mozilla Firefox web browser - Extended Support Release
  Firefox ESR is a powerful, extensible web browser with support for
  modern web application technologies.
```

Problem

How do you efficiently check if every package is potentially installable when you have tens of thousands of them?

Packages

package = files + configuration logic + metadata

```
Package: firefox-esr
Version: 52.5.0esr-1
Priority: optional
Section: web
Provides: gnome-www-browser, www-browser
Depends: libasound2 (>= 1.0.16), libatk1.0-0 (>= 1.12.4),
  libc6 (>= 2.17), libcairo-gobject2 (>= 1.10.0),
  libcairo2 (>= 1.10.0), libdbus-1-3 (>= 1.9.14),
  libdbus-glib-1-2 (>= 0.78), libevent-2.1-6 (>= 2.1.8),
  fontconfig, procps, [...]
Suggests: fonts-stix | otf-stix, fonts-lmodern, mozplugger, [...]
Conflicts: iceweasel (<< 45), j2re1.4, pango-graphite (<< 0.9.3)
Description: Mozilla Firefox web browser - Extended Support Release
  Firefox ESR is a powerful, extensible web browser with support for
  modern web application technologies.
```

Problem

How do you efficiently check if every package is potentially installable when you have tens of thousands of them?

Formal models for package relationships

Definition (package — concrete model)

A package (n, v, D, C) consists of

- a name $n \in \mathrm{N}$,
- a version $v \in \mathrm{V}$,
- a set of dependencies $D \subseteq \wp(\mathrm{N} \times \textsc{Con})$,
- a set of conflicts $C \subseteq \mathrm{N} \times \textsc{Con}$.

where $\textsc{Con} = \{\top, = v, > v, \leq v, \ldots\}$

Definition (installation — concrete model)

Let R be a repository. An R-installation is a set of packages $I \subseteq R$ such that $\forall p, q \in I$:

- abundance for each element $d \in p.D$ there exists $(n, c) \in d$ and a package $q \in I$ such that $q \in [[(n, c)]]_R$.
- peace for each $(n, c) \in p.C$: $I \cap [[(n, c)]]_R = \emptyset$
- flatness if $p \neq q$ then $p.n \neq q.n$

Formal models for package relationships (cont.)

Definition (repository — abstract model)

A repository consists of:

- a set of packages P,
- an anti-reflexive and symmetric conflict relation $C \subseteq P \times P$,
- a dependency function $D\colon P \longrightarrow \wp(\wp(P))$.

Theorem

(Co-)installability is NP-hard.

- abstract model result (EDOS, 2006)
- concrete model result (Mancoosi, 2012) for several component models: DEB, RPM, Eclipse plugins, OSGi bundles

Formal models for package relationships (cont.)

Definition (repository — abstract model)

A repository consists of:

- a set of packages P,
- an anti-reflexive and symmetric conflict relation $C \subseteq P \times P$,
- a dependency function $D\colon P \longrightarrow \wp(\wp(P))$.

Theorem

(Co-)installability is NP-hard.

- abstract model result (EDOS, 2006)
- concrete model result (Mancoosi, 2012) for several component models: DEB, RPM, Eclipse plugins, OSGi bundles

Package installability as SAT instance

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)
```

$\Rightarrow$

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

Adoption

- Debian: QA to mass-check the archive for installability issues
- Debian: pre-build checks for `buildd` autobuilders (fail-fast)

Package installability as SAT instance

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)
```

$\Rightarrow$

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

Adoption

- Debian: QA to mass-check the archive for installability issues
- Debian: pre-build checks for `buildd` autobuilders (fail-fast)

The upgrade problem

Definition (Dependency solving problem)

A *dependency solving problem* (AKA upgrade problem) as faced by packages managers, consists of:

1. repository R of all available packages (package universe)
2. $S \subseteq R$ denoting currently installed packages (package status)
3. user request U, asking to install/upgrade/remove packages

Desired output: new package status $S' \subseteq R$ s.t.

- S' is an installation,
- S' satisfies U.

Remarks

- the upgrade problem is harder than installability
- not all valid solutions are born equal

The upgrade problem

Definition (Dependency solving problem)

A *dependency solving problem* (AKA upgrade problem) as faced by packages managers, consists of:

1. repository R of all available packages (package universe)
2. $S \subseteq R$ denoting currently installed packages (package status)
3. user request U, asking to install/upgrade/remove packages

Desired output: new package status $S' \subseteq R$ s.t.

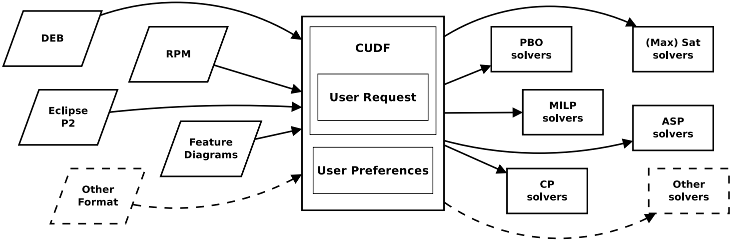
- S' is an installation,
- S' satisfies U.

Remarks

- the upgrade problem is harder than installability
- not all valid solutions are born equal

The Common Upgradeability Description Format (CUDF)

CUDF : a standard language to express upgrade problems across different component ecosystems, allowing to attack dependency solving with a multitude of technologies and research techniques.



Requirements

- distribution agnostic
- solver agnostic
- extensible
- formal semantics
- plain text
- close to original

CUDF — example

```
preamble:
property:
  bugs: int = 0,
  suite: enum(stable, unstable) = "stable"

package: car
version: 1
depends: engine, wheel > 2, door, battery <= 13
installed: true
bugs: 183

package: bicycle
version: 7
suite: unstable

package: gasoline-engine
version: 1
depends: turbo
provides: engine
conflicts: engine, gasoline-engine
installed: true

[...]

request:
install: bicycle, gasoline-engine = 1
upgrade: door, wheel > 3
```

- formal semantics based on the concrete package model
- type system for package properties
- plumbing for expressing user preferences (i.e., optimization criteria) that leverage package properties

libCUDF

- CUDF reference implementation
- native OCaml with C bindings

CUDF — example

```
preamble:
property:
  bugs: int = 0,
  suite: enum(stable, unstable) = "stable"

package: car
version: 1
depends: engine, wheel > 2, door, battery <= 13
installed: true
bugs: 183

package: bicycle
version: 7
suite: unstable

package: gasoline-engine
version: 1
depends: turbo
provides: engine
conflicts: engine, gasoline-engine
installed: true

[...]

request:
install: bicycle, gasoline-engine = 1
upgrade: door, wheel > 3
```

- formal semantics based on the concrete package model
- type system for package properties
- plumbing for expressing user preferences (i.e., optimization criteria) that leverage package properties

libCUDF

- CUDF reference implementation
- native OCaml with C bindings

CUDF — example

```
preamble:
property:
  bugs: int = 0,
  suite: enum(stable, unstable) = "stable"

package: car
version: 1
depends: engine, wheel > 2, door, battery <= 13
installed: true
bugs: 183

package: bicycle
version: 7
suite: unstable

package: gasoline-engine
version: 1
depends: turbo
provides: engine
conflicts: engine, gasoline-engine
installed: true

[...]

request:
install: bicycle, gasoline-engine = 1
upgrade: door, wheel > 3
```

- formal semantics based on the concrete package model
- type system for package properties
- plumbing for expressing user preferences (i.e., optimization criteria) that leverage package properties

libCUDF

- CUDF reference implementation
- native OCaml with C bindings

The MISC competition

The Mancoosi International Solver Competition (MISC)

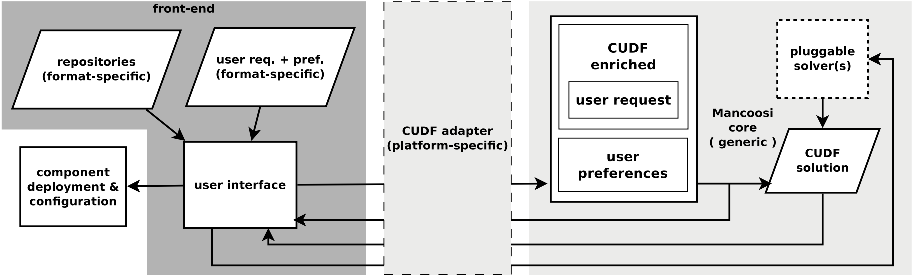
A gamification approach to grow a set of CUDF-compatible solvers, inspired by the SAT competition. Run successfully for 3 editions on a corpus of both real and synthetic upgrade problems.

Table: Sample of MISC competition entrants, ed. 2010 and 2011

name	**author**	**technique / solver**
apt-pbo	Trezentos	PBO[1]
aspcud	Matheis	ASP[2]
inesc	Lynce et. al	Max-SAT
p2cudf	Le Berre et. al	PBO / Sat4j
ucl	Gutierrez et al.	Graph constraints
unsa	Michel et. al	MILP[3] / CPLEX

[1] Pseudo-Boolean Optimization
[2] Answer Set Programming
[3] Mixed Integer Linear Programming

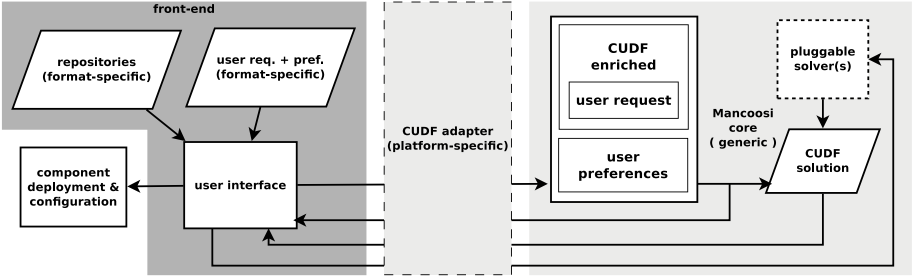
A modular package manager architecture



Adoption

- Eclipse P2: format to log/debug dependency solving issues
- Debian: apt-get optional bridge to CUDF solvers (e.g., aspcud)
- Opam: native package manager for the OCaml ecosystem

A modular package manager architecture



Adoption

- Eclipse P2: format to log/debug dependency solving issues
- Debian: apt-get optional bridge to CUDF solvers (e.g., aspcud)
- Opam: native package manager for the OCaml ecosystem

Outline

1. Modeling FOSS package relationships
2. Beyond host boundaries
3. Back to the source (code)
4. Scaling to the entire software commons

A package manager for the "cloud"?

To foster scalability and/or fault-tolerance, modern applications are usually deployed in production on multiple machines.

Example

Deploy a scalable Wordpress blog, with replicated frontend (web server) and backend (SQL database).

- package managers aren't up to the task due to locality assumptions and lack of a service layer
- can we design a component model suitable for automatically deploying networked and/or "cloud" applications?

A package manager for the “cloud”?

To foster scalability and/or fault-tolerance, modern applications are usually deployed in production on multiple machines.

Example

Deploy a scalable Wordpress blog, with replicated frontend (web server) and backend (SQL database).

- package managers aren’t up to the task due to locality assumptions and lack of a service layer
- can we design a component model suitable for automatically deploying networked and/or “cloud” applications?

Requirement #1: package installation

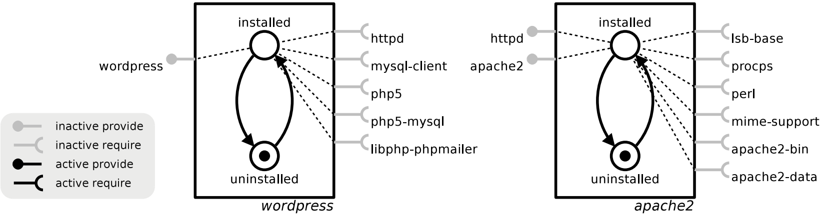


Figure: Available components, not installed

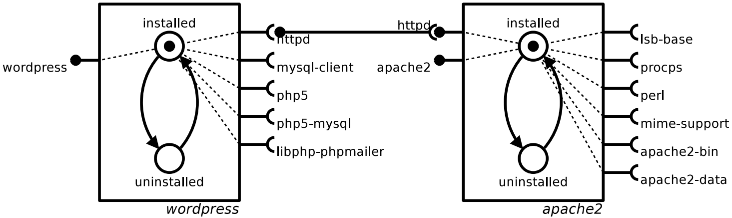
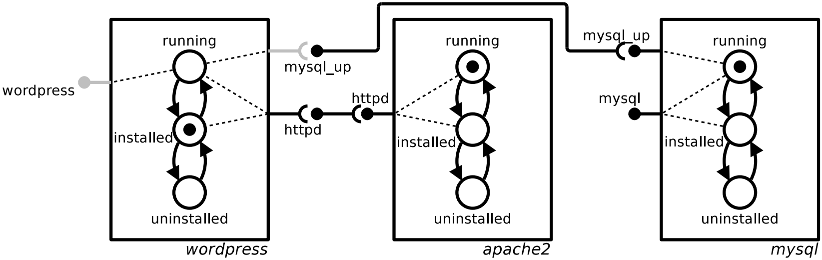
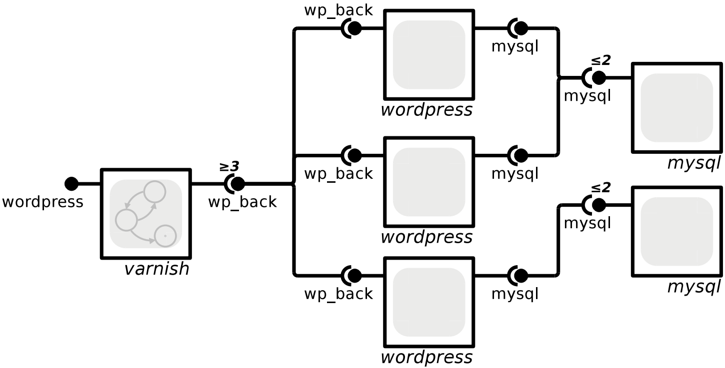


Figure: Installed components, bound together on the `httpd` port

Requirement #2: services and packages



Requirement #3: redundancy, capacity planning, conflicts



Requirement #4: provisioning

It should be possible to dynamically create and destroy components.

Use case: modeling up/down-scale to react to load changes.

The Aeolus component model

Definition (Component type)

The set Γ of component types of the Aeolus model, ranged over by $\mathcal{T}_1, \mathcal{T}_2, \ldots$ contains 5-ple $\langle Q, q_0, T, P, D\rangle$ where:

- Q is a finite set of states;
- $q_0 \in Q$ is the initial state and $T \subseteq Q \times Q$ is the set of transitions;
- $P = \langle \mathbf{P}, \mathbf{R} \rangle$, with $\mathbf{P}, \mathbf{R} \subseteq \mathcal{I}$, is a pair composed of the set of provide and the set of require ports, respectively;
- D is a function from Q to 2-ple in $(\mathbf{P} \nrightarrow \mathbb{N}_\infty) \times (\mathbf{R} \nrightarrow \mathbb{N}_0)$.

Definition (Configuration)

A configuration $\mathcal{C}$ is a 4-ple $\langle U, Z, S, B\rangle$ where:

- $U \subseteq \Gamma$ is the finite universe of all available component types;
- $Z \subseteq \mathcal{Z}$ is the set of the currently deployed components;
- S is the component state description [. . .];
- $B \subseteq \mathcal{I} \times Z \times Z$ is the set of bindings [. . .].

The Aeolus component model (cont.)

Definition (Configuration correctness)

[...] The configuration $\mathcal{C}$ is correct if for each component $z \in Z$, given $S(z) = \langle \mathcal{T}, q \rangle$ with $\mathcal{T} = \langle Q, q_0, T, P, D \rangle$ and $D(q) = \langle \mathcal{P}, \mathcal{R} \rangle$, we have that $(p \mapsto n_p) \in \mathcal{P}$ implies $\mathcal{C} \models_{prov} (z, p, n_p)$, and $(r \mapsto n_r) \in \mathcal{R}$ implies $\mathcal{C} \models_{req} (z, r, n_r)$.

Definition (Deployment actions)

The set $\mathcal{A}$ contains the following deployment actions:

- $stateChange(z, q_1, q_2)$ where $z \in \mathcal{Z}$;
- $bind(r, z_1, z_2)$ where $z_1, z_2 \in \mathcal{Z}$ and $r \in \mathcal{I}$;
- $unbind(r, z_1, z_2)$ where $z_1, z_2 \in \mathcal{Z}$ and $r \in \mathcal{I}$;
- $new(z : \mathcal{T})$ where $z \in \mathcal{Z}$ and $\mathcal{T}$ is a component type;
- $del(z)$ where $z \in \mathcal{Z}$.

The Aeolus component model (cont.)

Definition (Configuration correctness)

[...] The configuration $\mathcal{C}$ is correct if for each component $z \in Z$, given $S(z) = \langle \mathcal{T}, q \rangle$ with $\mathcal{T} = \langle Q, q_0, T, P, D \rangle$ and $D(q) = \langle \mathcal{P}, \mathcal{R} \rangle$, we have that $(p \mapsto n_p) \in \mathcal{P}$ implies $\mathcal{C} \models_{prov} (z, p, n_p)$, and $(r \mapsto n_r) \in \mathcal{R}$ implies $\mathcal{C} \models_{req} (z, r, n_r)$.

Definition (Deployment actions)

The set $\mathcal{A}$ contains the following deployment actions:

- $\mathit{stateChange}(z, q_1, q_2)$ where $z \in \mathcal{Z}$;
- $\mathit{bind}(r, z_1, z_2)$ where $z_1, z_2 \in \mathcal{Z}$ and $r \in \mathcal{I}$;
- $\mathit{unbind}(r, z_1, z_2)$ where $z_1, z_2 \in \mathcal{Z}$ and $r \in \mathcal{I}$;
- $\mathit{new}(z : \mathcal{T})$ where $z \in \mathcal{Z}$ and $\mathcal{T}$ is a component type;
- $\mathit{del}(z)$ where $z \in \mathcal{Z}$.

The Aeolus component model (cont.)

Definition (Reconfigurations)

Reconfigurations are denoted by transitions $\mathcal{C} \xrightarrow{\alpha} \mathcal{C}'$ meaning that the execution of $\alpha \in \mathcal{A}$ on the configuration $\mathcal{C}$ produces a new configuration $\mathcal{C}'$.

$$\mathcal{C} \xrightarrow{stateChange(z, q_1, q_2)} \langle U, Z, S', B \rangle$$

if $\mathcal{C}[z].\mathtt{state} = q_1$
and $(q_1, q_2) \in \mathcal{C}[z].\mathtt{trans}$
and $S'(z') = \begin{cases} \langle \mathcal{C}[z].\mathtt{type}, q_2 \rangle & \text{if } z' = z \\ \mathcal{C}[z'] & \text{otherwise} \end{cases}$

[. . .]

Definition (Deployment run)

A deployment run is a sequence $\alpha_1 \dots \alpha_m$ of actions such that there exist $\mathcal{C}_i$ such that $\mathcal{C} = \mathcal{C}_0$, $\mathcal{C}_{j-1} \xrightarrow{\alpha_j} \mathcal{C}_j$ for every $j \in \{1, \dots, m\}$, and the following conditions hold:

configuration correctness for every $i \in \{0, \dots, m\}$, $\mathcal{C}_i$ is correct;

[. . .]

Achievability

Definition (Achievability problem)

The achievability problem has as input a universe U of component types, a component type $\mathcal{T}$, and a target state q.
It returns **true** if there exists a deployment run $\alpha_1 \ldots \alpha_m$ such that $\langle U, \emptyset, \emptyset, \emptyset \rangle \xrightarrow{\alpha_1} \mathcal{C}_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_m} \mathcal{C}_m$ and $\mathcal{C}_m[z] = \langle \mathcal{T}, q \rangle$, for some component z in $\mathcal{C}_m$. Otherwise, it returns **false**.

Table: Decidability and complexity of achievability in (variants of) the Aeolus component model

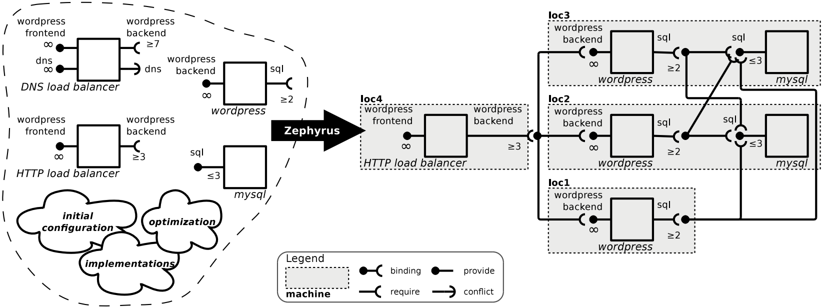
model	provides	requires	achievability
Aeolus	$\mathbb{N}_\infty$	$\mathbb{N}_0$	undecidable[4]
Aeolus core	$\{\infty\}$	$\{1, 0\}$	decidable, Ackermann-hard[5]
Aeolus$^-$	$\{\infty\}$	$\{1\}$	decidable, PTIME

[4] reduction from reachability in 2 counter machines

[5] reduction form coverability in reset Petri nets

Practical deployment planning

- contingency plan: split *stateless* provisioning from state change
- phase 1: Zephyrus



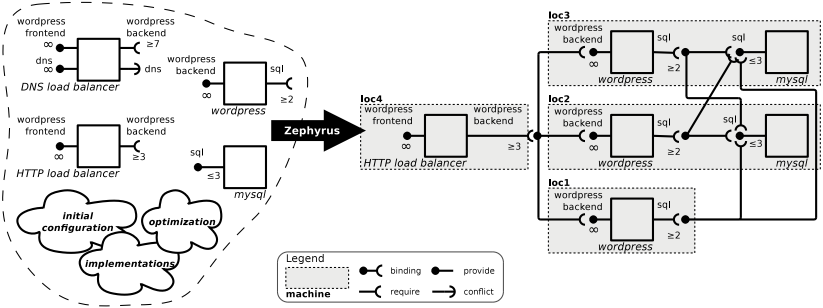
- phase 2: Metis (univ. of Bologna)

Adoption

- Mandriva, Kyriba

Practical deployment planning

- contingency plan: split *stateless* provisioning from state change
- phase 1: Zephyrus



- phase 2: Metis (univ. of Bologna)

Adoption

- Mandriva, Kyriba

Outline

1. Modeling FOSS package relationships
2. Beyond host boundaries
3. Back to the source (code)
4. Scaling to the entire software commons

Debsources in a nutshell

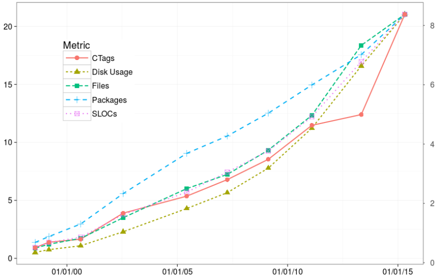
1. an infrastructure to publish Debian source code on the Web
2. a notable instance indexing *all* Debian source code to date

For developers:

- browse/search source code
- syntax highlighting
- pinpoint code lines, annotate

For researchers:

- Debian evolution over time
- 20+ years of FOSS history
- live change monitoring



Debsources for developers

Adoption

- quickly become a popular service among Debian developers
- frequently used on IRC to discuss source code snippets
- integrated with `codesearch.debian.net` and `tracker.debian.org`
- 15 code contributors; 5 interns

Debsources for developers

Adoption

- quickly become a popular service among Debian developers
- frequently used on IRC to discuss source code snippets
- integrated with `codesearch.debian.net` and `tracker.debian.org`
- 15 code contributors; 5 interns

Software evolution [in the large]

In software maintenance, software evolution refers to the process of repeatedly updating software, for various reasons, *after* initial development. FOSS distributions enabled a new scale of software evolution studies:

Software evolution in the large (Gonzalez-Barahona et. al, 2009)

The study of software evolution, at the scale of software collections, at the granularity they support (e.g., component release).

Pros

- relevant/popular software distribution model
- long lives (e.g., decades)
- uniform access to the history of contained software
- help with (researcher) selection bias

Cons

- *ad hoc* software ecosystems
- homegrown tools, conventions, social norms

Software evolution [in the large]

In software maintenance, software evolution refers to the process of repeatedly updating software, for various reasons, *after* initial development. FOSS distributions enabled a new scale of software evolution studies:

Software evolution in the large (Gonzalez-Barahona et. al, 2009)

The study of software evolution, at the scale of software collections, at the granularity they support (e.g., component release).

Pros

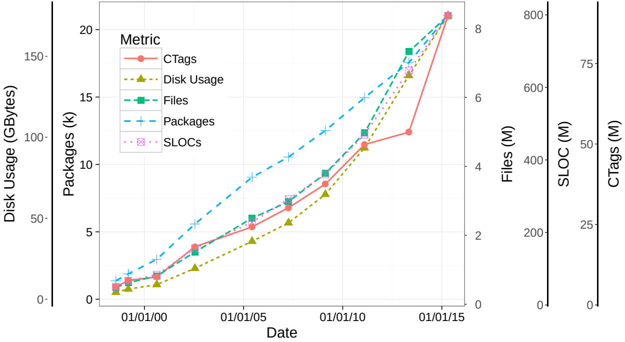
- relevant/popular software distribution model
- long lives (e.g., decades)
- uniform access to the history of contained software
- help with (researcher) selection bias

Cons

- *ad hoc* software ecosystems
- homegrown tools, conventions, social norms

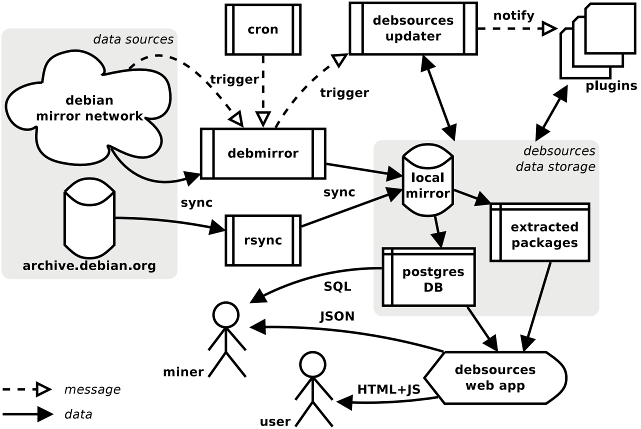
Debsources for researchers

- observation point on Debian macro-level evolution
- 20+ years of history
- both live and perennial monitoring



Debsources eases macro-level software evolution studies on FOSS, using Debian as a proxy.

Architecture



Debsources does the heavy lifting of maintaining a general purpose, always up to date storage for Debian source code, enabling plugin authors to focus on data extraction.

Plugins

- disk usage
- file type — MIME
- lines of code — `sloccount`, `wc`, `cloc`
- ctags — functions, classes, types, etc.
- checksums — SHA1, SHA256, TLSH
- license detection — `ninka`, `fossology`
- file count — (implicit)

Typical plugin: $\approx$ 100 SLOCs

Debsources dataset

- curated version of the data underpinning main Debsources instance
- focus on stable releases (sporadic updates)
- open data, available via Zenodo

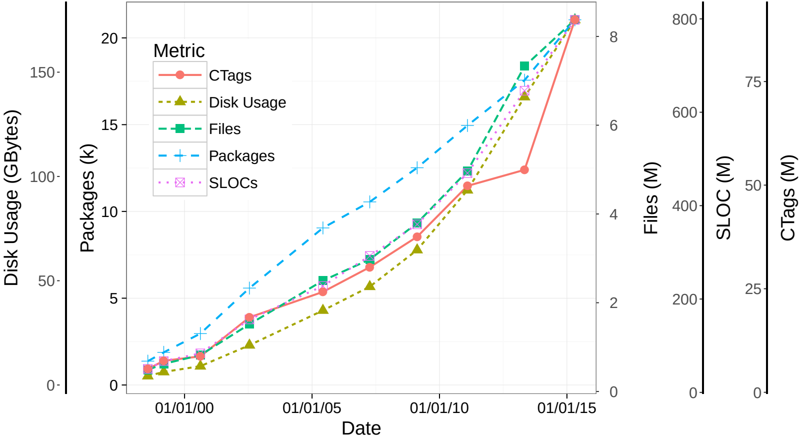
Table: Metadata

Table	Disk usage	Tuples
ctags	23 GB	186.5M
files	5944 MB	15.5M
metrics	3549 MB	46.7M
paths	3259 MB	30.5M
licenses	2976 MB	31.0M
path_info	1895 MB	11.7M
package_info	14 MB	82113
releases	7248 KB	97471
metric_info	32 KB	4
release_info	32 KB	10
	≈ 40 GB	

Table: Source code

	raw	dedup.
Files	30 M	15 M
Disk usage	320 GB	90 GB

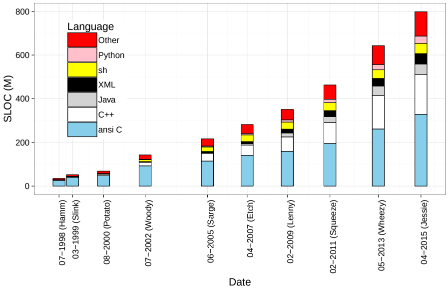
Highlight #1: total size

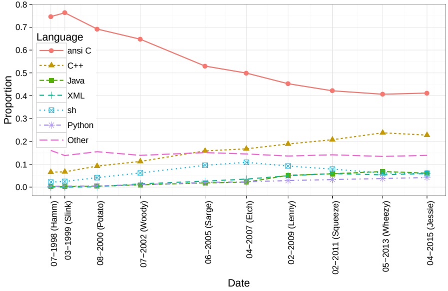


- correlation confirms Herraiz et. al, 2006 & 2007
- pre-*etch* (2007): growth rate slows down (allegedly, due to complexity ceiling)
- post-*etch*: growth rate increases

Highlight #2: programming languages

most popular programming languages in Debian over time

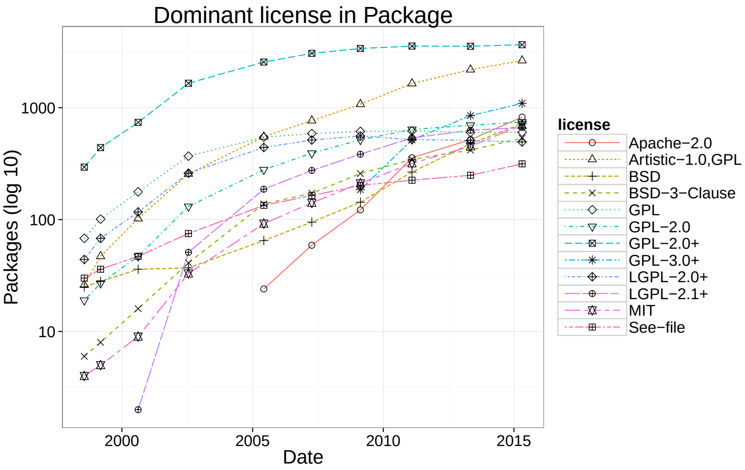




Recent trends (post-*etch*, 2007):

- C still leads, steady (absolute) growth
- C stops losing (relative) ground to C++
- decrease of Perl/Shell popularity
- Python rises
- Lisp halves its popularity
- Java no longer under-represented

Highlight #3: license usage



- the licenses census problem is hard to define
- the alleged decline of copyleft licensing is *not* evident here

Outline

1. Modeling FOSS package relationships
2. Beyond host boundaries
3. Back to the source (code)
4. Scaling to the entire software commons

Generalization opportunities

- time granularity: releases → commits
- space granularity: source packages → individual source files
- corpus coverage: Debian → all FOSS (i.e., the Software Commons)

The Software Heritage project

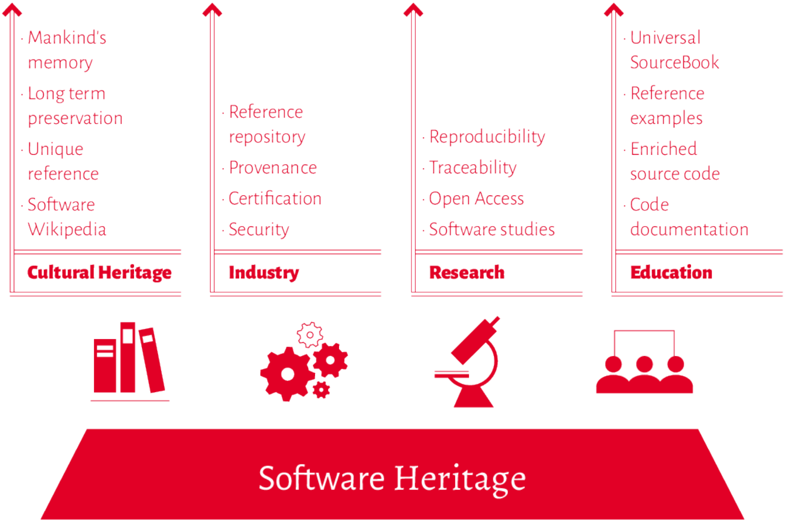
Software Heritage

THE GREAT LIBRARY OF SOURCE CODE

Our mission

Collect, preserve and share the *source code* of *all the software* that is publicly available.

A foundation for converging needs



Core principles

Artifacts

- file content
- directory structure
- commits, releases

Content

- no *a priori* selection
- intrinsic identifiers
- provenance and facts

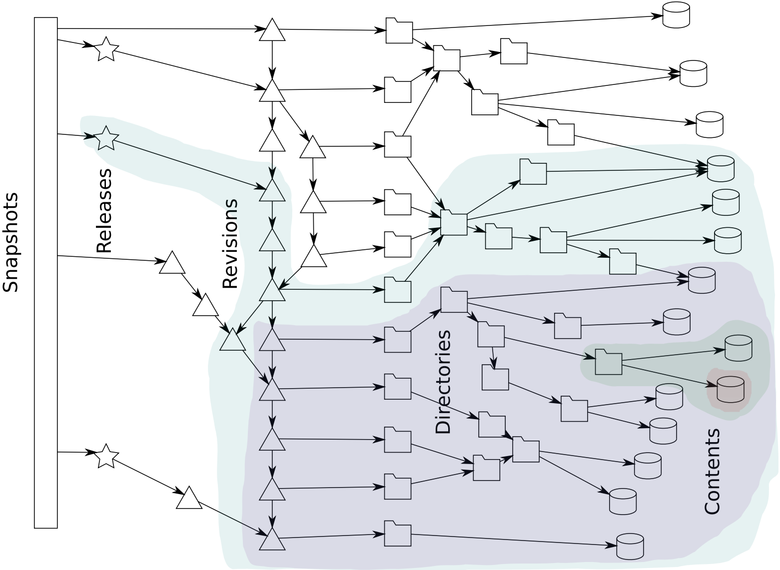
Accountability

- FOSS development
- replicas all the way down

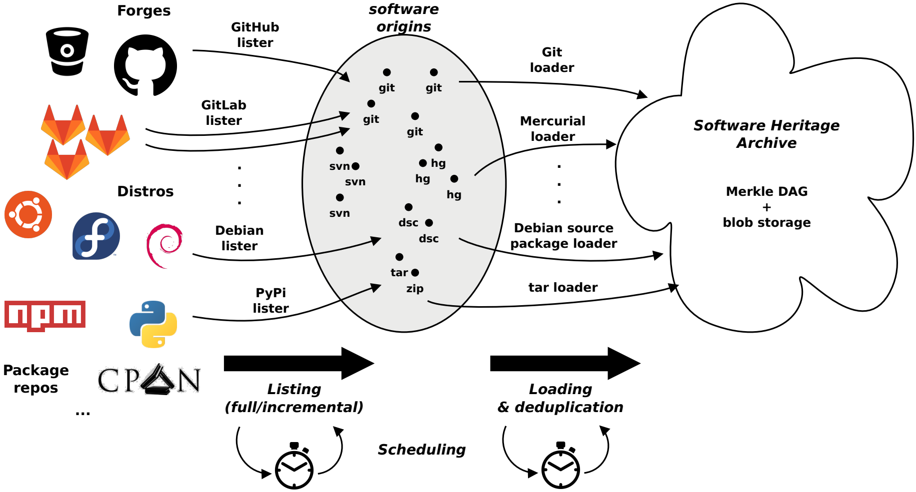
Business model

- multi-stakeholder
- non-profit

The archive: a (giant) Merkle DAG



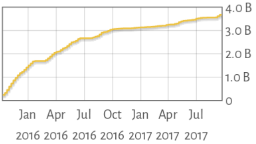
Data flow


Forges
GitHub lister
software origins
git
git
git
git
GitLab lister
svn
svn
svn
hg
hg
hg
dsc
dsc
tar
zip
Distros
Debian lister
PyPi lister
Package repos
CPAN
...
Git loader
Mercurial loader
Debian source package loader
tar loader
Software Heritage Archive
Merkle DAG
+
blob storage
Listing (full/incremental)
Loading & deduplication
Scheduling

Archive coverage

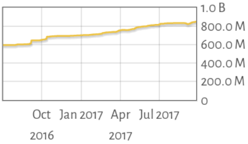
Source files

3,718,806,509



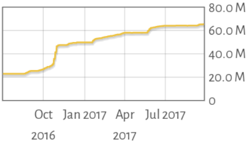
Commits

853,277,241



Projects

65,546,644



Current sources

- GitHub
- Debian, GNU
- WIP: Gitorious, Google Code, Bitbucket

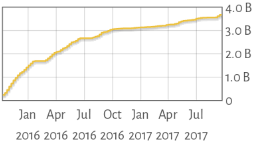
150 TB blobs, 5 TB database (as a graph: 7 B nodes + 60 B edges)

The *richest* source code archive already, . . . and growing daily!

Archive coverage

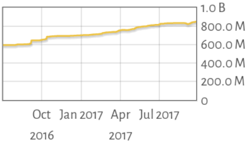
Source files

3,718,806,509



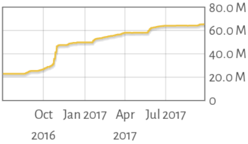
Commits

853,277,241



Projects

65,546,644



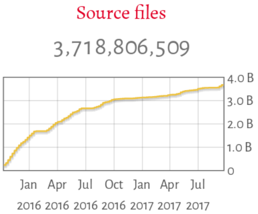
Current sources

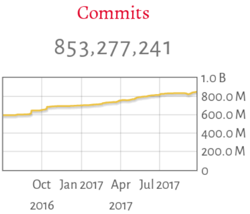
- GitHub
- Debian, GNU
- WIP: Gitorious, Google Code, Bitbucket

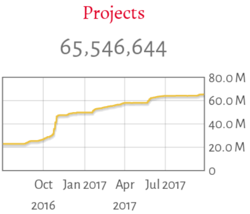
150 TB blobs, 5 TB database (as a graph: 7 B nodes + 60 B edges)

The *richest* source code archive already, . . . and growing daily!

Archive coverage







Current sources

- GitHub
- Debian, GNU
- WIP: Gitorious, Google Code, Bitbucket

150 TB blobs, 5 TB database (as a graph: 7 B nodes + 60 B edges)

The *richest* source code archive already, . . . and growing daily!

Technical roadmap

Features

- (done) lookup by content hash
- browsing: "wayback machine" for archived code
  - (done) via REST API
  - (todo) via Web UI
- (todo) download: `wget` / `git clone` from the archive
- (todo) deposit of source code bundles directly to the archive
- (todo) provenance lookup for all archived content
- (todo) full-text search on all archived source code files

. . . and much more than one could possibly imagine

the world's public software development history in a single graph!

Technical roadmap

Features

- (done) lookup by content hash
- browsing: "wayback machine" for archived code
  - (done) via REST API
  - (todo) via Web UI
- (todo) download: `wget` / `git clone` from the archive
- (todo) deposit of source code bundles directly to the archive
- (todo) provenance lookup for all archived content
- (todo) full-text search on all archived source code files

. . . and much more than one could possibly imagine

the world's public software development history in a single graph!

Research directions

Scalable analysis

- peculiar DAG, hard to treat with current large graph techniques
- we need abstractions, tools, and infrastructures to enable scale-out analysis of all this

Code search at scale

- how do you search 4 B source code files written in several thousand different programming languages?
- stemming and language detection alone become hard problems
- need to find a sweet spot in code understanding: *string* $\longleftrightarrow$ *AST*

Software phylogenetics

- the only corpus where all development ramifications of a code base are kept together
- clone detection at scale (for non-identical reuse)
- impact analysis: where did some code end up being used, a few thousand commits later?

Research directions

Scalable analysis

- peculiar DAG, hard to treat with current large graph techniques
- we need abstractions, tools, and infrastructures to enable scale-out analysis of all this

Code search at scale

- how do you search 4 B source code files written in several thousand different programming languages?
- stemming and language detection alone become hard problems
- need to find a sweet spot in code understanding: *string* $\longleftrightarrow$ *AST*

Software phylogenetics

- the only corpus where all development ramifications of a code base are kept together
- clone detection at scale (for non-identical reuse)
- impact analysis: where did some code end up being used, a few thousand commits later?

Research directions

Scalable analysis

- peculiar DAG, hard to treat with current large graph techniques
- we need abstractions, tools, and infrastructures to enable scale-out analysis of all this

Code search at scale

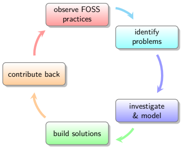
- how do you search 4 B source code files written in several thousand different programming languages?
- stemming and language detection alone become hard problems
- need to find a sweet spot in code understanding: *string* $\longleftrightarrow$ *AST*

Software phylogenetics

- the only corpus where all development ramifications of a code base are kept together
- clone detection at scale (for non-identical reuse)
- impact analysis: where did some code end up being used, a few thousand commits later?

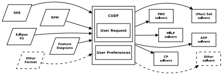
Wrapping up

A virtuous cycle in empirical software engineering



The Common Upgradeability Description Format (CUDF)

CUDF: a standard language to express upgrade problems across different component ecosystems, allowing to attack dependency solving with a multitude of technologies and research techniques.

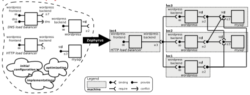


Requirements

- distribution agnostic
- solver agnostic
- extensible
- formal semantics
- plain text
- close to original

Practical deployment planning

- contingency plan: split *stateless* provisioning from state change
- phase 1: Zephyrus



- phase 2: Metis (univ. of Bologna)

Adoption

- Mandriva, Kyriba

Debsources for developers

Adoption

- quickly become a popular service among Debian developers
- frequently used on IRC to discuss source code snippets
- integrated with `codesearch.debian.net` and `tracker.debian.org`
- 15 code contributors; 5 interns

The Software Heritage project

Software Heritage
THE GREAT LIBRARY OF SOURCE CODE

Our mission

Collect, preserve and share the *source code* of *all the software* that is publicly available.

Research directions

Scalable analysis

- peculiar DAG, hard to treat with current large graph techniques
- we need abstractions, tools, and infrastructures to enable scale-out analysis of all this

Code search at scale

- how do you search 4 B source code files written in several thousand different programming languages?
- stemming and language detection alone become hard problems
- need to find a sweet spot in code understanding: *string* ⟷ *AST*

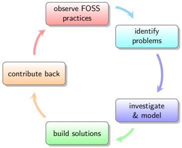
Software phylogenetics

- the only corpus where all development ramifications of a code base are kept together
- clone detection at scale (for non-identical reuse)
- impact analysis: where did some code end up being used, a few thousand commits later?

Thanks!

Wrapping up

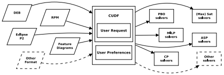
A virtuous cycle in empirical software engineering



Stefano Zacchiroli (IRIF) Habilitation à Diriger des Recherches 27 November 2017 3 / 47

The Common Upgradeability Description Format (CUDF)

CUDF: a standard language to express upgrade problems across different component ecosystems, allowing to attack dependency solving with a multitude of technologies and research techniques.



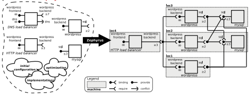
Requirements

- distribution agnostic
- extensible
- plain text
- solver agnostic
- formal semantics
- close to original

Stefano Zacchiroli (IRIF) Habilitation à Diriger des Recherches 27 November 2017 11 / 47

Practical deployment planning

- contingency plan: split *stateless* provisioning from state change
- phase 1: Zephyrus



- phase 2: Metis (univ. of Bologna)

Adoption

- Mandriva, Kyriba

Stefano Zacchiroli (IRIF) Habilitation à Diriger des Recherches 27 November 2017 25 / 47

Debsources for developers

Adoption

- quickly become a popular service among Debian developers
- frequently used on IRC to discuss source code snippets
- integrated with `codesearch.debian.net` and `tracker.debian.org`
- 15 code contributors; 5 interns

Stefano Zacchiroli (IRIF) Habilitation à Diriger des Recherches 27 November 2017 28 / 47

The Software Heritage project

Software Heritage
THE GREAT LIBRARY OF SOURCE CODE

Our mission

Collect, preserve and share the *source code* of *all the software* that is publicly available.

Stefano Zacchiroli (IRIF) Habilitation à Diriger des Recherches 27 November 2017 39 / 47

Research directions

Scalable analysis

- peculiar DAG, hard to treat with current large graph techniques
- we need abstractions, tools, and infrastructures to enable scale-out analysis of all this

Code search at scale

- how do you search 4 B source code files written in several thousand different programming languages?
- stemming and language detection alone become hard problems
- need to find a sweet spot in code understanding: *string* ⟷ *AST*

Software phylogenetics

- the only corpus where all development ramifications of a code base are kept together
- clone detection at scale (for non-identical reuse)
- impact analysis: where did some code end up being used, a few thousand commits later?

Stefano Zacchiroli (IRIF) Habilitation à Diriger des Recherches 27 November 2017 46 / 47

Thanks!