Towards the Formal Verification of Maintainer Scripts

Ralf Treinen

IRIF, Université Paris-Diderot

July 8, 2016

Plan

Disclaimer

- This talk is about work that just begun
- There are no results or tools yet!
- This is a collaborative research project over 4-5 years.
- Today I will just tell you about what we intend to do.
- There will hopefully be results to present at future debconfs.

Maintainer Scripts

A `.deb` package contains two sets of files:

1. *a set of files to install on the system when the package is installed,*
2. *and a set of files that provide additional metadata about the package or which are executed when the package is installed or removed. [...] Among those files are the package maintainer scripts [...]*

(Debian Policy, introduction to ch. 3)

Maintainer Scripts

Roughly:

preinst	executed before the package is unpacked
postinst	executed after the package is unpacked
prerm	executed before the package is removed
postrm	executed after the package is removed

(Debian Policy ch. 6.1)

Formal Verification

- Attempts to construct a formal *proof* of correctness.
- Needs a formalization of program execution, and a precise statement of what the program is supposed to do, like
  - *Whenever* I am in a state satisfying a given pre-condition
  - . . . and I execute a given program,
  - . . . then I get a state satisfying a given post-condition.
- This is not testing: it yields a guarantee (in the formal model) of correctness, for *any* initial state.
- There still is a connection to testing, see later.
- In our case, required properties of scripts are more involved. Let's first look at an example.

How it all started

Debian Bug report #431131

cmigrep: broken emacsen-install script

Package: cmigrep
Version: 1.3-1
Severity: critical

cmigrep's emacsen-install script is overzealous; it inappropriately attempts to compile all .el files in /usr/share/emacs/site-lisp [...]
and compounds the problem by removing /usr/share/$FLAVOR/site-lisp/*.el, which may contain files belonging to other packages (for instance, auctex's tex-site.el).

Why scripts are involved here

Elisp code

- Debian has decided to always byte-compile Emacs-Lisp (elisp) code that is installed by packages.
- There are several emacs packages available in debian.
- Different emacsen have different byte-code format.

Solution in Debian

- It was decided to not deploy compiled elisp for all emacsen, but rather to deploy elisp source code, and to compile it during package installation.

What the scripts are supposed to do

In the `postinst` script

For every emacs *flavor* available:

- compile elisp source for *flavor*
- place resulting byte-code in `/usr/share/`*`flavour`*`/site-lisp/`

This script is also executed when a new emacs flavour gets installed

In the `prerm` script

For every emacs *flavor* available:

- remove installed files in `/usr/share/`*`flavour`*`/site-lisp/`

A fatal change in the package

Initially, the package did something like

postinst

```
D=/usr/share/${FLAVOUR}/site-lisp/${PACKAGE}
mkdir -p ${D}
# create elc files in ${D}
```

prerm

```
D=/usr/share/${FLAVOUR}/site-lisp/${PACKAGE}
rm ${D}/*.elc
rmdir ${D}
```

A fatal change in the package

Then, the maintainer decided to get rid of the private directory:

postinst

```
D=/usr/share/${FLAVOUR}/site-lisp
mkdir -p ${D}
# create elc files in ${D}
```

prerm

```
D=/usr/share/${FLAVOUR}/site-lisp
rm ${D}/*.elc
```

What we can learn from this example

- The maintainer did a really stupid mistake ☹
- Testing in a *minimal* environment will not reveal the mistake, only testing with an already populated `.../site-lisp` does.
- One would like to know that package installation/removal does not do something bad, *whatever* the initial configuration.
- Finding stupid mistakes in a huge corpus of maintainer scripts: we need *automatic tools*.

Infrastructure shared between packages

What is at the root of this problem:

- We can not assume that each package acts only on a private part of the file system.
- Infrastructure has to be shared between packages.
- It may be necessary that different packages create and modify files in the same directory, or even modify the same configuration file.
- Example: TEX, emacs, ...
- Without resource sharing, everything would be much simpler.

Disclaimer, again

- Verifying automatically all maintainer scripts is an impossible task! Basically for two reasons:
  - The execution model is very rich (both the language in which scripts are written, and the state of the machine that is to be modified by the scripts)
  - Correctness of programs written in a Turing-complete programming language, for any non-trivial notion of correctness, is undecidable.
- We will have to simplify and approximate.
- Difficulty different to EDOS/Mancoosi projects, where the main challenge was *scale*.

Maintainer Scripts

- Policy does not require them to be scripts (`bash.preinst` is an ELF executable!), but they almost always are.
- `csh` and `tcsh` are discouraged (policy 10.4)
- The vast majority are written in Posix shell, with some embellishments mandated by policy 10.4:
  - `echo -n`
  - `test`, when built-in, must support `-a` and `-o`
  - local scopes
  - arguments to `kill` and `trap`
- We only look at Posix(+debian)-shell scripts

Our restricted view of maintainer scripts

- Typically small programs
- That means that we view them as describing a transformation of one file system tree into another (and which depends on various parameters, like current working directory, uid, gid, umask, environment)
- This already is an abstraction:
  - the filesystem is not really tree (symbolic and hard links)
  - we ignore anything else a script may be doing
- Policy 10.4 mandates strict semantics (`set -e`).

What makes our task easier

- No recursive functions
- Exit codes: we only care about whether $= 0$ or > 0
- Loops are mostly used in a restricted way (`for` loops, or `while read` loops)
- We ignore concurrency, and consider a sequential execution model (justified by the big `dpkg` lock)
- We will ignore *access* time stamps (justified by `relatime` mount option)

Working with shell scripts

- We wrote a shell script parser which allows us to do some statistical analysis about what is used and what is not.
- Shell is not the most convenient language when you want to do an analysis of scripts, and this concerns both syntax and semantics.
- We are currently defining a DSL (called *colis*), with the design goals of
  - having a sane semantics (in particular, being statically typed)
  - allowing for representing a large portion of our maintainer scripts

Properties of scripts

- When launched on a “reasonable” filesystem, execute without error.
- Maintain an invariant on the filesystem (e.g., FHS compliance)

Relations between Scripts

- Script r is the right-inverse of script i :

$$i \circ r = id$$

For instance : removing a package immediately after its installation always restores the original state.
(when purging, otherwise except configuration and log files)

- More precisely:, for any filesystem f :

$$f |> \mathrm{preinst} |> \textit{(unpack)} |> \mathrm{postinst} |> \mathrm{prerm} |> \textit{(removal)} |> \mathrm{postrm} = f$$

- . . . except for log and configuration files, unless when purging.

- Do we need a more general property?

$$\forall s \in S : i \circ s \circ r = s$$

For instance : installing a package, then doing some action s, then removing the package is the same as doing s alone. For instance, s might be installing or removing some other packages.

- Commutation of scripts?

$$s_1 \circ s_2 = s_2 \circ s_1$$

This would mean that is safe to reorder scripts.

Idempotency

- Debian policy (section 6.2) requires maintainer scripts to be idempotent.
- Mathematically, i is *idempotent* when

$$i \circ i = i$$

- The sense in Debian is much larger:

> *If the first call failed, or aborted half way through for some reason, the second call should merely do the things that were left undone the first time, if any, and exit with a success status if everything is OK.*

- What does that mean precisely?

A few words of software verification

- This is an almost impossible task!
- Verification problems are typically undecidable (like the halting problem).
- However:
  - We can simplify the problem and look at approximations
  - Our use case yields some simplifying assumptions (e.g., on loops)
- A failed proof attempt often can be used to extract interesting test cases.
- We should not be discouraged by theoretical undecidability!

First approach to verification

- Use a specialized computational model which can be analyzed by formal methods.
- Typically this either means that one looses information (approximation of the problem), or that this method applies only in specific cases.
- One well known example : model checking
- In our case we will use *tree transducers*, which have been studied for instance in the context of transformations of XML trees.
- Use case for tree transducers : invocations of `find`.
- Work in progress.

Second approach to verification

- Deductive Verification
- A deductive verification tools knows about the constructs of the programming language.
- A post-condition is threaded through the program: calculates the minimal condition at the beginning of the program that guarantees the post-condition to be true (the *weakest pre-condition*)
- Finally, one has to verify that the pre-condition (from the specification) implies the calculated weakest precondition.
- This *proof obligation* is then handed over to a solver for the specific data domain.

Examples: Weakest Pre-Conditions

Specification

$\underbrace{\{x > y\}}_{\text{pre-cond}}$ $\texttt{if}\ x > 0\ \texttt{then}\ x := 2{*}x\ \texttt{else}\ x := x{+}1\ \texttt{fi}$ $\underbrace{\{x > y\}}_{\text{post-cond}}$

Verification

- Weakest pre-condition:

$$(x > 0 \wedge 2 * x > y) \vee (x \leq 0 \wedge x + 1 > y)$$

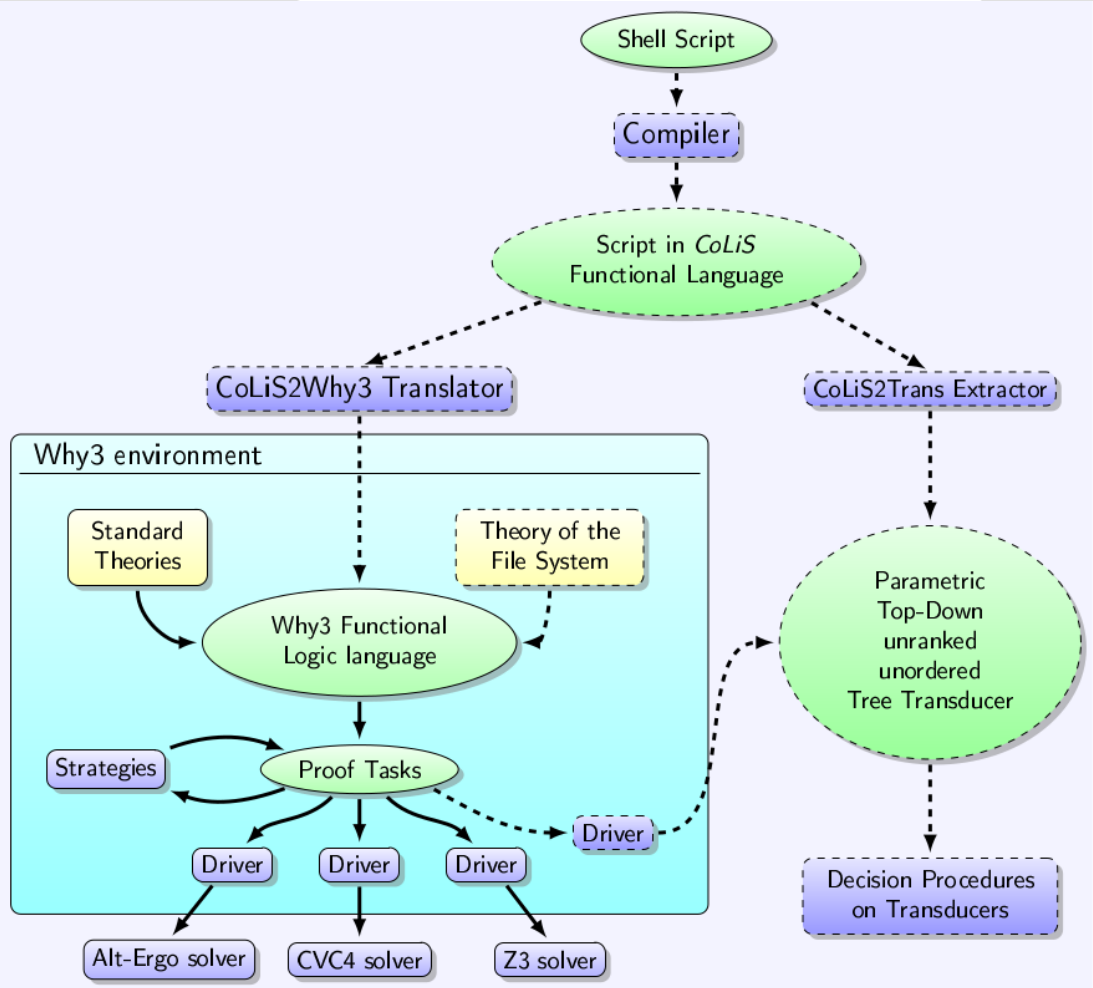
- Proof obligation:

$$(x > y) \Rightarrow \Big((x > 0 \wedge 2 * x > y) \vee (x \leq 0 \wedge x + 1 > y)\Big)$$

Existing tools

- Recent verification platforms know about programming constructs like polymorphic types, functions and procedures, exceptions handling, modules, ...
- Example: `why3` (in debian)
- Recently, specialized solvers for logics of data domains have become very strong (the SMT-revolution)
- Examples: `alt-ergo`, `cvc3`, `z3` (all in debian)

Global View of CoLiS



The CoLiS Project

- *Correctness of Linux Scripts*
- Project funded by *Agence Nationale de Recherche*
- October 2015 – September 2019 or later
- `http://colis.irif.univ-paris-diderot.fr/`
- 3 academic partner sites :
  - IRIF, University Paris-Diderot
  - INRIA Lille (team Links)
  - INRIA Saclay (team Toccata)

What I would like to hear from you

- What are you stories about faulty maintainer scripts ?
- What properties of maintainer scripts do you find useful to check ?
- Maintainer scripts — good or evil ?