Symbolic Execution of Maintainer Scripts

Nicolas Jeannerod and Ralf Treinen
joint work with Benedikt Becker, Claude Marché, Mihaela Sighireanu, Yann Régis-Gianas

IRIF, Université de Paris

July 21, 2019

Plan

1 Introduction

2 Symbolic Execution of Scripts

3 Symbolic Execution of Maintainer Scripts

4 Demo Time

5 Detected Bugs

6 Conclusions

Plan

1 Introduction

2 Symbolic Execution of Scripts

3 Symbolic Execution of Maintainer Scripts

4 Demo Time

5 Detected Bugs

6 Conclusions

The CoLiS project

- Goal: apply formal methods to the quality assessment of Debian maintainer scripts.
- Initial idea: use methods from formal *program verification*.
- Example of a use case: A *postrm* that deletes files from *unrelated packages*, see for instance Ralf's talk at Debconf'16 for a concrete example.
- We only look at Posix shell scripts which are more than 99% of our maintainer scripts.
- We knew from the beginning that this is an ambitious goal: We will at best succeed partially.

The CoLiS project

- Goal: apply formal methods to the quality assessment of Debian maintainer scripts.
- Initial idea: use methods from formal *program verification*.
- Example of a use case: A *postrm* that deletes files from *unrelated packages*, see for instance Ralf's talk at Debconf'16 for a concrete example.
- We only look at Posix shell scripts which are more than 99% of our maintainer scripts.
- We knew from the beginning that this is an ambitious goal: We will at best succeed partially.

The CoLiS project

- Goal: apply formal methods to the quality assessment of Debian maintainer scripts.
- Initial idea: use methods from formal *program verification*.
- Example of a use case: A *postrm* that deletes files from *unrelated packages*, see for instance Ralf's talk at Debconf'16 for a concrete example.
- We only look at Posix shell scripts which are more than 99% of our maintainer scripts.
- We knew from the beginning that this is an ambitious goal: We will at best succeed partially.

The CoLiS project

- Goal: apply formal methods to the quality assessment of Debian maintainer scripts.
- Initial idea: use methods from formal *program verification*.
- Example of a use case: A *postrm* that deletes files from *unrelated packages*, see for instance Ralf's talk at Debconf'16 for a concrete example.
- We only look at Posix shell scripts which are more than 99% of our maintainer scripts.
- We knew from the beginning that this is an ambitious goal: We will at best succeed partially.

The CoLiS project

- Goal: apply formal methods to the quality assessment of Debian maintainer scripts.
- Initial idea: use methods from formal *program verification*.
- Example of a use case: A *postrm* that deletes files from *unrelated packages*, see for instance Ralf's talk at Debconf'16 for a concrete example.
- We only look at Posix shell scripts which are more than 99% of our maintainer scripts.
- We knew from the beginning that this is an ambitious goal: We will at best succeed partially.

What happened previously

- *Static* syntactic analysis of Posix shell scripts.
- Talks in 2018 at Fosdem, Minidebconf Hamburg, Debconf.
- Static syntactical analysis of Posix shell scripts is far from trivial.
- The *Morbig* parser for Posix shell scripts.
- First report of bugs on a relatively trivial level, like:
  - Missing strict mode
  - Wrong redirections
  - Wrong test expressions

What happened previously

- *Static* syntactic analysis of Posix shell scripts.
- Talks in 2018 at Fosdem, Minidebconf Hamburg, Debconf.
- Static syntactical analysis of Posix shell scripts is far from trivial.
- The *Morbig* parser for Posix shell scripts.
- First report of bugs on a relatively trivial level, like:
  - Missing strict mode
  - Wrong redirections
  - Wrong test expressions

What happened previously

- *Static* syntactic analysis of Posix shell scripts.
- Talks in 2018 at Fosdem, Minidebconf Hamburg, Debconf.
- Static syntactical analysis of Posix shell scripts is far from trivial.
- The *Morbig* parser for Posix shell scripts.
- First report of bugs on a relatively trivial level, like:
  - Missing strict mode
  - Wrong redirections
  - Wrong test expressions

What happened previously

- *Static* syntactic analysis of Posix shell scripts.
- Talks in 2018 at Fosdem, Minidebconf Hamburg, Debconf.
- Static syntactical analysis of Posix shell scripts is far from trivial.
- The *Morbig* parser for Posix shell scripts.
- First report of bugs on a relatively trivial level, like:
  - Missing strict mode
  - Wrong redirections
  - Wrong test expressions

What happened previously

- *Static* syntactic analysis of Posix shell scripts.
- Talks in 2018 at Fosdem, Minidebconf Hamburg, Debconf.
- Static syntactical analysis of Posix shell scripts is far from trivial.
- The *Morbig* parser for Posix shell scripts.
- First report of bugs on a relatively trivial level, like:
  - Missing strict mode
  - Wrong redirections
  - Wrong test expressions

What happened previously

- *Static* syntactic analysis of Posix shell scripts.
- Talks in 2018 at Fosdem, Minidebconf Hamburg, Debconf.
- Static syntactical analysis of Posix shell scripts is far from trivial.
- The *Morbig* parser for Posix shell scripts.
- First report of bugs on a relatively trivial level, like:
    - Missing strict mode
    - Wrong redirections
    - Wrong test expressions

What happened previously

- *Static* syntactic analysis of Posix shell scripts.
- Talks in 2018 at Fosdem, Minidebconf Hamburg, Debconf.
- Static syntactical analysis of Posix shell scripts is far from trivial.
- The *Morbig* parser for Posix shell scripts.
- First report of bugs on a relatively trivial level, like:
  - Missing strict mode
  - Wrong redirections
  - Wrong test expressions

What happened previously

- *Static* syntactic analysis of Posix shell scripts.
- Talks in 2018 at Fosdem, Minidebconf Hamburg, Debconf.
- Static syntactical analysis of Posix shell scripts is far from trivial.
- The *Morbig* parser for Posix shell scripts.
- First report of bugs on a relatively trivial level, like:
  - Missing strict mode
  - Wrong redirections
  - Wrong test expressions

What we will present today

- Analyzing the *behavior* of Maintainer Scripts
- Caveat 1: we will never be able to analyze all the > 30.000 maintainer scripts.
- Caveat 2: we have to cut corners in the model, and perform *approximations*.
- Focus on finding bugs (as opposed to guaranteeing correctness).

What we will present today

- Analyzing the *behavior* of Maintainer Scripts
- Caveat 1: we will never be able to analyze all the > 30.000 maintainer scripts.
- Caveat 2: we have to cut corners in the model, and perform *approximations*.
- Focus on finding bugs (as opposed to guaranteeing correctness).

What we will present today

- Analyzing the *behavior* of Maintainer Scripts
- Caveat 1: we will never be able to analyze all the $>$ 30.000 maintainer scripts.
- Caveat 2: we have to cut corners in the model, and perform *approximations*.
- Focus on finding bugs (as opposed to guaranteeing correctness).

What we will present today

- Analyzing the *behavior* of Maintainer Scripts
- Caveat 1: we will never be able to analyze all the > 30.000 maintainer scripts.
- Caveat 2: we have to cut corners in the model, and perform *approximations*.
- Focus on finding bugs (as opposed to guaranteeing correctness).

Plan

1 Introduction

2 Symbolic Execution of Scripts

3 Symbolic Execution of Maintainer Scripts

4 Demo Time

5 Detected Bugs

6 Conclusions

Semantics of Shell Scripts

- First step: reasoning about one script at a time.
- Starting point: we need a language to talk about the semantics of scripts: symbolic representation.
- We do this both for the case of success and of failure of the script.
- We need a way to *calculate* effectively on these representations, and to combine them (sequential composition, conditional composition, . . .)
- Analogy: Using regular expressions to talk about sets of strings.

Semantics of Shell Scripts

- First step: reasoning about one script at a time.
- Starting point: we need a language to talk about the semantics of scripts: symbolic representation.
- We do this both for the case of success and of failure of the script.
- We need a way to *calculate* effectively on these representations, and to combine them (sequential composition, conditional composition, . . .)
- Analogy: Using regular expressions to talk about sets of strings.

Semantics of Shell Scripts

- First step: reasoning about one script at a time.
- Starting point: we need a language to talk about the semantics of scripts: symbolic representation.
- We do this both for the case of success and of failure of the script.
- We need a way to *calculate* effectively on these representations, and to combine them (sequential composition, conditional composition, . . .)
- Analogy: Using regular expressions to talk about sets of strings.

Semantics of Shell Scripts

- First step: reasoning about one script at a time.
- Starting point: we need a language to talk about the semantics of scripts: symbolic representation.
- We do this both for the case of success and of failure of the script.
- We need a way to *calculate* effectively on these representations, and to combine them (sequential composition, conditional composition, . . .)
- Analogy: Using regular expressions to talk about sets of strings.

Semantics of Shell Scripts

- First step: reasoning about one script at a time.
- Starting point: we need a language to talk about the semantics of scripts: symbolic representation.
- We do this both for the case of success and of failure of the script.
- We need a way to *calculate* effectively on these representations, and to combine them (sequential composition, conditional composition, . . .)
- Analogy: Using regular expressions to talk about sets of strings.

Tree Constraints

- Our current approach: use predicate logic.
- Predicate logic allows us to talk about *relations*: in our case the relation between the intial configuration, and the possible configurations obtained by the execution.
- Special purpose logic for talking about a restricted form of tree transformations.
- Effective calculations on formulas.

Tree Constraints

- Our current approach: use predicate logic.
- Predicate logic allows us to talk about *relations*: in our case the relation between the intial configuration, and the possible configurations obtained by the execution.
- Special purpose logic for talking about a restricted form of tree transformations.
- Effective calculations on formulas.

Tree Constraints

- Our current approach: use predicate logic.
- Predicate logic allows us to talk about *relations*: in our case the relation between the intial configuration, and the possible configurations obtained by the execution.
- Special purpose logic for talking about a restricted form of tree transformations.
- Effective calculations on formulas.

Tree Constraints

- Our current approach: use predicate logic.
- Predicate logic allows us to talk about *relations*: in our case the relation between the intial configuration, and the possible configurations obtained by the execution.
- Special purpose logic for talking about a restricted form of tree transformations.
- Effective calculations on formulas.

Example Specification: $\texttt{mkdir}\ q/f$

Success		$\exists x, x', y' \cdot$ $\texttt{resolve}(r, cwd, q, x) \wedge \texttt{dir}(x) \wedge x[f]\uparrow$ $\wedge\, \texttt{similar}(r, r', cwd, q, x, x') \wedge x \sim_{\{f\}} x'$ $\wedge\, \texttt{dir}(x') \wedge x'[f]y' \wedge \texttt{dir}(y') \wedge y'[\varnothing]$
Failure	*File exists*	$\exists y \cdot \texttt{resolve}(r, cwd, q/f, y) \wedge r \doteq r'$
Failure	*No such file*	$\texttt{noresolve}(r, cwd, q) \wedge r \doteq r'$
Failure	*Not a dir*	$\exists x \cdot \texttt{resolve}(r, cwd, q, x) \wedge \neg\texttt{dir}(x) \wedge r \doteq r'$

Example Specification: `mkdir` q/f

Success		$\exists x, x', y' \cdot$ $\mathtt{resolve}(r, cwd, q, x) \wedge \mathtt{dir}(x) \wedge x[f]\uparrow$ $\wedge\, \mathtt{similar}(r, r', cwd, q, x, x') \wedge x \sim_{\{f\}} x'$ $\wedge\, \mathtt{dir}(x') \wedge x'[f]y' \wedge \mathtt{dir}(y') \wedge y'[\varnothing]$
Failure	*File exists*	$\exists y \cdot \mathtt{resolve}(r, cwd, q/f, y) \wedge r \doteq r'$
	No such file	$\mathtt{noresolve}(r, cwd, q) \wedge r \doteq r'$
Failure	*Not a dir*	$\exists x \cdot \mathtt{resolve}(r, cwd, q, x) \wedge \neg\mathtt{dir}(x) \wedge r \doteq r'$

utcome of the
ecification Case

Example Specification: `mkdir` q/f

Success		$\exists x, x', y' \cdot$ $\texttt{resolve}(r, cwd, q, x) \wedge \texttt{dir}(x) \wedge x[f]\uparrow$ $\wedge\, \texttt{similar}(r, r', cwd, q, x, x') \wedge x \sim_{\{f\}} x'$ $\wedge\, \texttt{dir}(x') \wedge x'[f]y' \wedge \texttt{dir}(y') \wedge y'[\varnothing]$
Failure	*File exists*	$\exists y \cdot \texttt{resolve}(r, cwd, q/f, y) \wedge r \doteq r'$
	No such file	$\texttt{noresolve}(r, cwd, q) \wedge r \doteq r'$
Failure	*Not a dir*	$\exists x \cdot \texttt{resolve}(r, cwd, q, x) \wedge \neg\texttt{dir}(x) \wedge r \doteq r'$

utcome of the
ecification Case

Text
beings)

Example Specification: mkdir q/f

Formula in our logic

Success		$\exists x, x', y' \cdot \mathtt{resolve}(r, cwd, q, x) \wedge \mathtt{dir}(x) \wedge x[f]\uparrow \wedge \mathtt{similar}(r, r', cwd, q, x, x') \wedge x \sim_{\{f\}} x' \wedge \mathtt{dir}(x') \wedge x'[f]y' \wedge \mathtt{dir}(y') \wedge y'[\varnothing]$
Failure	*File exists*	$\exists y \cdot \mathtt{resolve}(r, cwd, q/f, y) \wedge r \doteq r'$
	No such file	$\mathtt{noresolve}(r, cwd, q) \wedge r \doteq r'$
Failure	*Not a dir*	$\exists x \cdot \mathtt{resolve}(r, cwd, q, x) \wedge \neg\mathtt{dir}(x) \wedge r \doteq r'$

Outcome of the
ecification Case

Text
beings)

Example Specification: `mkdir` q/f

Formula in our logic

Success		$\exists x, x', y' \cdot$ $\texttt{resolve}(r, cwd, q, x) \wedge \texttt{dir}(x) \wedge x[f]\uparrow$ $\wedge\, \texttt{similar}(r, r', cwd, q, x, x') \wedge x \sim_{\{f\}} x'$ $\wedge\, \texttt{dir}(x') \wedge x'[f]y' \wedge \texttt{dir}(y') \wedge y'[\varnothing]$
Failure	*File exists*	$\exists y \cdot$
	No such file	nore
Failure	*Not a dir*	$\exists x \cdot$ $\doteq r'$

utcome of the
ecification Case

n Text
beings)

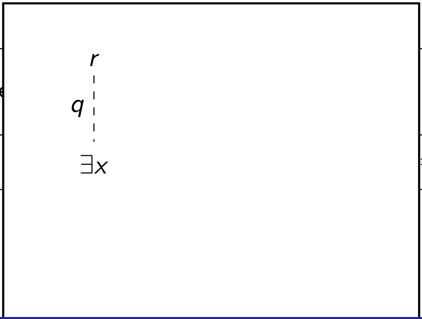
Example Specification: $\texttt{mkdir}\ q/f$

Formula in our logic

Success		$\exists x, x', y' \cdot$ $\texttt{resolve}(r, cwd, q, x) \wedge \texttt{dir}(x) \wedge x[f]\uparrow$ $\wedge\, \texttt{similar}(r, r', cwd, q, x, x') \wedge x \sim_{\{f\}} x'$ $\wedge\, \texttt{dir}(x') \wedge x'[f]y' \wedge \texttt{dir}(y') \wedge y'[\varnothing]$
Failure	*File exists*	$\exists y \cdot$
	No such file	nore
Failure	*Not a dir*	$\exists x \cdot$ … $\doteq r'$

utcome of the
ecification Case

Text
beings)



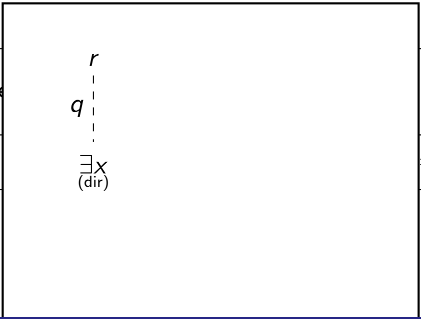
Example Specification: mkdir q/f

Formula in our logic

Success		$\exists x, x', y' \cdot$ $\texttt{resolve}(r, cwd, q, x) \wedge \texttt{dir}(x) \wedge x[f]\uparrow$ $\wedge\, \texttt{similar}(r, r', cwd, q, x, x') \wedge x \sim_{\{f\}} x'$ $\wedge\, \texttt{dir}(x') \wedge x'[f]y' \wedge \texttt{dir}(y') \wedge y'[\varnothing]$
Failure	*File exists*	$\exists y \cdot$
	No such file	nore
Failure	*Not a dir*	$\exists x \cdot$ $\doteq r'$

utcome of the
ecification Case

Text
beings)



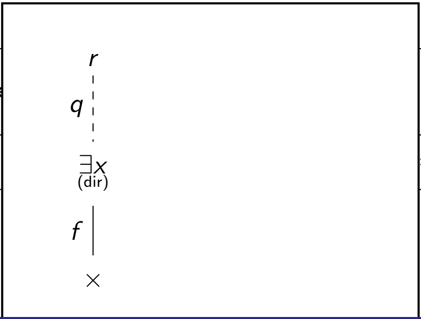
Example Specification: $\texttt{mkdir}\ q/f$

Formula in our logic

Success		$\exists x, x', y' \cdot$ $\texttt{resolve}(r, cwd, q, x) \land \texttt{dir}(x) \land x[f]\uparrow$ $\land \texttt{similar}(r, r', cwd, q, x, x') \land x \sim_{\{f\}} x'$ $\land \texttt{dir}(x') \land x'[f]y' \land \texttt{dir}(y') \land y'[\varnothing]$
Failure	*File exists*	$\exists y \cdot$
	No such file	nore
Failure	*Not a dir*	$\exists x \cdot$ … $\doteq r'$

utcome of the
ecification Case

Text
beings)



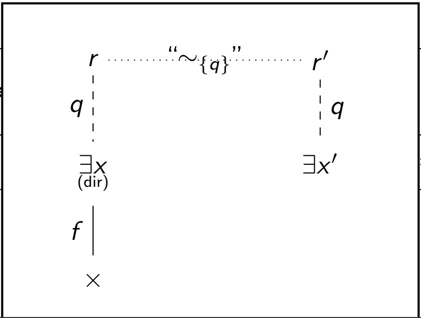
Example Specification: `mkdir` q/f

Formula in our logic

Success		$\exists x, x', y' \cdot$ $\texttt{resolve}(r, cwd, q, x) \wedge \texttt{dir}(x) \wedge x[f]\uparrow$ $\wedge\, \texttt{similar}(r, r', cwd, q, x, x') \wedge x \sim_{\{f\}} x'$ $\wedge\, \texttt{dir}(x') \wedge x'[f]y' \wedge \texttt{dir}(y') \wedge y'[\varnothing]$
Failure	*File exists*	$\exists y \cdot$
	No such file	nore
Failure	*Not a dir*	$\exists x \cdot$ $\doteq r'$

Outcome of the
ecification Case

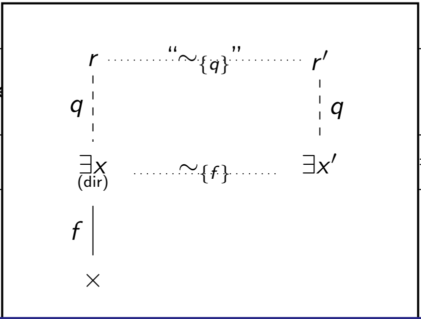
Text
beings)



Example Specification: `mkdir` q/f

Formula in our logic

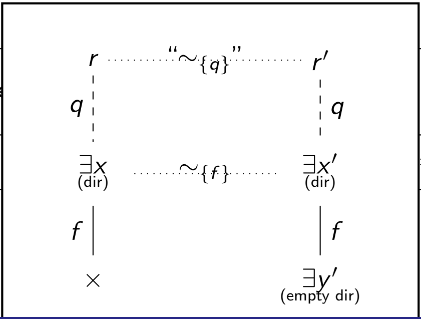
Success		$\exists x, x', y' \cdot$ $\texttt{resolve}(r, cwd, q, x) \wedge \texttt{dir}(x) \wedge x[f]\uparrow$ $\wedge\, \texttt{similar}(r, r', cwd, q, x, x') \wedge x \sim_{\{f\}} x'$ $\wedge\, \texttt{dir}(x') \wedge x'[f]y' \wedge \texttt{dir}(y') \wedge y'[\varnothing]$
Failure	*File exists*	$\exists y \cdot$
Failure	*No such file*	`nore`
Failure	*Not a dir*	$\exists x \cdot$ … $\doteq r'$



Example Specification: `mkdir` q/f

Formula in our logic

Success		$\exists x, x', y' \cdot$ $\texttt{resolve}(r, cwd, q, x) \wedge \texttt{dir}(x) \wedge x[f]\uparrow$ $\wedge\, \texttt{similar}(r, r', cwd, q, x, x') \wedge x \sim_{\{f\}} x'$ $\wedge\, \texttt{dir}(x') \wedge x'[f]y' \wedge \texttt{dir}(y') \wedge y'[\varnothing]$
Failure	*File exists*	$\exists y \cdot$
Failure	*No such file*	nore
Failure	*Not a dir*	$\exists x \cdot$ … $\doteq r'$

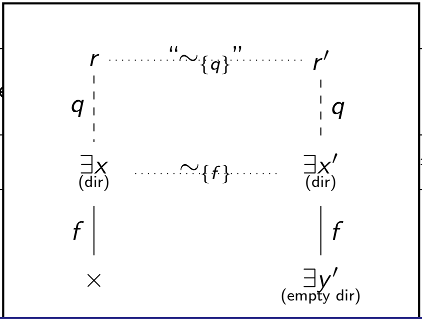


$$\begin{tikzcd}
r \arrow[r, dotted, dash, "\text{``}\sim_{\{q\}}\text{''}"] \arrow[d, dashed, dash, "q"'] & r' \arrow[d, dashed, dash, "q"] \\
\underset{\text{(dir)}}{\exists x} \arrow[r, dotted, dash, "\sim_{\{f\}}"] \arrow[d, dash, "f"'] & \underset{\text{(dir)}}{\exists x'} \arrow[d, dash, "f"] \\
\times & \underset{\text{(empty dir)}}{\exists y'}
\end{tikzcd}$$

Example Specification: `mkdir` q/f

Formula in our logic

Success		$\exists x, x', y' \cdot$ $\mathtt{resolve}(r, cwd, q, x) \wedge \mathtt{dir}(x) \wedge x[f]\uparrow$ $\wedge\, \mathtt{similar}(r, r', cwd, q, x, x') \wedge x \sim_{\{f\}} x'$ $\wedge\, \mathtt{dir}(x') \wedge x'[f]y' \wedge \mathtt{dir}(y') \wedge y'[\varnothing]$
Failure	*File exists*	$\exists y \cdot$
Failure	*No such file*	`nore`
Failure	*Not a dir*	$\exists x \cdot$ … $\doteq r'$



Example Specification: `mkdir` q/f

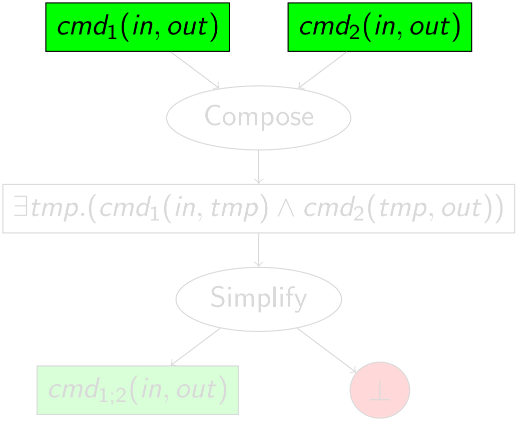
Formula in our logic

Success		$\exists x, x', y' \cdot$ $\texttt{resolve}(r, cwd, q, x) \land \texttt{dir}(x) \land x[f]\uparrow$ $\land\, \texttt{similar}(r, r', cwd, q, x, x') \land x \sim_{\{f\}} x'$ $\land\, \texttt{dir}(x') \land x'[f]y' \land \texttt{dir}(y') \land y'[\varnothing]$
Failure	*File exists*	$\exists y \cdot \texttt{resolve}(r, cwd, q/f, y) \land r \doteq r'$
	No such file	$\texttt{noresolve}(r, cwd, q) \land r \doteq r'$
Failure	*Not a dir*	$\exists x \cdot \texttt{resolve}(r, cwd, q, x) \land \neg\texttt{dir}(x) \land r \doteq r'$

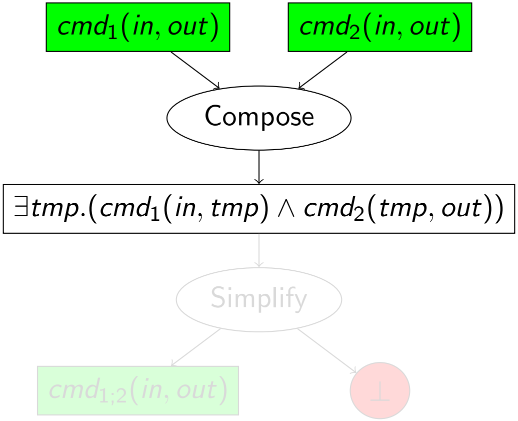
utcome of the
ecification Case

Text
beings)

Using the Logic: sequential composition



Using the Logic: sequential composition



Using the Logic: sequential composition

$$\begin{tikzcd}
\boxed{cmd_1(in, out)} \arrow[dr] & & \boxed{cmd_2(in, out)} \arrow[dl] \\
& \text{Compose} \arrow[d] & \\
& \boxed{\exists tmp.(cmd_1(in, tmp) \land cmd_2(tmp, out))} \arrow[d] & \\
& \text{Simplify} \arrow[dl] \arrow[dr] & \\
\boxed{cmd_{1;2}(in, out)} & & \bot
\end{tikzcd}$$

Symbolic Execution

- Idea: We simulate the script, and collect in our logical formalism its effect on the file system.
- More precisely: Mixed concrete/symbolic execution: We only describe symbolically the effect on the file system, other effects like variable assignements etc. are simulated concretely.
- We know the parameters the script is invoked on, and we make reasonable assumptions on environment variables.

Symbolic Execution

- Idea: We simulate the script, and collect in our logical formalism its effect on the file system.
- More precisely: Mixed concrete/symbolic execution: We only describe symbolically the effect on the file system, other effects like variable assignements etc. are simulated concretely.
- We know the parameters the script is invoked on, and we make reasonable assumptions on environment variables.

Symbolic Execution

- Idea: We simulate the script, and collect in our logical formalism its effect on the file system.
- More precisely: Mixed concrete/symbolic execution: We only describe symbolically the effect on the file system, other effects like variable assignements etc. are simulated concretely.
- We know the parameters the script is invoked on, and we make reasonable assumptions on environment variables.

Plan

1 Introduction

2 Symbolic Execution of Scripts

3 Symbolic Execution of Maintainer Scripts

4 Demo Time

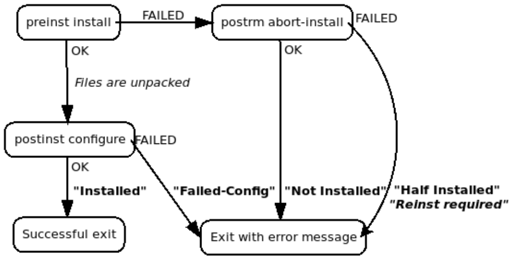
5 Detected Bugs

6 Conclusions

Installation Scenarios

- Second Step: scenarios, like this one:

Installation of foo (Not Installed)

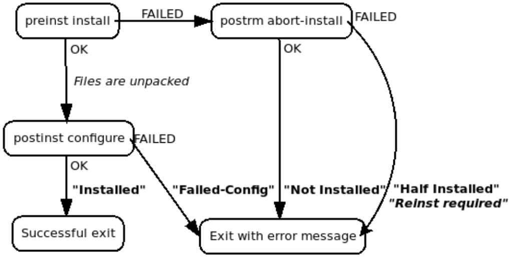


- More (and more complex) scenarios: see the policy.

Installation Scenarios

- Second Step: scenarios, like this one:

Installation of foo (Not Installed)



- More (and more complex) scenarios: see the policy.

Failures and bad states

- Three different kinds of observations:
  1. The failure (exit code > 0) of a maintainer script
  2. The failure of a request to dpkg
  3. The state a package is in at the end of the process
- As one can see in the scenarios:
  - it is possible that a request fails, but still all packages are in a consistent state: when the *error unwind* has worked.
  - there are situations where some script may fail, and still the request succeeds in the end.

Failures and bad states

- Three different kinds of observations:
  1. The failure (exit code > 0) of a maintainer script
  2. The failure of a request to dpkg
  3. The state a package is in at the end of the process
- As one can see in the scenarios:
  - it is possible that a request fails, but still all packages are in a consistent state: when the *error unwind* has worked.
  - there are situations where some script may fail, and still the request succeeds in the end.

Failures and bad states

- Three different kinds of observations:
  1. The failure (exit code > 0) of a maintainer script
  2. The failure of a request to dpkg
  3. The state a package is in at the end of the process
- As one can see in the scenarios:
  - it is possible that a request fails, but still all packages are in a consistent state: when the *error unwind* has worked.
  - there are situations where some script may fail, and still the request succeeds in the end.

Failures and bad states

- Three different kinds of observations:
  1. The failure (exit code > 0) of a maintainer script
  2. The failure of a request to dpkg
  3. The state a package is in at the end of the process
- As one can see in the scenarios:
  - it is possible that a request fails, but still all packages are in a consistent state: when the *error unwind* has worked.
  - there are situations where some script may fail, and still the request succeeds in the end.

Failures and bad states

- Three different kinds of observations:
  1. The failure (exit code > 0) of a maintainer script
  2. The failure of a request to dpkg
  3. The state a package is in at the end of the process
- As one can see in the scenarios:
  - it is possible that a request fails, but still all packages are in a consistent state: when the *error unwind* has worked.
  - there are situations where some script may fail, and still the request succeeds in the end.

Failures and bad states

- Three different kinds of observations:
  1. The failure (exit code > 0) of a maintainer script
  2. The failure of a request to dpkg
  3. The state a package is in at the end of the process
- As one can see in the scenarios:
  - it is possible that a request fails, but still all packages are in a consistent state: when the *error unwind* has worked.
  - there are situations where some script may fail, and still the request succeeds in the end.

Failures and bad states

- Three different kinds of observations:
  1. The failure (exit code > 0) of a maintainer script
  2. The failure of a request to dpkg
  3. The state a package is in at the end of the process
- As one can see in the scenarios:
  - it is possible that a request fails, but still all packages are in a consistent state: when the *error unwind* has worked.
  - there are situations where some script may fail, and still the request succeeds in the end.

Failures and Bugs

- Policy 6.1 says:

  The package management system looks at the exit status from these scripts. It is important that they exit with a non-zero status if there is an error, so that the package management system can stop its processing... It is also important, of course, that they exit with a zero status if everything went well.

- Consequence: A possible failure case of a script is not necessarily a bug!

Failures and Bugs

- Policy 6.1 says:

  > *The package management system looks at the exit status from these scripts. It is important that they exit with a non-zero status if there is an error, so that the package management system can stop its processing... It is also important, of course, that they exit with a zero status if everything went well.*

- Consequence: A possible failure case of a script is not necessarily a bug!

Failures and Bugs

- Policy 6.1 says:

  > *The package management system looks at the exit status from these scripts. It is important that they exit with a non-zero status if there is an error, so that the package management system can stop its processing... It is also important, of course, that they exit with a zero status if everything went well.*

- Consequence: A possible failure case of a script is not necessarily a bug!

Plan

1 Introduction

2 Symbolic Execution of Scripts

3 Symbolic Execution of Maintainer Scripts

4 Demo Time

5 Detected Bugs

6 Conclusions

Infrastructure

- Corpus of 13906 packages containing 33320 maintainer scripts extracted on 2019-03-18 from a Debian mirror
- Corpus of 165 additional files which are included by maintainer scripts
- Using the Contents file to simulate dpkg -L
- Running for 20 minutes on a 80 cores Intel(R) Xeon(R) CPU at 2.20GHz.

Infrastructure

- Corpus of 13906 packages containing 33320 maintainer scripts extracted on 2019-03-18 from a Debian mirror
- Corpus of 165 additional files which are included by maintainer scripts
- Using the Contents file to simulate dpkg -L
- Running for 20 minutes on a 80 cores Intel(R) Xeon(R) CPU at 2.20GHz.

Infrastructure

- Corpus of 13906 packages containing 33320 maintainer scripts extracted on 2019-03-18 from a Debian mirror
- Corpus of 165 additional files which are included by maintainer scripts
- Using the Contents file to simulate dpkg -L
- Running for 20 minutes on a 80 cores Intel(R) Xeon(R) CPU at 2.20GHz.

Infrastructure

- Corpus of 13906 packages containing 33320 maintainer scripts extracted on 2019-03-18 from a Debian mirror
- Corpus of 165 additional files which are included by maintainer scripts
- Using the Contents file to simulate dpkg -L
- Running for 20 minutes on a 80 cores Intel(R) Xeon(R) CPU at 2.20GHz.

Plan

1 Introduction

2 Symbolic Execution of Scripts

3 Symbolic Execution of Maintainer Scripts

4 Demo Time

5 Detected Bugs

6 Conclusions

sgml-base preinst

- Script snippet:

```
if [ ! -d /var/lib/sgml-base ]
then
  mkdir /var/lib/sgml-base 2>/dev/null
fi
```

- Problem: If `/var/lib/sgml-base` exists and is not a directory this fails *silently*
- We have asked on the mailing list for confirmation that this is a bug.
- `https://bugs.debian.org/929706`

sgml-base preinst

- Script snippet:

```
if [ ! -d /var/lib/sgml-base ]
then
  mkdir /var/lib/sgml-base 2>/dev/null
fi
```

- Problem: If `/var/lib/sgml-base` exists and is not a directory this fails *silently*
- We have asked on the mailing list for confirmation that this is a bug.
- `https://bugs.debian.org/929706`

sgml-base preinst

- Script snippet:

```
if [ ! -d /var/lib/sgml-base ]
then
  mkdir /var/lib/sgml-base 2>/dev/null
fi
```

- Problem: If `/var/lib/sgml-base` exists and is not a directory this fails *silently*
- We have asked on the mailing list for confirmation that this is a bug.
- `https://bugs.debian.org/929706`

sgml-base preinst

- Script snippet:

```
if [ ! -d /var/lib/sgml-base ]
then
  mkdir /var/lib/sgml-base 2>/dev/null
fi
```

- Problem: If `/var/lib/sgml-base` exists and is not a directory this fails *silently*
- We have asked on the mailing list for confirmation that this is a bug.
- `https://bugs.debian.org/929706`

armagetronad-dedicated postrm

- Script snippet:

```
if [ "$1" = "purge" ]; then
  rm -r /var/games/armagetronad
  rmdir --ignore-fail-on-non-empty /var/games
fi
```

- Will fail if `/var/games/armagedtronad` does not exist.
- Do we have to account for this case?
- Policy, section 6.2: Maintainer scripts have to be idempotent.
- Note that if a `postrm purge` succeeds the package is gone completely.
- We still think this is a bug since the script may fail later.

armagetronad-dedicated postrm

- Script snippet:

```
if [ "$1" = "purge" ]; then
  rm -r /var/games/armagetronad
  rmdir --ignore-fail-on-non-empty /var/games
fi
```

- Will fail if `/var/games/armagedtronad` does not exist.
- Do we have to account for this case?
- Policy, section 6.2: Maintainer scripts have to be idempotent.
- Note that if a `postrm purge` succeeds the package is gone completely.
- We still think this is a bug since the script may fail later.

armagetronad-dedicated postrm

- Script snippet:

```
if [ "$1" = "purge" ]; then
  rm -r /var/games/armagetronad
  rmdir --ignore-fail-on-non-empty /var/games
fi
```

- Will fail if `/var/games/armagedtronad` does not exist.
- Do we have to account for this case?
- Policy, section 6.2: Maintainer scripts have to be idempotent.
- Note that if a `postrm purge` succeeds the package is gone completely.
- We still think this is a bug since the script may fail later.

armagetronad-dedicated postrm

- Script snippet:

```
if [ "$1" = "purge" ]; then
  rm -r /var/games/armagetronad
  rmdir --ignore-fail-on-non-empty /var/games
fi
```

- Will fail if `/var/games/armagedtronad` does not exist.
- Do we have to account for this case?
- Policy, section 6.2: Maintainer scripts have to be idempotent.
- Note that if a `postrm purge` succeeds the package is gone completely.
- We still think this is a bug since the script may fail later.

armagetronad-dedicated postrm

- Script snippet:

```
if [ "$1" = "purge" ]; then
  rm -r /var/games/armagetronad
  rmdir --ignore-fail-on-non-empty /var/games
fi
```

- Will fail if `/var/games/armagedtronad` does not exist.
- Do we have to account for this case?
- Policy, section 6.2: Maintainer scripts have to be idempotent.
- Note that if a `postrm purge` succeeds the package is gone completely.
- We still think this is a bug since the script may fail later.

armagetronad-dedicated postrm

- Script snippet:

```
if [ "$1" = "purge" ]; then
  rm -r /var/games/armagetronad
  rmdir --ignore-fail-on-non-empty /var/games
fi
```

- Will fail if `/var/games/armagedtronad` does not exist.
- Do we have to account for this case?
- Policy, section 6.2: Maintainer scripts have to be idempotent.
- Note that if a `postrm purge` succeeds the package is gone completely.
- We still think this is a bug since the script may fail later.

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.

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.

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.

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.

courier-filter-perl postrm

- Script snippet:

```
case "$1" in
  purge )
    rm /etc/courier/filters/courier-filter-perl.conf
    ;;
esac
```

- Will fail when `.../courier-filter-perl.conf` does not exist: script not idempotent.
- However, this is at the end of script, so when it succeeds and removes the file the package is gone, so this seems purely formal.

courier-filter-perl postrm

- Script snippet:

```
case "$1" in
  purge )
    rm /etc/courier/filters/courier-filter-perl.conf
    ;;
esac
```

- Will fail when `.../courier-filter-perl.conf` does not exist: script not idempotent.
- However, this is at the end of script, so when it succeeds and removes the file the package is gone, so this seems purely formal.

courier-filter-perl postrm

- Script snippet:

```
case "$1" in
  purge )
    rm /etc/courier/filters/courier-filter-perl.conf
    ;;
esac
```

- Will fail when `.../courier-filter-perl.conf` does not exist: script not idempotent.
- However, this is at the end of script, so when it succeeds and removes the file the package is gone, so this seems purely formal.

oz postrm

- Script snippet:

```
FILE="/etc/oz/id_rsa-icicle-gen"
case "$1" in
    purge)
    if [ -f $FILE ]; then
        rm $FILE $FILE.pub
    fi
    ;;
esac
```

- Fails if `$FILE` exists but `$FILE.pub` does not.
- In that case, a second invocation of `postrm purge` will succeed!
- Even if it is not against idempotency, this behavior is at least strange and annoying.

oz postrm

- Script snippet:

```
FILE="/etc/oz/id_rsa-icicle-gen"
case "$1" in
    purge)
    if [ -f $FILE ]; then
        rm $FILE $FILE.pub
    fi
    ;;
esac
```

- Fails if `$FILE` exists but `$FILE.pub` does not.
- In that case, a second invocation of `postrm purge` will succeed!
- Even if it is not against idempotency, this behavior is at least strange and annoying.

oz postrm

- Script snippet:

```
FILE="/etc/oz/id_rsa-icicle-gen"
case "$1" in
    purge)
    if [ -f $FILE ]; then
        rm $FILE $FILE.pub
    fi
    ;;
esac
```

- Fails if `$FILE` exists but `$FILE.pub` does not.
- In that case, a second invocation of `postrm purge` will succeed!
- Even if it is not against idempotency, this behavior is at least strange and annoying.

oz postrm

- Script snippet:

```
FILE="/etc/oz/id_rsa-icicle-gen"
case "$1" in
    purge)
    if [ -f $FILE ]; then
        rm $FILE $FILE.pub
    fi
    ;;
esac
```

- Fails if `$FILE` exists but `$FILE.pub` does not.
- In that case, a second invocation of `postrm purge` will succeed!
- Even if it is not against idempotency, this behavior is at least strange and annoying.

Bugs found by Colis

- Listing: `https://bugs.debian.org/cgi-bin/pkgreport.cgi?tag=colis-shparser;users=treinen@debian.org`
- 148 bugs filed so far, 90 of which are solved.
- So far a great majority are on a trivial level (like missing `set -e`), or on the level of syntactic structure (requires `morbig`, hence is not trivial).
- How did we find the last four bugs:
  - The first two from bad package states detected by our tool, then investigation by hand.
  - The last two where found by running our tool on a dedicated scenario for testing a subcase of idempotency.

Bugs found by Colis

- Listing: `https://bugs.debian.org/cgi-bin/pkgreport.cgi?tag=colis-shparser;users=treinen@debian.org`
- 148 bugs filed so far, 90 of which are solved.
- So far a great majority are on a trivial level (like missing `set -e`), or on the level of syntactic structure (requires `morbig`, hence is not trivial).
- How did we find the last four bugs:
  - The first two from bad package states detected by our tool, then investigation by hand.
  - The last two where found by running our tool on a dedicated scenario for testing a subcase of idempotency.

Bugs found by Colis

- Listing: `https://bugs.debian.org/cgi-bin/pkgreport.cgi?tag=colis-shparser;users=treinen@debian.org`
- 148 bugs filed so far, 90 of which are solved.
- So far a great majority are on a trivial level (like missing `set -e`), or on the level of syntactic structure (requires `morbig`, hence is not trivial).
- How did we find the last four bugs:
  - The first two from bad package states detected by our tool, then investigation by hand.
  - The last two where found by running our tool on a dedicated scenario for testing a subcase of idempotency.

Bugs found by Colis

- Listing: `https://bugs.debian.org/cgi-bin/pkgreport.cgi?tag=colis-shparser;users=treinen@debian.org`
- 148 bugs filed so far, 90 of which are solved.
- So far a great majority are on a trivial level (like missing `set -e`), or on the level of syntactic structure (requires `morbig`, hence is not trivial).
- How did we find the last four bugs:
  - The first two from bad package states detected by our tool, then investigation by hand.
  - The last two where found by running our tool on a dedicated scenario for testing a subcase of idempotency.

Bugs found by Colis

- Listing: `https://bugs.debian.org/cgi-bin/pkgreport.cgi?tag=colis-shparser;users=treinen@debian.org`
- 148 bugs filed so far, 90 of which are solved.
- So far a great majority are on a trivial level (like missing `set -e`), or on the level of syntactic structure (requires `morbig`, hence is not trivial).
- How did we find the last four bugs:
  - The first two from bad package states detected by our tool, then investigation by hand.
  - The last two where found by running our tool on a dedicated scenario for testing a subcase of idempotency.

Bugs found by Colis

- Listing: `https://bugs.debian.org/cgi-bin/pkgreport.cgi?tag=colis-shparser;users=treinen@debian.org`
- 148 bugs filed so far, 90 of which are solved.
- So far a great majority are on a trivial level (like missing `set -e`), or on the level of syntactic structure (requires `morbig`, hence is not trivial).
- How did we find the last four bugs:
  - The first two from bad package states detected by our tool, then investigation by hand.
  - The last two where found by running our tool on a dedicated scenario for testing a subcase of idempotency.

Plan

1 Introduction

2 Symbolic Execution of Scripts

3 Symbolic Execution of Maintainer Scripts

4 Demo Time

5 Detected Bugs

6 Conclusions

Ongoing Work

- Include simulation of the *unpack* phase.
- Increase the number of script we can handle, by modeling more commands.
- Being more precise about idempotency: checking *equivalence* of the executing a script once or twice.
- This uses our result on *decidability* of the logic.
- Investigate other properties, like commutation of scripts.
- Using *tree transducers* to represent the semantics of scripts.

Ongoing Work

- Include simulation of the *unpack* phase.
- Increase the number of script we can handle, by modeling more commands.
- Being more precise about idempotency: checking *equivalence* of the executing a script once or twice.
- This uses our result on *decidability* of the logic.
- Investigate other properties, like commutation of scripts.
- Using *tree transducers* to represent the semantics of scripts.

Ongoing Work

- Include simulation of the *unpack* phase.
- Increase the number of script we can handle, by modeling more commands.
- Being more precise about idempotency: checking *equivalence* of the executing a script once or twice.
- This uses our result on *decidability* of the logic.
- Investigate other properties, like commutation of scripts.
- Using *tree transducers* to represent the semantics of scripts.

Ongoing Work

- Include simulation of the *unpack* phase.
- Increase the number of script we can handle, by modeling more commands.
- Being more precise about idempotency: checking *equivalence* of the executing a script once or twice.
- This uses our result on *decidability* of the logic.
- Investigate other properties, like commutation of scripts.
- Using *tree transducers* to represent the semantics of scripts.

Ongoing Work

- Include simulation of the *unpack* phase.
- Increase the number of script we can handle, by modeling more commands.
- Being more precise about idempotency: checking *equivalence* of the executing a script once or twice.
- This uses our result on *decidability* of the logic.
- Investigate other properties, like commutation of scripts.
- Using *tree transducers* to represent the semantics of scripts.

Ongoing Work

- Include simulation of the *unpack* phase.
- Increase the number of script we can handle, by modeling more commands.
- Being more precise about idempotency: checking *equivalence* of the executing a script once or twice.
- This uses our result on *decidability* of the logic.
- Investigate other properties, like commutation of scripts.
- Using *tree transducers* to represent the semantics of scripts.

Thank you

- Joint work with the people from the Colis project.
- Project ANR-15-CE25-0001 funded by *Agence Nationale de Recherche*.
- October 2015 – September 2020
- `http://colis.irif.fr/`

Academic Papers

- NJ, CM, RT: *A Formally Verified Interpreter for a Shell-like Programming Language*, VSTTE 2017, `https://hal.archives-ouvertes.fr/hal-01534747`
- YRG, NJ, RT: *Morbig: A Static Parser for POSIX Shell*, SLE 2018, `https://hal.archives-ouvertes.fr/hal-01890044`
- NJ, RT: *Deciding the First-Order Theory of an Algebra of Feature Trees with Updates*, IJCAR 2018, `https://hal.archives-ouvertes.fr/hal-01807474`
- BB, CM: *Ghost Code in Action: Automated Verification of a Symbolic Interpreter*, VSTTE 2019.

dpkg-maintscript-helper

- This is a utility that may be used *by* maintainer scripts
- Script snippet:

```
find "$PATHNAME" -mindepth 1 -print0 | \
    xargs -0 -i% mv -f "%" "$ABS_SYMLINK_TARGET/"
```

- Fails when `"$PATHNAME"` contains subdirectories
- Solution: add option `"-maxdepth 1"` to `find`
- `https://bugs.debian.org/922799` (our proposed fix was accepted)

dpkg-maintscript-helper

- This is a utility that may be used *by* maintainer scripts
- Script snippet:

```
find "$PATHNAME" -mindepth 1 -print0 | \
    xargs -0 -i% mv -f "%" "$ABS_SYMLINK_TARGET/"
```

- Fails when `"$PATHNAME"` contains subdirectories
- Solution: add option `"-maxdepth 1"` to `find`
- `https://bugs.debian.org/922799` (our proposed fix was accepted)

dpkg-maintscript-helper

- This is a utility that may be used *by* maintainer scripts
- Script snippet:

```
find "$PATHNAME" -mindepth 1 -print0 | \
    xargs -0 -i% mv -f "%" "$ABS_SYMLINK_TARGET/"
```

- Fails when `"$PATHNAME"` contains subdirectories
- Solution: add option `"-maxdepth 1"` to `find`
- `https://bugs.debian.org/922799` (our proposed fix was accepted)

dpkg-maintscript-helper

- This is a utility that may be used *by* maintainer scripts
- Script snippet:

```
find "$PATHNAME" -mindepth 1 -print0 | \
    xargs -0 -i% mv -f "%" "$ABS_SYMLINK_TARGET/"
```

- Fails when `"$PATHNAME"` contains subdirectories
- Solution: add option `"-maxdepth 1"` to `find`
- `https://bugs.debian.org/922799` (our proposed fix was accepted)

dpkg-maintscript-helper

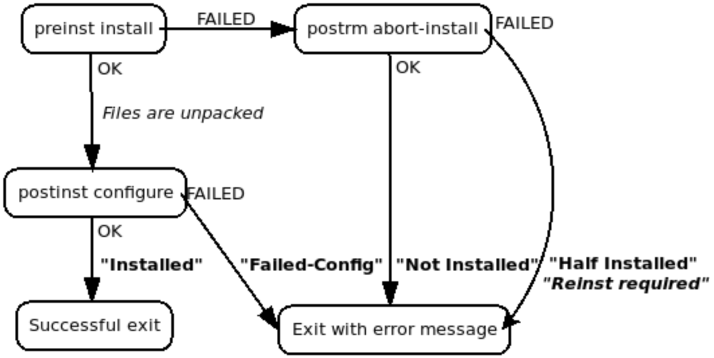
- This is a utility that may be used *by* maintainer scripts
- Script snippet:

```
find "$PATHNAME" -mindepth 1 -print0 | \
    xargs -0 -i% mv -f "%" "$ABS_SYMLINK_TARGET/"
```

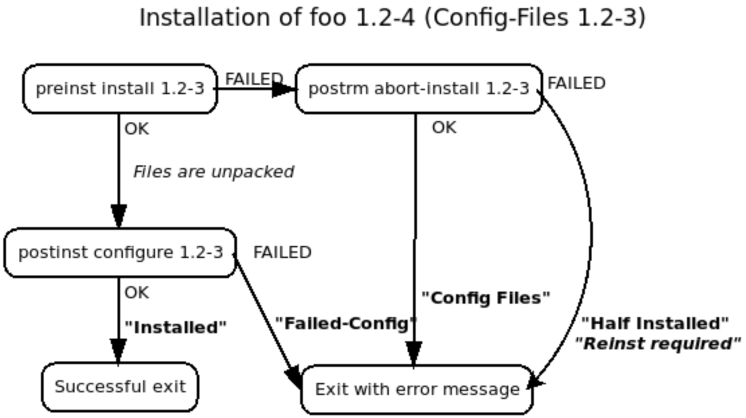
- Fails when `"$PATHNAME"` contains subdirectories
- Solution: add option `"-maxdepth 1"` to `find`
- `https://bugs.debian.org/922799` (our proposed fix was accepted)

Scenario: fresh installation

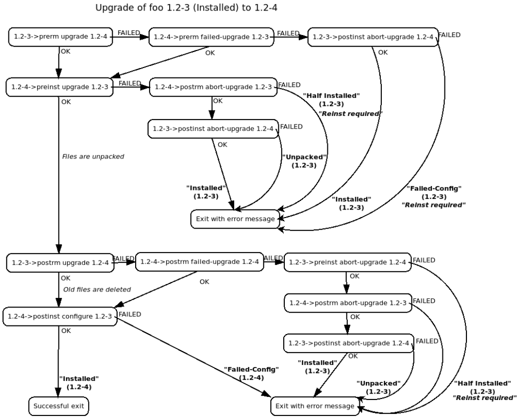
Installation of foo (Not Installed)



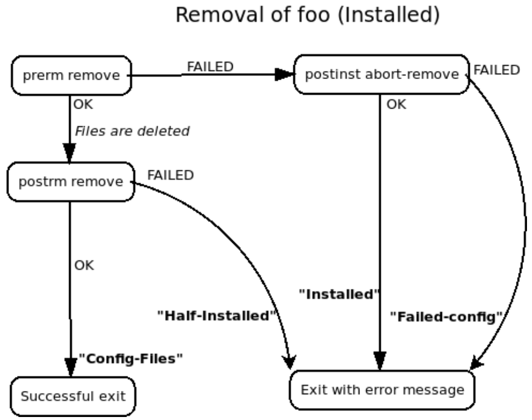
Scenario: installation of previously removed package



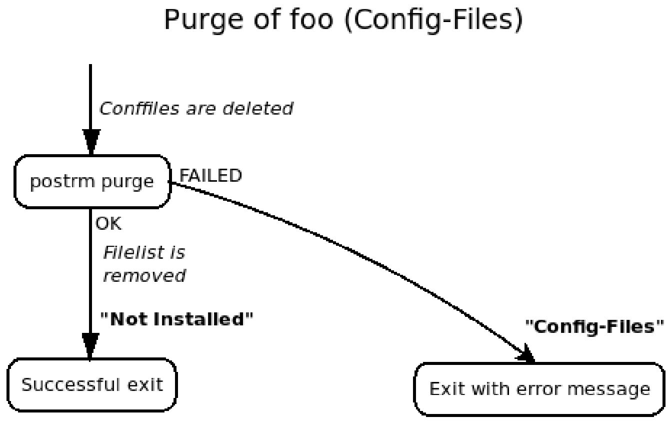
Scenario: upgrade of an installed package


Upgrade of foo 1.2-3 (Installed) to 1.2-4
1.2-3->prerm upgrade 1.2-4
FAILED
1.2-4->prerm failed-upgrade 1.2-3
FAILED
1.2-3->postinst abort-upgrade 1.2-4
FAILED
OK
OK
OK
1.2-4->preinst upgrade 1.2-3
FAILED
1.2-4->postrm abort-upgrade 1.2-3
FAILED
"Half Installed" (1.2-3) "Reinst required"
OK
OK
1.2-3->postinst abort-upgrade 1.2-4
FAILED
OK
Files are unpacked
"Unpacked" (1.2-3)
"Installed" (1.2-3)
"Installed" (1.2-3)
"Failed-Config" (1.2-3) "Reinst required"
Exit with error message
1.2-3->postrm upgrade 1.2-4
FAILED
1.2-4->postrm failed-upgrade 1.2-4
FAILED
1.2-3->preinst abort-upgrade 1.2-4
FAILED
OK
OK
OK
Old files are deleted
1.2-4->postrm abort-upgrade 1.2-3
FAILED
1.2-4->postinst configure 1.2-3
FAILED
OK
OK
1.2-3->postinst abort-upgrade 1.2-4
FAILED
OK
"Installed" (1.2-3)
"Installed" (1.2-4)
"Failed-Config" (1.2-4)
"Unpacked" (1.2-3)
"Half Installed" (1.2-3) "Reinst required"
Successful exit
Exit with error message

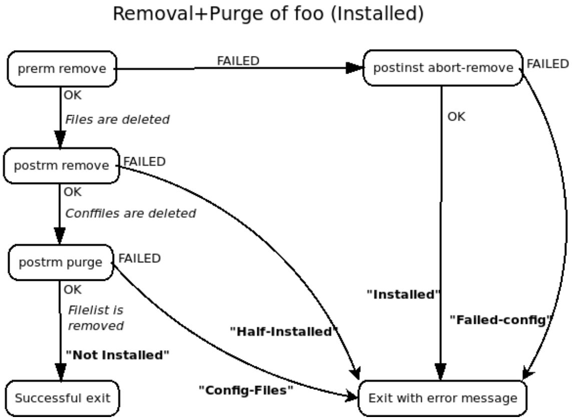
Scenario: removal of an installed package



Scenario: purge of a removed package



Scenario: purge of an installed package


Removal+Purge of foo (Installed)
prerm remove
FAILED
postinst abort-remove
FAILED
OK
Files are deleted
OK
postrm remove
FAILED
OK
Conffiles are deleted
postrm purge
FAILED
OK
Filelist is removed
"Not Installed"
"Installed"
"Failed-config"
"Half-Installed"
"Config-Files"
Successful exit
Exit with error message