The CoLiS Platform for the Analysis of Maintainer Scripts in Debian Software Packages

Benedikt Becker, Nicolas Jeannerod, Claude Marché, Yann Régis-Gianas, Mihaela Sighireanu, Ralf Treinen

Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

The CoLiS Platform for the Analysis of Maintainer Scripts in Debian Software Packages

Benedikt Becker · Nicolas Jeannerod · Claude Marché · Yann Régis-Gianas · Mihaela Sighireanu · Ralf Treinen

Received: date / Accepted: date

Abstract The software packages of the Debian distribution include more than twenty-seven thousand maintainer scripts in total, almost all of them being written in the Posix shell language. These scripts are executed with root privileges at installation, update, and removal of a package, which makes them critical for system maintenance. While the Debian policy provides guidance for package maintainers producing the scripts, only few tools exist to check the compliance of a script to that policy. We present CoLiS, a software platform for discovering violations of non-trivial properties required by the Debian policy in maintainer scripts. We describe our methodology which is based on symbolic execution and feature tree constraints, and we give an overview of the toolchain. We obtain promising results: our toolchain is effective in analysing a large set of Debian maintainer scripts, and it has already detected over 150 policy violations that have lead to bug reports, more than two-third of them now being fixed.

Keywords Quality Assurance · Safety Properties · Debian distribution · Software Package Installation · Shell Scripts · High-Level View of File Hierarchies · Symbolic Execution · Feature Tree Constraints

This work has been partially supported by the ANR project CoLiS, contract number ANR-15-CE25-0001.

Benedikt Becker
Université Paris-Saclay, CNRS, Inria, LMF, 91190 Gif-sur-Yvette, France

Nicolas Jeannerod
Université Paris Cité, CNRS, IRIF, 75013 Paris, France; moving to Tweag I/O

Claude Marché
Université Paris-Saclay, CNRS, Inria, LMF, 91190 Gif-sur-Yvette, France

Yann Régis-Gianas
Université Paris Cité, CNRS, IRIF, 75013 Paris, France; on leave at Nomadic Labs

Mihaela Sighireanu
Université Paris-Saclay, CNRS, ENS Paris-Saclay, LMF, 91190 Gif-sur-Yvette, France

Ralf Treinen
Université Paris Cité, CNRS, IRIF, 75013 Paris, France

1 Introduction

The Debian distribution is one of the oldest free software distributions. The latest released version 11 (code name "bullseye") provides 59,551 binary packages built from 31,358 software source packages with an official support for nine different CPU architectures. It is one of the most used GNU/Linux distributions, and serves as the basis for some derived distributions like Ubuntu.

A software package of Debian contains an archive of files to be placed on the target machine when installing the package. The package may come with a number of so-called *maintainer scripts* that are executed when installing, upgrading, or removing the package. The released version "bullseye" of Debian (for the `amd64` architecture and including the `contrib` and `non-free` collections, as of August 14, 2021) contains 27,324 maintainer scripts in 11,640 different packages, 10,468 of which are completely or partially written by hand. These scripts are used for tasks like cleaning up, configuration, and repairing mistakes introduced in older versions of the distribution. Since they may have to perform any action on the target machine, the scripts are almost exclusively written in some general-purpose scripting language that allows for invoking any Unix command.

The whole installation process is orchestrated by `dpkg`, a Debian-specific tool, which executes the maintainer scripts of each package according to *scenarios*. Executing the `dpkg` tool and the scripts requires *root* privileges, that is, maximal administrator rights for reading, writing files and directories, and executing commands. For this reason, the failure of one of these scripts may lead to effects ranging from mildly annoying (like spurious warnings) to catastrophic (*e.g.*, removal of files belonging to unrelated packages, as already reported [43] in 2007 against a package maintained by one of the authors of this paper). When an execution error of a maintainer script is detected, the `dpkg` tool attempts an *error unwind*, but the success of this operation depends again on the correct behaviour of maintainer scripts. There is no general mechanism to simply undo the unwanted effects of a failed installation attempt, short of using a file system implementation providing for snapshots.

The *Debian policy* [3] aims to normalise, in natural language, important technical aspects of packages. Concerning the maintainer scripts we are interested in, it states that the standard shell interpreter is the Posix shell, with the consequence that 99% of all maintainer scripts are written in this language. The policy also sets down the control flow of the different stages of the package installation process, including attempts for error recovery, defines how `dpkg` invokes maintainer scripts, and states some requirements on the execution behaviour of scripts. One of these requirements is the *idempotency* of scripts, that is, if an installation operation is done, successfully or signaling a failure (*e.g.*, because the disc is full), and is repeated afterwards (*e.g.*, after freeing space on the disc), then the system state must be the same as if it was already successful the first time. Most of these properties are until today checked on a very basic syntactic level (using tools like Lintian [31]), by automated testing (like the `piuparts` suite [35]), or simply left until someone stumbles upon a bug and reports it to Debian.

1.1 Objectives of the Study

The goal of our study is to improve the quality of the installation of software packages in the Debian distribution using a formal (*i.e.*, based on a mathematical approach), static (*i.e.* without actually running scripts on a real system) and automated approach. We focus on bug finding, as opposed to a complete formal proof of absence of bugs, for three reasons. Firstly, a real Unix-like operating system is obviously too complex to be described completely and accurately by some formal model. Besides, the formal correctness properties may be difficult to apprehend when they are expressed on an abstract model. Finally, when a bug is detected, even on a system abstraction, one can try to reproduce it on a real system and, if confirmed, report it to the authors. This has a real and immediate impact on the quality of the software and helps to promote the usage of formal methods to a community that often is rather sceptical towards methods and tools coming from academic research.

The bugs in Debian maintainer scripts that we attempt to find may come at different levels: simple syntax errors (which may go unnoticed due to the unsafe design of the Posix shell language), non-compliance with the requirements of the Debian policy, usage of unofficial or undocumented features, or failure of a script in a situation where it is supposed to succeed.

1.2 Challenges of the Study

The challenges are multiple: the Posix shell language is highly dynamic and recalcitrant to static analysis, both on a syntactic and semantic level. A Unix file system implementation contains many features that are difficult to model, like file ownership, permissions, timestamps, symbolic links, and multiple hard links to regular files. There are an immense variety of Unix commands that may be invoked from scripts, all of which have to be modelled in order to be treated by our approach.

To address properties of scripts required by the Debian policy, we need to capture the transformation done by the script on a file system hierarchy. For this, we need some kind of logic that is expressive enough, and still allows for automated reasoning methods. A particular challenge is checking the idempotency property for script execution because it requires relational reasoning. For this, we encode the semantics of a script as a logic formula specifying the relation between the input and the output of the script, and we check that it is equivalent to its composition with itself. Finally, all these challenges have to be met at the scale of tens of thousands of scripts.

1.3 Summary of our Contributions

The contributions of the work for this study are:

1. A translation of Debian maintainer scripts into a language with formal semantics, and a formalisation of properties required for the execution of these scripts by the Debian policy.

```
if [ -h /etc/rancid/lg.conf ]; then
  rm /etc/rancid/lg.conf
fi
if [ -e /etc/rancid/apache.conf ]; then
  rm /etc/rancid/apache.conf
fi
```

Fig. 1 The `preinst` script of the `rancid-cgi` package. Informally, if the symbolic link `/etc/rancid/lg.conf` exists then it is removed; and then, if the file `/etc/rancid/apache.conf` exists, no matter its type, it is also removed. Both removal operations use the Posix command `rm` which, without options, cannot remove directories. Hence, we can immediately notice that if `/etc/rancid/apache.conf` is a directory, this script fails while trying to remove it.

2. A verification toolchain for maintainer scripts based on an existing symbolic execution engine [7,8] and a symbolic representation [30]. Some components of this toolchain have been published independently; we improved them to cope with this study. The toolchain is free software available online [40].
3. A formal specification of the transformations done by an important set of Posix commands [28] in *feature tree constraints* [30].
4. A number of bugs found by our method in recent versions of Debian packages.

We start in the next section with an overview of our method illustrated on a concrete example. Section 3 explains in greater detail the elements of our toolchain, the particular challenges, the hypotheses that we could make for the specific Debian use case at hand, and the solution we are proposing. Section 4 presents the results we obtained so far on the Debian packages, and the lessons learnt. We conclude in Section 5 by discussing additional outcomes of this study, and the related and future work.

2 Overview of the Study and Analysis Methodology

2.1 Structure of a Debian Package

Three components of a Debian binary package play an important role in the installation process: the *static content*, that is the archive of files to be placed on the target machine when installing the package; the lists of *dependencies* and *pre-dependencies*, which tell us which packages can be assumed present at different moments; and the *maintainer scripts*, that is a possibly empty subset of four scripts called `preinst`, `postinst`, `prerm`, and `postrm`. We found (see Section 4.2) that 99% of the maintainer scripts in Debian are written in the Posix shell language [25].

Our running example is the binary package `rancid-cgi` [37]. It comes with only two maintainer scripts: `preinst` and `postinst`. Figure 1 presents the `preinst` script.

We did a statistical analysis of maintainer scripts in Debian to help us design our intermediate language, see Section 4.2 for details. We found that, for instance, most variables in these scripts can be expanded statically and hence are used like constants; most `while` loops can be translated into `for` loops; recursive functions

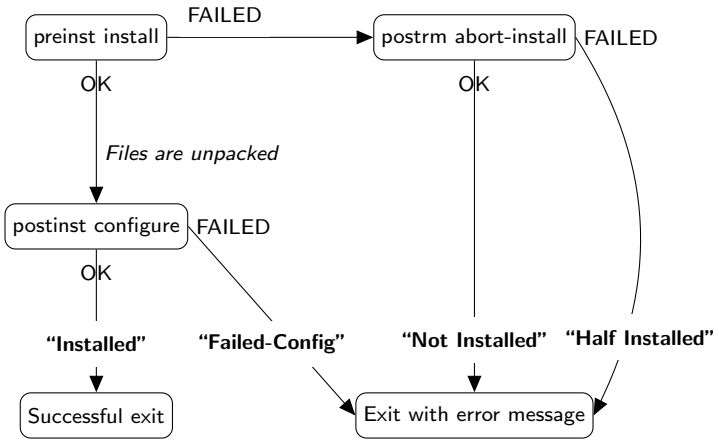

Fig. 2 Debian flowchart for installing a package [3, Appendix 9]. The states represent calls to maintainer scripts with their arguments and actions performed externally to scripts are in italics; the status returned by `dpkg` at the end of the process is in bold.

are not used at all; redirections are almost always used to discard the standard output of commands.

2.2 Workflow of Installation, Upgrade, and Removal of a Package

The maintainer scripts are invoked by the `dpkg` utility when installing, removing or upgrading a package. Roughly speaking, for installation, `dpkg` calls the `preinst` before the package static content is unpacked, and calls the `postinst` afterwards. For deinstallation, `dpkg` calls the `prerm` before the static content is removed and calls the `postrm` afterwards. The precise sequence of script invocations and the actual script parameters are defined by informal flowcharts in the Debian policy [3, Appendix 9]. Figure 2 shows the flowchart for the package installation. The `dpkg` utility may be asked to: install a package that was not previously installed (Figure 2), install a package that was previously removed but not purged, upgrade a package, remove a package, purge a package previously removed, remove and purge a package. These tasks include 39 possible execution paths, 4 of them presented in Figure 2.

2.3 Requirements on Maintainer Scripts

The Debian policy contains [3, Chapters 6 and 10] several requirements on maintainer scripts. This case study targets checking the requirements regarding the execution of scripts, and considers out of scope some other kinds of requirements, for example the permissions of script files. The requirements of interest are checked by different tools of our toolchain presented in Section 3. For example, the different ways to invoke a maintainer script are handled by the analysis of scenarios

(Section 3.5) calling the scripts. Different requirements on the usage of the shell language are checked by the syntactic analysis (Section 3.1.1), like the usage of `-e` mode or of authorised shell features that are optional in the Posix standard. Some of the usage requirements can be detected by a semantic analysis; this is done in our toolchain by a translation into a formally defined language, called CoLiS (Section 3.1.2). Finally, requirements concerning the behaviour of scripts include the usage of exit codes and the *idempotency* of scripts. The last property is difficult to formalise since it refers to possible unforeseen failures (see discussion in Section 4.4). Formally checking behavioural properties of scripts requires reasoning about their semantics, which is done by a symbolic execution in our toolchain (Section 3.3). We also check some requirements that are simply common sense and that are not stated in the policy, for example invoking Unix commands with correct options. This is done by the semantic analysis (Section 3.1.2).

2.4 Principles and Workflow of the Analysis Method

Our goal is to check the above properties of maintainer scripts in a formal and static way, by analysing each script and the composition of scripts in the execution paths exhibited by the flowcharts of `dpkg`. More precisely, we check the conformity of execution of scripts with respect to an expected *scenario*. In our setting, a scenario is either (1) an execution path of `dpkg` as exemplified in Figure 2, (2) a single execution of a script, or (3) a double execution of a script with the same parameters (to check idempotency). Section 3.5 presents scenarios in more details.

The analysis should consider a variety of states for the system on which the execution takes place. Yet we assume the following hypotheses:

- the scripts are executed in a root-privileged process without concurrency with other (user or root) processes,
- the static content of the package is successfully unpacked,
- the dependencies defined by the package are present (fact checked by `dpkg`), and
- the `/bin/sh` command implements the standard Posix.1-2017 Shell Command Language with the additional features described in the Debian policy [3, Chapter 10].

The components of our toolchain for the analysis of a scenario are summarised in Figure 3 and detailed in Section 3.

2.5 Presentation of Results

The results computed by the scenario player are presented in a set of web pages, one per scenario, and a summary page for the package [6]. Each scenario may have several computed exit codes; for an error code, the associated symbolic relation is translated automatically into a diagnosis message.

For example, consider the simple scenario of a call to the script `preinst` given in Figure 1. The resulting web page includes the diagram in Figure 4, which is obtained by the interpretation of the symbolic relation computed by the scenario player for the error exit code.

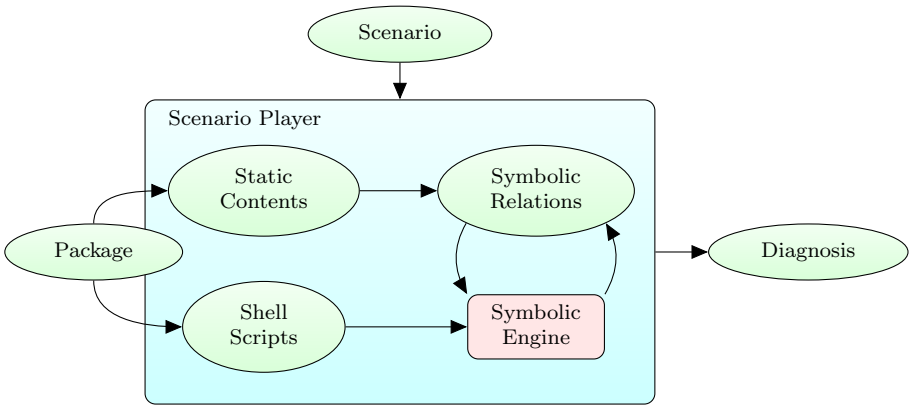

Fig. 3 The CoLiS toolchain for analysis of a scenario on a given package. Given a package and one scenario, the scenario player extracts the static content and the maintainer scripts, prepares the initial *symbolic state* of the scenario, symbolically executes the steps of the scenario to compute a *symbolic relation* between the input and the output states of the file system for each outcome of the scenario, and produces a diagnosis.

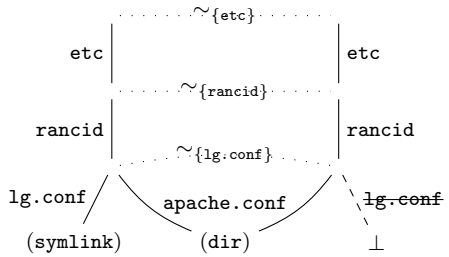

Fig. 4 Example of diagnosis: error case for `preinst` call in the package `rancid-cgi`. This diagram displays knowledge about the initial file system (on the left side), and about the file system resulting from an attempt to execute the script (on the right). The respective roots of the two filesystem are on the top, downward edges represent links in the file system. A dotted edge describes a similarity relation, *e.g.*, the $\sim_{\{\texttt{rancid}\}}$ on the second dotted line means that the trees rooted at `/etc` coincide except possibly on the child named `rancid`. $\perp$ denotes the absence of a node. Finally, a leaf can be annotated by a property, *e.g.*, the annotation (`dir`) rooted at `/etc/rancid/apache.conf` expresses that this node is a directory.

The diagram shows that the `preinst` script leads to an error state when the file `/etc/rancid/apache.conf` is a directory, which is expected as we already noticed that the `rm` command cannot remove directories. Note that in this case the failed attempt to execute the script leaves the file system modified since the symbolic link `/etc/rancid/lg.conf` has been removed.

Finally, another set of generated web pages [6] provides statistics on the coverage and the errors found for the full set of scenarios of the Debian distribution.

3 Design and Implementation of the Tool Chain

The toolchain, as described in Figure 3, hinges on a *symbolic execution engine* that computes the overall effect of a script on the file system as a symbolic relation between the input and the output file system. This section details this toolchain. It is composed first of a front-end that parses the script and translates it into a script in an intermediate language called CoLiS, having a formally defined semantics (see Section 3.1). Second, a back-end symbolically executes the CoLiS scripts, so as to get, for each outcome of a script, the relation between the input and the output file systems, encoded by a *tree constraint* (Section 3.2). The core of the symbolic execution engine (Section 3.3) is a program written in the WhyML language, allowing us to certify, thanks to the Why3 verification environment [9], the adequacy of the symbolic execution with respect to the semantics of CoLiS. A basic component for such an engine is, for each Unix utility, a formal specification of its behaviour, again as an input-output relation (Section 3.4).

3.1 The Front-End

3.1.1 The morbig Parser for Shell Scripts

The syntax of the Posix shell language is unconventional in many aspects. For this reason, the implementation of a parser for Posix shell scripts cannot simply reuse the standard techniques solely based on code generators (such as `yacc`). Most of the shell implementations fall back to manually written character-level parsers, which are difficult to maintain and to trust.

The morbig parser was designed by some of the authors in 2018 [36]. Its implementation aims at employing code generators as much as possible to keep the implementation at a high level of abstraction, to simplify its maintenance, and to ease checking of compliance with the Posix standard.

To validate the parser, we compare its behavior to another Posix compliant parser, namely `dash`, using a large corpus of scripts found in the Software Heritage Archive [1]. The morbig parser and `dash` agree on 95% of these $7,436,215$ files. By studying the remaining 5%, we found bugs in both parsers.

The morbig parser was used in our project to build several tools. For instance, the Debian Policy defines some rules that are easily expressible as pattern-matching over the concrete syntax trees produced by morbig. As another example, morbig provides the basis for a statistical analysis tool for the corpus of Debian maintainer scripts. This analysis aims at identifying the most frequently used, or the most rarely used constructions of Posix shell. The results of this analysis had an impact on the design of the CoLiS language presented in the next section as it shows that the full complexity of the Posix shell language is not used by programmers to write critical pieces of code. For instance, `while` loops are mostly used in the corpus of maintainer scripts in order to bypass a restriction of the shell that makes it difficult to iterate over a list of strings that contain the string separator. Hence, this idiom can be translated to a `for` loop in the CoLiS language which is properly typed; see also Section 4.2.

3.1.2 The CoLiS *language*

Our initial design of the CoLiS language was presented in 2017 [27]. This design was guided by the following requirements.

- A need to avoid some pitfalls of the Posix shell, and to make explicit the dangerous constructs we cannot eliminate.
- A need to have a clear syntax and semantics. Indeed, by providing a *formal* semantics, we are sure that no ambiguity on the semantics remains.
- An automated and fairly straightforward translation from shell must be possible: since the correctness of the translation from shell to CoLiS cannot be proven formally, it must be trusted on the basis of manual review of translations and tests. For this reason, the CoLiS language cannot be *fundamentally* different from shell.

For the purpose of application to the full corpus of Debian maintainer scripts, we had to improve on the initial version of the CoLiS language so as to increase the ratio of maintainer scripts that can be translated to CoLiS. On the one hand, we had to add a few more constructs to the language, such as the `export` special built-in utility, or the redirection of standard output. On the other hand, we extended the formal semantics for the new constructs, but we also had to align the previous semantics to the one of the Posix shell for a few other constructs. These changes and a complete description of the final design of the CoLiS language appear in a technical report that we published in 2019 [8]. We provide here a quick overview of its syntax and semantics.

Syntax of CoLiS. An important originality of the CoLiS toolchain for analyzing scripts is its design with formal verification in mind: its syntax, its semantics, and interpreters for it are designed using the Why3 environment [9] for formal verification. This is achieved by defining the syntax of CoLiS as abstract syntax trees (AST for short) by an algebraic datatype in Why3, and then defining the semantics by a set of inductive predicates [8].

The AST are described in detail in the technical report [8]; we present here only an excerpt of the corresponding concrete syntax, shown in Figure 5. This excerpt is more than half of the full grammar [8] and corresponds to the constructs we use in this paper for our running example `rancid` and for illustrating the concrete and symbolic interpreters.

As an llustrative example, Figure 6 presents the CoLiS version of the `preinst` script of the `rancid-cgi` package, shown previously in Figure 1.

Operational Semantics. The operational semantics of CoLiS is defined by a set of formal judgements and inductive rules for them, in a quite standard way for defining a big-step operational semantics. Yet, for the purpose of expressing properties of the symbolic execution (described later on in Section 3.3), we introduced a non-standard parameter to these judgements: a bound on the maximal number of loop iterations. This bound is either a non-negative integer or $+\infty$. The general form of an evaluation judgement for an instruction i is

$$(\mathcal{I}, \mathcal{C}, \mathcal{S}), i \Downarrow_s^{\mathsf{i}} (\mathcal{S}', \mathcal{C}', \beta)$$

⟨*program*⟩ ::= ⟨*function-definition*⟩* `begin` ⟨*instr*⟩* `end`

⟨*function-definition*⟩ ::= `function` ⟨*identifier*⟩ `begin` ⟨*instr*⟩* `end`

⟨*instr*⟩ ::= — *Program instruction*
| ⟨*identifier*⟩ `:=` ⟨*string-expr*⟩ — *Variable assignment*
| `if` ⟨*instr*⟩ `then` ⟨*instr*⟩* (`else` ⟨*instr*⟩*)? `fi` — *Conditional*
| `for` ⟨*identifier*⟩ `in` ⟨*list-expr*⟩ `do` ⟨*instr*⟩* `done` — *For loop*
| `while` ⟨*instr*⟩ `do` ⟨*instr*⟩* `done` — *While loop*
| `call` ⟨*identifier*⟩ ⟨*list-expr*⟩? — *Function call*
| ⟨*identifier*⟩ ⟨*list-expr*⟩? — *Utility call*

⟨*string-expr*⟩ ::= ⟨*sfrag*⟩+ — *String expression*

⟨*sfrag*⟩ ::= — *String fragment*
| ⟨*literal*⟩ — *String literal*
| ⟨*identifier*⟩ — *Variable*
| `embed {` ⟨*instr*⟩ `}` — *Output from instr*

⟨*list-expr*⟩ ::= `[` (⟨*lfrag*⟩ (`,` ⟨*lfrag*⟩)*)? `]` — *Non-empty, bracket-delimited, comma-separated list of fragments*

⟨*lfrag*⟩ ::= `split`? ⟨*string-expr*⟩ — *List fragment*

Fig. 5 Excerpt of the concrete syntax of the CoLiS language. A CoLiS program is a sequence of function definitions followed by a main body. Function bodies and the main body are sequences of instructions, including compound statements for conditional and "for" and "while" loops. Atomic instructions include variable assignment, function calls, and invocations of Unix utilities. The syntax of expressions, which are in fact restricted to expressions manipulating strings, is quite involved because of the need to make explicit some features that are implicit in the shell. These include in particular the splitting of strings into lists of strings which is performed by the shell at separators (typically the space character). In CoLiS the string literals are explicitly written between single quotes, the lists are given between square brackets and their elements are separated by commas. String splitting only occurs when explicitly required by a `split` expression as shown at the bottom of the figure.

```
1 if test [ '-h', '/etc/rancid/lg.conf' ] then
2   rm [ '/etc/rancid/lg.conf' ]
3 fi
4 if test [ '-e', '/etc/rancid/apache.conf' ] then
5   rm [ '/etc/rancid/apache.conf' ]
6 fi
```

Fig. 6 The `preinst` script of the `rancid-cgi` package in CoLiS. Notice the calls to the `test` and `rm` utilities. Notice also the syntax for string arguments and for lists of arguments, which require mandatory usage of delimiters. This exemplifies how the syntax of CoLiS was designed so as to remove potential ambiguities [8].

where $\mathcal{I}$ is an evaluation context (specifying parameters like limits of stack size and loop iterations, but also whether evaluation currently occurs under a condition), $\mathcal{C}$ and $\mathcal{C}'$ are program contexts (storing in particular the current values of variables, including the result of the last command), $\mathcal{S}$ and $\mathcal{S}'$ are system states (holding in particular the current state of the file system), and β is a *result behavior*, that can be either normal (Normal) or one of the abnormal behaviors (Exit, Return or even

$$\textsc{eval-if-true}\quad \frac{(\{\mathcal{I}\ \texttt{with}\ \mathsf{under\text{-}condition{=}True}\},\mathcal{C},\mathcal{S}),i_1 \Downarrow_s^{\mathsf{i}} (\mathcal{S}_1,\mathcal{C}_1,\mathsf{Normal}) \quad \mathcal{C}_1.\mathsf{result} = \mathsf{True} \quad (\mathcal{I},\mathcal{C}_1,\mathcal{S}_1),i_2 \Downarrow_s^{\mathsf{i}} (\mathcal{S}_2,\mathcal{C}_2,\beta_2)}{(\mathcal{I},\mathcal{C},\mathcal{S}),\texttt{if}\ i_1\ \texttt{then}\ i_2\ \texttt{else}\ i_3 \Downarrow_s^{\mathsf{i}} (\mathcal{S}_2,\mathcal{C}_2,\beta_2)}$$

$$\textsc{eval-if-false}\quad \frac{(\{\mathcal{I}\ \texttt{with}\ \mathsf{under\text{-}condition{=}True}\},\mathcal{C},\mathcal{S}),i_1 \Downarrow_s^{\mathsf{i}} (\mathcal{S}_1,\mathcal{C}_1,\mathsf{Normal}) \quad \mathcal{C}_1.\mathsf{result} = \mathsf{False} \quad (\mathcal{I},\mathcal{C}_1,\mathcal{S}_1),i_3 \Downarrow_s^{\mathsf{i}} (\mathcal{S}_3,\mathcal{C}_3,\beta_3)}{(\mathcal{I},\mathcal{C},\mathcal{S}),\texttt{if}\ i_1\ \texttt{then}\ i_2\ \texttt{else}\ i_3 \Downarrow_s^{\mathsf{i}} (\mathcal{S}_3,\mathcal{C}_3,\beta_3)}$$

$$\textsc{eval-if-transmit-condition}\quad \frac{(\{\mathcal{I}\ \texttt{with}\ \mathsf{under\text{-}condition{=}True}\},\mathcal{C},\mathcal{S}),i_1 \Downarrow_s^{\mathsf{i}} (\mathcal{S}_1,\mathcal{C}_1,\beta_1) \quad \beta_1 \neq \mathsf{Normal}}{(\mathcal{I},\mathcal{C},\mathcal{S}),\texttt{if}\ i_1\ \texttt{then}\ i_2\ \texttt{else}\ i_3 \Downarrow_s^{\mathsf{i}} (\mathcal{S}_1,\mathcal{C}_1,\beta_1)}$$

$$\textsc{eval-call-utility}\quad \frac{s,(\mathcal{I},\mathcal{C},\mathcal{S}),es \Downarrow^{\mathsf{I}} (\mathcal{S}',\mathsf{Success}\ ss) \quad \mathit{interp}\,(\mathcal{C}.\mathsf{cwd},\mathit{exported}(\mathcal{C}.\mathsf{vars}),ss)\,(id,\mathcal{S}') = (\mathcal{S}'',b)}{(\mathcal{I},\mathcal{C},\mathcal{S}),id\ \ es \Downarrow_s^{\mathsf{i}} (\mathcal{S}'',\{\mathcal{C}\ \texttt{with}\ \mathsf{result}{=}b\},bhv_{\mathcal{I}}(b))}$$

Fig. 7 Semantic rules for the evaluation of conditions and calls to utilities.

Failure when some limit is exceeded). An excerpt of the semantic rules is given in Figure 7, and we refer to [8] for the complete set and related details.

Notice that although this formalizes very precisely the operational semantics of CoLiS, it does not provide any guarantee that it is conformant with the semantics of the regular Posix shell. We had to check this conformity by testing, that is comparing the results of `sh` and our CoLiS concrete interpreters on a set of scripts hopefully covering sufficiently the constructs of the language. It is worth noting that thanks to those tests, we identified a non-conformity related to the interpretation of errors on strict or non-strict modes, that we fixed later on [8]. This fix concerns precisely the rules given in Figure 7, which is related to how an abnormal behavior is propagated: when inside a condition of an "if" or a loop, the abnormal behavior must be captured and execution should proceed. This can be seen in the three first rules for evaluating an if-expression: when evaluating the condition, we set the Boolean under-condition true in the evaluation context. Then, in rule EVAL-CALL-UTILITY, the resulting behavior of an utility call is $bhv_{\mathcal{I}}(b)$ which denotes the Normal behavior when $\mathcal{I}$.under-condition is true and b is true, and Exit otherwise.

A concrete interpreter for the CoLiS language is implemented in WhyML. It is programmed in an imperative style, using mutable variables, and using the native exception mechanism of WhyML to encode abnormal behaviours. The profile of the interpretation procedure for instructions is given in Figure 8.

Using the Why3 environment, the code is checked for conformity with respect to the specifications, by generating 245 verification conditions, and then checking their validity using a combination of automated theorem provers [27]. Figure 9 show the details.

```
1 let ghost ref s = 0  (* size of call stack *)
2
3 let rec interp_instruction (I:input) (sta:concrete_state)
4                            (i:instruction) : unit
5   requires { I.loop-limit = ∞ ∧ I.stack-size = ∞ }
6   ensures { (I,C(old sta),S(old sta)),i ⇓^i_{old s} (S(sta),C(sta),Normal)) }
7   raises { (Exit | Return) as β →
8              (I,C(old sta),S(old sta)),i ⇓^i_{old s} (S(sta),C(sta),β)) }
```

Fig. 8 Profile for the procedure interpreting CoLiS instructions. It includes a pre-condition and two post-conditions respectively for normal termination and abnormal termination. The pre-condition (line 5) imposes that the concrete interpretation is done without any limit on loop iterations or stack size. The first post-condition (line 6) expresses that on normal termination, the resulting concrete state is admissible with respect to the inductively defined operational semantics. The second post-condition (lines 7 and 8) expresses the analogous property in case an exception is raised. In all, the full contract expresses the conformity of the interpreter with respect to the formal operational semantics.

Prover	VCs	Fastest	Slowest	Average
CVC4 1.6	230	0.10	0.68	0.26
Alt-Ergo 2.2.0	13	0.12	2.45	0.54
Z3 4.6.0	2	0.12	0.25	0.18

Fig. 9 The use of different automatic theorem provers in the verification conditions (VCs) of the concrete interpreter with processing time in seconds. The provers are tried in order, that is Alt-Ergo is run only in the 15 VCs that were not discharged by CVC4, and Z3 is run only on the 2 remaining unproved VCs. Most VCs are solved in a fraction of seconds, the most complicated one requiring around 2 seconds and half.

3.1.3 Translation from shell to CoLiS

This translation is done automatically, but it is not formally proven. Indeed, a formal semantics of Posix shell was missing until very recently [24]. For the control flow constructs, the AST of the shell script is translated into the AST of CoLiS. For the strings (words in shell), the translation generates either a CoLiS string-expression or a list of CoLiS expressions depending on the content of the shell string. This translation makes explicit the string evaluation in shell, in particular the implicit string splitting. At the present time, the translator rejects 23% of shell scripts because it does not cover the full constructs of the shell, for example the usage of *globs* (restricted sets of regular expressions), of parameters with modifiers (like inline tests, or value substitutions), and advanced uses of redirections.

The conformity of the CoLiS script with the original shell script is not proven formally but checked by manual review and some automatic tests. For the latter, we developed a tool that automatically compares the results of the CoLiS interpreter on the CoLiS script with the results of the Debian default shell (`dash`) on the original shell script. This tool uses a test suite of shell scripts built to cover the whole set of constructs of the CoLiS language. The test suite allowed us to fix the translator and the formal semantics of CoLiS and, as an interesting additional outcome, it revealed a lack of conformity between the Debian default shell and Posix [5].

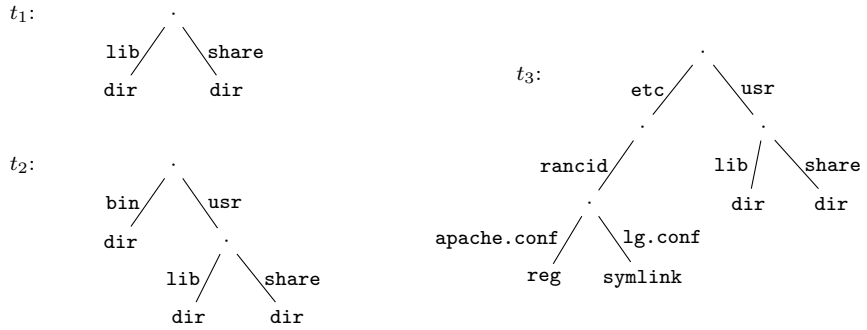

Fig. 10 Examples of feature trees showing directories (t_1), sub-directories (t_2), a regular file and a symbolic link (t_3). The `dir` kind of internal nodes is usually omitted.

For more technical details in the translation scheme we refer to N. Jeannerod's PhD thesis [26, Chapter 6].

3.2 Feature Trees and Constraints

We employ first-order logic to describe transformations of UNIX file systems. Feature trees [38,2,39] turn out to be suitable models for this study since they have names attached not to the nodes but to the edges going from a node to its children, as is the case for UNIX file systems [4]. Since previously existing feature logics where not able to express local structural differences between trees, as for instance created by the addition or removal of children, some of the authors [30, 26] proposed an extended feature constraint system that is suitable for expressing file system transformations. We provide below a concise overview of the model and the logic used in this study.

3.2.1 Feature Trees

The values of our model are trees with edges labeled by *features* taken from $\mathcal{F}$, which is an infinite set of legal file names. This is an abstraction of real POSIX systems where each implementation chooses its own limit on the length of the filenames[1]. Since the maximal length of a filename is a finite but unknown number, any error that may happen on some POSIX implementation due to a filename exceeding that limit will be avoided on an implementation with a larger limit. As a consequence, we decided to ignore possible errors that may occur due to long filenames, which is achieved by taking an infinite set of possible filenames.

Nodes in these trees are labeled by the `dir` *kind*, and leaves labeled by any *kind* (`dir`, `reg` or `symlink`). Examples of feature trees are given in Figure 10.

[1] According to the POSIX standard, the maximal number of bytes in a filename is defined by the constant `NAME_MAX` in the file `limits.h`, the value of which must not be smaller than 14.

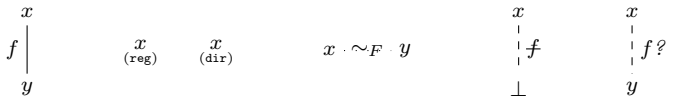

Fig. 11 Basic constraints, from left to right: a feature, a regular file node, a directory node, a tree similarity, a feature *absence*, a *maybe*.

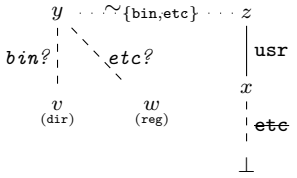

Fig. 12 A conjunctive clause.

3.2.2 Constraints

Properties of feature trees, and more generally relations between feature trees, are expressed by formulas of a feature tree logic with similarity constraints [30]. For the sake of presentation, we will use in the following a graphical representation of quantifier-free conjunctive clauses of this logic. The logic used in this study extends the one presented in the original publication [30] by the addition of *kinds* that are attached to nodes of trees.

The core basic constraints are presented in Figure 11. The *feature* constraint expresses that y is a subtree of x accessible from the root of x *via* feature f. The *kind* constraints express that the root of a tree has the given kind (`dir`, `reg` or `symlink`). The *similarity* constraint expresses that x and y have the same children with the same names except for the children whose names are in F, a finite set of features, where they may differ.

For performance reasons, we added two more constraints; these do not increase the expressive power but help to prevent combinatorial explosion of formulas. The *absence* constraint expresses that either x is not a directory or x does not have a feature f at its root. The *maybe* constraint expresses that either x is not a directory, or it does not have a feature f at its root, or it has one that leads to y.

A model of a formula is a valuation that maps variables to feature trees. For instance, consider the valuation that associates t_1 to x, t_2 to y and t_3 to z, where t_1, t_2 and t_3 are the trees defined in Figure 10; it satisfies the formula which is depicted in Figure 12. This formula reads

- if y has a feature `bin` then it leads to some directory v,
- if y has a feature `etc` then it leads to some regular file w,
- z has a feature `usr` leading to x, and x does not have feature `etc`,
- y and z coincide, except possibly for features `bin` and `etc`.

For a more detailed presentation of these extensions see [28,26].

3.2.3 Satisfiability of Feature Tree Constraints

We designed a set of transformation rules [30] that turns any Σ_1-formula into an *irreducible form* that is either *false* or a satisfiable formula. This allows us to simplify formulas *incrementally* and always keep the irreducible form of a formula, instead of the original formula. This speeds up computations, and allows the system to purge useless branches since unsatisfiable formulas are detected as soon as possible. Our toolchain includes an implementation of this system, using an efficient representation of irreducible Σ_1-formulas as trees themselves. Finally, the system of rules is also extended to a quantifier elimination procedure (showing in particular that the whole first-order theory of feature tree logic is decidable), which has application to verifying properties of specifications, as shown in Section 3.4.5.

3.3 Analysis by Symbolic Execution

With a similar approach as for the concrete interpreter (Section 3.1.2), we designed, and implemented in WhyML, a symbolic interpreter for the CoLiS language. Guided by a proof-of-concept symbolic interpreter for a simple IMP language [7], the main design choices for the symbolic interpreter for CoLiS are:

- Variables are *not* interpreted abstractly: when executing an installation script, the concrete values of variables are known. On the other hand, the state of the file system is not known precisely, and it is represented symbolically using a feature tree constraint.
- The symbolic engine is generic with respect to the utilities: their specifications in terms of symbolic input/output relations are taken as parameters. These specifications will be presented later on in Section 3.4.
- To control potentially infinite symbolic execution, or even to limit it in practice, the number of loop iterations and the number of (recursively) nested function calls [8] is bounded a priori, the bound is given by a global parameter set at the interpreter call.

The WhyML code for the symbolic interpreter is annotated with post-conditions to express that it computes an *over-approximation* [7] of the concrete states that are reachable without exceeding the given bound on loop iterations. Figure 13 presents the profile of the main procedure for symbolically interpreting CoLiS instructions.

As for the concrete interpreter, the WhyML code we wrote is shown conforming to the specification above, by generating verification conditions and proving them valid using automated provers. Table 14 summarises the prover results on the 780 generated VCs.

An executable code for the symbolic interpreter is automatically extracted, as OCaml code, using Why3's extraction mechanism. That code provides an executable symbolic interpreter with strong guarantees of soundness with respect to the concrete formal semantics.

Notice that our symbolic engine neither supports parallel executions, nor file permissions or file timestamps. This is another source of over-approximation, but also under-approximation, meaning that our approach can miss bugs whose triggering relies on the former features.

```
let rec sym_interp_instruction (s) (I, C, S) (i : instruction) : set of (S' × C')_β
  requires { s ≤ I.stack-size ≠ ∞ ∧ I.loop-limit ≠ ∞ }
  variant { I.stack-size − s, size(i) }
  ensures { ∀ S', C', β. (I, C, S), i ⇓^i_s (S', C', β) → (S', C')_β ∈ result }
```

Fig. 13 WhyML procedure for symbolic execution of CoLiS instructions. It returns a tuple of sets of resulting states, one set for each possible behaviour β. The pre-condition states that (contrarily to the concrete interpreter, see Figure 8) the bounds for the number of loop iterations and nested function calls are finite. This is a mandatory requirement in order to prove that the symbolic interpreter *terminates on all inputs*, that termination property being stated thanks to the `variant` clause above, giving a lexicographic measure that decreases at each recursive call to the `sym_interp_instruction` function. The post-condition expresses the wanted over-approximation property, which is a kind of coverage property: any concrete execution is covered by the symbolic execution. This property states that when no undesired symbolic states are reachable by symbolic execution, then no undesired concrete states are reachable by any concrete execution.

Prover	VCs	Fastest	Slowest	Average
CVC4 1.6	702	0.13	1.92	0.48
Alt-Ergo 2.2.0	65	0.13	43.26	2.60
Z3 4.6.0	13	0.07	1.14	0.31

Fig. 14 The use of different automatic theorem provers in the verification conditions of the symbolic interpreter functions with processing time in seconds. The same strategy as for the concrete interpreter (see Figure 9) is used, Alt-Ergo being run only on VCs not solved by CVC4, and then Z3. Most VCs need a fraction of a second to be discharged, with a few exception including one VC needing more than 40 seconds to be solved by Alt-Ergo.

3.3.1 Visualisation of the Symbolic Semantics of Scripts

The symbolic interpreter provides a symbolic semantics for the given script: given an initial symbolic state that represents the possible initial shape of the file system, it returns a triple of sets of symbolic input/output relations, respectively for normal result, error result (corresponding to non-zero exit code) and result when a loop limit is reached. Error results are unexpected for Debian maintainer scripts, and these cases have to be inspected manually. To help this inspection, a visualisation of symbolic relations was designed, as already described in Figure 4.

3.4 Specifications of Unix Commands

The specification of the Unix commands uses our feature tree logic to express their effect on the file system. The specification formalises the description given in natural language in the Posix standard [25, Chapter Utilities] and, for some commands, in GNU manual pages. We only specified the Unix commands that are most frequently called by the maintainer scripts.

The full specification is available in a separate technical report [28]. We present here its main ingredients. A Unix command has the form: "`cmd options paths`", where "`cmd`" is a command name, "`options`" is a list of options, and "`paths`" is one or more absolute or relative paths (*i.e.*, sequences of file names and symbols "`.`" and "`..`"). Section 3.4.1 describes path resolution.

For each combination of command name and options we provide a list of cases. A success or failure case formula has two free variables r and r', which represent respectively the file system before and after the execution of a command, as explained in Section 3.4.3.

For some combinations of command names and options, the specification is not provided directly as a set of formulas, but computed by the symbolic execution (as seen in Section 3.3) of a CoLiS script. This script captures the command behaviour by multiple but simpler invocations of the same command, or even different commands (see Section 3.4.4).

3.4.1 Path Resolution

It is important to note that specifications of commands are usually *parameterised* by their path argument(s): for each concrete value of such paths, an appropriate constraint is produced. This fact is essential for using our symbolic engine, because the variables of a constraint denote subtrees of a file system tree. Paths, however, are not first-class values of our logic. This is justified by the fact that the symbolic execution engine can in the vast majority of cases keep track of the values of shell variables, and hence it can fill in these values when variables are used in command invocations.

Hence, an important ingredient in command specification is the constraint encoding the resolution of a path in the file system. For this, we define a predicate $\texttt{resolve}(r, \texttt{cwd}, p, z)$ stating that "when variable r holds the file system, and `cwd` is the sequence of features leading from the root to the current working directory, then the path p resolves and leads to the tree denoted by the variable z". The constraint defining this predicate is a conjunction of existentially quantified basic constraints defined by a straightforward recursion on p and distinguishing the cases of resolving an absolute or a relative path. For example, the constraint $\texttt{resolve}(r, \texttt{cwd}, \texttt{/etc/rancid}, y)$ is represented by the path starting from the root of r and ending in y in the left part of Figure 15. Since in this example we are resolving an absolute path, the current working directory does not matter.

Notice a significant limitation of this constraint-based representation of path resolution: the possible presence of symbolic links is simply ignored. Supporting the *potential* presence of symbolic links is a challenge since there is no way to express finitely, at least in our feature tree logic, the infinite variety of symbolic links that may occur. Yet, this limitation was not noticeable in practice in the experiments reported in the next section.

3.4.2 Path Similarity

When specifying a modification of a file tree at one of its subtrees we need a more complex variant of the `resolve` predicate. The predicate $\texttt{similar}(r, r', \texttt{cwd}, p, z, z')$ expresses that p resolves in r and leads to z (taking into account that `cwd` is the current working directory), that p resolves in r' and leads to z', and that r and r' coincide except possibly in the subtrees z and z'. For example, the constraint $\texttt{similar}(r, r', \texttt{cwd}, \texttt{/etc/rancid}, y, y')$ is shown in the right part of Figure 15.

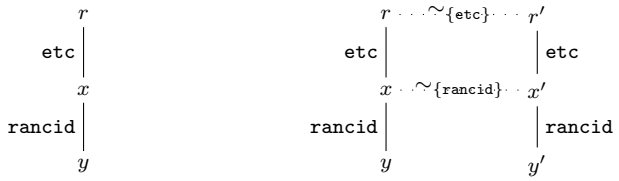

Fig. 15 On the left, the graphical representation of the constraint `resolve`$(r, \texttt{cwd}, \texttt{/etc/rancid}, y)$ which states the existence of the path `/etc/rancid` starting at r and ending at y. On the right, the graphical representation of the constraint `similar`$(r, r', \texttt{cwd}, \texttt{/etc/rancid}, y, y')$, which states that the path `/etc/rancid` exists in r and leads to y, and exists in r' and leads to y', and that r and r' coincide everywhere except possibly in y and y'. Note that in both cases the variable *cwd* is not used since we resolve an absolute path.

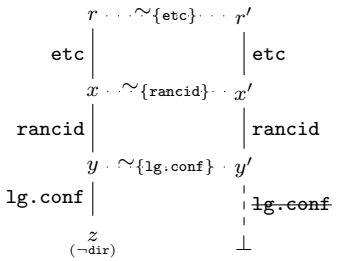

Fig. 16 Specification of the success case for command `rm /etc/rancid/lg.conf`. The resolution of the path `/etc/rancid/lg.conf` succeeds in the initial file system denoted by r and depicted on the left part of the diagram, and leads to node that is not a directory. The resulting file system, denoted by r' and shown on the right of the diagram, is similar to r except for the absence of the feature `lg.conf` at the node reached by `/etc/rancid`.

3.4.3 Success and Error Cases

Let us consider the command `rm /etc/rancid/lg.conf`. Its specification includes one success case, given in Figure 16: the resolution of the path `/etc/rancid/lg.conf` succeeded in the initial file system denoted by r, and the resulting file system, denoted by r' is similar to r except for the absence of the feature `lg.conf`. The specification also includes several error cases given in Figure 17, where the path cannot be resolved to a regular path, and therefore the initial and final file systems are the same.

The error case demands special care since the failure of the path resolution typically causes the failure of the command. To specify these failure cases, we have to use the negation of the predicate `resolve`, which generates a number of clauses which is linear in the length of the resolved path. Figure 17 shows, in the three left-most constraints, the error cases for the resolution of the path to `/etc/rancid/lg.conf`. Because the internal representation of formulas keeps only conjunctive clauses, this may produce a state explosion of constraints when the command uses several paths. To obtain a compact internal representation of these error cases we employ the *maybe* shorthand as shown on the right of Figure 17. This compact representation is denoted `noresolve`(r,cwd,q).

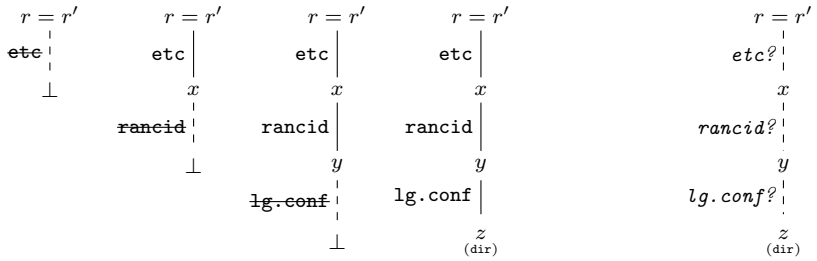

Fig. 17 Specification of error cases for command "`rm /etc/rancid/lg.conf`": explicit cases on the left, compact specification on the right

Outcome	Explanation	Corresponding input-output-formula
Success		$\exists x, x', y \cdot \texttt{resolve}(r, \texttt{cwd}, \texttt{q/f}, y) \wedge \texttt{similar}(r, r', \texttt{cwd}, \texttt{q}, x, x')$ $\wedge \neg\texttt{dir}(y) \wedge\ x \sim_{\{\texttt{f}\}} x' \wedge\ \texttt{dir}(x') \wedge x'[\texttt{f}]\uparrow$
Failure	Is a directory	$\exists y \cdot \texttt{resolve}(r, \texttt{cwd}, \texttt{q/f}, y) \wedge \texttt{dir}(y) \wedge r \doteq r'$
Failure	No such file	$\texttt{noresolve}(r, \texttt{cwd}, \texttt{q/f}) \wedge r \doteq r'$

Fig. 18 Specification of the `rm` command with a single path argument `q/f`: First line specifies the relation between the input file system r and the output one r' in case of a successful execution; the second line specifies the failure case caused by `q/f` being a directory; the third line specifies the failure case due to a missing file at `q/f`.

Figure 18 gives the specification of the command `rm q/f` where `q/f` is a path argument that is legal for `rm`, that is it is neither the empty absolute path, nor does it end on a single or double dot. Note that the recursion on the path `q/f` is delegated to the predicates `resolve`, `noresolve`, and `similar`.

3.4.4 Handling multiple path arguments

We model command invocations with multiple path arguments by a CoLiS script that iterates over all path arguments. For instance, an invocation like `rm pl` where `pl` is a list of paths is translated to this script :

```
1  failure := 'false'
2  for f in pl do
3    if rm f then else failure := 'true' fi
4  done
5  if test [ failure, '-eq', 'true' ] then fail fi
```

Note that this models the real behaviour of the `rm` command applied to multiple path arguments, that is it tries to remove all paths given as arguments, and in the end signals failure when one of the removals failed. This is in contrast to just serially invoking `rm` on each argument, which would fail if and only the *last* removal operation fails. Doing the same with the `-e` flag set (the so-called *strict mode*) would abort the sequence at the first failure, and not attempt to remove the remaining files.

3.4.5 Properties of specifications

The specifications of Unix commands satisfy some properties that are checked using the decision procedure designed for feature tree constraints (see Section 3.2.3). Firstly, these specifications are expressed as a disjunction of cases $\vee_i \phi_i$, (see Table 18), each case ϕ_i being the conjunction of a precondition formula $\texttt{Pre}_i(r)$ on the initial file system represented by the variable r and a transformation formula $\texttt{Trans}_i(r, r')$ specifying the transformation performed by the command to obtain the resulting file system represented by r'. Due to this specific form, we are able to check that the specification of a command is *complete*, that is, the disjunction of preconditions for all cases of a command's specification is a valid formula. Moreover, we checked that the specification of each case is *coherent*, that is the precondition formula $\texttt{Pre}_i(r)$ implies the satisfiability of the associated transformation formula $\texttt{Trans}_i(r, r')$. Coherence and completeness together imply that a specification is *total*, that is the relation it represents covers all the input trees. A total specification ensures that the symbolic execution does not lose traces of command's execution.

The above properties are not enough to characterize the precision of specifications. In particular, they do not guarantee that we will not get false positives, that is bugs that are not actually reachable. The properties that carry such information are the determinism and the functionality of a specification. Intuitively, a specification of a command is *deterministic* if there is no pair (i, j) $(i \neq j)$ of cases such that their preconditions $\texttt{Pre}_i(r)$ resp. $\texttt{Pre}_j(r)$ share a model. Determinism is particularly important between preconditions of cases with different status – success or error. Indeed, if it holds, then it ensures that the input trees are classified correctly. A specification case ϕ_i is functional if every input tree is related to at most one output tree. A command specification is *functional* if it is deterministic and all its cases are functional.

All our specifications are written to be complete and coherent. Most of them are also functional. However, for some of them, the specification logic is not expressive enough to capture the exact behaviour of the command. For example, the command `cp` with recursive option and overlapping source and destination paths may produce a (potentially partial) interleaving of the two input file systems, that is a (potentially strict) subset of the union of the two input trees, which our logic can simply not express. In that case, our specification over-approximates the behaviour of the command, allowing one input tree to lead to several output trees, giving up on the functionality property. We notice that such situations occur very rarely in the maintainer scripts of Debian, which justifies that we do not extend the logic to handle them.

3.4.6 Testing the specifications

To ensure that the specifications of Unix commands are correct abstractions of what they are actually doing, we employ specification-based testing. The overall process for a command call `cmd args` is summarised in Figure 19. The symbolic engine is used to instantiate the specification of the the command `cmd` with its actual parameters `args`. The result is a formula $\phi(r, r')$ which is given to a model extractor to produce a model M. The model extractor follows the simplification rules of the decision procedure (see Section 3.2.3) and additional heuristics to

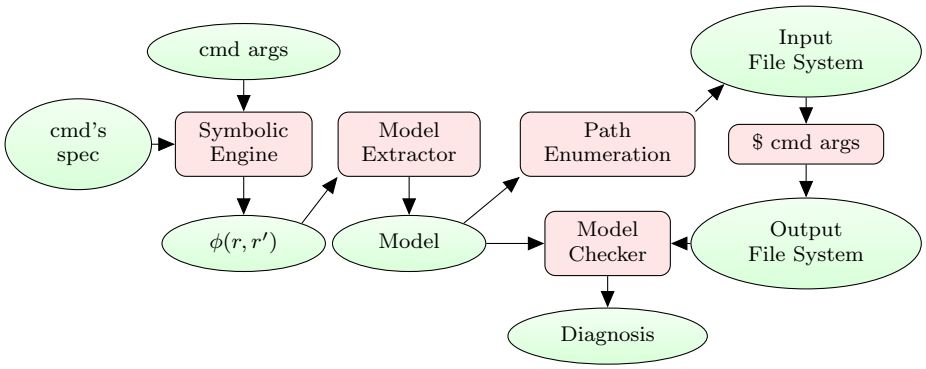

Fig. 19 Testing behaviour of a given command against its specification: green components are input and output data, red parts execute commands or tools. The formula ϕ is generated by the symbolic engine from the input command specification and the command arguments; ϕ is given to a model extractor which produces a finite model. The model is explored to create an input file system on which the tested command is executed and creates an output file system. The model-checker tests that the output file system complies with the model extracted from the specification.

obtain a small model M of a satisfiable constraint $\phi(r, r')$. The model M built by our model extractor is a forest with two roots labelled by the logic variables r and r'. An example of such model for the command `rm /etc/rancid/apache.conf` is given in Figure 17. The model is then used to create an input file system by enumerating all paths reachable from the root r. The command `cmd args` is called on the input file system to produce an output file system. The paths reachable from the root r' in the model M are computed and a simple model-checker verifies that all these paths belong to the output file system.

There are several challenges for this testing process. The one that required most attention was in the model extractor which has to deal with negative constraints (*e.g.*, the absence of a file or the difference between two directories). We proposed a sound algorithm for extracting models in the presence of such constraints. Another challenge concerns the choice of input arguments that exhibit interesting test cases. We enumerate exhaustively arguments that are input paths for our commands by fixing a finite set of file names including here ('.') and parent ('..').

The main outcome of the testing process is the increased confidence in our specification of the UNIX commands. We did not find specification errors, but we found errors in the implementation in OCaml of the UNIX commands' specifications. These bugs concern the handling of input paths containing special file names like here ('.') and parent ('..'), or sequences of slashes ('/').

3.5 Scenarios

So far, we have presented how we analyse individual maintainer scripts. In reality, the Debian policy specifies in natural language in which order and with which arguments these scripts are invoked during package installation, upgrade, or removal (see, for instance, Figure 2). We have specified these scenarios in a loop-free

custom language. These scenarios define what happens after the success or the failure of a script execution. They also specify when the static content is unpacked. Furthermore, our toolchain allows us to define the assumptions that can be made on an initial filesystem before executing a scenario, for instance the File System Hierarchy Standard [42]. Our toolchain reports on packages that may remain in an unexpected state after the execution of one of these scenarios.

For instance, the installation scenario of the package `rancid-cgi` may leave that package in the state NotInstalled, which is reported by our toolchain using the diagram in Figure 4.

4 Results and impact

4.1 Coverage of the case study

We executed the analysis on a machine equipped with 16 hyperthreaded Intel Xeon CPU @ 2.50GHz, and 64GB of RAM. To obtain a reasonable execution time, we limit the processing of one script to 60 seconds and 8GB of RAM. The time limit might seem low, but the experience shows that the few scripts (in 30 packages) that exceed this limit actually require hours of processing because they heavily use the `dpkg-maintscript-helper` script. On our corpus of 11,640 packages with 27,324 scripts, the analysis runs in about half an hour.

All of those scripts that are syntactically correct with respect to the Posix standard (99.9%) are parsed successfully by our parser. The translation of the parsed scripts into our intermediary language CoLiS succeeds for 77% of them; the translation fails mainly because of the limitations already mentioned in Section 3.1.3, namely the use of globs, parameter modifiers and advanced uses of redirections.

Our toolchain then attempts to run 104,760 scenarios (11,640 packages with scripts, 9 scenarios per package). Out of those, 42,738 scenarios (41%) are run completely and 12,471 (12%) partially. This is because scenarios have several branches and although a branch might encounter failure, we try to get some information on execution of other branches. For the same reason, one scenario might encounter several failures. In total, we encounter 62,023 failures. The origins of failures are multiple, but the two main ones are (i) trying to execute a scenario that includes a script that we cannot convert (21% of failures), or (ii) the scripts might use commands unsupported by our tools, or unsupported features of supported commands (76% of failures).

Among the scenarios that we manage to execute at least partially, 14 reach an unexpected end state. These are potential bugs. We have examined them manually to remove false positives due to approximations done by our methodology or the toolchain. We discuss in Section 4.3 the main classes of true bugs revealed by this process.

The tools used during the current study are available in our Github repository [40].

Bugs	*Closed*		*Detected by*	*Reports*	*Examples*
	2020	*2021*			
95	56	64	parser	[11]	not using `-e` mode
6	4	5	parser & manual	[17]	unsafe or non-Posix constructs
34	24	26	corpus mining	[10,12]	wrong options, mixed redirections
9	7	7	translation	[13]	wrong test expressions
5	2	4	symbolic execution	[15,19,17]	try to remove a directory with `rm`
3	3	3	formalisation	[14]	bug in `dpkg-maintscript-helper`
152	96	109			

Fig. 20 Classification of bugs found between 2016 and 2019 in Debian *sid* and *stable* distributions, with the numbers of resolved bugs in February 2020 and September 2021.

4.2 Corpus mining

We ran our tools on the "bullseye" Debian distribution which was released on August 14, 2021. It contains 59,551 packages, 11,640 of which contain at least one maintainer script, which leads to 27,324 scripts. In total, these scripts contain 683,926 source lines of code, 25 lines on average, and up to 1,393 for the largest script. Among them we find 150 `bash` scripts, 2 `dash` scripts, 7 `perl` scripts, and one ELF executable – the rest are Posix shell scripts.

In the process of designing our tools, and in order to validate our hypotheses, we ran statistical analysis on this corpus of scripts. The construction of our tool for statistical analysis is described in a technical report [29] where we also detail a few of our findings. To summarise, analysing the corpus revealed that:

- Most variables in scripts were used as constants: only 3,008 scripts contain variables whose value actually changes.
- There are no recursive functions in the whole corpus.
- There are 2,300 scripts that include a while loop. 93% of the while loops occur in a pipe reading the output of `dpkg -L` and are an idiosyncrasy that is proper to some shell languages. They can be translated to "foreach" loops in a properly typed language.
- The huge majority of redirections are used to hide the standard output or merge it into the error output.

This analysis had an important impact on the project by guiding the design choices of CoLiS, which Unix commands we should specify and in which order, etc. This also helped us to discover a few bugs, for example scripts invoking Unix commands with invalid options.

4.3 Bugs found

We ran our toolchain on several snapshots of the Debian *sid* distribution taken between 2016 and 2019, and on the latest released version named "bullseye" which was released August 14, 2021. We reported over this period a total of 152 bugs

to the Debian Bug Tracking System [41]. Some of them have immediately been confirmed by the package maintainer (for instance, [18]), and 109 of them have already been resolved.

Table 20 summarizes the main categories of bugs we reported. Simple lexical analysis already detects 95 violations of the Debian Policy, for instance scripts that do not specify the interpreter to be used, or that do not use the `-e` mode [11]. The shell parser (Section 3.1.1) detects 3 scripts that use shell constructs not allowed by the Posix standard, or in a context where the Posix standard states that the behaviour is undefined [17]. There are also 3 miscellaneous bugs, like using unsafe shell constructs. The mining tool (Section 4.2) detects 5 scripts that invoke Unix commands with wrong options and 29 scripts that mix up redirection of standard-output and standard-error. The translation from the shell to the CoLiS language (Section 3.1.3) detects 9 scripts with wrong test expressions [13]. These may stay unnoticed during superficial testing since the shell confuses, when evaluating the condition of an if-then-else, an error exception with the Boolean value *False*. Inspection of the symbolic semantics extracted by the symbolic execution (Section 3.3) finds 5 scripts with semantic errors. Among these is the bug [18] of the package `rancid-cgi` already explained in Section 2.5. During the formalisation of Debian tools (see Section 3.4), we found 3 bugs. These include in particular a bug [14] in the `dpkg-maintscript-helper` command which is used 10,306 times in our corpus of maintainer scripts, and was fixed in the meantime.

4.4 Lessons learnt

One basic problem when trying to analyse maintainer scripts is to understand precisely the meaning of the policy document. For instance, one of the most intriguing requirements is that maintainer scripts have to be idempotent (Section 6.2 in [3]). While it is common knowledge that a mathematical function f is idempotent when $f(f(x)) = f(x)$ for any x, the meaning is much less clear in the context of Debian maintainer scripts as the policy goes on to explain: "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." We suppose that this refers to causes of error external to the script itself (power failure, full disk, etc.), and that there might be an intervention by the system administrator between the two invocations. Since we cannot even explain in natural language what precisely that means, let alone formalise it, we decided to model at the moment only a rough under-approximation of that property that only compares executions by their exit code. Even if this is a rough approximation, it allowed us to detect an idempotency bug in a package [16].

We found that identifying bugs in maintainer scripts always requires human examination. Automated tools are good at pointing out potential problems in a large corpus, but deciding whether such a problem actually deserves a bug report, and of what severity level, requires some experience with the Debian processes. This is most visible with semantic bugs in scripts, since an error exit code does not imply that there is a bug. Indeed, if a script detects a situation it cannot handle then it *must* signal an error and produce a useful error message. Deciding whether a detected error case is justified or accidental requires human judgement.

Filing bug reports demands some caution, and observance of rules and common practices in the community. For instance, the Debian Developers Reference [20] requires approval by the community before so-called *mass bug filing*. Consequently, we always sought advice before sending batches of bugs, either on the Debian developers mailing list, or during Debian conferences.

5 Conclusion

The corpus of Debian maintainer scripts is an interesting case study for analysis due to its size, the challenging features of the scripting language, and the relational properties that are required to be analysed. The results are very promising. First, we reported 152 bugs [41] to the Debian Bug Tracking system, 109 of which have already been resolved by Debian maintainers. Second, the toolchain performs the analysis of a package in seconds and of the full distribution in less than one hour, which makes it fit for integration in the workflow of Debian maintainers or for quality assurance at the level of the whole distribution. Integration of our toolchain in the Lintian [31] tool will not be possible since it would add a lot of external dependencies to that tool, and since the reports generated by our tool still require human evaluation (see Section 4.4).

This study had several additional outcomes. The toolchain includes tools for parsing and light static analysis of shell scripts [36], an engine for the symbolic execution of imperative languages based on first-order logics representation of program configurations [7], and an efficient decision procedure for feature tree logic. We also provide a formal specification of Posix commands used in Debian scripts in terms of a first-order logic [28].

We are not aware of a project dealing with this kind of problem or obtaining comparable results. To our knowledge, the only existing attempt to analyse a complete corpus of package maintainer scripts was done in the context of the Mancoosi project [21]. In this work, the analysis, mainly syntactic, resulted in a set of building blocks used in maintainer scripts that may be used in a DSL. In a series of papers [23,34,33], Ntzik et al. consider the formal reasoning on the Posix scripts manipulating the file system based on (concurrent) separation logic. Not only do they employ a different logic (a second-order logic), but they also focus on (manual) proof techniques for correctness and not on automatic techniques for finding bugs. Moreover, they consider general scripts and properties that are not relational (like idempotency). There have been few attempts to formalise the shell. Greenberg [24] recently offers an executable formal semantics of Posix shell that will serve as a foundation for shell analysis tools. Abash [32] contains a formalisation of parts of the bash language and an abstract interpretation tool for the analysis of arguments passed by scripts to Unix commands; this work focused on identifying security vulnerabilities.

The successful outcome of this case study revealed new challenges that we aim to address in future work. In order to increase the coverage of our analysis and the acceptance by Debian maintainers, the translation from shell should cover more features, additional Unix commands should be formally specified, and the model should capture more features of the file system (*e.g.*, permissions or symbolic links). The efficiency of the analysis can still be improved by using a more compact representation of disjunctive constraints in feature tree logics or by

exploiting the genericity of the symbolic execution engine to include other logic based symbolic representations that may be more efficient and precise. A recent work [22] goes in this direction by using tree transducer techniques. Finally, we want to use the computed constraints on scenarios to check new properties of scripts like equivalence of behaviours.

Acknowledgements We would like to thank all the other members of the CoLiS project, in particular Ilham Dami for her internship about the early design of the CoLiS language, Abinandan Pal for his internship about the test of specifications, and Paul Gallot et Sylvain Salvati for their feedback on the usage of the CoLiS platform, towards the integration of tree transducer techniques in addition to feature constraints.

References

1. Jean-François Abramatic, Roberto Di Cosmo, and Stefano Zacchiroli. Building the universal archive of source code. *Communications of the ACM*, 61(10):29–31, 2018. doi:10.1145/3183558.
2. Hassan Aït-Kaci, Andreas Podelski, and Gert Smolka. A feature-based constraint system for logic programming with entailment. *Theoretical Computer Science*, 122(1–2):263–283, January 1994.
3. Russ Allbery and Sean Whitton. Debian policy manual. https://www.debian.org/doc/debian-policy/, October 2019. [Online; last accessed 2022-June-14].
4. Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, 1986.
5. Benedikt Becker. `dash` ignores `-e` in substitution under test. https://www.mail-archive.com/dash@vger.kernel.org/msg01683.html, 2018. [Online; last accessed 2022-June-14].
6. Benedikt Becker, Nicolas Jeannerod, Claude Marché, Yann Régis-Gianas, Mihaela Sighireanu, and Ralf Treinen. Report generated by colis-batch on Debian bullseye. Zenodo Repository, October 2021. doi:10.5281/zenodo.5560955.
7. Benedikt Becker and Claude Marché. Ghost Code in Action: Automated Verification of a Symbolic Interpreter. In Supratik Chakraborty and Jorge A.Navas, editors, *Verified Software: Tools, Techniques and Experiments*, Lecture Notes in Computer Science, 2019. URL: https://hal.inria.fr/hal-02276257.
8. Benedikt Becker, Claude Marché, Nicolas Jeannerod, and Ralf Treinen. Revision 2 of CoLiS language: formal syntax, semantics, concrete and symbolic interpreters. Technical report, HAL Archives Ouvertes, October 2019. URL: https://hal.inria.fr/hal-02321743.
9. François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's verify this with Why3. *International Journal on Software Tools for Technology Transfer (STTT)*, 17(6):709–727, 2015. URL: http://hal.inria.fr/hal-00967132/en, doi:10.1007/s10009-014-0314-5.
10. Debian Bug Tracker. `dibbler-server`: `postinst` contains invalid command. Debian Bug Reports 841934, October 2016. URL: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=841934.
11. Debian Bug Tracker. `authbind`: maintainer script(s) not using strict mode. Debian Bug Report 866249, June 2017. URL: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=866249.
12. Debian Bug Tracker. `dict-freedict-all`: `postinst` script has a wrong redirection. Debian Bug Report 908189, September 2018. URL: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=908189.
13. Debian Bug Tracker. `python3-neutron-fwaas-dashboard`: incorrect test in `postrm`. Debian Bug Report 900493, May 2018. URL: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=900493.
14. Debian Bug Tracker. `dpkg-maintscript-helper`: bug in `finish_dir_to_symlink`. Debian Bug Report 922799, February 2019. URL: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=922799.
15. Debian Bug Tracker. `ndiswrapper`: when `postrm purge` fails it may have deleted some config files. Debian Bug Report 942392, October 2019. URL: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=942392.

16. Debian Bug Tracker. `oz`: non-idempotent `postrm` script. Debian Bug Report 942395, October 2019. URL: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=942395.
17. Debian Bug Tracker. `preinst` script not Posix compliant. Debian Bug Report 925006, March 2019. URL: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=925006.
18. Debian Bug Tracker. `rancid-cgi`: `preinst` may fail and not rollback a change. Debian Bug Report 942388, October 2019. URL: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=942388.
19. Debian Bug Tracker. `sgml-base`: `preinst` may fail silently. Debian Bug Report 929706, May 2019. URL: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=929706.
20. Developer's Reference Team. Debian developers reference. https://www.debian.org/doc/manuals/developers-reference/, October 2019. [Online; last accessed 2022-June-14].
21. Roberto Di Cosmo, Davide Di Ruscio, Patrizio Pelliccione, Alfonso Pierantonio, and Stefano Zacchiroli. Supporting software evolution in component-based FOSS systems. *Science of Computer Programming*, 76(12):1144–1160, 2011. doi:10.1016/j.scico.2010.11.001.
22. Paul Gallot. *Safety of transformations of data trees*. Phd thesis, Université de Lille, 2021. URL: https://hal.archives-ouvertes.fr/tel-03517128.
23. Philippa Gardner, Gian Ntzik, and Adam Wright. Local reasoning for the Posix file system. In *European Symposium On Programming*, volume 8410 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2014. doi:10.1007/978-3-642-54833-8_10.
24. Michael Greenberg and Austin J. Blatt. Executable formal semantics for the Posix shell. *CoRR*, abs/1907.05308, 2019. arXiv:1907.05308.
25. IEEE and The Open Group. The open group base specifications issue 7. http://pubs.opengroup.org/onlinepubs/9699919799/, 2018. [Online; last accessed 2022-June-14].
26. Nicolas Jeannerod. *Verification of Shell Scripts Performing File Hierarchy Transformations*. PhD thesis, Université de Paris, March 2021. URL: https://hal.archives-ouvertes.fr/tel-03369452.
27. Nicolas Jeannerod, Claude Marché, and Ralf Treinen. A Formally Verified Interpreter for a Shell-like Programming Language. In *9th Working Conference on Verified Software: Theories, Tools, and Experiments*, volume 10712 of *Lecture Notes in Computer Science*, 2017. URL: https://hal.archives-ouvertes.fr/hal-01534747.
28. Nicolas Jeannerod, Yann Régis-Gianas, Claude Marché, Mihaela Sighireanu, and Ralf Treinen. Specification of UNIX utilities. Technical report, HAL Archives Ouvertes, October 2019. URL: https://hal.inria.fr/hal-02321691.
29. Nicolas Jeannerod, Yann Régis-Gianas, and Ralf Treinen. Having fun with 31.521 shell scripts. Technical report, HAL Archives Ouvertes, 2017. URL: https://hal.archives-ouvertes.fr/hal-01513750.
30. Nicolas Jeannerod and Ralf Treinen. Deciding the first-order theory of an algebra of feature trees with updates. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *9th International Joint Conference on Automated Reasoning*, volume 10900 of *Lecture Notes in Computer Science*, pages 439–454, Oxford, UK, July 2018. Springer. URL: https://hal.archives-ouvertes.fr/hal-01807474.
31. The Lintian expert system. https://lintian.debian.org. [Online; last accessed 2022-June-14].
32. Karl Mazurak and Steve Zdancewic. ABASH: finding bugs in bash scripts. In *Workshop on Programming Languages and Analysis for Security*, pages 105–114, 2007.
33. Gian Ntzik, Pedro da Rocha Pinto, Julian Sutherland, and Philippa Gardner. A concurrent specification of Posix file systems. In *European Conference on Object-Oriented Programming*, volume 109 of *LIPIcs*, pages 4:1–4:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.ECOOP.2018.4.
34. Gian Ntzik and Philippa Gardner. Reasoning about the Posix file system: local update and global pathnames. In *Object-Oriented Programming, Systems, Languages and Applications*, pages 201–220. ACM, 2015. doi:10.1145/2814270.2814306.
35. The `piuparts` tool for quality assurance of Debian packages. https://piuparts.debian.org/. [Online; last accessed 2022-June-14].
36. Yann Régis-Gianas, Nicolas Jeannerod, and Ralf Treinen. Morbig: A static parser for POSIX shell. *J. Comput. Lang.*, 57:100944, 2020. URL: https://doi.org/10.1016/j.cola.2020.100944, doi:10.1016/j.cola.2020.100944.
37. Roland Rosenfeld. Package rancid-cgi: looking glass cgi based on rancid tools, 2019. https://packages.debian.org/en/sid/rancid-cgi.
38. Gert Smolka. Feature constraint logics for unification grammars. *Journal of Logic Programming*, 12:51–87, 1992.

39. Gert Smolka and Ralf Treinen. Records for logic programming. *Journal of Logic Programming*, 18(3):229–258, April 1994.
40. The CoLiS project. The CoLiS toolchain. https://github.com/colis-anr.
41. The Debian Project. Bugs tagged colis. https://bugs.debian.org/cgi-bin/pkgreport.cgi?tag=colis-shparser;users=treinen@debian.org.
42. The Linux Foundation. Filesystem hierarchy standard, version 3.0, March 2015. URL: https://refspecs.linuxfoundation.org.
43. Aaron M. Ucko. `cmigrep`: broken `emacsen-install` script. Debian Bug Report 431131, June 2007. URL: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=431131.