Specification of UNIX Utilities

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

CoLiS project

http://colis.irif.fr/

Specification of UNIX Utilities

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

February 2019

Contents

1 Introduction

This report aims at providing the detailed technical informations about the specifications of UNIX utilities that we designed in the context of the CoLiS project.

The specification of utilities mainly concerns the effect they perform on the UNIX file system. For that purpose, the file system is specified in an abstract manner using a contraint language called the *feature tree constraints*. The important aspect of feature trees are presented in a paper by Jeannerod and Treinen [JT18]. For practical reasons the tree constraint language is a variant of the former. That variant is described in details in the first sections of this document.

General speaking, our sources for the expected behaviour of UNIX utilities are:

1. the GNU man pages,
2. the GNU info pages,
3. the POSIX standard [IEE], Chapter*Shell & Utilities*, Section *Utilities*.

After the description of the contraint language, this document give an exhaustive set of specifications of the behavior of UNIX utilities, as there are now implemented in the tool suite for CoLiS. This should serve as a basis of the design of a symbolic execution machine, or a translation into tree transducers. Notice however that it is not the purpose of this document to describe optimizations that will be necessary for the design of these engines.

2 Preliminaries

2.1 Set-Theoretic Notions

$A \rightsquigarrow B$ denotes the set of partial functions from the set A to the set B with a finite domain. The domain of a partial function f is written $\mathtt{dom}(f)$.

2.2 First-Order Logic

We assume logical conjunction and disjunction to be associative and commutative, and equality to be symmetric.

3 File Trees

Our model of a file hierarchy is an abstraction of the file systems as they are implemented on real UNIX systems. The model presented here is a variant of the one used in [JT18].

We assume given an infinite set $\mathcal{F}$ of *features*, where we assume that $., .., / \notin \mathcal{F}$. The letters f, g, h will always denote features.

Definition 1 *The set of* path components *is defined as*

$$\mathcal{PC} = \mathcal{F} \cup \{., ..\}$$

The set of paths *is defined as*

$$\mathcal{P} = \mathit{relative}(\mathcal{PC}^+) \cup \mathit{absolute}(\mathcal{PC}^*)$$

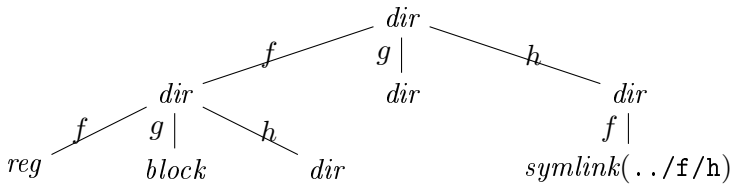


Figure 1: Example of a Feature Tree.

We will often write paths following the traditional UNIX notation: the path components are listed in order and separated by the symbol `/`, with a leading `/` if and only if the path is absolute.

Definition 2 *The set* $\mathcal{FT}$ *of* feature trees *is inductively defined as*

$$\mathcal{FT} = \mathit{leaves} \cup \mathit{dir}(\mathcal{F} \rightsquigarrow \mathcal{FT}) \cup \mathit{symlink}(\mathcal{P})$$

where leaves is a set of constants indicating different types of files that are leaves in a file system hierarchy:

Constant	*UNIX file type*
reg	*regular file*
block	*block device*
char	*character device*
fifo	*named pipe*
socket	*socket*

Since a partial function may have an empty domain we have in fact three base cases of the induction: either a special file, or a symlink, or an empty directory. Hence, this amounts to saying that a feature tree is a finite unordered tree where nodes are unlabeled, and edges are labeled by features. Each node in a feature tree has a finite number of outgoing edges, and all outgoing edges of a node carry different names. Leaves may be either special files, regular files, symbolic links, or empty directories. Inner nodes must be directories.

Our notion of equality on trees is *structural equality*, i.e. $t_1 = t_2$ iff

- both t_1 and t_2 are the same kind of leaf,
- or $t_1 = \mathit{symlink}(p_1)$, $t_2 = \mathit{symlink}(p_2)$, and $p_1 = p_2$, that is both are absolute or both are relative, and both have the same sequence of path components.
- or $t_1 = \mathit{dir}(f_1)$, $t_2 = \mathit{dir}(f_2)$, $\mathtt{dom}(f_1) = \mathtt{dom}(f_2)$ and $f_1(f) = f_2(f)$ for every $f \in \mathtt{dom}(f_1)$.

An example of a feature tree is given in Figure 1.

4 Constraints and their Interpretation

The set of predicate symbols (or atomic constraints) of our logic is

$x \doteq y$	Equality
$x[f]y$	Feature f from x to y
$x[F]$	Fence constraint, for any finite $F \subset \mathcal{F}$
$z \sim_F y$	Similarity, for any finite $F \subset \mathcal{F}$
$\texttt{dir}(x), \texttt{reg}(x), \texttt{symlink}(x)$	Directories, regular files, symbolic links
$\texttt{fifo}(x), \texttt{block}(x), \texttt{char}(x), \texttt{socket}(x)$	Special files
$\texttt{abslink\_}q(x)$	Absolute link, for any $q \in \mathcal{F}^*$
$\texttt{rellink\_}q(x)$	Relative link, for any $q \in \mathcal{F}^+$

We will use the usual syntactic sugar and write $x \neq y$ for $\neg(x \doteq y)$.

We have one model which has as universe the set $\mathcal{FT}$. As usual, we use the same symbol $\mathcal{FT}$ for the model and for its universe. The predicate symbols are interpreted as follows, where ρ is a valuation of the free variables of the formula in the model $\mathcal{FT}$:

$\mathcal{FT}, \rho \models x \doteq y$	iff $\rho(x) = \rho(y)$
$\mathcal{FT}, \rho \models x[f]y$	iff $\rho(x) = \mathit{dir}(\phi), f \in \texttt{dom}(\phi)$ and $\phi(f) = \rho(y)$
$\mathcal{FT}, \rho \models x[F]$	iff $\rho(x) = \mathit{dir}(\phi), \texttt{dom}(\phi) \subseteq F$
$\mathcal{FT}, \rho \models x \sim_F y$	iff $\rho(x) = \mathit{dir}(\phi), \rho(y) = \mathit{dir}(\psi), \phi(f) = \psi(f)$ for all $f \notin F$
$\mathcal{FT}, \rho \models \texttt{dir}(x)$	iff $\rho(x) = \mathit{dir}(\phi)$ for some ϕ
$\mathcal{FT}, \rho \models \texttt{symlink}(x)$	iff $\rho(x) \in \mathit{symlink}(\mathcal{P})$
$\mathcal{FT}, \rho \models \texttt{reg}(x)$	iff $\rho(x) = \mathit{reg}$
$\mathcal{FT}, \rho \models \texttt{block}(x)$	iff $\rho(x) = \mathit{block}$
$\mathcal{FT}, \rho \models \texttt{char}(x)$	iff $\rho(x) = \mathit{char}$
$\mathcal{FT}, \rho \models \texttt{fifo}(x)$	iff $\rho(x) = \mathit{fifo}$
$\mathcal{FT}, \rho \models \texttt{socket}(x)$	iff $\rho(x) = \mathit{socket}$
$\mathcal{FT}, \rho \models \texttt{abslink\_}q(x)$	iff $\rho(x) = \mathit{symlink}(\mathit{absolute}(q))$
$\mathcal{FT}, \rho \models \texttt{rellink\_}q(x)$	iff $\rho(x) = \mathit{symlink}(\mathit{relative}(q))$

Remark 1 *We have that $x \sim_{\{f,g\}} y$ is equivalent to $\exists z.(x \sim_{\{f\}} z \wedge z \sim_{\{g\}} y)$. Note, however, that the following three formulas have different meanings, assuming that $f \neq g$:*

$$x \sim_{\{f\}} y \wedge x \sim_{\{g\}} y$$
$$x \sim_{\{f\}} y \vee x \sim_{\{g\}} y$$
$$x \sim_{\{f,g\}} y$$

The first formula expresses that x and y have the same children, it is in fact equivalent to $x \sim_\emptyset y$. The second formula expresses that x and y may differ in at most one child, which is f or g. The last formula expresses that x and y have the same children except for f and g, and includes the possibility that they differ in both.

5 Macros for constraints

5.1 The absence of feature: $x[f]\uparrow$

$$x[f]\uparrow = \texttt{dir}(x) \wedge \neg\exists y \cdot x[f]y$$

5.2 `resolve`(r, cwd, p, z)

This macro creates a constraint that states that "when the root of the filesystem is r and the current working directory is the sequence of features cwd, the path p resolves and goes to z".

5.2.1 `resolve_s`(x, π, q, z)

It is easier to define `resolve_s`. Instead of working on a root, a current working directory (a sequence of features) and any path, it works on any variable in the filesystem, a stack of variables that contains all the parents of this variable up to the root (not including the variable itself) and a sequence of path components.

$$\texttt{resolve\_s} : \begin{cases} (x, \ \pi, \ \ \varepsilon, z) & \mapsto & x \doteq z \\ (x, \ \pi, f \cdot q, z) & \mapsto & \exists y \cdot x[f]y \wedge \texttt{resolve\_s}(y, \pi x, q, z) \\ (x, \ \pi, \ .. \cdot q, z) & \mapsto & \texttt{resolve\_s}(x, \pi, q, z) \\ (x, \ \varepsilon, .. \cdot q, z) & \mapsto & \texttt{resolve\_s}(x, \varepsilon, q, z) \\ (x, \pi y, .. \cdot q, z) & \mapsto & \texttt{resolve\_s}(y, \pi, q, z) \end{cases}$$

5.2.2 `resolve`(r, cwd, q, z)

The macro `resolve` can then be defined using `resolve_s`:

$$\texttt{resolve}(r, cwd, p, z) = \begin{cases} \texttt{resolve\_s}(r, \varepsilon, \quad\ \ q, z) & \text{if } p = absolute(q) \\ \texttt{resolve\_s}(r, \varepsilon, cwd@q, z) & \text{if } p = relative(q) \end{cases}$$

5.3 `similar`(r, r', cwd, p, z, z')

5.3.1 `similar_n`(x, x', q, z, z')

It only makes sense to define the similarity of two trees on a certain sequence of features.

$$\texttt{similar\_n} : \begin{cases} (x, x', \quad \varepsilon, z, z') & \mapsto & x \doteq z \wedge x' \doteq z' \\ (x, x', f \cdot q, z, z') & \mapsto & \exists y, y' \cdot \begin{pmatrix} x[f]y \wedge x'[f]y' \wedge x \sim_{\{f\}} x' \\ \wedge \ \texttt{similar\_n}(y, y', q, z, z') \end{pmatrix} \end{cases}$$

5.3.2 `normalize`(cwd, q)

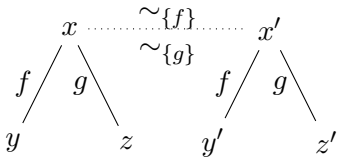
This is the **syntactic** normalization of a sequence of path components into a sequence of features.

$$\texttt{normalize} : \begin{cases} (\quad cwd, \quad \varepsilon) & \mapsto & cwd \\ (\quad cwd, f \cdot q) & \mapsto & \texttt{normalize}(cwd \cdot f, q) \\ (\quad cwd, \ .. \cdot q) & \mapsto & \texttt{normalize}(cwd, q) \\ (\qquad \varepsilon, .. \cdot q) & \mapsto & \texttt{normalize}(\varepsilon, q) \\ (cwd \cdot f, .. \cdot q) & \mapsto & \texttt{normalize}(cwd, q) \end{cases}$$

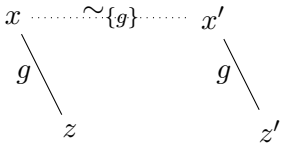
5.3.3 `similar`(r, r', cwd, p, z, z')

We define `similar` on paths using `similar_n` and `normalize`. See Fig. 2 as to why we do that this way.

$$\texttt{similar}(r, r', cwd, p, z, z') = \begin{cases} \texttt{similar\_n}(r, r', \texttt{normalize}(\varepsilon \quad\ , q), z, z') & \text{if } p = absolute(q) \\ \texttt{similar\_n}(r, r', \texttt{normalize}(cwd, q), z, z') & \text{if } p = relative(q) \end{cases}$$



(a) Naive version of `similar`$(f/../g)$



(b) `similar`$(f/../g) =$ `similar_n`(g)

Figure 2: Why we want to normalize paths given to the similarity. Note that if $f \neq g$, then $x \sim_{\{f\}} x' \wedge x \sim_{\{g\}} x' = x \sim_{\varnothing} x'$

X.2.1 `util -a -i` q/f

Success	Bla blo	$\exists x, x' \cdot$ `similar`$(r, r', cwd, q, x, x') \wedge x'[f]\uparrow$

Figure 3: Example of a Specification Case

6 Specifications of Utilities

The specification of each UNIX utility consists of one or several subsections.

There are first a few subsections about the options supported by the utility and how we chose to model them. In a lot of cases, the options have no impact in our model and can be safely ignored. Some options do change the semantics of the utility. It is to be noted that we are only interested in the options that change something in the action of the utility on the file system. In particular, a few options do change what the utility does on stdin, stdout and errout, but it is not the topic of this document.

The subsequent subsections describe specifications of particular uses of the utility. Their title is a pattern of arguments to the utility. For instance, `mkdir` p/f indicates that the `mkdir` utility is applied to a single nonempty path, the last component of which is the feature f.

These subsections contain a specification of the modification of the file system by the utility call. This is a list of specification cases. Each specification case (See Fig. 3) is a triple with the outcome (success, failure or unknown), a message describing the case and a constraint whose free variables are r (input root), r' (output root), cwd (the current working directory) and variables that are bound by the subsection (in this example, q and f such that `util -a -i` q/f is the specified utility.

6.1 cp

6.1.1 POSIX options

`-H`: Follow command-line symbolic links. Ignored.

`-i`: Forbidden because interactive.

`-L`: Always follow symbolic links. Ignored.

`-f`: Ignored.

`-p`: Ignored.

`-P`: Never follow symbolic links. Ignored.

`-r`, `-R`: Recursive.

6.1.2 `cp` q_s/f_s q_d/f_d

When q_s/f_s and q_d/f_d do not represent the same file.

Success	q_d/f_d is not a directory	$\exists y_s, x_d, x'_d, y_d \cdot$ $\texttt{resolve}(r, cwd, q_s/f_s, y_s) \wedge \neg\texttt{dir}(y_s)$ $\wedge\ \texttt{resolve}(r, cwd, q_d, x_d)$ $\wedge\ \texttt{dir}(x_d) \wedge (x_d[f_d]\uparrow \vee (x_d[f_d]y_d \wedge \neg\texttt{dir}(y_d)))$ $\wedge\ \texttt{similar}(r, r', cwd, q_d, x_d, x'_d) \wedge x_d \sim_{\{f_d\}} x'_d$ $\wedge\ \texttt{dir}(x'_d) \wedge x'_d[f_d]y_s$
Success	q_d/f_d is a directory, $q_d/f_d/f_s$ is not	$\exists y_s, y_d, y'_d, z_d \cdot$ $\texttt{resolve}(r, cwd, q_s/f_s, y_s) \wedge \neg\texttt{dir}(y_s)$ $\wedge\ \texttt{resolve}(r, cwd, q_d/f_d, y_d)$ $\wedge\ \texttt{dir}(y_d) \wedge (y_d[f_s]\uparrow \vee (y_d[f_s]z_d \wedge \neg\texttt{dir}(z_d)))$ $\wedge\ \texttt{similar}(r, r', cwd, q_d/f_d, y_d, y'_d) \wedge y_d \sim_{\{f_s\}} y'_d$ $\wedge\ \texttt{dir}(y'_d) \wedge y'_d[f_s]y_s$
Failure	No such file	$(\neg\exists y_s \cdot \texttt{resolve}(r, cwd, q_s/f_s, y_s)) \wedge r \doteq r'$
Failure	Is directory	$\exists y_s \cdot \texttt{resolve}(r, cwd, q_s/f_s, y_s) \wedge \texttt{dir}(y_s) \wedge r \doteq r'$
Failure	No such file	$(\neg\exists x_d \cdot \texttt{resolve}(r, cwd, q_d)) \wedge r \doteq r'$
Failure	Not a directory	$\exists x_d \cdot \texttt{resolve}(r, cwd, q_d, x_d) \wedge \neg\texttt{dir}(x_d) \wedge r \doteq r'$
Failure	Is directory	$\exists z_d \cdot \texttt{resolve}(r, cwd, q_d/f_d/f_s, z_d) \wedge \texttt{dir}(z_d) \wedge r \doteq r'$

6.1.3 Several arguments: `cp` q_s^1 ... q_s^n q_d

```
if test [ '-d' ; qd ] then
  success = true
  for qs in [ qs1 ; qs2 ; ...; qsn ] do
    if not (cp [ qs ; qd ]) then
```

```
        success = false
      fi
    done
    return success
  else
    return false
  fi
```

6.1.4 `cp -R` q_s/f_s q_d/f_d

Because `cp -R` behaves on non-directory just like `cp`, we only specify what happens when the source is a directory.

Success	Dir does not exist	$\exists y_s, x_d, x'_d.$ $\texttt{resolve}(r, cwd, q_s/f_s, y_s) \wedge \texttt{dir}(y_s)$ $\wedge\, \texttt{resolve}(r, cwd, q_d, x_d) \wedge x_d[f_d]\uparrow$ $\wedge\, \texttt{similar}(r, r', cwd, q_d, x_d, x'_d) \wedge x_d \sim_{\{f_d\}} x'_d$ $\wedge\, x'_d[f_d]y_s$
Success	Subdir does not exist	$\exists y_s, y_d, y'_d.$ $\texttt{resolve}(r, cwd, q_s/f_s, y_s) \wedge \texttt{dir}(y_s)$ $\wedge\, \texttt{resolve}(r, cwd, q_d/f_d, y_d) \wedge \texttt{dir}(y_d) \wedge y_d[f_s]\uparrow$ $\wedge\, \texttt{similar}(r, r', cwd, q_d/f_d, y_d, y'_d) \wedge y_d \sim_{\{f_s\}} y'_d$ $\wedge\, y'_d[f_s]y_s$
Success	Subdir exists and is empty	$\exists y_s, z_d.$ $\texttt{resolve}(r, cwd, q_s/f_s, y_s) \wedge \texttt{dir}(y_s)$ $\wedge\, \texttt{resolve}(r, cwd, q_d/f_d/f_s, z_d) \wedge \texttt{dir}(z_d) \wedge z_d[\varnothing]$ $\wedge\, \texttt{similar}(r, r', cwd, q_d/f_d/f_s, z_d, y_s)$
Unknown	Subdir exists and is not empty	$\exists y_s, z_d, z'_d.$ $\texttt{resolve}(r, cwd, q_s/f_s, y_s) \wedge \texttt{dir}(y_s)$ $\wedge\, \texttt{resolve}(r, cwd, q_d/f_d/f_s, z_d) \wedge \texttt{dir}(z_d) \wedge \neg z_d[\varnothing]$ $\wedge\, \texttt{similar}(r, r', cwd, q_d/f_d/f_s, z_d, z'_d)$

We have an overapproximation of the behaviours in the case where the subdirectory already exists.

6.1.5 Several arguments: `cp -R` q_s^1 ... q_s^n q_d

The destination directory may be created by the first copy and used in the next ones.

```
success = true
if not (cp -R [ qs1; qd ]) then
  success = false
fi
if test [ '-d' ; qd ] then
  for qs in [ qs2 ; ...; qsn ] do
    if not (cp [ qs ; qd ]) then
      success = false
    fi
  done
fi
return success
```

6.2 dpkg-maintscript-helper

The different forms of this command (except the first one) expect to receive all the arguments of the maintainer script, passed to the command in the form `$@`. We know already that any invocation of a maintainer script has either one or two arguments, that the first argument is one of the actions described in policy, and that the second argument if present is a debian version number. The idea is that the second argument indicates the old installed version of a package in case of an upgrade action. Note that this second argument may be present even in case the first argument is "install" in case the package previously was simply removed, and not purged.

We assume furthermore that

- the variable `DPKG_MAINTSCRIPT_NAME` holds the name of the maintainer script, that is one of `preinst`, `postinst`, `prerm`, `postrm`
- the function `absolute-pathname` succeeds if and only its argument starts with the `/` symbol
- the function `validate-optional-version` succeeds if and only if its argument is either the empty string or a correct debian version number.
- the function `debian-le-nl`, when applied to two arguments, succeeds if and only either the second argument is empty, or both arguments are debian versions and the first one is smaller in the debian version comparison order than the second.
- the function `package-owns-file`, when applied to two arguments, returns true when the second argument is in the static contents of the package named by the first argument, and false otherwise.
- the function `conffiles`, when applied to a package name, returns the list of absolute pathnames of the conffiles of that packages.
- the function `contents`, when applied to a package name, returns the list of absolute pathnames that constitute the static contents of that package.
- the function `is-prefix`, when applied to two absolute path names, returns `true` if and only if the first is a prefix of the second.
- the function `remove-prefix`, when applied to an absolute path p and a list l of absolute path names, returns the list of all path names q such that p/q is in l.
- FIXME explain recursive-fence.

6.2.1 `dpkg-maintscript-helper supports` *subcmd*

```
case subcmd in
  rm_conffile|mv_conffile|symlink_to_dir|dir_to_symlink) true
  *) false
  esac
```

6.2.2 `dpkg-maintscript-helper rm_conffile` *args*

where *args* is of the form:

```
conffile [prior-version [package]] -- $@
```

We use the following variables:

- conffile; as given as a parameter.
- lastversion: as given as the optional parameter `prior-version` if present, otherwise empty
- package: as given as an optional parameter, defaults to the name of the package owning the maintscript (variable `$DPKG_MAINTSCRIPT_PACKAGE`)
- scriptarg1: first argument given after - (must be non-emptyl)
- scriptarg2: second argument given after - (may be empty)

```
function prepare_rm_conffile(conffile,package)
  if [ -e "$conffile" ] && package-owns-file($package,$conffile)
  then
    choice
      mv -f "$conffile" "$conffile.dpkg-backup"
    or
      mv -f "$conffile" "$conffile.dpkg-remove"
    eciohc
  fi
end

function finish_rm_conffile(conffile)
  if [ -e "$conffile.dpkg-backup" ]
  then
    echo "Obsolete␣conffile␣$conffile␣has␣been␣modified␣by␣you."
    echo "Saving␣as␣$conffile.dpkg-bak␣..."
    mv -f "$conffile.dpkg-backup" "$conffile.dpkg-bak"
  fi
  if [ -e "$conffile.dpkg-remove" ]
  then
    echo "Removing␣obsolete␣conffile␣$conffile␣..."
    rm -f "$conffile.dpkg-remove"
  fi
end

function abort_rm_conffile(conffile,package)
  if package-owns-file($package,$conffile)
  then
    if [ -e "$conffile.dpkg-remove" ]
    then
      echo "Reinstalling␣$conffile␣that␣was␣moved␣away"
      mv "$conffile.dpkg-remove" "$conffile"
    fi
    if [ -e "$conffile.dpkg-backup" ]
    then
      echo "Reinstalling␣$conffile␣that␣was␣backed-up"
      mv "$conffile.dpkg-backup" "$conffile"
    fi
  fi
end
```

```
[ -n "$package" ] || exit 1
[ -n "$scriptarg1" ] || exit 1
[ -n "$dpkg_maintscript_name" ] || exit 1
[ -n "$dpkg_maintscript_package" ] || exit 1
absolute-pathname($conffile) || exit 1
validate_optional_version($lastversion) || exit 1

case "$dpkg_maintscript_name" in

preinst)
  if [ "$scriptarg1" = "install" -o "$scriptarg1" = "upgrade" ] &&
     [ -n "$scriptarg2" ] && debian-le-nl($scriptarg2,$lastversion)
  then
     prepare_rm_conffile(conffile,package)
  fi

postinst)
  if [ "$scriptarg1" = "configure" ] && [ -n "$scriptarg2" ] &&
     debian-le-nl($scriptarg2,$lastversion)
  then
    finish_rm_conffile(conffile)
  fi

postrm)
  if [ "$scriptarg1" = "purge" ]
  then
    rm -f "$conffile.dpkg-bak" "$conffile.dpkg-remove" \
      "$conffile.dpkg-backup"
  fi
  if [ "$scriptarg1" = "abort-install" -o "$scriptarg1" = "abort-upgrade" ]
       && [ -n "$scriptarg2" ] &&
       debian-le-nl($scriptarg2,$lastversion)
  then
        abort_rm_conffile(conffile,package)
  fi

esac
```

Remark : The non-deterministic choice in the `preinst` case corresponds in the real implementation to the comparison of the hashsum of the config file with a locally stored value, in order to determine whether the file has been modified since installation of the package.

6.2.3 `dpkg-maintscript-helper mv_conffile`

where *args* is of the form:

`old-conffile new-conffile [prior-version [package]] -- $@`

We use the following variables:

- oldconffile; as given as a parameter.
- newconffile; as given as a parameter.
- lastversion: as given as the optional parameter `prior-version` if present, otherwise empty

- package: as given as an optional parameter, defaults to the name of the package owning the maintscript (variable `$DPKG_MAINTSCRIPT_PACKAGE`)
- scriptarg1: first argument given after - (must be non-null)
- scriptarg2: second argument given after - (may be empty)

```
function prepare_mv_conffile(conffile,package)
  if [ -e "$conffile" ] && package-owns-file($package,$conffile)
  then
    choice
      mv -f "$conffile" "$conffile.dpkg-remove"
    or
      true
    eciohc
  fi
end

function finish_mv_conffile(oldconfile,newconffile,package)
  rm -f "$oldconffile.dpkg-remove"
  if [ -e "$oldconffile" ] && package-owns-file($package,$oldconffile)
  then
    echo "Preserving␣user␣changes␣to␣$newconfile␣(renamed␣from␣$oldconffile)"
    if [ -e $newconffile ]
    then
      mv -f $newconffile "$newconffile.dpkg-new"
    fi
    mv -f $oldconffile $newconffile
  fi
end

function abort_mv_conffile(conffile,package)
  if package-owns-file($package,$conffile) &&
     [ -e $conffile.dpkg-remove ]
  then
    echo "Reinstalling␣$conffile␣that␣has␣been␣move␣away"
    mv $conffile.dpkg-remove $conffile
  fi
end

[ -n "$package" ] || exit 1
[ -n "$scriptarg1" ] || exit 1
[ -n "$dpkg_maintscript_name" ] || exit 1
[ -n "$dpkg_maintscript_package" ] || exit 1
absolute-pathname($oldconffile) || exit 1
absolute-pathname($newconffile) || exit 1
validate_optional_version($lastversion) || exit 1

case "$dpkg_maintscript_name" in
preinst)
  if [ "$scriptarg1" = "install" -o "$scriptarg1" = "upgrade" ] &&
     [ -n "$scriptarg2" ] && debian-le-nl($scriptarg2,$lastversion)
  then
    prepare_mv_conffile($conffile,$package)
```

```
  fi

postinst)
  if [ $scriptarg1 = "configure" ] && [ -n $scriptarg2 ] &&
     debian-le-nl($scriptarg2,$lastversion)
  then
    finish_mv_conffile($oldconffile,$newconffile,$package)
  fi

postrm)
  if [ "$scriptarg1" = "abort-install" -o "$scriptarg1" = "abort-upgrade" ]
     && [ -n "$scriptarg2" ] &&
     debian-le-nl($scriptarg2,$lastversion)
  then
    abort_mv_conffile($oldconffile,$package)
  fi
esac
```

Remark : The non-deterministic choice in the `preinst` case corresponds in the real implementation to the comparison of the hashsum of the config file with a locally stored value, in order to determine whether the file has been modified since installation of the package.

6.2.4 `dpkg-maintscript-helper symlink_to_dir`

where *args* is of the form:

`pathname old-target [prior-version [package]] -- $@`

We use the following variables:

- symlink; as given by the parameter `pathname`.
- symlink_target; as given by the parameter `old-target`.
- lastversion: as given as the optional parameter `prior-version` if present, otherwise empty
- package: as given as an optional parameter, defaults to the name of the package owning the maintscript (variable `$DPKG_MAINTSCRIPT_PACKAGE`)
- scriptarg1: first argument given after `-` (must be non-null)
- scriptarg2: second argument given after `-` (may be empty)

```
[ -n "$dpkg_maintscript_name" ] || exit 1
[ -n "$dpkg_maintscript_package" ] || exit 1
[ -n "$package" ] || exit 1
[ -n "$symlink" ] || exit 1
absolute-pathname($symlink) || exit 1
ends-on-slash($symlink) || exit 1
[ -n "$symlink_target" ] || exit 1
[ -n "scriptarg1" ] || exit 1
validate_optional_version($lastversion) || exit 1

case "$dpkg_maintscript_name" in
```

```
preinst)
  if   [ "$1" = "install" -o "$1" = "upgrade" ] &&
       [ -n "$scriptarg2" ] && [ -h "$symlink" ] &&
            symlink-match($symlink,$symlink_target) &&
            debian-le-nl($scriptarg2,$lastversion)
  then
     mv -f "$symlink" "$symlink.dpkg-backup"
  fi

postinst)
  if [ "scriptarg1" = "configure" ] && [ -h "$symlink.dpkg-backup" ] &&
     symlink-match($symlink.dpkg-backup,$symlink_target)
  then
    rm -f "$symlink.dpkg-backup"
  fi

postrm)
  if [ "$scriptarg1" = "purge" ] && [ -h "$symlink.dpkg-backup" ]
  then
    rm -f "$symlink.dpkg-backup"
  fi
  if [ "$scriptarg1" = "abort-install" -o "$scriptarg1" = "abort-upgrade" ]
     && [ -n "$scriptarg2" ] && [ ! -e "$symlink" ]
     && [ -h "$symlink.dpkg-backup" ]
     && symlink-match("$symlink.dpkg-backup","$symlink_target")
     && dpkg-le-nl($scriptarg2,$lastversion)
  then
    echo "Restoring␣backup␣of␣$symlink␣..."
    mv "$symlink.dpkg-backup" "$symlink"
  fi

esac
```

6.2.5 `dpkg-maintscript-helper dir_to_symlink`

where *args* is of the form:

`pathname new-target [prior-version [package]] -- $@`

We use the following variables:

- pathname; as given as a parameter, but with a terminal `/` deleted.
- symlink_target; as given by the parameter `new-target`.
- lastversion: as given as the optional parameter `prior-version` if present, otherwise empty
- package: as given as an optional parameter, defaults to the name of the package owning the maintscript (variable `$DPKG_MAINTSCRIPT_PACKAGE`)
- scriptarg1: first argument given after `-` (must be non-null)
- scriptarg2: second argument given after `-` (may be empty)

```
function prepare_dir_to_symlink(package,pathname)
  for f in conffiles($package) do
    if is-prefix($package,$pathname) then exit 1 fi
  done
  recursive-fence($pathname,remove-prefix($pathname,contents($package)))
  mv -f $pathname $pathname.dpkg-backup
  mkdir $pathname
  touch $pathname/.dpkg-staging-dir
end

function finish_dir_to_symlink(pathname,symlink_target)
  if absolute-pathname($symlink_target)
  then
    abs_symlink_target = $symlink_target
  else
    abs_symlink_target = $pathname/$symlink_target
  fi
  rm $pathname/.dpkg-staging-dir
  for all f in $pathname do (* FIXME *)
    mv -f $pathname/$f $symlink_target
  done
  rmdir $pathname
  ln -s $symlink_target $pathname
  rm -r -f $pathname/.dpkg-backup
end

function abort_dir_to_symlink(pathname)
  echo "Restoring␣backup␣of␣$pathname␣..."
  if [ -h $pathname ]
  then
    rm -f $pathname
  else
    rm -f $pathname/.dpkg-staging-dir
    rmdir $pathname
  fi
  mv $pathname.dpkg-backup $pathname
end

[ -n "$dpkg_maintscript_name" ] || exit 1
[ -n "$dpkg_maintscript_package" ] || exit 1
[ -n "$package" ] || exit 1
[ -n "$pathname" ] || exit 1
absolute-pathname($pathname) || exit 1
[ -n "$symlink_target" ] || exit 1
[ -n "$scriptarg1" ] || exit 1
validate_optional_version($lastversion) || exit 1

case "$dpkg_maintscript_name" in
preinst)
  if [ "$scriptarg1" = "install" -o "$scriptarg1" = "upgrade" ] &&
     [ -n "$scriptarg2" ] && [ ! -h "$pathname" ] && [ -d "$pathname" ]
        && dpkg-le-nl($scriptarg2,$lastversion)
  then
```

```
    prepare_dir_to_symlink($package,$pathname)
  fi

postinst)
  if [ "$scriptarg1" = "configure" ] && [ -d "$pathname.dpkg-backup" ] &&
     [ ! -h "$pathname" ] && [ -d "$pathname" ] &&
          [ -f "$pathname/.dpkg-staging-dir" ]
  then
    finish_dir_to_symlink($pathname,$symlink_target)
  fi

postrm)
  if [ "$scriptarg1" = "purge" ] && [ -d "$pathname.dpkg-backup" ]
  then
    rm -rf "$pathname.dpkg-backup"
  fi
  if [ "$scriptarg1" = "abort-install" -o "$scriptarg1" = "abort-upgrade" ]
     && [ -n "$scriptarg2" ] && [ -d "$pathname.dpkg-backup" ]
     && ( ( [ ! -h "$pathname" ] && [ -d "$pathname" ]
                 && [ -f "$pathname/.dpkg-staging-dir" ] )
          ||
          ( [ -h pathname ] && symlink-match($pathname,$symlink_target) ))
     && dpkg-le-nl($scriptarg2,$lastversion)
  then
    abort_dir_to_symlink($pathname)
  fi
esac
```

6.3 mkdir

6.3.1 Options

- `-m`$^{(P)}$, `-v`, `-Z` and `-context` can be ignored.
- `-help` and `-version` make `mkdir` behave like noop.
- `-p`$^{(P)}$ makes `mkdir` create any missing intermediate pathname components.

6.3.2 `mkdir` $q/.$, `mkdir` $q/..$

Failure	File exists	$\exists x \cdot \texttt{resolve}(r, cwd, q, x) \wedge r \doteq r'$
Failure	No such file	$(\neg\exists x \cdot \texttt{resolve}(r, cwd, q)) \wedge r \doteq r'$

6.3.3 `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	$(\neg\exists x \cdot \texttt{resolve}(r, cwd, q, x)) \wedge r \doteq r'$
Failure	Not a dir	$\exists x \cdot \texttt{resolve}(r, cwd, q, x) \wedge \neg\texttt{dir}(x) \wedge r \doteq r'$

6.3.4 `mkdir -p` $c_1/c_2/.../c_n$

`mkdir -p` applied to one path $c_1/c_2/.../c_n$ where c_i are path components can be replaced by the following CoLiS script:

```
for path in [ 'c1/'; 'c1/c2/'; ...; 'c1/c2/.../cn/' ] do
  if not test [ '-d'; path ] then
    mkdir [ path ]
  fi
```

6.3.5 Several arguments: `mkdir opts` q^1 ... q^n

```
success = true
for q in [ q1 ; ... ; qn ] do
  if not (mkdir [ otps ; q1 ]) then
    success = false
  fi
done
return success
```

6.4 mv

6.4.1 POSIX options

`-f`: do not prompt for confirmation if the destination path exists; any previous occurrence of `-i` is ignored. Set by default in our model.

`-i`: prompt for confirmation if the destination path exists; any previous occurrence of `-f` is ignored. Forbidden in our model.

6.4.2 GNU options

`-b`: Make backup for overwritten or removed file. Forbidden.

`-f`: Do not prompt the user before removing a destination file. By default.

`-i`: Prompt whether to overwrite each existing destination file, regardless of its permissions. Forbidden.

`-n`: Do not overwrite an existing file; silently do nothing instead. Forbidden.

`-u`: Do not update a non-directory that has an existing destination. Forbidden.

`-v`: Print the name of each file before moving it. Forbidden.

`trailing-slashes`: Remove any trailing slashes from each source argument. Forbidden.

`-S suffix`: Append suffix to each backup file. Forbidden.

`-t directory`: Specify the destination directory. Forbidden.

`-T`: Do not treat the last operand specially when it is a directory or a symbolic link to a directory. Forbidden.

`-Z`: Adjust the SELinux security context according to the system default type for destination files and each created directory.

6.4.3 Two arguments: `mv` q_1 q_2

```
if test [ '-d'; q2 ] then
  mv2dir q1 q2
else
  rename q1 q2
fi
```

6.4.4 `rename` q_o q_n

The `rename` function is defined in the System Interface volume of POSIX.1-17.

The specification requires to test the following boolean constraints on the arguments q_o (*old*) and q_n (*new*):

- b_{dot} is true iff one of q_o or q_n ends in `'.'` or `'..'`;
- b_{slash} is true iff q_n contains characters different from `'/'` and ends by a `'/'` (after link processing);

MS: and trailing '/'

- $b_{ancestor}$ is true iff q_o is an ancestor of q_n. This situation is specified by the test: "`normalize`(cwd, q_o) is a strict prefix of `normalize`(cwd, q_n)" This test is an *under-approximation* of initial condition because `normalize` does not expand symbolic links and therefore the paths computed may not be "fully" normalized. A precise test for $b_{ancestor}$ shall use a reachability predicate in the logic.

6.4.4.1 Paths end in dot: i.e. b_{dot} is true

Failure	Path ends in dot or dot-dot	$r \doteq r'$

6.4.4.2 `rename` q_s/f_s q_d/f_d

Failure	Old does not resolve	$(\neg\exists y_s \cdot \texttt{resolve}(r, cwd, q_s/f_s, y_s)) \wedge r \doteq r'$
Failure	No path to new	$(\neg\exists x_d \cdot \texttt{resolve}(r, cwd, q_d, x_d)) \wedge r \doteq r'$
Success	Old same as new	$\exists y_s, y_d \cdot \texttt{resolve}(r, cwd, q_s/f_s, y_s) \wedge \texttt{resolve}(r, cwd, q_d/f_d, y_d)$ $\wedge\, y_s \doteq y_d \wedge r \doteq r'$
Failure	Old is dir, new not dir	$\exists y_s, y_d \cdot \texttt{resolve}(r, cwd, q_s/f_s, y_s) \wedge \texttt{dir}(y_s)$ $\wedge\, \texttt{resolve}(r, cwd, q_d/f_d, y_d) \wedge \neg\texttt{dir}(y_d) \wedge r \doteq r'$
Success	Old is dir, new is empty dir	$\exists r^{(i)}, x_s, x_s^{(i)}, y_s, x_d, x_d^{(i)}, x'_d, y_d \cdot$ $\texttt{resolve}(r, cwd, q_s/f_s, y_s) \wedge \texttt{dir}(y_s)$ $\wedge\, \texttt{resolve}(r, cwd, q_d/f_d, y_d) \wedge \texttt{dir}(y_d) \wedge y_s \not\doteq y_d \wedge y_d[\varnothing]$ $\wedge\, \texttt{similar}(r, r^{(i)}, cwd, q_s, x_s, x_s^{(i)}) \wedge x_s \sim_{\{f_s\}} x_s^{(i)} \wedge x_s^{(i)}[f_s]\uparrow$ $\wedge\, \texttt{similar}(r^{(i)}, r', cwd, q_d, x_d^{(i)}, x'_d) \wedge x_d^{(i)} \sim_{\{f_d\}} x'_d \wedge x'_d[f_d]y_s$
Failure	Old is dir, new not empty dir	$\exists y_s, y_d \cdot \texttt{resolve}(r, cwd, q_s/f_s, y_s) \wedge \texttt{dir}(y_s)$ $\wedge\, \texttt{resolve}(r, cwd, q_d/f_d, y_d) \wedge \texttt{dir}(y_d) \wedge y_s \not\doteq y_d \wedge \neg y_d[\varnothing] \wedge r \doteq r'$
Success	Old is file, new is not dir	$\exists r^{(i)}, x_s, x_s^{(i)}, y_s, x_d, x_d^{(i)}, x'_d, y_d \cdot$ $\texttt{resolve}(r, cwd, q_s/f_s, y_s) \wedge \neg\texttt{dir}(y_s)$ $\wedge\, \texttt{resolve}(r, cwd, q_d, x_d) \wedge (x_d[f_d]\uparrow \vee (x_d[f_d]y_d \wedge y_s \not\doteq y_d \wedge \neg\texttt{dir}(y_d)))$ $\wedge\, \texttt{similar}(r, r^{(i)}, cwd, q_s, x_s, x_s^{(i)}) \wedge x_s \sim_{\{f_s\}} x_s^{(i)} \wedge x_s^{(i)}[f_s]\uparrow$ $\wedge\, \texttt{similar}(r^{(i)}, r', cwd, q_d, x_d^{(i)}, x'_d) \wedge x_d^{(i)} \sim_{\{f_d\}} x'_d \wedge x'_d[f_d]y_s$
Failure	Old is file, new is a dir	$\exists y_s, y_d \cdot \texttt{resolve}(r, cwd, q_s/f_s, y_s) \wedge \neg\texttt{dir}(y_s)$ $\wedge\, \texttt{resolve}(r, cwd, q_d/f_d, y_d) \wedge \texttt{dir}(y_d) \wedge r \doteq r'$

Failure	Old is dir, new is absent and b_{slash}	$\exists y_s, x_d \cdot \texttt{resolve}(r, cwd, q_s/f_s, y_s) \wedge \texttt{dir}(y_s)$ $\wedge\, \texttt{resolve}(r, cwd, q_d, x_d) \wedge x_d[f_d]\uparrow \wedge r \doteq r'$
Success	Old is dir, new is absent and not b_{slash}	$\exists r^{(i)}, x_s, x_s^{(i)}, y_s, x_d, x_d^{(i)}, x'_d, y_d \cdot$ $\texttt{resolve}(r, cwd, q_s/f_s, y_s) \wedge \texttt{dir}(y_s)$ $\wedge\, \texttt{resolve}(r, cwd, q_d, x_d) \wedge x_d[f_d]\uparrow$ $\wedge\, \texttt{similar}(r, r^{(i)}, cwd, q_s, x_s, x_s^{(i)}) \wedge x_s \sim_{\{f_s\}} x_s^{(i)} \wedge x_s^{(i)}[f_s]\uparrow$ $\wedge\, \texttt{similar}(r^{(i)}, r', cwd, q_d, x_d^{(i)}, x'_d) \wedge x_d^{(i)} \sim_{\{f_d\}} x'_d \wedge x'_d[f_d]y_s$
Failure	Old ancestor of new, $b_{ancestor}$	$\exists y_s, x_d \cdot \texttt{resolve}(r, cwd, q_s/f_s, y_s) \wedge \texttt{dir}(y_s)$ $\wedge\, \texttt{resolve}(r, cwd, q_d, x_d) \wedge r \doteq r'$

The IEEE Std 1003.1-2017 does not specify what happens when the source is a file and the destination ends in trailing slashes. Trailing slashes are ignored for destination in the GNU implementation.

6.4.5 `mv2dir` q_s q_d

Assumption: q_d resolves to a directory.

Let f_s be the last path component of q_s. We denote by q_{dp} the path built by concatenation of q_d, `'/'` and f_s.

"The `mv` utility shall perform actions equivalent to the `rename` function" (IEEE Std 1003.1-2017) with q_s as old and destination path q_{dp} as new. The behaviors specified in the standard in addition to the general case concern the following cases: (a) the destination path exists, (b) the permissions for removing source, and (c) the characteristics duplicated by the move of a directory. We should ignore (b) and (c) because these attributes are not dealt in our logic. For (a), the standard repeats the behavior of `rename`.

MS: Why?

	Rename	`rename` q_s q_{dp}

6.4.6 More than two arguments: `mv` q_1 ... q_n

```
success = true
if test [ '-d' ; qn ] then
  for q in [ q1 ; ... ; qn-1 ] do
    if not (mv2dir q qn) then
      success = false
    fi
  done
else
  success = false
fi
return success
```

6.5 rm

6.5.1 POSIX options

`-f`: Do not ask anything; do not complain in case of lack of operands or in case an operand does not exist.

`-i`: Prompt before removal. Forbidden model.

`-r`,`-R`: Recursive.

6.5.2 GNU options

`-force`: Same as `-f`.

`-I`: Prompt less often than `-i`. Forbidden.

`teractive[=WHEN]`: Prompt according to WHEN. Forbidden.

`-one-file-system`: When removing a hierarchy recursively, skip any directory that is on a file system different from that of the corresponding command line argument

`no-preserve-root`: Do not treat '/' specially

`serve-root[=all]`: Do not remove '/' (default); with 'all', reject any command line argument on a separate device from its parent

`-recursive`: Same as `-R`.

`-d`, `-dir`: Remove empty directories

`-v`, `-verbose`: Forbidden.

`-help`: Forbidden.

`-version`: Forbidden.

6.5.3 `rm` $q/.$, `rm` $q/..$, `rm /`

Failure	invalid argument	$r \doteq r'$

6.5.4 `rm` p/f

Success		$\exists x, x', y \cdot$ $\texttt{resolve}(r, cwd, q, y) \wedge \neg\texttt{dir}(y)$ $\wedge\ \texttt{similar}(r, r', cwd, q, x, x') \wedge x \sim_{\{f\}} x'$ $\wedge\ \texttt{dir}(x') \wedge x'[f]\uparrow$
Failure	Is a directory	$\exists y \cdot \texttt{resolve}(r, cwd, q, y) \wedge \texttt{dir}(y) \wedge r \doteq r'$
Failure	No such file	$(\neg\exists y \cdot \texttt{resolve}(r, cwd, q)) \wedge r \doteq r'$

6.5.5 `rm -f args...`

In our model, this is equivalent to:

```
|| NoOutput(rm args...) || true
```

How is NoOutput represented in CoLiS?

6.5.6 `rm -r` q/f, `rm -R` q/f

Success		$\exists x, x', y\cdot$ $\texttt{resolve}(r, cwd, q/f, y)$ $\wedge\, \texttt{similar}(r, r', cwd, q, x, x') \wedge x \sim_{\{f\}} x'$ $\wedge\, \texttt{dir}(x') \wedge x'[f]\uparrow$
Failure	No such file	$(\neg\exists y \cdot \texttt{resolve}(r, cwd, q/f)) \wedge r \doteq r'$

6.5.7 Several arguments

We apply the single-argument `rm` to all the arguments, without stopping on errors. The return code of the whole is 0 if all the `rm`s went well and $\neq 0$ otherwise.

6.6 rmdir

POSIX options:

`-p`: Remove all directories in a pathname.

6.6.1 `rmdir` $q/.$, `rmdir` $q/..$

Failure	Invalid argument	$r \doteq r'$

6.6.2 `rmdir` q/f

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

6.6.3 `rmdir -p` $c_1/c_2/.../c_n$

`rmdir -p` applied to one path $c_1/c_2/.../c_n$ (where c_i are path components) can be replaced by the following CoLiS script:

```
for path in [ 'c1/c2/.../cn/'; ...; 'c1/c2/'; 'c1/' ] do
  rmdir [ path ]
fi
```

6.6.4 Several arguments

We apply the single-argument `rmdir` to all the arguments, without stopping on errors. The return code of the whole is 0 if all the `rmdir`s went well and $\neq 0$ otherwise.

6.7 test, [

The `[` command is a variant of the `test` command which checks, in addition to the action performed by `test`, that its last argument is `]`. Hence, when specifying the semantics we may identify these two commands.

6.7.1 `test -b` q

Checks for existence of a block device.

Success	Is a block device	$\exists y \cdot \texttt{resolve}(r, cwd, q, y) \wedge \texttt{block}(y) \wedge r \doteq r'$
Failure	No such file	$(\neg \exists y \cdot \texttt{resolve}(r, cwd, q, y)) \wedge r \doteq r'$
Failure	Not a block device	$\exists y \cdot \texttt{resolve}(r, cwd, q, y) \wedge \neg\texttt{block}(y) \wedge r \doteq r'$

6.7.2 `test -c` q

Checks for existence of a character device.

Success	Is a char device	$\exists y \cdot \texttt{resolve}(r, cwd, q, y) \wedge \texttt{char}(y) \wedge r \doteq r'$
Failure	No such file	$(\neg \exists y \cdot \texttt{resolve}(r, cwd, q, y)) \wedge r \doteq r'$
Failure	Not a char device	$\exists y \cdot \texttt{resolve}(r, cwd, q, y) \wedge \neg\texttt{char}(y) \wedge r \doteq r'$

6.7.3 `test -d` q

Checks for existence of a directory.

Success	Is a directory	$\exists y \cdot \texttt{resolve}(r, cwd, q, y) \wedge \texttt{dir}(y) \wedge r \doteq r'$
Failure	No such file	$(\neg \exists y \cdot \texttt{resolve}(r, cwd, q, y)) \wedge r \doteq r'$
Failure	Not a directory	$\exists y \cdot \texttt{resolve}(r, cwd, q, y) \wedge \neg\texttt{dir}(y) \wedge r \doteq r'$

6.7.4 `test -e` q

Checks for existence of a file, no matter the file type.

Success	Exists	$\exists y \cdot \texttt{resolve}(r, cwd, q, y) \wedge r \doteq r'$
Failure	No such file	$(\neg \exists y \cdot \texttt{resolve}(r, cwd, q, y)) \wedge r \doteq r'$

6.7.5 `test -f` q

Checks for existence of a regular file.

Success	Is a regular file	$\exists y \cdot \texttt{resolve}(r, cwd, q, y) \wedge \texttt{reg}(y) \wedge r \doteq r'$
Failure	No such file	$(\neg\exists y \cdot \texttt{resolve}(r, cwd, q, y)) \wedge r \doteq r'$
Failure	Not a regular file	$\exists y \cdot \texttt{resolve}(r, cwd, q, y) \wedge \neg\texttt{reg}(y) \wedge r \doteq r'$

6.7.6 `test -h` q

Checks for existence of a symbolic link.

Success	Is a symlink	$\exists y \cdot \texttt{resolve}(r, cwd, q, y) \wedge \texttt{symlink}(y) \wedge r \doteq r'$
Failure	No such file	$(\neg\exists y \cdot \texttt{resolve}(r, cwd, q, y)) \wedge r \doteq r'$
Failure	Not a symlink	$\exists y \cdot \texttt{resolve}(r, cwd, q, y) \wedge \neg\texttt{symlink}(y) \wedge r \doteq r'$

6.7.7 `test -p` q

Checks for existence of a named pipe.

Success	Is a pipe	$\exists y \cdot \texttt{resolve}(r, cwd, q, y) \wedge \texttt{fifo}(y) \wedge r \doteq r'$
Failure	No such file	$(\neg\exists y \cdot \texttt{resolve}(r, cwd, q, y)) \wedge r \doteq r'$
Failure	Not a pipe	$\exists y \cdot \texttt{resolve}(r, cwd, q, y) \wedge \neg\texttt{fifo}(y) \wedge r \doteq r'$

6.7.8 `test -S` q

Checks for existence of a socket.

Success	Is a socket	$\exists y \cdot \texttt{resolve}(r, cwd, q, y) \wedge \texttt{socket}(y) \wedge r \doteq r'$
Failure	No such file	$(\neg\exists y \cdot \texttt{resolve}(r, cwd, q, y)) \wedge r \doteq r'$
Failure	Not a socket	$\exists y \cdot \texttt{resolve}(r, cwd, q, y) \wedge \neg\texttt{socket}(y) \wedge r \doteq r'$

6.7.9 `test -[GOgkrsuwx]` q

These are all POSIX or GNU test operators testing certain file permissions, non-zero size, or ownership attributes of files. Since we decided not to include file attributes in our model we obtain on our level of abstraction a non-deterministic semantics. If the path does not resolve then we certainly have a failure, if it does resolve the command may or may not fail.

Success	Attribute OK	$\exists y \cdot \texttt{resolve}(r, cwd, q, y) \wedge r \doteq r'$
Failure	No such file	$(\neg\exists y \cdot \texttt{resolve}(r, cwd, q, y)) \wedge r \doteq r'$
Failure	Attribute not OK	$\exists y \cdot \texttt{resolve}(r, cwd, q, y) \wedge r \doteq r'$

6.7.10 `test` q_1 `-nt` q_2, `test` q_1 `-ot` q_2

These binary tests are a GNU extension comparing the dates of two files. Our semantics is non-deterministic for the same reasons as explained in Section 6.7.9.

Success	Attribute OK	$(\exists y_1 \cdot \texttt{resolve}(r, cwd, p_1, y_1) \wedge (\exists y_2 \cdot \texttt{resolve}(r, cwd, p_2, y_2) \wedge r \doteq r'$
Failure	No such file 1	$(\neg\exists y \cdot \texttt{resolve}(r, cwd, p_1, y)) \wedge r \doteq r'$
Failure	No such file 2	$(\neg\exists y \cdot \texttt{resolve}(r, cwd, p_2, y)) \wedge r \doteq r'$
Failure	Attribute not OK	$(\exists y_1 \cdot \texttt{resolve}(r, cwd, p_1, y_1) \wedge (\exists y_2 \cdot \texttt{resolve}(r, cwd, p_2, y_2) \wedge r \doteq r'$

6.7.11 `test !` $-op\ arg$

```
not (test -op arg)
```

6.8 `test` arg_1 `-a` arg_2

Remark: In reality, the recognition of an `-a` expression is not done on a textual level but by a special-purpose parser which builds an abstract syntax tree of test expressions.

```
if test arg1 then test arg2 else false fi
```

6.9 `test` arg_1 `-o` arg_2

The same remark about the parsing of test expression as in Section 6.8 applies.

```
if test arg1 then true else test arg2 fi
```

6.10 touch

6.10.1 POSIX options

`-a`: Change only the access time. Ignored in our model.

`-c`: Do not create if the target does not exist. Only useful when considering timestamps. In our model, `touch -c` behaves like `true`.

`-d`: Use the given datetime. Ignored in our model.

`-m`: Change only the modification time. Ignored in our model.

`-r`: Use the given file as a reference for the time. In our model, this, `touch -r f args...` behaves like `test -e f && touch args...`.

`-t`: Use the given time. Ignored in our model.

RT: this means that the *option* is ignored, not the command, right?

6.10.2 `touch` $q/.$, `touch` $q/..$

Success	Exists	$\exists x \cdot \texttt{resolve}(r, cwd, q, x) \wedge r \doteq r'$
Failure	No such file	$(\neg\exists x \cdot \texttt{resolve}(r, cwd, q)) \wedge r \doteq r'$

6.10.3 `touch` q/f

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

6.10.4 Several arguments

We apply the single-argument `touch` to all the arguments, without stopping on errors. The return code of the whole is 0 if all the `touch`s went well and $\neq 0$ otherwise.

6.11 which

The `which` command is somehow an extension of the `test -x` command, which checks if its arguments appears in one of the directories listed in the `PATH` variable. The following specifies the `which` utilities installed by the `debianutils` package of Debian.

6.11.1 `which` $q \cdot f$

When the argument given is not limited to a single path component, but contains some `/`, then `which` does not look in the `PATH`: it just checks existence of the given path: indeed it then acts exactly as `test -f` $q \cdot f$ `-a -x` $q \cdot f$, plus the effect of printing the path in case of success.

Notice below that we over-approximate the knowledge of executability, since that attribute is not supported in our contraint setting.

			STDOUT
Success	Is a regular and executable file	$\exists y \cdot \texttt{resolve}(r, cwd, q \cdot f, y) \wedge \texttt{reg}(y) \wedge r \doteq r'$	$q \cdot f$
Failure	No such file	$(\neg \exists x \cdot \texttt{resolve}(r, cwd, q \cdot f)) \wedge r \doteq r'$	
Failure	Not a regular file, or not executable	$\exists y \cdot \texttt{resolve}(r, cwd, q \cdot f, y) \wedge r \doteq r'$	

6.11.2 `which` f

Checks for existence of regular and executable file named f occuring in the `PATH`

```
for q in [ PATH ] do
    if which [ q/f ] then
      (* echo not necessary: `which` already prints it *)
      return true
    fi
done
return false
```

6.11.3 `which -a` f

With option `-a`, `which` greedily searches in the PATH for all occurrences.

```
success = false
for q in [ PATH ] do
    if which [ q/f ] then
      (* echo not necessary: `which` already prints it *)
      success = true
    fi
done
return success
```

6.11.4 Several arguments: `which` q^1 ... q^n

Searches for all arguments, succeeds if all are found.

```
if n > 0 then
  success = true
  for q in [ q1 ; q2 ; ...; qn ] do
    if not (which [ q ]) then
      success = false
    fi
  done
  return success
else
  return false
fi
```

Notice that `which` returns false when given no argument

References

[IEE] IEEE and The Open Group. *POSIX.1-2008/Cor 1-2013*. http://pubs.opengroup.org/onlinepubs/9699919799/.

[JT18] 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*, number 10900 in Lecture Notes in Artificial Intelligence, pages 439–454, Oxford, UK, July 2018. Springer Verlag.