Statically Analyzing Shell Scripts and Their Effects in Context

ANIRUDH NARSIPUR, Brown University

The shell is used for a variety of tasks, including data processing, system administration, and continuous deployment. However, its elaborate semantics and use of opaque commands make writing shell programs prone to bugs, which can have catastrophic and irreversible consequences—e.g., overwriting files, deleting file systems, or crashing scripts. This project presents a system for identifying bugs and unintended behavior in shell programs through ahead-of-time semantic reasoning. At its core, the system is powered by a symbolic execution engine that models the execution of scripts over abstract inputs. Its reasoning capabilities extend to the file system by way of a model that handles realistic paths with variables, non-linear references, and globs.

1 CONTRIBUTIONS

This project builds on my undergraduate thesis [11]. In relation to the undergraduate thesis, this work covers a larger subset of the POSIX shell semantics and introduces an initial version of a filesystem logic that can express constraints on symbolic pathnames. I have solely implemented the system described in this report in roughly 9k LOC of Python code. This research has contributed to a forthcoming publication [8]. While this report focuses on my contributions, this work was undertaken in collaboration with others namely, Eric Zhao, Evangelos Lamprou, Lukas Lazarek, and Nikos Vasilakis.

2 INTRODUCTION

The shell's ability to easily compose opaque commands has led to it remaining a popular choice for orchestration, configuration, and data pipelining workloads causing it to steadily remain among the top ten programming languages on GitHub finishing at eighth in 2024 [6].

Unfortunately, shell programs are prone to bugs [4, 5, 9]. At best, buggy scripts crash the execution of a long-running task; at worst, they silently corrupt the broader environment in which they execute, affecting user data, modifying system files, and rendering entire systems unusable. High-profile examples of such computations gone wrong include Valve's Steam game distribution engine, whose update script deleted entire file systems due to the contents of a variable [13], and an open-source Nvidia driver script, which caused deletion of the `/usr` directory due to an unintended space character [10].

Preventing shell script bugs requires reasoning about the script behavior ahead-of-time before the script executes on a concrete environment with concrete input (or sometimes even before it is even deployed to be later run on any user environment). Although such reasoning for robust static analysis is widely used for other languages, shell scripts are only supported by very limited analyses for bug finding in the form of linting [14].

Challenges: Developing a static analysis tool for the shell requires addressing two key technical challenges.

First, the shell's semantics is bi-modal: shell constructs and expansion are evaluated by the interpreter, while opaque commands are evaluated through fork-execve. Any static reasoning should support precise reasoning for both modes of execution. Second, shell programs extensively utilize the file system, locating resources in this hierarchy by way of paths. Though merely strings within the shell, these paths take on much richer behavior when resolved, and they may contain variables, non-linear references, and glob patterns. Each of these features requires further complexity to model. Ahead-of-time reasoning must compactly represent the (potentially infinite) sets of possible file system states.

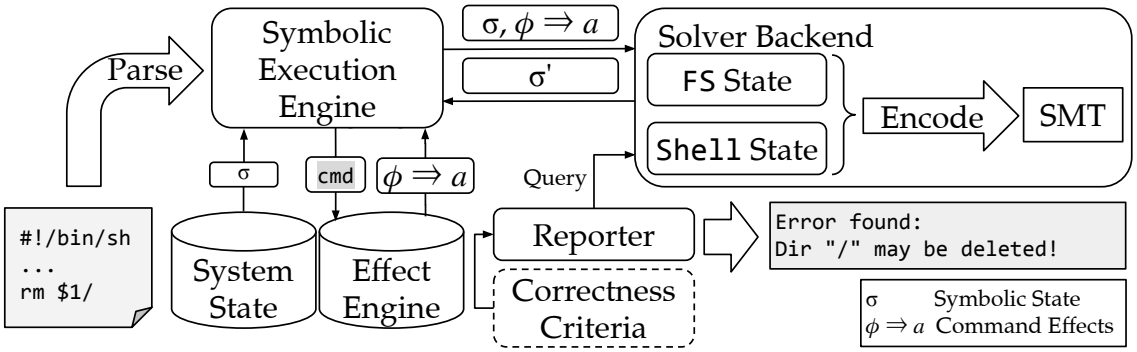


Fig. 1. **Shseer overview.** Shseer parses the input shell script, builds a symbolic representation of the system state, symbolically executes it, and converts the symbolic state into a set of constraints, where unsatisfiability indicates a bug.

Our approach: In this project, building on an initial prototype [11], we present Shseer, a system to find bugs in POSIX shell scripts by statically analyzing them before they execute. Shseer achieves this by analyzing the semantics of a script and not just its syntax: its reasoning is driven by models of the individual opaque commands, the language of the shell composing them, and the interactions of these commands and the shell with the filesystem and the broader environment.

Shseer identifies bugs in two categories. First, Shseer detects filesystem effects that may delete a system directory (*e.g.*, filesystem root or `/usr`) or incorrectly compose (*e.g.*, reading from a deleted file). Second, Shseer detects bugs that arise in the shell from confusions over shell semantics (*e.g.*, using an unpassed positional parameter).

To do so, Shseer supports both modes of the shell's semantics by combining a symbolic execution engine, for reasoning about shell construct evaluation and expansion, and expressive command specifications, for reasoning about the execution of opaque binary commands (*e.g.*, `rm` or `mv`).

Finally, Shseer introduces a *novel specification logic for file system client programs* that is amenable to a precise, fully automated symbolic reasoning of a program's effects on file resources. The underlying model of the file hierarchy supports realistic paths containing symbolic variables, non-linear references, and glob patterns. To compactly represent infinitely many states, the representation is composed of a *partial file tree* and a *constraint system.*

3 SYSTEM OVERVIEW

Consider the core of a bug in the Steam for Linux updater (Fig. 2), which resulted in the deletion of the filesystem of several Steam users [12]. The updater first deletes the current version, whose location it identifies via a variable, `STEAMROOT`. Its value is determined using runtime expansion, all in a subshell: identify the path of the current script (`${0}`), expand it to remove anything after the last slash (`%/*`), change to that path (`cd`…), and report the current directory (`$PWD`). For some paths (*e.g.*, `~/.steam/upd.sh`), expansion results in the parent directory as intended (*e.g.*, `/home/jcarb/.steam`); for other paths (*e.g.*, ones lacking any directories like `upd.sh`), expansion results in the script name, causing `cd` to fail(as the argument is not a directory) and `STEAMROOT` to end up empty. The result: `rm -fr /*` deletes everything user-writable on the system.

To develop a semantics based analysis for the above bug first requires defining correctness criteria. Rather than detecting a syntactic pattern— `rm` is passed a string of the form `\$var/*` — a system needs to check a correctness predicate over the behavior of the script—does the script delete the root directory? Checking such correctness predicates

```
1 #!/bin/sh
2 STEAMROOT="$(cd "${0%/*}" && echo $PWD)"
3 # ...
4 rm -fr $STEAMROOT/*
```

Fig. 2. **Core of a Steam updater bug** [13]. When expansion results in an empty `STEAMROOT` string (ln. 2), the script deletes everything user-writable (ln. 4).

requires developing significant machinery. The shell environment must be tracked to compute the range of values for `\$STEAMROOT` and `\$0` and mutations/restrictions applied to them. Next, the control flow of the script must be computed to determine if the `rm` command going to be invoked and with what arguments i.e the possible execution traces of the script. If the `rm` command is invoked, a *specification* is required to determine the possible effect of the invocation. As the `rm` command modifies the filesystem state, it is insufficient to model just the shell state—a model of the filesystem state is required as well. With the knowledge of the shell/filesystem state and the possible transformations over them—the correctness predicate can now be checked to see if the script behaves dangerously. Shseer addresses these challenges through a symbolic execution engine that tracks the shell/filesystem state and symbolically executes commands to determine possible execution traces of script. By checking correctness predicates over these traces using a constraint solver, Shseer can report dangerous bugs to the user while being robust to syntactic permutations.

Inputs to Shseer: Shseer reports bugs in POSIX shell scripts. The POSIX specification [1] is the de-facto standard defining a common core language which many popular shells (*e.g.*, Bash) build upon, making it the natural choice of semantics. The POSIX semantics is ambiguous and popular shells such as Bash and Dash differ from it in subtle ways [7]. Shseer targets the semantics of Bash running in POSIX mode—essentially corresponding to the POSIX standard with Bash's particular choices for unspecified behavior. Shseer uses the POSIX-compliant libdash parser [2] to obtain an abstract syntax tree (AST) which is then passed to the symbolic execution engine. As the behavior of a script depends on it's execution environment (shell/filesystem state) (*e.g.*, . the value of the `PWD` variable), Shseer also takes as input the execution environment of the script. By default Shseer considers the execution environment abstractly (§4.1) but alternatively can take in the current *concrete* environment. For example, instead of considering some arbitrary value of the current working directory(`PWD`), Shseer will analyze the script with respect to the current working directory.

Symbolic execution: Having initialized a starting shell/filesystem state Shseer symbolically executes the script exploring multiple execution paths. For the script in Fig. 2, Shseer first steps through the subshell invocation that computes the value of `STEAMROOT`. To evaluate the `cd` command Shseer first symbolically applies the shell's word expansion (string rewriting) rules to compute the value of `$0%/*`. Interpreting the `cd` command requires an interplay of the shell-filesystem state-the *PWD* variable is possibly updated depending on whether the `cd` argument can be resolved to an existing directory. By passing the current state and the specification for the `cd` command to the constraint solver, Shseer can determine with the information available that the command could succeed/fail and explores both possible execution branches—including the one where the `cd` command fails.

Next,Shseer must evaluate the opaque `rm` command. Shseer includes a set of specifications for core filesystem manipulation utilities that describe their transformation of the filesystem state. Each command invocation pattern (the command with its set of flags/options) has an associated parametric specification. When analyzing a script, Shseer first parses a command invocation `rm -fr \$STEAMROOT/*` to identify the command and its arguments making some simplifying assumptions (§4.3) to make such symbolic parsing tractable. Matching this command invocation against the parametric invocation library Shseer can generate a specification for the behavior of this command.

File system effects: The specification for the `rm` command is defined in Shseer's filesystem logic (§5) as a disjunction of preconditions and effects that tracks the possible behaviors of the command depending on the current state of the filesystem. The constraint solver combines constraints about strings such as the value of `STEAMROOT` with constraints about the filesystem state. What the symbolic execution treated as strings, the constraint solver interprets as paths, resolving the path `$STEAMROOT/*` to identify the possible entries being updated. It leverages an SMT solver to check the feasibility of this update and generates a new symbolic filesystem state. Finally, the correctness predicate—the filesystem root should not be deleted— can be checked against this updated filesystem state. In this case, the correctness predicate is violated and Shseer can report a bug.

Tradeoffs: Shseer must make some tradeoffs to ensure analysis is tractable. Shseer is a bug finder and not a verification tool. Most real scripts contain custom commands whose behavior Shseer cannot determine statically. The only sound choice is for Shseer to declare that the command could have arbitrary behavior. However, the sound choice may be *worse* than an unsound one, that proceeds with optimistic assumptions such as assuming the custom command does not touch the filesystem. Similarly, a command invocation such as `rm $1 $2` could at runtime correspond to the concrete invocations `rm -r file1` or `rm file1 file2`. The `-r` flag modifies the behavior of `rm` command and the corresponding output. Considering the full set of possible concrete command invocations would lead to an exponential blowup making analysis intractable. Hence, Shseer assumes that symbolic variables such as `\$1` are command arguments, while making smarter choices when possible.

4 SYMBOLIC EXECUTION FOR SHELL PROGRAMS

State	σ	::=	$\langle \text{opts}, \ \rho_g, \overline{\rho_\ell}, \ F,$ $s_{\text{cwd}}, \overline{fd}, \ n_{\$?}, n_{\text{loop}}$ $\text{optoff}^?, \mathbb{C}, \mathbb{FS}, p_c \rangle$
Options	opts	$\subseteq$	$\{\texttt{allexport}, \ldots\}$
Global environment	ρ_g	:	$str \mapsto s \times b_{\text{mut}} \times b_{\text{export}}$
Local environment	ρ_ℓ	:	$str \mapsto s \times b_{\text{mut}} \times b_{\text{export}}$
Function	F	:	$str \mapsto command$
File descriptor	fd	:	$\mathbb{N}$
Symbolic string	s	::=	$str \mid var$
Symbolic integer	i	::=	$\mathbb{Z} \mid var$
Symbolic boolean	b	::=	$\top \mid \bot \mid var$
Symbolic variable	var	$\in$	$\mathcal{V} \subseteq \mathcal{P}(\Sigma^+)$
String	str	$\in$	Σ (*e.g.*, UTF-8)
Constraint store	$\mathbb{C}$	::=	$\overline{ac}$
Constraint	ac	::=	$s = s \mid s < s \mid slen(s) = i$ $\mid i = i \mid i < i \mid i = i - i \mid \mho$ $\mid b \mid ac \wedge ac \mid ac \vee ac \mid \neg ac$
File system	$\mathbb{FS}$ ::= (omitted)		
File system constraints	$\mho$ ::= (omitted)		

Fig. 3. Shseer's modeled shell state.

At the core of Shseer lies a symbolic execution engine which tracks the possible states of the shell program's execution.

- The executor maintains a compact, *symbolic* representation of the shell's possible states (§4.1). The state is transformed as commands are evaluated, producing a trace of states.
- Before evaluation, a command undergoes *expansion*: a set of rewriting rules process shell *words* into lists of strings. The engine gives symbolic treatment to this expansion phase (§4.2).
- Expanded commands are symbolically evaluated against the shell's state (§4.3).

4.1 Modeling shell state

As shown in Fig. 3 Shseer models shell state during symbolic execution with the tuple of twelve components. Key among these is the (global) mapping ρ_g from variable names to their (symbolic) values and the path condition p_c which tracks the condition that must be satisfiable for the execution path to be feasible.

Most of the treatment of shell state is standard [7] with some omissions (signals, job management) and a few additions to store symbolic information. The constraint language $\mathbb{C}$ of Shseer is guided by the constraints imposed by shell builtins such as `test` and `shift`. Note the distinction between shell variables (*e.g.*, $1, $name, $PATH ..) which are mutable and symbolic constants (*e.g.*, x,y,z ...) which are immutable. The shorthand $\sigma(\$vr)$ represents the current value of the shell variable $\$vr$ in the shell state σ. As denoted by the grammar, constraints are only expressed over symbolic constants and hence can be directly dispatched to the constraint solver.

As the shell state affects the semantics of the shell, Shseer makes certain assumptions to guide its analysis. The shell options *opt* are assumed to be initially empty. The global variable environment ρ_g is empty expect for the variables for field splitting, the home directory, and current/previous working directories. The current working directory s_{cwd} is assumed to be arbitrary. Similarly Shseer assumes that there are initially three open file descriptors: the standard input, output and error.

4.2 Word Expansion

Word expansion is an effectful operation that transforms a pair of command AST and model shell state to a new command AST and new shell state. It is mutually recursive with command evaluation–word expansion occurs before command evaluation while expansion can lead to evaluation due to command substitution.Expansion also depends on the filesystem state via tilde/pathname expansion.

Table 1. **Expansion examples.** Each example demonstrates a different case of word expansion with alternative start states. Inputs are one-line snippets, and the result state reflects the shell state after expansion. The final column shows the result of word expansion, assuming only the global environment.

Expansion case	Start state	Input	Result state	Expansion
Known value	any σ	`hello`	*unchanged*	`hello`
	σ w/ $\rho_g[\$\text{name} \mapsto \text{"john"}]$	`hi $name`	*unchanged*	`hi john`
	σ w/ $\rho_g[\$\text{name} \mapsto \text{""}]$	`hi ${name:-joe}`	*unchanged*	`hi joe`
Unknown variable	σ w/ $\$\text{name} \notin \rho_g$	`hi $name`	$\sigma + \mathit{fresh}\, x,\ \mathbb{C}$ += {$name = x}	`hi` x
	σ w/ $\rho_g[\$\text{name} \mapsto x]$	`hi $name`	*unchanged*	`hi` x
	σ w/ $\mathbb{C}$ and $\rho_g[\$\text{name} \mapsto x]$	`hi ${name:-joe}`	$\mathbb{C}$ += $\{x = \text{""} \implies y = \mathit{joe},\ x \neq \text{""} \implies y = x\}$	`hi` y
Complex	any σ	`$(cat a.txt)`	$\sigma + \mathit{fresh}\, x$	`hi` x
	σ w/ $\mathbb{C}$ and $\rho_g[\$\text{name} \mapsto x]$	`${#name}`	$\mathbb{C}$ += $\{z = \mathit{slen}(x)\}, \sigma + \mathit{fresh}\, z$	`$z`

As Shseer symbolically interprets scripts, it needs to expand words which contain variables with unknown values. In such cases, Shseer imposes constraints that track the effect of the expansion. As shown in Table 1 if in the word

$$expand : c \times \sigma \to c \times \sigma$$

$$\begin{array}{rrl} \text{ASTs/commands} & c & ::= \overline{(str = w)}\ w\ \bar{r} \mid c\& \mid c_1; c_2 \mid !c \\ & & \mid c_1 \&\& c_2 \mid c_1 || c_2 \mid \texttt{while}\ c_1\ c_2 \\ & & \mid \texttt{for}\ str\ w\ c \mid \texttt{if}\ c_1\ c_2\ c_3 \\ & & \mid \ldots \\ \text{Words} & w & ::= \overline{(s^+ \mid \_ \mid k \mid var)} \\ \text{Control codes} & k & ::= \texttt{\$\{}str\ \phi\texttt{\}} \mid \texttt{\$(}c\texttt{)} \mid \texttt{\$((}w\texttt{))} \mid \texttt{"}w\texttt{"} \mid \ldots \end{array}$$

Fig. 4. Expansion, and immediately relevant subset of shell syntax.

`${name:-joe}` the value of `$name` is unknown then Shseer expands the word to `y` and imposes constraints linking the two expressions.

Shseer performs symbolic word expansion which results in an AST containing a mixture of concrete strings and symbolic variables: a symbolic variable could be a word (Fig. 4). There are three main cases, and Table 1 illustrates examples from each case:

(1) **Fully expand concrete values.** When expanding (c, σ) if the (shell) command c only contains variables which have concrete values in σ and doesn't involve tilde/pathname expansion (depend on symbolic filesystem state) then Shseer's word expansion is equivalent to normal word expansion.

(2) **Partially expand unknown shell variables.** For command invocations containing variables with symbolic values or references to unknown shell variables, Shseer:
   (a) expands shell-variables mapped to symbolic variables to the referent (*e.g.*, `$name` to x), or a fresh one plus constraints depending on the type of reference Table 1.
   (b) A shell program may expect certain shell-variables such as *JAVA_HOME* to be exported by the local environment. Hence Shseer expands unknown shell-variables by mapping the identifier to a fresh symbolic constant.

(3) **Approximate complex expansions.** For commands containing command substitutions or pathname expansion, Shseer expands command substitutions by first recursively expanding the sub-command to obtain a new command and state, then replacing the command substitution with a fresh symbolic variable—approximating the subshell's possible output as any possible string—and expands path globs to fresh symbolic variables. Shseer generates SMT string constraints for some forms of parameter expansion such as string length (Table 1). For other cases such as expansion involving suffix/prefix patterns Shseer simply assumes the expansion result is arbitrary as such constraints can be hard for the SMT based constraint solver to solver while adding little analysis value. For example, in the case of the steam bug, knowing whether `$0` is empty is sufficient and adding substring constraints does not benefit analysis.

4.3 Command Evaluation

Parsing: Following word expansion, Shseer parses commands to identify the command and its arguments. For example, `rm -f "$1"` could expand to `rm -f foo.txt`. Shseer then needs to parse this result to identify the command (`rm`) being evaluated and its flags(`-f`) and arguments (`foo.txt`). However, even expanded commands might contain symbolic parts posing a challenge for parsing. For example `rm $1` could expand to `rm -f dir1` or `rm file1 file2` depending on the value of `\$1`. Analyzing all such possible combinations of flags and options leads to an exponential blowup making analysis intractable. Hence, Shseer assumes that symbolic words in a command represent a single word

(no field splitting) that is not an argument. Given `rm -r $1`, Shseer parses the `rm` command as having a single flag `-r` and a single argument `\$1`. By identifying the command, Shseer can then decide how to evaluate it.

Builtins: Builtins are evaluated directly by the shell. If a command is a builtin, Shseer interprets it, updating the symbolic state. For example to interpret the builtin command `set`:

```
set -- "name1" "name2"
```

Shseer updates the sequence of positional parameters to `name1 name2`. The value of `\$\#` which tracks the number of positional parameters is update to two. Builtin commands could also generate constraints: `set`:

```
test $1 = $2
```

The `test` command invocation succeeds and returns an exit code of 0 only if `$1` is equal to `$2`. If either of the values is unknown then the exit code of the command is symbolic: The exit code of this command ec is symbolic and given by the constraint: $ec = ITE(\sigma(\$1) = \sigma(\$2), 0, 1)$

Shell constructs: Similar to other languages the shell includes constructs for conditionals, loops and pattern matching. Such constructs result in a fork in execution paths. For example, an `if` statement splits the execution path in two–one along the `then` branch and one along the `else` branch. Copying the state and exploring both branches will lead to an exponential explosion in the number of execution paths making analysis intractable. Hence, Shseer adopts a (naive) copy and merge strategy. When the execution path splits, Shseer copies the full state, adds path constraints and explores both branches.

```
if test "$1" = "$2" ; then # Fork
    x=10
    echo "Following true branch"
else
    x=20
fi
```

For the if statement above, Shseer copies the shell state σ, imposing the path constraint $p_1 \iff (\$? == 0)$ along the true branch and $\neg p_1$ along the false branch where $? is the exit code of the last evaluated command which in this case is the conditional. After interpreting the body of the if statement, the two shell states σ_1, σ_2 are merged back into a single state $\sigma' = merge(\sigma_1, \sigma_2)$ where $\sigma'(x) = if(p_1, 10, 20)$. Similarly any added constraints (*e.g.*, equality between variables) must be guarded by the corresponding path condition. In general, the shell options ($opts$), function definitions (F), and the getopts argument offset ($optoff^?$) are approximated by picking the first branch, while variable values (ρ_g), constraints, the last exit code ($n_{\$?}$) and the value of the current working directory(s_{cwd}) are merged as described above.

Shseer uses the same strategy to handle loops and case statements. Shseer statically unrolls loop expressions to a small fixed bound while adopting smarter strategies if possible. For example, a common pattern is to loop over the `getopts` command to parse a script's flags and options. Shseer unrolls the loop until all possible flags (which are almost always known) are parsed.

The handling of other shell constructs is straightforward. Function invocations create a new local environment $\overline{\rho_l}$ and enforce a recursion limit to prevent blowups. Subshells create a copy of the shell state while ensuring that filesystem effects are persisted. As Shseer cannot reason about concurrency, pipelines and background nodes are interpreted sequentially.

Executables: Executable commands are external commands that are executed by the shell in a separate process and could have arbitrary behavior. Shseer currently only has a library of specifications for common filesystem utilities (`rm`,`mv`,`cp`,`mkdir`,`cat`,`touch`) that capture their behavior. A specification describes the behavior of a specific command invocation pattern that consists of the command name, flags/options and a list of arguments.

$$rm\ \$HOME/.bashrc \xRightarrow{match\ to\ abstract\ cmd\ inv} rm\ \ arg$$

Shseer matches the parsed command invocation against one of these specifications and enforces the relevant constraints. Command constraints are in the form of pre-postconditions in the style of Hoare logic and are described in detail in the following section.

Utilizing the constraint solver Shseer can then check if this filesystem mutation will/could fail or perform a dangerous mutation such as deleting a system directory. If Shseer cannot match a command invocation against a specification, Shseer errors on the side of underapproximation to be helpful and assumes the command to be side-effect free. Alternatively if a configuration flag is set, Shseer assumes the command can have arbitrary behaviour and discards all filesystem information it has learnt.

5 SYMBOLIC REASONING FOR THE FILE SYSTEM

Up to this point, consideration has been given to Shseer's reasoning about the shell's internal state alone. One of the key uses of shell programs, however, is to interact with the file system, usually through utilities like `rm` and `mkdir`; we now turn to the Shseer's treatment of such commands.

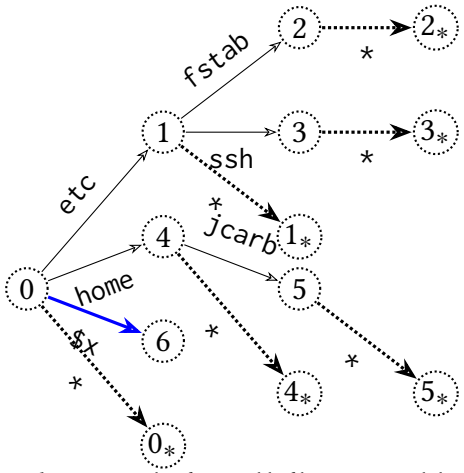


Fig. 5. **Filesystem graph example.** An example of a possible filesystem graph beginning at the filesystem root.

The state of the file system may be regarded as finite graph of *locations* with labelled edges. Each location has a *value* which may be a *file*, *directory* or *absent*. For example, Fig. 5 visualizes a file system whose root directory, denoted ⓪, has two children: the `etc` and `home` directories, denoted ① and ④, respectively. The `etc` directory contains only a file ② named `fstab` and a directory ③ named `ssh`. While we use integer values to denote a location, they should be understood as an uninterpreted sort [3] rather than integers as it possibly for two locations to be equal as we shall

see shortly. Note the *value* of a location is not attached a location in the graph as locations are constant but values change. For example, the file `/etc/fstab` may be deleted and a directory may be created at the same location. The filesystem itself can be thought of simply as a mutable map M from locations to values which has an initial state M_0 that is updated by effects to produce new maps at $M_1 ... M_n$. We use the notation $\textcircled{m}_i$ to refer to the value of the location $\textcircled{m}$ in the map M_i.

Paths and resolution: Commands such as `mkdir` and `rm` mutate the file system at location(s) specified by *paths*, which are `/`-separated sequences of names (*e.g.*, `/home/me`). Paths with a leading `/` are *absolute*; all others are *relative* and resolved from the current working directory (s_{cwd} in the shell state). For now, we only treat absolute paths; relative paths will be discussed in §5.3.

Symbolic resolution: The key insight of Shseer is developing a symbolic resolution procedure to map symbolic pathnames to locations in the filesystem graph. *Symbolic resolution* of an absolute path begins at the root ⓪, from which the *cursor* recursively traverses the tree according to the names. For each intermediate name (all but the last), the cursor must be on a directory; if the corresponding name is absent or file, then resolution fails.

To resolve the path `/home/me` on the tree in Fig. 5 Shseer traverses from ⓪ to ④ and then to the location ⑤ by following the edges with labels `home` and `me` The sequence of intermediate locations (⓪ and ④) we call the resolution *footprint* and the final location (⑤) we call the *target*. The names `".."`,`"."` refer to the parent directory and the current directory respectively. The possibility of such *non-linear* paths means that a single target location may be described by many paths—in fact, infinitely many. Non-linear references are represented as normal edges with the edge labels `".."` and `"."` but are omitted from Fig. 5 for the sake of brevity.

What makes the example in Fig. 5 a graph and not a tree? Pathnames that contain symbolic values such as `/$x` result in symbolic edge names. The symbolic value `$x` could be empty (in which case ⓪ = ⑥), refer to a single path component (*e.g.*, `$x` = bin) or multiple path components (*e.g.*, `$x` = usr/local). A symbolic value could also include non-linear components (*e.g.*, `$x` = ../etc). Hence, while edges with constant values point to children of the source location, edges with symbolic values (denoted in blue) could point to *location* any arbitrary location. Thus, edges with constant values are bidirectional (via `..`) edges with symbolic values are not. Resolving `..` from ⑥ would require creating a fresh location that represents the *target* of the path `/$x/..` It is often required to express a cofinite constraint which is where $*$ edges ((dashed edges) come in. The location $\textcircled{0_*}$ represents all other *unnamed* children of ⓪. Hence if $\textcircled{0_*}$ is marked absent, then the root has exactly two children ① and ④. Afterwards, whenever $\textcircled{m}$ is newly introduced with an explicit name as a child of $\textcircled{n}$, it *inherits* the state of $\textcircled{n}$. Precisely,

$$\left(\textcircled{m}_j \neq \text{abst}\right) \vee \left(\textcircled{m}_j = \textcircled{n}_j\right)$$

5.1 Specifying Constraints

We can now specify constraints over the filesystem with the machinery developed. Given path resolution, we can consider a set of predicates to answer questions about the shape of the file tree. In particular, given a path, one might consider if the path successfully resolves to a location or not, and if the location is a file, a directory, or absent. We write $p \twoheadrightarrow \alpha$ to say that the path p successfully resolves to a location in the tree of the shape α, where α is one of file, dirc, or abst. For convenience, we write $p \twoheadrightarrow \neg\text{abst}$ to mean $p \twoheadrightarrow \text{file} \vee p \twoheadrightarrow \text{dirc}$, *i.e.*, that the path resolves to a location that is

`Val`	v	::=	.. | . | $*$
`Path`	p	::=	/ | p/v
`PathExpr`	e	::=	/ | e/e | base e | v
`Constraint`	α	::=	file | dirc | abst
`Prop`	ϕ	::=	$e \twoheadrightarrow \alpha$
`Action`	ϵ	::=	noop | $\epsilon;\epsilon$ | mkfile e
		|	mkdirc e | mkabst e | dup e e
`Spec`	Φ	::=	$\phi \Longrightarrow \epsilon \mid \Phi \vee (\phi \Longrightarrow \epsilon)$

Fig. 6. Command specification grammar.

not absent. For instance, for the tree of Fig. 5 we can express constraints:

$$\texttt{/home/jcarb} \twoheadrightarrow \text{abst}$$

$$\texttt{/etc/fstab} \twoheadrightarrow \text{file}$$

$$\neg\,(\texttt{/home/jcarb} \twoheadrightarrow \text{dirc})$$

$$\neg\,(\texttt{/usr/bin} \twoheadrightarrow \text{dirc})$$

Effects: We also introduce a set of effects that describe mutations to specific locations in the tree. These locations are, of course, specified by paths: mkfile p, mkdirc p, and mkabst p each resolve p and mutate the corresponding location to be a file, to be an empty directory, or to be absent, respectively. Lastly, dup p_1 p_2 recursively duplicates the state of the location resolved from p_1 to the location of p_2. As Shseer does not reason about the contents of filesystem entries, copying a file (non recursive copy) can be represented using the existing actions.

5.2 Specifying command behavior

Consider the invocation `mkdir /home/jcarb`. The invocation requires that `/home/jcarb` resolve to an absent location in order to succeed: `/home/jcarb` $\twoheadrightarrow$ abst. If so, it introduces an empty directory at that location: mkdirc `/home/jcarb`. Otherwise, the invocation fails. How is this pre-condition answered and the effect modelled for the symbolic tree?

Pre-condition. As in the concrete setting, the path is first be resolved to its corresponding location. *Symbolic resolution* of the path traverses from the root ⓪ along `home` to ④, noting that both must be of the shape dirc for the resolution to succeed. Let $\phi_{\texttt{pre}}$ denote this pre-condition:

$$\phi_{\texttt{pre}} \triangleq \big(⓪_0 = ①_0 = \text{dirc}\big) \wedge \big(⑤_0 = \text{abst}\big)$$

Effect. If the pre-condition holds, the target location is made an empty directory at time `T1`, *i.e.*, ⑤ is made dirc where its $*$ child is abst. This child is also introduced into the tree as ⑤$_*$. More precisely,

$$\begin{aligned}\phi_{\texttt{post}} \triangleq\ & \big(⑤_1 = \text{dirc}\big) \wedge \big(⑤_{*2} = \text{abst}\big) \wedge \\ & \text{maintain}_0^1\{⑤\} \wedge \text{maintain}_1^2\{⑤_*\} \\ & \text{maintain}_i^j\{ⓜ\} \triangleq \forall ⓛ \in \text{Loc}, ⓛ \neq ⓜ \implies ⓛ_j = ⓛ_i\end{aligned}$$

Readers familiar with SMT will notice that semantics of the location-value map M described here are precisely that of the SMT theory of arrays. On the other hand, if the pre-condition fails, the state is left unchanged and the corresponding

$$
\begin{array}{rcl}
[\![\texttt{cat}\ p]\!] & \triangleq & p \twoheadrightarrow \text{file} \implies \text{noop} \\
[\![\texttt{touch}\ p]\!] & \triangleq & \bigvee \left\{ \begin{array}{lcl} p \twoheadrightarrow \text{abst} & \implies & \text{mkfile}\, p \\ p \twoheadrightarrow \neg\text{abst} & \implies & \text{noop} \end{array} \right\} \\
[\![\texttt{rm}\ p]\!] & \triangleq & p \twoheadrightarrow \text{file} \implies \text{mkabst}\, p \\
[\![\texttt{rm -r}\ p]\!] & \triangleq & [\![\texttt{rm}\ p]\!] \vee (p \twoheadrightarrow \text{dirc} \implies \text{mkabst}\, p) \\
[\![\texttt{rm -dir}\ p]\!] & \triangleq & [\![\texttt{rm}\ p]\!] \vee (p \twoheadrightarrow \text{emp} \implies \text{mkabst}\, p) \\
[\![\texttt{cp}\ p_1\ p_2]\!] & \triangleq & \bigvee \left\{ \begin{array}{lcl} p_1 \twoheadrightarrow \text{file} \wedge p_2 \twoheadrightarrow \text{dirc} & \implies & \text{dup}\, p_1\ (p_2 / \text{base}\, p_1) \\ p_1 \twoheadrightarrow \text{file} \wedge p_2 \twoheadrightarrow \neg\text{dirc} & \implies & \text{dup}\, p_1\, p_2 \end{array} \right\} \\
[\![\texttt{cp -r}\ p_1\ p_2]\!] & \triangleq & \bigvee \left([\![\texttt{cp}\ p_1\ p_2]\!] \cup \left\{ \begin{array}{lcl} p_1 \twoheadrightarrow \text{dirc} \wedge p_2 \twoheadrightarrow \text{dirc} & \implies & \text{dup}\, p_1\ (p_2 / \text{base}\, p_1) \\ p_1 \twoheadrightarrow \text{dirc} \wedge p_2 \twoheadrightarrow \text{abst} & \implies & \text{dup}\, p_1\, p_2 \end{array} \right\} \right)
\end{array}
$$

Fig. 7. **Specifications for common coreutils commands.** Commands specifications are manually written by domain experts and capture the possible effects of each command invocation on the file tree.

shell exit code of the command is set to 1 to indicate command failure. This is ideal for two reasons. First, filesystem effect failures might be handled within the shell script and throwing an error in such a case would be noisy. Second, a shell command such as `rm foo boo goo` is mapped to a *sequence* of effects: `mkabst foo ; mkabst boo ; mkabst goo`. If `foo` does not exist but `boo`,`goo` exist then while the command fails `boo`,`goo` are deleted. Rather than prematurely failing (by asserting false), it is better to accurately accumulate constraints over the filesystem state and then query the constraint solver on the feasibility of various failures.

Altogether. Thus, the invocation `mkdir /home/jcarb` is interpreted to impose the following pair of conditional constraints on the nodes of the symbolic tree where *ec* is the exit code of the shell command:

$$\phi_{\texttt{pre}} \rightarrow \phi_{\texttt{post}}$$

$$\neg\phi_{\texttt{pre}} \rightarrow \text{noop}$$

$$ec = ITE(\phi_{\text{pre}}, 0, 1)$$

If a shell command represents the sequencing on multiple effects, then the exit code is zero if all of them succeed. More generally, for any symbolic path p, we write for the specification of `mkdir` p:

$$[\![\texttt{mkdir}\ p]\!] \triangleq \underbrace{p \twoheadrightarrow \text{abst}}_{\text{pre-condition}} \implies \underbrace{\text{mkdirc}\, p}_{\text{effect}}$$

More generally, $\phi \implies \epsilon$ describes a single behavior for a pre-condition ϕ and an effect ϵ. Then, an invocation's specification, denoted Φ, consists of the disjunction of such possible behaviors: Thus, the full specification is given by:

$$suc \triangleq \vee_i^k\ (\phi_i \implies \epsilon_i)$$

$$ec = ITE(suc, 0, 1)$$

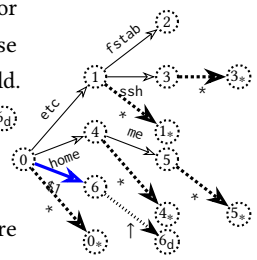
5.3 Complexities: non-linear and relative paths

In the previous sections, we neglected to demonstrate how symbolic resolution behaves when faced with paths that contain parent references or are not absolute.

Parent references: What should happen if a parent reference `..` is encountered during symbolic resolution? For a path such as `/home/../etc`, the cursor traverses along the concrete name `home` to reach ④; it may simply move back to its parent ⓪ after constraining ④ to be a directory. But if the cursor's node is along a symbolic name—consider the

path `/$1/..`—then the parent *node* does not necessarily correspond to the parent *location*; the cursor cannot simply move from ⑥ back to ⓪.

Thus, each node is associated with a *directory node* representing the parent location. For nodes along concrete names, the directory node is the same as the parent; but for those along symbolic names, it is a separate node reachable only by resolving `..` from the child. For instance, symbolic resolution of the path `/$1/..` on §5.3, would introduce a node ⑥d for the directory node of ⑥:



Relative paths: By initiating resolution at the root, we have assumed that all paths are *absolute*. In practice, programs often work with *relative* paths that should be resolved relative to the *current working directory* (CWD). In the case that the current working directory is known, a relative path can be made absolute by prepending it. More often, it is not known ahead-of-time. Even if directory from which Shseer is invoked is assumed to be the working directory, the working directory is often changed by the script using the `cd` builtin and hence ends up being symbolic. The current working directory may therefore be treated as a symbolic variable—in fact, `$PWD` is already standard in the POSIX shell. As `$PWD` is usually absolute (barring developer mischief), relative paths can be made absolute by prepending this variable.

…or possibly so: More troublesome is a path that begins with a symbolic variable, such as `$1` (note the absence of a leading slash). Since it is unknown *a priori* if the value of `$1` begins with a slash, it cannot be known if the path is absolute or relative.

The solution is to introduce a fresh variable, say `$X`, whose value is dependent on `$1`. If the first character of `$1` is a slash, `$X` is empty; otherwise, `$X` is `$PWD`:

$$((\texttt{\$1}[0] = \text{`/'}) \rightarrow (\texttt{\$X} = \text{``''}))$$
$$\wedge\ ((\texttt{\$1}[0] \neq \text{`/'}) \rightarrow (\texttt{\$X} = \texttt{\$PWD}))$$

Then, the path `$1` is treated (in the normal way) as `/$X/$1`.

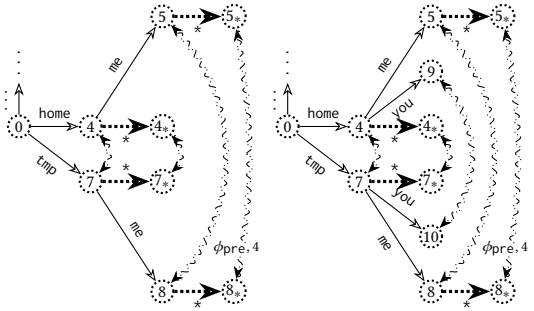
5.4 Complexities: glob patterns

As discussed in §4, paths containing glob patterns are expanded into a list of names matching the pattern. Paths with glob patterns are also symbolic because the names against which to match is not known ahead-of-time.

Consider `rm /etc/*`, which tries to delete all the entries in the `etc` directory ①. Since `rm` is invoked without the `-r` flag, the invocation fails if any of the entries are directories. In the symbolic setting, `rm /etc/*` does not add any new nodes but imposes new constraints on their states. In particular, because `*` means all the entries in the `etc` directory, we would like to impose constraints over the finite collection Δ of ②, ③, ①*, corresponding to the `fstab` entry, the `ssh` entry, and any other unnamed entries. Note that in general, Δ is a superset of the actual expansion set, since some of its members may be absent. The constraints imposed on Δ take this into consideration.

Firstly, the glob pattern must be able to expand, *i.e.*, at least one of these nodes must not be absent:

$$\begin{aligned}\phi_1 &\triangleq \exists ⓝ \in \Delta, ⓝ_2 \neq \text{abst} \\ &\equiv (②_2 \neq \text{abst}) \vee (④_2 \neq \text{abst}) \vee (①*_2 \neq \text{abst})\end{aligned}$$



(a) (b)

Fig. 8

Secondly, the non-absent entries must be files, since `rm` is invoked without `-r`:

$$\phi_2 \triangleq \forall ⓝ \in \Delta, (ⓝ_2 \neq \mathsf{abst} \rightarrow ⓝ_2 = \mathsf{file}) \tag{5.1}$$

Of course, the path `/etc` must also be resolvable:

$$\phi_3 \triangleq \left(⓪_2 = ①_2 = \mathsf{dirc}\right)$$

Again writing $\phi_{\texttt{pre}}, \phi_{\texttt{post}}$ for the pre-condition and postcondition,

$$\phi_{\texttt{pre}} \triangleq \phi_1 \wedge \phi_2 \wedge \phi_3$$
$$\phi_{\texttt{post}} \rightarrow (\forall ⓝ \in \Delta, ⓝ \rightarrow \mathsf{abst}\ )$$

Note that the quantifier can be eliminated as Δ is a known finite set.

5.5 Complexities: recursive copies

The hierarchical nature of the file system also introduces some complexity to symbolically model the copying of directories. Consider the invocation `cp -r /home /tmp`. This invocation is interpreted per Fig. 7 into four sets of conditional constraints on the nodes of the symbolic tree. We will not fully detail each of these—they follow the pattern illustrated throughout this section—but instead focus on the effect dup `/home` `/tmp`.

As usual, a new ⑦ is introduced for the `tmp` child of the root ⓪. Then, the state of `home` ④ is copied to ⑦ for time `T4`. It proceeds recursively, copying the states of the ④'s children ⑤ and ⑦* to corresponding (fresh) child nodes ⑧ and ⑦*, with the same names `me` and `*`, respectively. Recursing again copies the state of ⑤'s child ⑤* to a corresponding descendant ⑧*:

$$⑦_4 = ④_4 \quad ⑧_4 = ⑤_4 \quad ⑦*_4 = ④*_4 \quad ⑧*_4 = ⑤*_4.$$

However, in general, it must do more. Specifically, any new information discovered about the original locations may also apply to the copies. For instance, if the symbolic execution determines that the `home` directory may contain another entry, say `you`, then the `tmp` directory should also have such an entry—and vice-versa.

To handle this complexity, we make one direction of this relationship explicit at the node-state level by way of a binary relation of *unidirectional ties*, denoted $ⓝ_j \rightsquigarrow ⓝ'_j$, atop the symbolic tree. At the constraint level, each tie is an

implication: if the property P applies to $\textcircled{n}_j$, then it applies to $\textcircled{n'}_j$: $P(\textcircled{n}_j) \to P(\textcircled{n'}_j)$. The symmetric *bidirectional tie* $\textcircled{n}_j \leftrightsquigarrow \textcircled{n'}_j$ consists of a unidirectional tie $\textcircled{n}_j \rightsquigarrow \textcircled{n'}_j$ and its converse $\textcircled{n'}_j \rightsquigarrow \textcircled{n}_j$. Whenever a child is introduced to $\textcircled{n}$, a corresponding child is introduced for all the other nodes to which $\textcircled{n}$ may be tied. These children are then tied in the same manner as their parents. This is performed across the recursive-transitive closure of unidirectional ties.

Returning to our example, the effect dup `/home` `/tmp` introduces (conditionally on some $\phi_{\texttt{pre}}$) bidirectional ties, as shown in Fig. 8a, for each pair of corresponding original and copied nodes:

$$\textcircled{7}_4 \leftrightsquigarrow \textcircled{4}_4 \quad \textcircled{8}_4 \leftrightsquigarrow \textcircled{5}_4 \quad \textcircled{7}_{*4} \leftrightsquigarrow \textcircled{4}_{*4} \quad \textcircled{8}_{*4} \leftrightsquigarrow \textcircled{5}_{*4}.$$

Note that, in fact, these ties subsume the equalities given above. Now, suppose that after the copy comes an invocation `test -f /home/you`. When a fresh ⑨ for the `you` child of `home` ④ is introduced, as in Fig. 8b (still at time `T4`), since there exists (conditionally on $\phi_{\texttt{pre}}$) a bidirectional tie $\textcircled{7}_4 \leftrightsquigarrow \textcircled{4}_4$, a corresponding child ⑩ is introduced to ⑦ with a bidirectional tie $\textcircled{10}_4 \leftrightsquigarrow \textcircled{9}_4$ on the *same* condition.

REFERENCES

[1] [n. d.]. "https://pubs.opengroup.org/onlinepubs/9799919799.2024edition/".

[2] [n. d.]. "https://github.com/binpash/libdash/tree/master".

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2025. *The SMT-LIB Standard: Version 2.7.* Technical Report. Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

[4] Ting Dai, Alexei Karve, Grzegorz Koper, and Sai Zeng. 2020. Automatically detecting risky scripts in infrastructure code. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 358–371. https://doi.org/10.1145/3419111.3421303

[5] Yiwen Dong, Zheyang Li, Yongqiang Tian, Chengnian Sun, Michael W. Godfrey, and Meiyappan Nagappan. 2023. Bash in the Wild: Language Usage, Code Smells, and Bugs. *ACM Trans. Softw. Eng. Methodol.* 32, 1 (2023), 8:1–8:22. https://doi.org/10.1145/3517193

[6] GitHub. 2024. Octoverse: AI leads Python to top language as the number of global developers surges - The GitHub Blog. https://github.blog/news-insights/octoverse/octoverse-2024/.

[7] Michael Greenberg and Austin J. Blatt. 2019. Executable formal semantics for the POSIX shell. *Proc. ACM Program. Lang.* 4, POPL, Article 43 (Dec. 2019), 30 pages. https://doi.org/10.1145/3371111

[8] Lukas Lazarek, Seong-Heon Jung, Evangelos Lamprou, Zekai Li, Anirudh Narsipur, Eric Zhao, Michael Greenberg, Konstantinos Kallas, Konstantinos Mamouras, and Nikos Vasilakis. [n. d.]. From Ahead-of- to Just-in-Time and Back Again: Static Analysis for Unix Shell Programs *(HOTOS '25)*. https://nikos.vasilak.is/p/sash:hotos:2025.pdf

[9] Karl Mazurak and Steve Zdancewic. 2007. ABASH: finding bugs in bash scripts. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security (PLAS '07)*. Association for Computing Machinery, New York, NY, USA, 105–114. https://doi.org/10.1145/1255329.1255347

[10] MrMEEE. 2011. install script does rm -rf /usr for ubuntu. https://github.com/MrMEEE/bumblebee-Old-and-abbandoned/issues/123. Accessed: 2025-03-06.

[11] Anirudh Narsipur. [n. d.]. Towards Automated Reasoning For Shell Programs. https://cs.brown.edu/media/filer_public/0b/c7/0bc731f0-fe4a-4fe8-a23c-60764f7298cf/narsipuranirudh.pdf.

[12] Shaun Nichols. 2015. Scary Code of the Week: Steam cleans Linux PCs. *The Register* (17 January 2015). https://www.theregister.com/2015/01/17/scary_code_of_the_week_steam_cleans_linux_pcs/ Accessed: 2025-03-06.

[13] Valve Software. 2023. Moved /.local/share/steam. Ran steam. It deleted everything on system owned by user. https://github.com/ValveSoftware/steam-for-linux/issues/3671 Accessed: 2023-10-04.

[14] Wolf. 2024. ShellCheck. https://github.com/CICDToolbox/shellcheck.