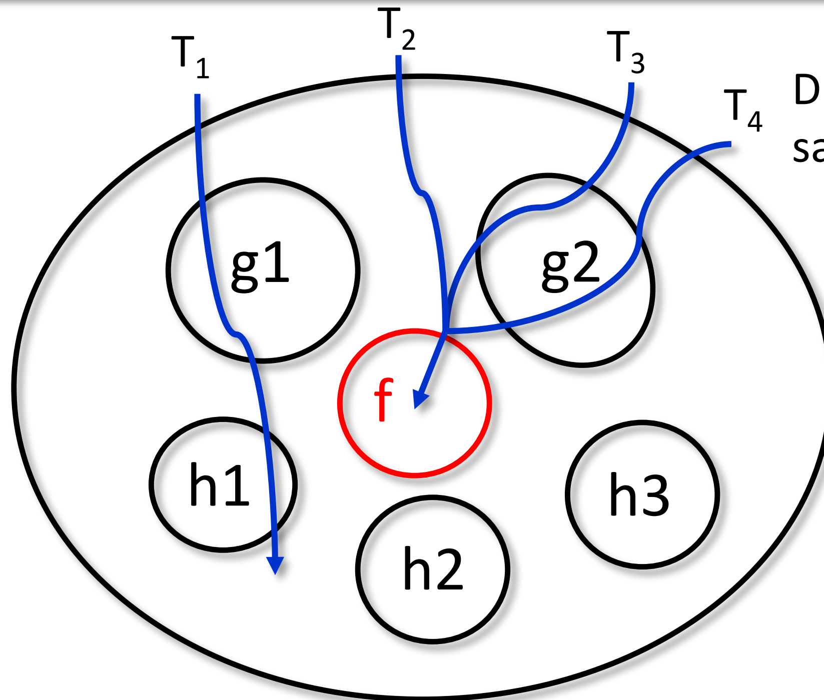
Automated Test Generation in System-Level

Pros
+ Can be easy to generate system TCs due to clear interface specification
+ No false alarm (i.e., no assert violation caused by infeasible execution scenario)

Cons
- **Low controllability of each unit**
- **Large and complex search space to explore in a limited time**
- **Hard to detect bugs in corner cases**

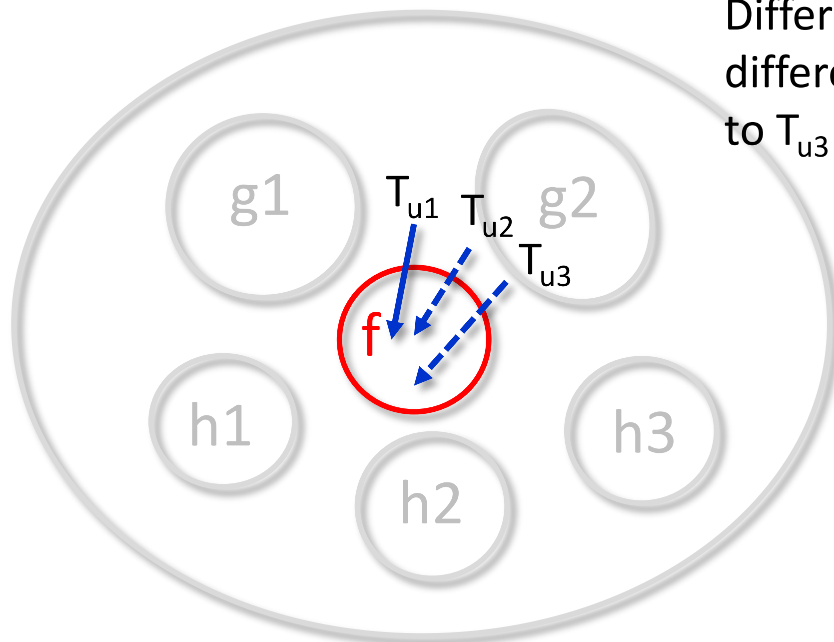


Different system tests T_2 to T_4 exercise the same behavior of the target unit

Automated Test Generation in Unit-Level

Pros
+ High controllability of a target unit
+ Smaller search space to explore than system testing
+ High effectiveness for detecting corner cases bugs

Cons
- Hard to write down accurate unit test drivers/stubs due to unclear unit specification
- High false/true alarm ratio



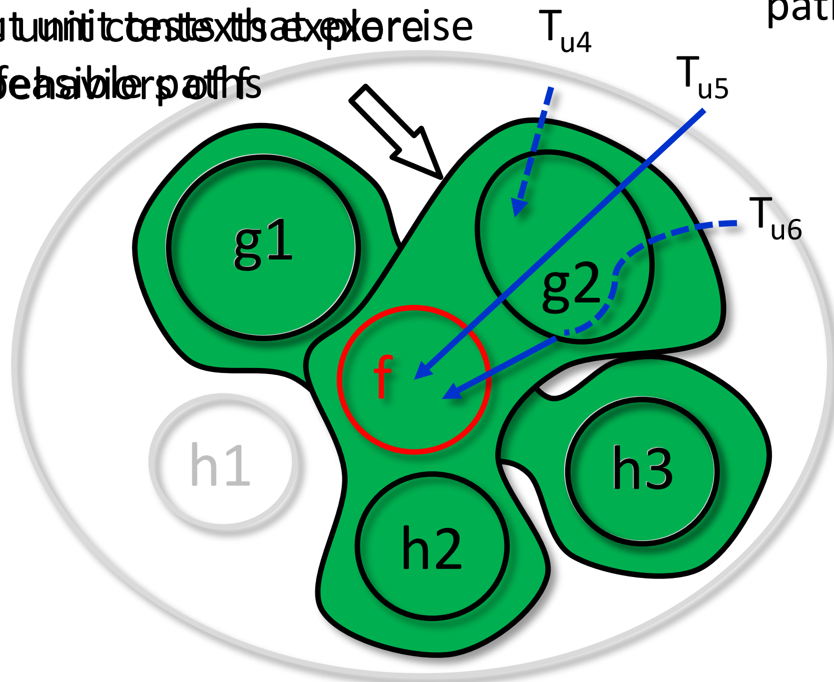
Different unit tests T_{u1} to T_{u3} directly exercise different behaviors of the target unit, but T_{u2} to T_{u3} exercise infeasible paths

Legend
A feasible execution
An infeasible execution

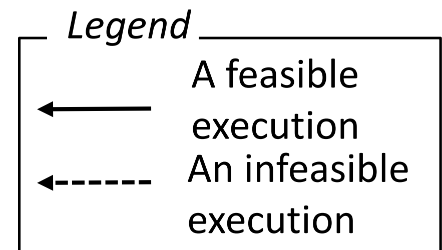
Automated Unit Test Generation with Realistic Unit Context

Pros
+ High controllability of a target unit
+ High effectiveness for detecting corner cases bugs
+ Low false alarm ratio

Realistic unit context filters out unit tests that exercise infeasible paths



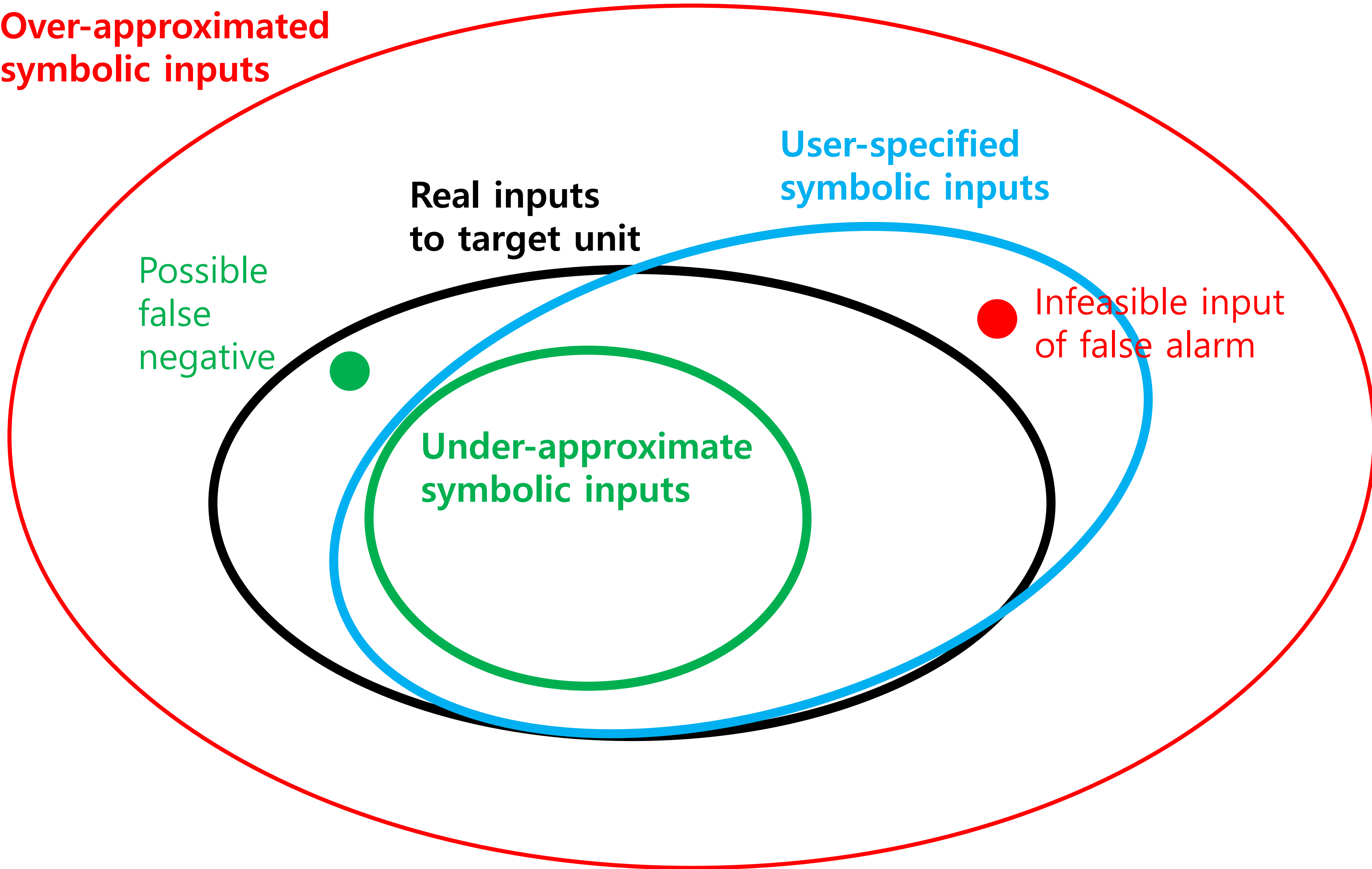
Unit tests T_{u4} that exercises an infeasible path is filtered out by the unit context



Related Work on Automated Unit Testing

	Bug detection ability	False/True alarm ratio	Target languages
Function input generation [PLDI 05][FSE 05][EMSOFT 06][TAP 08][ISSTA 08][SEC 15]	**High**	**High**	**Procedural or OO languages**
Method-sequence generation [ICSE 07] [ICST 10][FSE 11] [ICSE 13]	**High**	Medium	**Object-oriented languages**
Capture system tests to generate unit tests [TSE 09] [STTT 09][ISSTA 10]	**Low**	**Low**	**Object-oriented languages**
Automated Unit Test Generation with Realistic Unit Context	**High** (86.7% of target bugs in SIR and SPEC2006)	**Low** (2.4 false alarms per one true alarm)	**Procedural languages**

Approximate Input Space for Symbolic Unit Testing



Unit-testing Busybox ls

- We performed unit-testing Busybox ls by using CREST
  - We tested 14 functions of Busybox ls (1100 lines long)
  - Note that this is a refined testing activity compared to the previous testing activity for 10 Busybox utilities in a system-level testing

Busybox "ls" Requirement Specification

- POSIX specification (IEEE Std 1003.1, 2004 ed.) is a good requirement specification document for ls
  - A4 ~10 page description for all options
- We defined test oracles using **`assert`** statements based on the POSIX specification
  - Automated oracle approach
- However, it still required human expertise on Busybox ls code to define concrete assert statements from given high-level requirements

Copyright © 2001-2004 The IEEE and The Open Group, All Rights reserved.

NAME

ls - list directory contents

SYNOPSIS

[XSI] `ls [-CFRacdilqrtu1][-H | -L ]`[XSI]`[-fgmnopsx]`[XSI]`[`*file...*`]`

DESCRIPTION

For each operand that names a file of a type other than directory or symbolic link to a directory, *ls* shall write the name of the file as well as any requested, associated information. For each operand that names a file of type directory, *ls* shall write the names of files contained within the directory as well as any requested, associated information. If one of the **-d**, **-F**, or **-l** options are specified, and one of the **-H** or **-L** options are not specified, for each operand that names a file of type symbolic link to a directory, *ls* shall write the name of the file as well as any requested, associated information. If none of the **-d**, **-F**, or **-l** options are specified, or the **-H** or **-L** options are specified, for each operand that names a file of type symbolic link to a directory, *ls* shall write the names of files contained within the directory as well as any requested, associated information.

If no operands are specified, *ls* shall write the contents of the current directory. If more than one operand is specified, *ls* shall write non-directory operands first; it shall sort directory and non-directory operands separately according to the collating sequence in the current locale.

The *ls* utility shall detect infinite loops; that is, entering a previously visited directory that is an ancestor of the last file encountered. When it detects an infinite loop, *ls* shall write a diagnostic message to standard error and shall either recover its position in the hierarchy or terminate.

OPTIONS

The *ls* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-2001, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

-C Write multi-text-column output with entries sorted down the columns, according to the collating sequence. The number of text columns and the column separator characters are unspecified, but should be adapted to the nature of the output device.

-F Do not follow symbolic links named as operands unless the **-H** or **-L** options are specified. Write a slash (`'/'`) immediately after each pathname that is a directory, an asterisk (`'*'`) after each that is executable, a vertical bar (`'|'`) after each that is a FIFO, and an at sign (`'@'`) after each that is a symbolic link. For other file types, other symbols may be written.

Four Bugs Detected

1. Missing '@' symbol for a symbolic link file with –F option
2. Missing space between adjacent two columns with –i or –b options
3. The order of options is ignored
   - According to the ls specification, the last option should have a higher priority (i.e., -C -1 and -1 -C are different)
4. Option –n does not show files in a long format
   - -n enforces to list files in a long format and print numeric UID and GID instead of user/group name

Missing '@' symbol for symbolic link with –F option

- Busybox ls does not print a type marker '@' after a symbolic link file name, when -F is specified and a file name is specified in the command line.
  1. Output of linux ls:

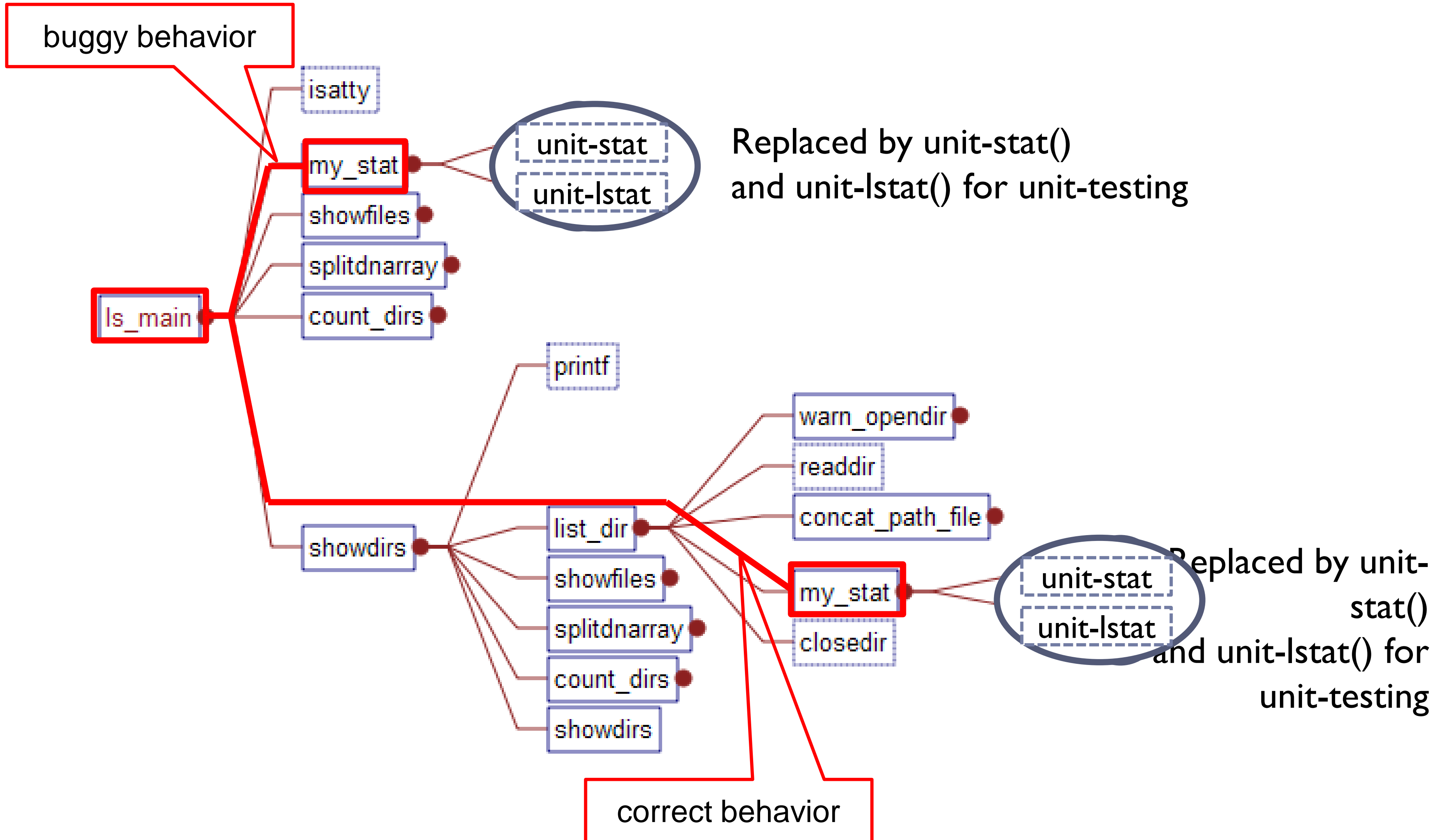
     ```
     $ ls -F t.lnk
     t.lnk@
     ```

  2. Output of Busybox ls (incorrect behavior):

     ```
     $ ./busybox ls -F t.lnk
     t.lnk
     ```

  - -F means write a marker (/*|@=) for different type of files.
  - `t.lnk` is a symbolic link, which links to file `t` in the directory ~yang/
- We found that the bug was due to the violation of a precondition of `my_stat()`

Calls Graph of Busybox ls



ls_main

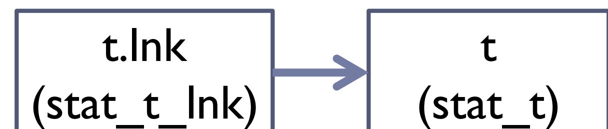
```
int ls_main(int argc UNUSED_PARAM,
            char **argv)
```

- Purpose:
  1. `ls_main` obtains specified options and file/dir names from command line input.
  2. Calculate runtime features by specified options.
     1) `unsigned int` **`opt`** is defined to represent all options given in the command line arguments
        - `opt = getopt32(argv, ls_options, &tabstops, &terminal_width)`
     2) `unsigned int` **`all_fmt`** is defined to represent all runtime features based on the given command line options and compile time option **`opt_flags`**`[]`
        - `all_fmt=`fun (`opt, opt_flags`)
  3. Print file/dir entries of a file system whose names and corresponding options are given in the command line input by calling sub-functions

my_stat

```
static struct dnode *my_stat(const char
  *fullname, const char *name, int force_follow)
```

- Purpose:
  1. `my_stat` gets file status by `fullname`, and store file status in `struct dnode *cur` which is returned by `my_stat`
  2. If a file/dir entry corresponding to `fullname` is available in the file system, `cur->stat` should stores the corresponding file info. Otherwise, NULL is turned.
- Pre condition:
  1. `len (fullname) >= len(name)`
  2. If any of -d, -F, or -l options is given, and -L option is not given, `follow_symlink` should be false
     - `((-d || -F || -l) && !-L) -> !follow_symlink`
     - -d: llist directory entries instead of contents, and do not dereference symbolic links
     - -F: append indicator (one of */=>@|) to entries
     - -l: use a long listing format
     - -L: when showing file information for a symbolic link, show information for the file the link references rather than for the link itself



my_stat

```
static struct dnode *my_stat(const char
*fullname, const char *name, int force_follow)
```

- Post condition:
    1. When `fullname` is a real file name, the following condition should be satisfied:

       `(cur!=NULL && cur->fullname==fullname && cur->name==name)`

```
struct dnode {
  const char *name;
  const char *fullname;
  /* point at the next node */
  struct dnode *next;
  smallint fname_allocated;
  /* the file stat info */
  struct stat dstat;
}
```

```
struct stat {
dev_t     st_dev;     /* ID of device containing file */
ino_t     st_ino;     /* inode number */
mode_t    st_mode;    /* protection */
nlink_t   st_nlink;   /* number of hard links */
uid_t     st_uid;     /* user ID of owner */
gid_t     st_gid;     /* group ID of owner */
dev_t     st_rdev;    /* device ID (if special file) */
off_t     st_size;    /* total size, in bytes */
blksize_t st_blksize; /* blocksize for filesystem I/O */
blkcnt_t  st_blocks;  /* number of blocks allocated */
time_t    st_atime;   /* time of last access */
time_t    st_mtime;   /* time of last modification */
time_t    st_ctime;   /* time of last status change */};
```

Assertions in `my_stat`

```
1.  static struct dnode *my_stat(const char *fullname, const char
    *name, int force_follow)
2.  {
3.  #ifdef ASSERTION
4.  assert(strlen(fullname) >= strlen(name));
5.  #endif
6.      struct stat dstat;
7.      struct dnode *cur;
8.      IF_SELINUX(security_context_t sid = NULL;)
9.  #ifdef ASSERTION
10. /* If any of -d, -F, or -l options is given, and -L
11.  * option is not given, ls should print out the status
12.  * of the symbolic link file. I.e.,
13.  * ((d || F || l) && !L) -> !FOLLOW_SYM_LNK
14.  */
15. unsigned char follow_symlink =
16.             (all_fmt & FOLLOW_LINKS) || force_follow;
17. assert(!((opt_mask[2] || opt_mask[17] || opt_mask[4])
18.           && !opt_mask[19]) || !follow_symlink);
19. #endif
20.     if (follow_symlink) { /*get file stat of link itself*/
21. //......
22. #if !CREST
23.         if (stat(fullname, &dstat))
24. #else
25.         if (unit_stat(fullname, &dstat))
26. #endif
27.         {
28.             bb_simple_perror_msg(fullname);
29.             exit_code = EXIT_FAILURE;
30.             return 0;
31.         }
32.     } else { /*get file stat of real file which sym_lnk linked to*/
33. //......
34. #if !CREST
35.         if (lstat(fullname, &dstat))
36. #else
37.         if (unit_lstat(fullname, &dstat))
38. #endif
39.         {
40.             bb_simple_perror_msg(fullname);
41.             exit_code = EXIT_FAILURE;
42.             return 0;
43.         }
44.     }
44.     cur = xmalloc(sizeof(*cur));
45.     cur->fullname = fullname;
46.     cur->name = name;
47.     cur->dstat = dstat;
48.     IF_SELINUX(cur->sid = sid;)
49. #ifdef ASSERTION
50. assert(strcmp(fullname, cur->fullname)==0);
51. assert(strcmp(name, cur->name)==0);
52. #endif
53.     return cur;
54. }
```

Symbolic Environment Setting

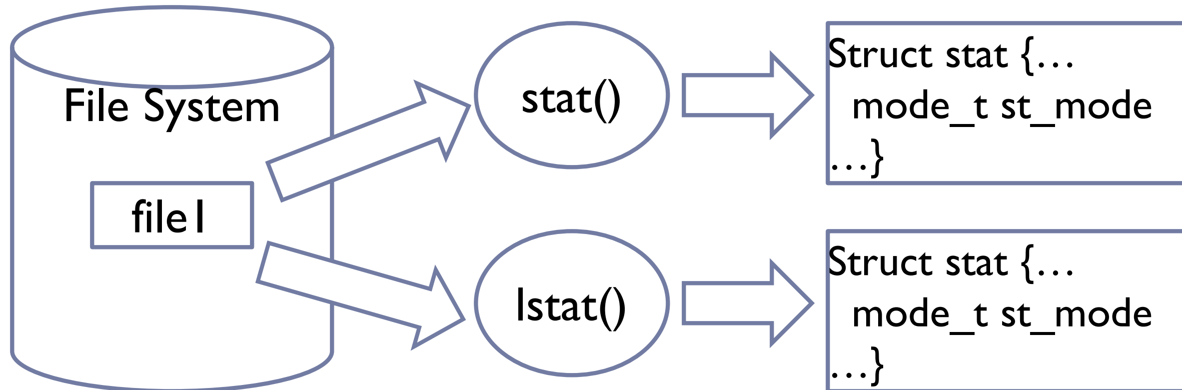
- Symbolic variables:
  - Command line options
  - Target file status
    - we partially simulate status of a file (`struct stat dstat`) in a file system by a symbolic value

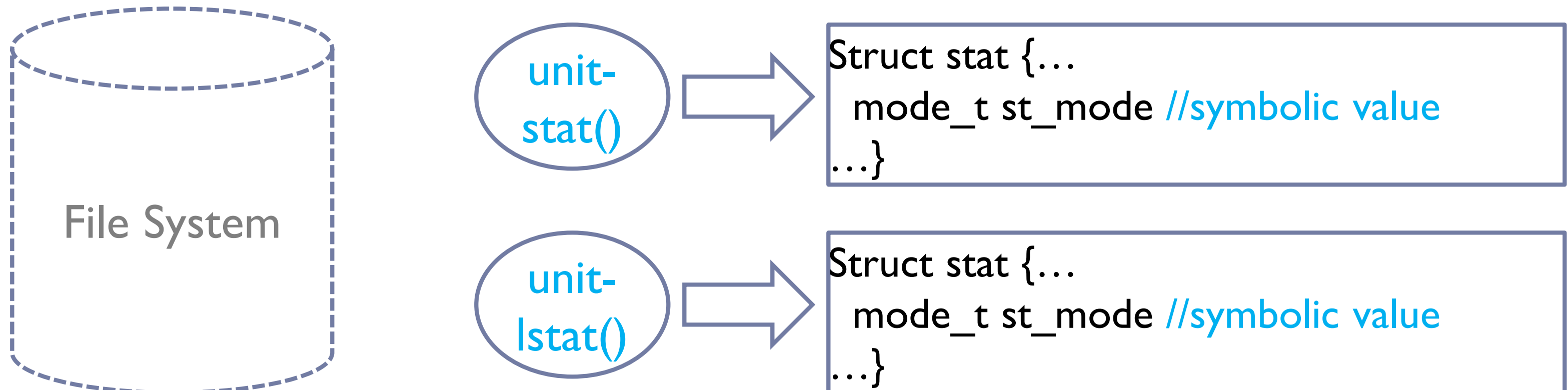
- Test stubs:
  - stat() and lstat() are replaced by unit-stat() and unit-lstat() for generating symbolic file status
    - `stat(const char *path, struct stat *buf)`
    - `lstat(const char *path, struct stat *buf)`

Symbolic Variables

- Symbolic options
  - Replacing `unsigned int opt` with a symbolic value .
    - `opt = getopt32(argv, ……);`
    - `unsigned char opt_mask[22]` is defined to replace `opt`
      - each element in `opt_mask[]` specifies a symbolic value for each bit of `opt`
- Symbolic file status
  - Replacing `mode_t dstat.st_mode` with a symbolic value
    - `st_mode` stores information on a file type, file permission, etc. in each bit.
      - `sizeof(st_mode)=sizeof(unsigned int)`

Stub Function – unit-stat(), unit-lstat()





Test Driver for Symbolic Command Line Options

```
50.int ls_main(int argc UNUSED_PARAM, char **argv)
51.{
52.//……
53./* process options */
54.    IF_FEATURE_LS_COLOR(applet_long_options
55.     = ls_longopts;)
56.#if ENABLE_FEATURE_AUTOWIDTH
57.    opt_complementary="T+:w+"; /* -T N, -w N */
58.    opt = getopt32(argv, ls_options, &tabstops,
59.        & terminal_width, IF_FEATURE_LS_COLOR
60.        (&color_opt));
61.#else
62.    opt = getopt32(argv, ls_options
63.         IF_FEATURE_LS_COLOR(&color_opt));
64.#endif
65.
66.//START of calculating all_fmt value
67.    for (i = 0; opt_flags[i] != (1U<<31); i++) {
68.   if (opt & (1 << i))
69.   {
70.           unsigned flags = opt_flags[i];
71.           //refresh value when trigger is on
72.           if (flags & LIST_MASK_TRIGGER) //0
73.               all_fmt &= ~LIST_MASK;
74.//……
75.           all_fmt |= flags;
76.       }
77.   }
78.//processing all_fmt for some special cases
79.   if (argv[1])
80.       all_fmt |= DISP_DIRNAME; /* 2 or more items? label
    directories */
81.//END of calculating all_fmt value
82.//calling internal functions
83.}
```

Test Driver for Symbolic File Status

```
1.  #define OPTSIZE 22
2.  #define MODESIZE 32
3.  unsigned char opt_mask[OPTSIZE];
4.  unsigned char modemask[2][MODESIZE];
5.  static char file_no=0;
6.
7.  #define SYM_S_ISLNK(mode) ((mode)[12]==0 &&
    (mode)[13]==1 && (mode)[14]==0 && (mode)[15]==1)

8.  //simulating file status by symbolic value
9.  static char gen_sym_file_stat(struct stat *buf){
10.     int i;
11.     for(i=0 ; i<MODESIZE ; i++){ //file type
12.         CREST_unsigned_char(modemask[file_no][i]);
13.     }
14.     return 1;
15.}
16.
17.static int unit_stat(const char *path, struct stat *buf){
18.     unsigned char local_mode[32];
19.     char ret;
20.     CREST_char(ret);
21.     if(ret==(char)0){
22.         if(gen_sym_file_stat(buf)){
23.             memcpy(local_mode, modemask[file_no], 32);
24.             if(SYM_S_ISLNK(local_mode)){ //link
25.                 local_mode[13]=0; //change to reg file
26.             }
27.             buf->st_mode = bstoi(local_mode, MODESIZE);
28.         }
29.         file_no++;
30.         return 0;
31.     }else
32.         return -1;
33.}
34.
35.static int unit_lstat(const char *path, struct stat *buf){
36.     unsigned char local_mode[32];
37.     char ret;
38.     CREST_char(ret);
39.     if(ret==(char)0){
40.         if(gen_sym_file_stat(buf)){
41.             memcpy(local_mode, modemask[file_no],
    MODESIZE);
42.             buf->st_mode = bstoi(local_mode, MODESIZE);
43.         }
44.         file_no++;
45.         return 0;
46.     }else
47.         return -1;
48.}
```