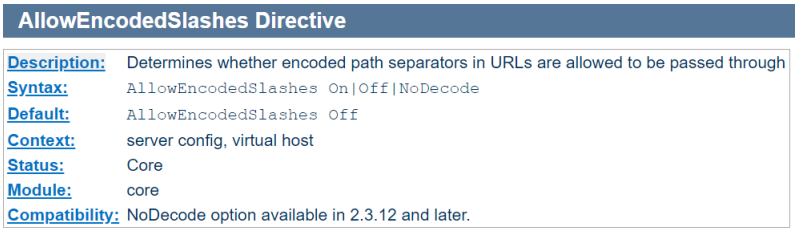

Fig. 1. Example of specifications in documentation

A APPENDIX

Our principles of specification summarizing are as follows.

A.1 Process

First, two authors are assigned with a same list of all the parameters from studied SUT. Second, they start to summarize specifications individually from documentations according to the specification rules in Table 2. Third, if an author fails to summarize certain specifications. He/She will look into the source code, including configuration data-structures, checking code, log statements and etc. Forth, when they complete, each author will check the other's results. When they diverge, the third author is consulted for additional discussion until consensus is reached.

A.2 Author Experience

All the authors have been working on software configuration research for several years, ranging from five to nine years, with prior work published at top Software Engineering venues. The authors have a good understanding of the six systems under study (HTTPD, Nginx, MySQL, PostgreSQL, Squid and VSFTPD)—they have used those studied systems as subjects of evaluation in their prior research. We expect that a similar level of experiences and expertise are needed for a team to reproduce specification summarizing, including the understanding of the designs of the systems and specifications rule of configuration under study. We believe a fair understanding of software configuration design and implementation is important, too.

A.3 Common Scenarios of Specification Summarizing

Documentation

Documentation is a common scenario to summarize specifications. Some SUTs (e.g., HTTPD) have well-formed parameter information. As shown in Figure 1, we could quickly summarize the specifications of *AllowEncodeSlashes*: parameter type is *ENUM*, and the value set is {"On", "Off", "NoDecode" }.

Source code (Configuration data structure)

Some other SUTs (e.g., PostgreSQL) maintain their parameters' specifications in well-formed data-structures. For example, in PostgreSQL-*guc.c*,

```
{
    /* This is PGC_SUSET to prevent hiding from log_lock_waits. */
    {"deadlock_timeout", PGC_SUSET, LOCK_MANAGEMENT,
        gettext_noop("Sets the time to wait on a lock before checking for deadlock."),
        NULL,
```

```
6           GUC_UNIT_MS
7     },
8     &DeadlockTimeout,
9         1000, 1, INT_MAX,
10        NULL, NULL, NULL
11 },
```

Line 9 indicates that the default value of parameter *deadlock_timeout* is 1000. And its *MIN* value and *MAX* value are 1 and *INT_MAX* respectively.

Source code (Checking code)

Checking code is also useful. For example, in Nginx documentation, parameter *err_page*'s description is vague, and it looks like an integer but no other specifications could be found. We track the data flow of the parameter and find specifications in *ngx_http_core_module.c*,

```
1 if (err->status < 300 || err->status > 599) {
2     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
3                        "value \"%V\" must be between 300 and 599",
4                        &value[i]);
5     return NGX_CONF_ERROR;
6 }
```

From the checking code, we find that it actually needs to be between 300 and 599.

Source code (Log statements)

Log statements might provide information we need when parameters are with complex semantics, e.g., path. For example in HTTPD-*core.c*,

```
1 if ((apr_filepath_merge((char**)&ap_server_root, NULL, arg,
2                        APR_FILEPATH_TRUENAME, cmd->pool) != APR_SUCCESS)
3     || !ap_is_directory(cmd->temp_pool, ap_server_root)) {
4     return "ServerRoot must be a valid directory";
5 }
```

Despite checking code in line 1-3 is vague, log statement in line 4 indicates that parameter *ServerRoot* must be an existent directory path.