Методика автоматизированной проверки возвращаемых кодов ошибок при тестировании программных интерфейсов

К. А. Власов, А. С. Смачёв

1. Введение

В современном мире компьютеры играют всё большую роль, а с ними — и программное обеспечение. В настоящее время наблюдается тенденция ко всё возрастающему усложнению и увеличению программных комплексов, состоящих из различных модулей и подсистем. Но чем больше программа, тем сложнее её отлаживать и проверять на соответствие спецификационным требованиям. Есть области, где ошибки совершенно недопустимы, например, медицина, атомная энергетика. Легко представить, что может натворить небольшая ошибка в программе, управляющей атомным реактором. Поэтому важность задачи верификации программного обеспечения трудно переоценить.

Данное исследование было проведено в рамках проекта OLVER (Open Linux VERification) [1], задачей которого была разработка тестового набора, позволяющего выполнять автоматическую проверку дистрибутивов операционной системы Linux на соответствие стандарту LSB (Linux Standard Base). Тестовый сценарий заключается в вызове всех функций из тестируемой подсистемы с заданным набором параметров и проверке возвращаемых значений. Одним из важных пунктов в этом тестировании является проверка кодов ошибок, возвращаемых функциями, а именно:

- проверка, что функция не возвращает код ошибки, когда она не должна этого делать;
- проверка, что функция возвращает код ошибки, если это требуется по спецификации;
- проверка, что в случае ошибки возвращается именно тот код ошибки, который функция должна вернуть согласно требованиям.

В процессе работы над тестовым набором задача проверки кодов ошибок возникает для большинства функций, причём обычно эта задача решается однотипно. Поэтому было решено автоматизировать написание исходного

кода проверок, насколько это возможно. В данной статье мы расскажем, как была решена эта задача. Во втором разделе будет дана исходная постановка задачи: какие конкретно действия должны быть автоматизированы, и как должна измениться работа по написанию тестов после внедрения этой автоматизации. В третьем разделе описаны технические подробности реализации, возникшие проблемы и их решение. И в заключении рассмотрены возможные пути развития системы в будущем.

2. Постановка задачи

При написании тестов для пакета OLVER используется система CTesK [2]. Она включает в себя язык SeC [3] — спецификационное расширение языка C, а также библиотеки для работы с различными типами данных, такими как массив, список, строка и т. д.

Добавление новых тестируемых функций в тестовый набор обычно выглядит следующим образом. Сначала программист читает текст стандарта и размечает его, выделяя атомарные требования при помощи конструкций языка HTML и присваивая им уникальные идентификаторы. Затем при открытии в браузере веб-страницы с текстом стандарта автоматически запускается скрипт, написанный на языке JavaScript, который анализирует список отмеченных требований, строит по нему шаблонный код на языке SeC и отображает его на странице. Теперь программисту достаточно просто скопировать этот код в SEC-файлы проекта, и бóльшая часть рутинной работы оказывается выполненной. Дальнейшая работа — это, в основном, создание тестовых сценариев и написание непосредственно кода проверки для всех атомарных требований, т. е. творческая работа, которая практически не поддаётся автоматизации.

В процессе работы над тестовым набором выяснилось, что проверки кодов ошибок, возвращаемых различными тестируемыми функциями, имеют много общего, в результате чего существенная часть времени тратится на одну и ту же работу по организации взаимодействия проверок для разных кодов ошибок друг с другом и с проверками других требований. Поэтому было решено, насколько это возможно, попытаться автоматизировать и эту задачу.

Наиболее удобный путь для такого рода автоматизации — это модификация существующего скрипта, чтобы он генерировал шаблонный код ещё и для проверки ошибок. Тем самым будет сохранена общая методика работы по разметке стандарта.

3. Генерируемый SeC-код

В первую очередь необходимо чётко определить, как должен выглядеть генерируемый код, что именно и как именно он будет проверять. За основу, разумеется, были взяты созданные к этому времени наработки, написанные вручную. Весь однотипный код удобно заменить набором макросов (они

поддерживаются в языке SeC, т. к. он включает в себя все возможности языка C). Это сделало бы исходный текст более наглядным и более простым для написания и отладки.

В стандарте LSB присутствуют несколько разных типов требований на поведение функций при возникновении ошибочной ситуации. Возможные варианты поведения функции в случае ошибки могут быть описаны следующим образом:

- **SHALL**: в стандарте присутствует фраза «*The function **shall** fail if …*», т. е. стандарт требует, чтобы функция всегда обнаруживала описанную ошибочную ситуацию и возвращала в этом случае определённый код ошибки.
- **MAY**: в стандарте присутствует фраза «*The function **may** fail if …*», т. е. стандарт не требует непременного обнаружения данной ошибочной ситуации, но в случае, если функция всё же реагирует на данную ошибку, её поведение должно быть таким, как описано в стандарте.
- **NEVER**: в стандарте присутствует фраза «*The function **shall not** return an error code …*», т. е. данный код ошибки не может быть возвращён ни при каких условиях.

Эти три типа требований и были взяты за основу.

Проверка кодов ошибок должна производиться до начала проверки остальных требований (за исключением, быть может, самых базовых аспектов, обеспечивающих непосредственное функционирование самой тестовой системы — таких как проверки на NULL). Это связано с тем, что если произошла ошибка, то функция не выполнила требуемое от неё действие, и, следовательно, проверки функциональных требований сообщат, что функция работает некорректно, т. е. не удовлетворяет стандарту. Однако ошибка могла быть вызвана тестовым сценарием намеренно (например, специально для того, чтобы проверить, как поведёт себя функция в этом случае), и тогда сообщение о некорректности поведения функции окажется ложной тревогой. Именно поэтому в первую очередь должен отработать блок проверки ошибочных ситуаций, и только если ошибка не была возвращена (и не ожидалась!), управление передаётся дальше, на код проверки основных требований.

Для удобства проверка кодов ошибок оформлена в виде блока, ограниченного операторными скобками **ERROR_BEGIN** и **ERROR_END**, которые являются макросами. **ERROR_BEGIN** при этом содержит параметры, глобальные для всего блока (например, в какой переменной хранится собственно код ошибки). Внутри блока располагаются индивидуальные проверки для каждого пункта, упомянутого в стандарте, которые также являются вызовами макросов с определённым набором параметров. Упомянутым выше трём основным типам требований соответствуют макросы **ERROR_SHALL**, **ERROR_MAY** и **ERROR_NEVER**. Помимо них, есть ещё некоторые дополнительные макросы, но о них мы расскажем ниже.

Приведём небольшой пример, демонстрирующий, как выглядит блок проверки кодов ошибок в одной из подсистем:

Текст стандарта:

```
The pthread_setspecific() function shall fail if:

[ENOMEM]
    Insufficient memory exists to associate the non-
NULL value with the key.

The pthread_setspecific() function may fail if:

[EINVAL]
    The key value is invalid.

These functions shall not return an error code of
[EINTR].
```

Текст спецификации:

```
ERROR_BEGIN(POSIX_PTHREAD_SETSPECIFIC,
            "pthread_setspecific.04.02",
            pthread_setspecific_spec != 0,
            pthread_setspecific_spec)
   /*
    * The pthread_setspecific() function shall fail
if:
    * [ENOMEM]
    *     Insufficient memory exists to associate the
non-NULL value with
    *     the key.
    */
    ERROR_SHALL(POSIX_PTHREAD_SETSPECIFIC,
                ENOMEM,
                "!pthread_setspecific.05.01",
                /* условие */)


   /*
    * The pthread_setspecific() function may fail if:
    * [EINVAL]
    *     The key value is invalid.
    */
```

```
    ERROR_MAY(POSIX_PTHREAD_SETSPECIFIC,
              EINVAL,
              "pthread_setspecific.06.01",
              !containsKey_Map(thread->key_specific,
key))

   /*
    * These functions shall not return an error code
of [EINTR].
    */
    ERROR_NEVER(POSIX_PTHREAD_SETSPECIFIC,
                EINTR,
                "pthread_setspecific.07")

ERROR_END()
```

Параметры макросов имеют следующий смысл:

ERROR_BEGIN*(ERR_FUNC, REQID, HAS_ERROR, ERROR_VAL):*

- **ERR_FUNC** — идентификатор функции, используемый в качестве префикса для конфигурационных констант. Например, с префикса POSIX_PTHREAD_SETSPECIFIC начинаются соответствующие конфигурационные константы, такие как:
  - POSIX_PTHREAD_SETSPECIFIC_HAS_EXTRA_ERROR_CODES
  - POSIX_PTHREAD_SETSPECIFIC_FAILS_WITH_EINVAL

  и др.
- **REQID** — идентификатор проверяемого требования (строковая константа).
- **HAS_ERROR** — предикат, определяющий условие возникновения ошибки (булевское выражение).
- **ERROR_VAL** — код ошибки (обычно это переменная *errno* или возвращаемое значение функции).

ERROR_MAY*(ERR_FUNC, ERRNAME, REQID, ERROR_PREDICATE),*

ERROR_SHALL*(ERR_FUNC, ERRNAME, REQID, ERROR_PREDICATE),*

ERROR_NEVER*(ERR_FUNC, ERRNAME, REQID):*

- **ERR_FUNC** — то же, что и выше.
- **ERRNAME** — имя константы, соответствующей ожидаемому коду ошибки. Например, ENOMEM, EINVAL.
- **REQID** — идентификатор проверяемого требования.
- **ERROR_PREDICATE** — предикат, определяющий условие возникновения ошибки.

ERROR_END*:*

параметров не требует.

3.1. Конфигурационные константы.

Стандарт LSB не всегда достаточно чётко определяет ситуации ошибочного завершения функции. В нем допускается, что функции могут возвращать коды ошибок, не описанные в стандарте, а также возвращать описанные коды ошибок в каких-то иных, определяемых реализацией случаях (см. [LSB, System Interfaces, Chapter 2, 2.3 Error Numbers]). На практике же такие случаи достаточно редки. К тому же вполне возможна ситуация, когда ошибочное завершение функции с кодом, не указанным в стандарте, является не предусмотренным разработчиком случаем, а ошибкой реализации. Поэтому была добавлена возможность выбрать в каждом конкретном случае желаемый уровень тестирования — в соответствии со стандартом, или более жёстко. Соответствующие константы именуются ***XX*_HAS_EXTRA_ERROR_CODES** и ***XX*_HAS_EXTRA_CONDITION_ON_*YY***, где ***XX*** — идентификатор функции, обычно передаваемый в макросы как параметр **ERR_FUNC**, а ***YY*** — имя константы ошибки.

Ситуация с требованиями типа **MAY** аналогична. В соответствии со стандартом, функция может не возвращать код ошибки, даже если условие выполняется. Однако в большинстве случаев разработчики «предпочитают ясность» и возвращают указанный код, хотя стандарт и не обязывает их к этому. Поэтому была введена специальная конфигурационная константа ***XX*_FAILS_WITH_*YY***, определяющая, следует ли считать требование нарушенным, если функция не возвращает код ошибки при выполнении условия типа **MAY**. Как можно заметить, при включённой константе ***XX*_FAILS_WITH_*YY*** требования **SHALL** и **MAY** по сути перестают отличаться друг от друга. Фактически, сами макросы **ERROR_SHALL** и **ERROR_MAY** реализованы как один макрос **ERROR_MAY_SHALL**, принимающий на вход ещё один дополнительный аргумент с именем **SHALL**, который и определяет, должна ли функция возвращать код ошибки, если предикат **ERROR_PREDICATE** истинен. Соответственно, макрос **ERROR_SHALL** реализован как вызов **ERROR_MAY_SHALL** с параметром **SHALL**, равным ***true***, а **ERROR_MAY** — как вызов **ERROR_MAY_SHALL** с параметром **SHALL**, зависящим от значения константы ***XX*_FAILS_WITH_*YY***.

3.2. Проверка

В макросе **ERROR_BEGIN** в первую очередь проверяется, что если функция завершилась с ошибкой, то код ошибки не равен **EOK** (коду, обозначающему отсутствие ошибки). Обычно признаком ошибки является сам факт отличия кода от **EOK**, и эта проверка превращается в тавтологию. Но бывают и другие случаи, когда, например, признаком ошибки является возвращаемое значение функции, равное -1, и тогда эта проверка необходима.

Проверка кодов ошибок не ограничивается случаем, когда функция завершилась с ошибкой. Условие типа **SHALL** должно обеспечивать также проверку в обратную сторону: если условия вызова функции ошибочны, то функция обязана вернуть ошибку, и если этого не произошло, поведение считается некорректным.

Заметим, что независимо от того, успешно ли отработала функция, необходимо проверять **все** требования к кодам возврата, поскольку выполнение одного требования ещё не означает, что другие требования не нарушены. Таким образом, решение о правильной работе функции должно приниматься только в макросе **ERROR_END**, не раньше. На первый взгляд, это соображение очевидно, но именно поэтому его так легко упустить из виду.

Для каждого требования имеется четыре базовых случая:

1. Ошибка ожидалась и произошла.
2. Ошибка не ожидалась, но произошла.
3. Ошибка ожидалась, но не произошла.
4. Ошибка не ожидалась, и её не было.

В каждом из этих случаев есть свои тонкости, которые легко упустить из виду. Возьмём, например, первый случай, когда ошибка ожидалась и произошла. Первым побуждением будет выдать вердикт, что проверка завершилась успешно, и выйти из блока проверки ошибок. Однако этого делать ни в коем случае нельзя, т. к. должны быть проверены абсолютно все требования — то самое «очевидное» соображение, высказанное выше! Ведь даже в случае возврата ожидаемого кода ошибки есть вероятность, что другое требование окажется нарушенным. То же самое относится к последнему случаю, когда ошибка не ожидалась и не произошла.

Случай второй, когда ошибка не ожидалась, но произошла. Если это требование типа **NEVER**, или если это **MAY/SHALL**, и соответствующая константа ***XX*_HAS_EXTRA_CONDITION_ON_*YY*** равна нулю, то можно констатировать неправильную работу функции. При ненулевом значении этой константы поведение функции не противоречит стандарту.

Случай третий, когда ошибка ожидалась, но не произошла. Если это условие типа **SHALL** или если это **MAY**, и соответствующая константа ***XX*_FAILS_WITH_*YY*** не равна нулю, то можно констатировать неправильную работу функции. Если же функция вернула другой код ошибки, то поведение также является некорректным. Однако в этом случае дополнительно стоит обратить внимание на причины возникновения этой проблемы. Не исключено, что возвращённый код взялся не «с потолка», а явился следствием того, что оказались выполнены условия возникновения и этой второй ошибки наряду с первой. Если оба требования являются требованиями типа **SHALL** (или **MAY** с ненулевой константой ***XX*_FAILS_WITH_*YY***), то можно утверждать, что в стандарте имеет место противоречие: для одних и тех же условий требуется вернуть одновременно

два различных кода. На таких ситуациях мы сейчас остановимся несколько подробнее.

3.3. Пересечение требований

Может оказаться, что один и тот же код ошибки встречается в нескольких требованиях. Хорошо, если эти требования лишь дополняют друг друга. Но может сложиться ситуация, когда эти требования противоречат друг другу, а именно, когда одно из них требует, чтобы функция вернула код ошибочного завершения, а другое — утверждает, что в этой ситуации данный код ошибки не может быть возвращён. Такая ситуация может и не встретиться в ходе выполнения тестов, но подобное противоречие условий в любом случае является ошибкой в стандарте.

Другая ситуация: под одно и то же условие попадают разные коды ошибок, при этом стандарт для обоих кодов требует обязательного возврата этого кода. В этом случае при выполнении условий появления ошибки тесты зафиксируют некорректность в работе тестируемой системы, независимо от того, какой из этих кодов ошибки был возвращён функцией, так как проверка другого кода зафиксирует некорректность поведения. В такой ситуации обычно из текста стандарта понятно, какой код ошибки должен возвращаться в каждом конкретном случае, и, следовательно, условия возникновения ошибок должны быть дополнены так, чтобы не выполняться одновременно ни в каких ситуациях. Если же из контекста неясно, какой код ошибки должен быть возвращён, налицо явное противоречие в стандарте.

Таким образом, пересечение требований является серьёзной проблемой, и разработчик тестов должен обращать на них особое внимание.

3.4. Трёхзначная логика

Иногда случается, что мы не во всех ситуациях можем определить, выполняется данное условие или нет. В этом случае логика становится трёхзначной: «да» — «нет» — «не знаю». Конечно, можно было бы поместить проверку внутрь условного ветвления, в котором условие всегда было бы проверяемым, но это значительно усложнило бы читаемость и понятность кода. К тому же для человека такая трёхзначная логика в достаточной мере «естественна».

Поэтому к существующим макросам были добавлены **ERROR_SHALL3** и **ERROR_MAY3**, в которых предикат может принимать одно из трёх значений: **False_Bool3**, **True_Bool3**, **Unknown_Bool3**.

Наиболее используемы эти макросы в случаях, когда значение проверяемого параметра подсистемы может быть неизвестно в силу закрытости тестовой модели. Пример:

```
    /*
```

```
     * [EINVAL]
     *     Invalid speed argument.
     */
    ERROR_MAY3(LSB_CFSETSPEED, EINVAL,
"cfsetspeed.04.01",
               (isKnown_Speed(speed) ?
                False_Bool3 :
not_Bool3(isValid_Speed(speed)))
             )
```

Другими частыми случаями употребления этих макросов являются проверки на корректность параметров или на наличие определённых ресурсов. Зачастую стандарт не указывает, как определить некорректность нужного значения, а проверить наличие ресурсов (например, оперативной памяти) не представляется возможным, так как почти никогда не известно точно, какое их количество будет «достаточным». С другой стороны, про отдельные значения может быть известно, что они заведомо корректные или заведомо некорректные. Пример:

```
    /*
     * The confstr() function shall fail if:
     * [EINVAL]
     *     The value of the name argument is invalid.
     */
    /* [LSB: 18.1.1. Special Requirements]
     * A value of -1 shall be an invalid "_CS_..."
value for confstr().
     */
    ERROR_SHALL3(POSIX_CONFSTR, EINVAL, "confstr.12",
                 ((name   ==   -1)   ?   True_Bool3   :
Unknown_Bool3)
                )
```

В подобных случаях макросы с трёхзначной логикой позволяют естественным образом формализовать условие с помощью одной проверки, что обеспечивает читабельность и наглядность программного кода.

3.5. Оформление непроверяемых требований

В силу различных причин проверка некоторых требований может быть затруднена или даже невозможна. Для таких ситуаций были введены два дополнительных макроса.

1. **ERROR_UNCHECKABLE**. Этот макрос используется в том случае, когда проверка требования не может быть выполнена по каким-либо

причинам. Сам макрос просто сигнализирует о том, что данный код ошибки не противоречит стандарту.
2. **TODO_ERR(*errname*)**. Этот макрос-заглушка используется вместо предиката, определяющего условия возникновения ошибки, в тех случаях, когда проверка в принципе возможна, но либо она чрезмерно сложна, либо её реализация по каким-то причинам на время отложена. Макрос **TODO_ERR** возвращает истину тогда и только тогда, когда возвращаемый код ошибки совпадает с проверяемым. Таким образом, вместо реальной проверки в данном случае мы имеем тавтологию. Этот макрос подставляется по умолчанию вместо конкретных условий в коде, который генерируется непосредственно из разметки. Это позволяет достаточно быстро и с минимальными усилиями создать компилирующийся и работающий проект для тестирования новой подсистемы.

4. Заключение

Несмотря на то, что работа над пакетом OLVER в настоящее время завершается, описанная в данной статье методика будет применяться при написании тестовых наборов, расширяющих и дополняющих существующие сертификационные тесты на соответствие стандарту LSB, поскольку она очень проста в применении, и при этом генерируемый ей код достаточно универсален. Конечно, нужно понимать, что заранее учесть и продумать все мыслимые комбинации различных ошибочных ситуаций невозможно. Всегда может найтись какая-то очень специфичная функция, требующая нестандартного подхода. И именно эта возможность задаёт направление для дальнейшего развития системы. С одной стороны, в подобных случаях можно попросту подходить к каждой функции индивидуально и писать код, не опираясь на сгенерированные шаблоны. Однако если обнаруженная зависимость не единична, а встречается в нескольких функциях, то стоит сначала оценить трудозатраты: возможно, в такой ситуации имеет смысл доработать систему генерации кода так, чтобы она теперь учитывала и эти новые ситуации. Учитывая, что стандарт LSB активно развивается и включает в себя всё больше и больше функций, можно с уверенностью сказать, что описанная в данной статье система будет развиваться и дальше.

Литература

1. http://linuxtesting.ru/
2. CTesK 2.2 Community Edition: Руководство пользователя. http://www.unitesk.ru/download/papers/ctesk/ce/CTesK2.2CEUserGuide.rus.pdf
3. CTesK 2.2 Community Edition: Описание языка SeC. http://www.unitesk.ru/download/papers/ctesk/ce/CTesK2.2CELanguageReference.rus.pdf