Алгоритм генерации тестов работоспособности на основе расширенной базы данных LSB

А. В. Пономаренко, Е. С. Чернов
{susanin, ches}@ispras.ru

Аннотация. В статье рассматривается алгоритм автоматической генерации тестов на основе базы данных LSB, содержащей описания сигнатур функций и системы типов, и ее расширения, представляющего собой описание некоторых ограничений стандарта для этих функций и служащее для корректной инициализации параметров и проверки ряда требований к корректности результатов.

1. Введение

Стандарт Linux Standard Base (LSB) включает описание более 30000 функций, число которых растет с каждой его новой версией. Для проверки реализаций функций на соответствие стандарту LSB необходимо создать для них тесты. В настоящее время существуют тесты примерно для 5000 функций, поэтому задача создания тестов для остальных функций является актуальной, но создание качественных тестов для них требует больших затрат. В этой ситуации привлекательной выглядит идея создания на первом этапе *тестов работоспособности* для всех непротестированных функций, проверяющих работу этих функций только в простейших ситуациях, а на втором этапе — создание тестов, детально проверяющих корректность работы наиболее значимых из них [1].

Задачу первого этапа — массового создания тестов работоспособности — решает технология Azov. Высокая производительность создания тестов достигается в ней за счет высокой степени автоматизации, поэтому существенную роль играют алгоритм генерации тестов.

В качестве исходных данных алгоритм генерации используют базу данных LSB, содержащую сигнатуры функций и описание системы типов данных, и ее специальное расширение, содержащее описания уточнений входных и выходных данных функций. Эти уточнения являются формализацией части требований документации на функции (например, стандарта) и представлены в виде абстрактных конструкций — *специализированных типов* [2].

Целью алгоритма генерации является построение тестов работоспособности на основе имеющихся исходных данных.

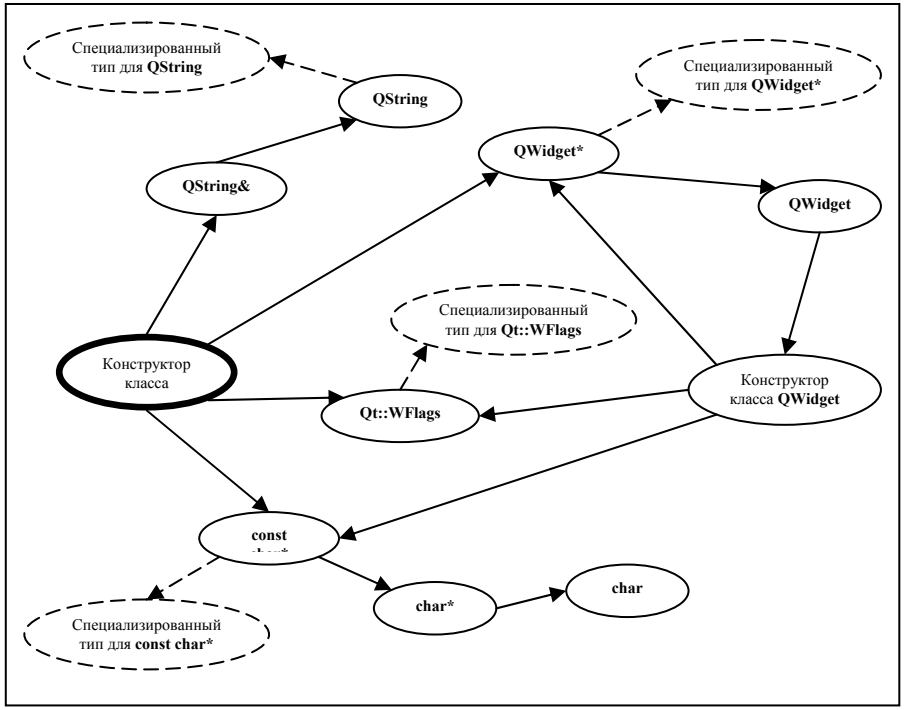

Рис. 1. Граф зависимостей исходных данных для конструктора класса QLabel

В первом разделе статьи рассматриваются основные принципы алгоритма генерации тестов. Во втором и третьем разделе раскрываются шаги алгоритма. В четвертом разделе приведен пример теста работоспособности.

2. Основные принципы алгоритма генерации

2.1. Представление исходных данных в виде графа зависимостей

Исходные данные можно разбить на две группы — описания сигнатур функций с исходными типами их параметров и результатов и описания специализированных типов для этих же параметров и результатов. Данные из первой группы можно представить в виде *графа зависимостей*. Для этого функции отождествляются со структурными типами — параметры функции суть поля структурного типа. В этом графе каждый узел — это некоторый тип данных (в том числе и функция). Узел графа указывает на другие узлы, если

он от них зависит. Например, тип зависит от своего базового типа (от которого данный тип является производным) и от функции возвращающей этот тип, функция зависит от узлов, отвечающих за типы ее параметров. На Рис. 1 изображен граф зависимостей исходных данных для конструктора класса `QLabel`, имеющего сигнатуру "`QLabel::QLabel(QString&, QWidget*, const char*, Qt::WFlags)`". Второй параметр этого конструктора имеет тип QWidget*, базовым типом которого является класс `QWidget`. Объект этого класса можно получить с помощью конструктора "`QWidget::QWidget(QWidget*, const char*, Qt::WFlags)`".

Данные из второй группы — специализированные типы — дополняют граф зависимостей (на Рис. 1 специализированные типы изображены пунктиром). Зависимость в графе обычного типа от специализированного типа означает, что при построении экземпляра этого типа будут использоваться атрибуты соответствующего специализированного типа.

2.2. Структура теста

Выбрана традиционная структура генерируемых тестов, состоящая из различных блоков (см. Рис. 2).

1	Информация о тесте и комментарии
2	Подключение заголовочных файлов
3	Определение вспомогательных функций (**SpecCode)**
4	Определение классов-наследников
5	**int main () {...**
6	Программный код в начале теста (**Preamble**)
7	Конструирование параметров целевой функции
8	Программный код перед вызовом целевой функции (**InitCode**)
9	Вызов целевой функции
10	Проверка некоторых требований (**NormalResult**)
11	Программный код после вызова целевой функции (**FinalCode**)
12	Освобождение памяти
13	Программный код в конце теста (**Finalization**)
14	**... return 0; }**

Рис. 2. Структура теста работоспособности

- Блок 1 "Информация о тесте и комментарии" — информация о том, когда и для какой аппаратной архитектуры сгенерирован тест, а также о сигнатуре целевой функции.
- Блок 2 "Подключение заголовочных файлов" — содержит директивы на подключение заголовочных файлов вида "`#include <qstring.h>`" необходимых для теста.
- Блок 3 "Определение вспомогательных функций" — взятые непосредственно из атрибута **SpecCode** определения вспомогательных функций.
- Блок 4 "Определение классов-наследников" — содержит определение используемых в тесте классов-наследников некоторых классов (см. раздел 3.2).
- Блоки 5 и 14 обозначающие начало и конец функции `main()`.
- Блоки 6 "Программный код в начале теста" и 13 "Программный код в конце теста" — соответствуют специальным ограничениям на целевую функцию **Preamble** и **Finalization**.
- Блок 7 "Инициализация параметров целевой функции" — конструирование фактических параметров целевой функции.
- Блоки 8 "Программный код перед вызовом целевой функции" и 11 "Программный код после вызова целевой функции" — соответствуют атрибутам специализированных типов **InitCode** и **FinalCode**.
- Блок 9 "Вызов целевой функции".
- Блок 10 "Проверка некоторых требований" — код, полученный из атрибута **NormalResult** специализированных типов и сигнализирующий об ошибке в случае нарушения требований наложенных на целевую функцию.
- Блок 12 "Освобождение памяти" — содержит вызовы деструкторов для всех созданных объектов в тесте.

Инициализацию фактических параметров целевой функции (блок 7) можно представить в виде цепочки вызовов других функций. Одной из основных задач алгоритма генерации является построение таких *цепочек инициализации*.

2.3. Представление цепочки инициализации параметров функции в виде дерева

Цепочку инициализации фактических параметров целевой функции можно представить в виде *дерева вывода* графа зависимостей исходных данных для этой функции. Корень дерева — это узел, содержащий целевую функцию.

Узлы дерева — типы данных (или функции), используемые при инициализации параметров целевой функции. Листья дерева содержат некоторое значение, с помощью которого инициализируется соответствующий тип данных.

Дерево вывода для конструктора "`QLabel::QLabel(QString&, QWidget*, const char*, Qt::WFlags)`" изображено на Рис. 3. Первый параметр этого конструктора инициализируется с помощью конструктора "`QString::QString(const char*)`", второй — с помощью конструктора "`QWidget::QWidget(QWidget*, const char*, Qt::WFlags)`". Третий и четвертый параметры инициализируются соответственно произвольной строкой и одним из возможных значений перечисления `Qt::WFlags`.

2.4. Шаги алгоритма генерации тестов

Генерация тестов основана на выполнении двух шагов: построении дерева вывода в графе зависимостей исходных данных и генерации кода теста на основе этого дерева. Далее в разделе 3 описана последовательность построения дерева вывода, а в разделе 4 — последовательность генерации программного кода на основе построенного дерева вывода.

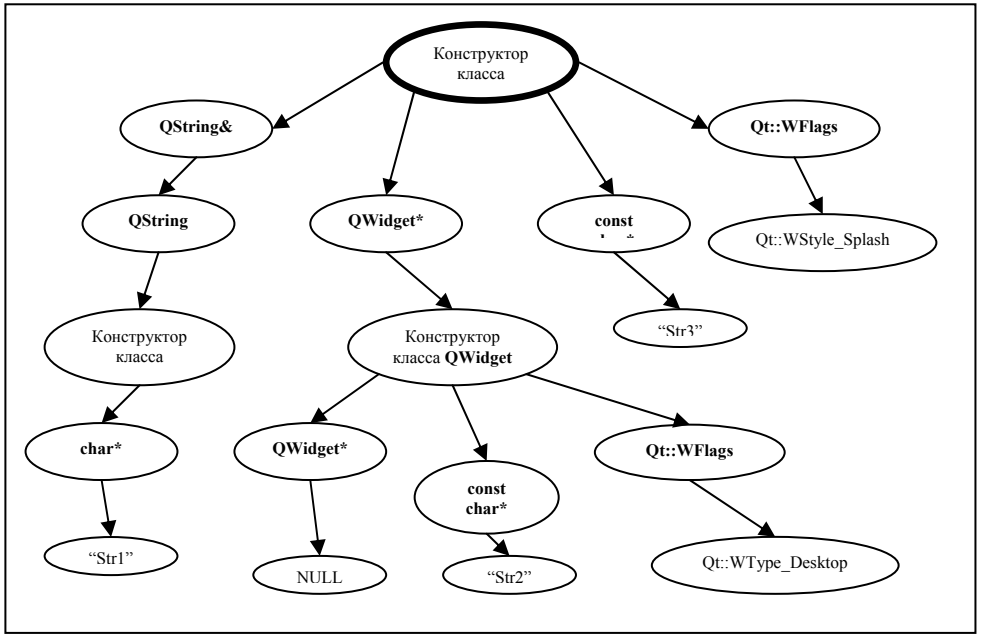

Рис. 3. Пример дерева вывода для конструктора класса QLabel

3. Построение дерева вывода

3.1. Основные характеристики типов данных

Дерево вывода строится на основе графа зависимостей исходных данных, узлами которого являются различные типы и функции. Основными характеристиками любого типа данных **Type** в графе зависимостей с точки зрения построения дерева вывода являются следующие.

- *Порождающий тип* (**originative type**) — тип данных, который не зависит ни от какого другого типа в графе зависимостей исходных данных, при этом **Type** является производным от него. Порождающий тип не построен ни из какого другого при помощи модификаторов `*`, `&`, **`const`** или определения синонима с помощью **`typedef`**. Например, порождающим типом для типа "`QThread**`" является "`QThread`", а для типа "**`const char*`**" — тип "**`char`**".

- *Базовый тип* (**base type**) — тип данных, от которого непосредственно зависит **Type** в графе зависимостей исходных данных. **Type** строится из базового с помощью одного из модификаторов `*`, `&`, `const` или с помощью **`typedef`**.

- *Уровень вложенности указателей* (**pointer level**). Это количество модификаторов `*`, использованных при построении данного типа из порождающего. Уровень вложенности указателей типа "`QThread**`" равен двум, а для типа "**`const char*`**" — равен единице.

- *Множество родственных типов* (**type family**) — определяется, как множество типов имеющих один и тот же порождающий тип. Например, для типа `Display*` множеством родственных типов является {`_XDisplay`, `_XDisplay*`, `Display`, `Display*`, `Display**`, `Display***`}, так как порождающим типом у всех является тип `_XDisplay`.

3.2. Расширение графа зависимостей исходных данных виртуальными специализированными типами

Перед построением дерева вывода граф зависимостей исходных данных дополняется особым видом специализированных типов — *виртуальными специализированными типами*. Такие специализированные типы генерируются автоматически. Специализированные типы содержат следующие атрибуты:

- **InitCode —** программный код, помещаемый перед вызовом функции;
- **FinalCode —** программный код, помещаемый после вызова функции;
- **NormalResult —** логическое выражение для проверки правильности результата функции;

- **SpecValue —** значение, с помощью которого инициализируется параметр;
- **SpecCode —** программный код, содержащий определения функций используемых в других атрибутах;
- **DataType —** тип данных, который уточняется специализированным типом.

Различают два вида виртуальных специализированных типов: для параметров и для возвращаемых значений.

3.2.1. Виртуальные специализированные типы для параметров

Этот вид специализированных типов содержит атрибуты **SpecValue** и **SpecCode**, тем самым, достраивая до конца дерево инициализации параметра. В зависимости от порождающего типа для разных множеств родственных типов виртуальные специализированные типы создаются следующим образом.

- Встроенные типы (**intrinsic**) — значения **SpecValue** для типа `int` задаются последовательно из множества {1, 2, 3 …}, для `char` — из множества {‘A’, ‘B’, ‘C’ …}, для `double` — из множества {1.0, 2.0, 3.0 …}, для `bool` — из множества {true, false} и т.д.
- Перечисления (**enum**) — значения **SpecValue** задаются некоторым значением из списка возможных значений данного перечисления.
- Любой тип (**intrinsic, struct, union, class**) с уровнем указателя большим единицы, для которого нет других возможностей инициализации — значение **SpecValue** этого специализированного типа выбирается равным `NULL`.
- Классы (**class**) — атрибут **SpecCode** содержит определение класса-наследника, значение **SpecValue** содержит вызов одного из конструкторов этого класса-наследника. В классе-наследнике переопределяются автоматически чисто виртуальные методы, определяются необходимые конструкторы и деструкторы, а так же методы, напрямую вызывающие protected-методы исходного класса.

В последнем случае виртуальные специализированные типы создаются, если необходимо вызвать protected-метод для объекта или сконструировать объект абстрактного класса.

3.2.2. Виртуальные специализированные типы для возвращаемых значений

Если возвращаемое значение функции имеет тип с уровнем указателя большим единицы, то генерируется виртуальный специализированный тип атрибутом **NormalResult** содержащим проверку на неравенство `NULL`.

3.3. Выбор направления построения дерева вывода в узле графа зависимостей исходных данных

Узлы графа зависимостей исходных данных могут быть трех видов: типы, функции и специализированные типы. Поскольку узлы, отвечающие за типы данных, могут зависеть от нескольких других узлов, то построение дерева вывода может быть продолжено в направлении любого из них. Поддерживаются следующие политики выбора направления построения дерева в узле: конструирование, вызов функции и присваивание явно указанного значения.

3.3.1. Конструирование

Конструирование осуществляется разными способами в зависимости от типа узла:

- Указатель на тип (**pointer**) — конструирование базового типа с последующим приведением к типу параметра.
- Структуры (**struct**) и объединения (**union**) — инициализация полей.
- Массивы (**array**) — инициализация элементов массива.
- Указатель на функцию (**func-ptr**) — определяется простейшая функция с нужными параметрами и проинициализированным возвращаемым значением.
- Классы (**class**) — вызов конструктора.

3.3.2. Вызов функции

По графу зависимостей исходных данных для данного типа определяется множество функций с возвращаемым значением этого же типа. Это множество сортируется по разным характеристикам:

- Уровень доступа (public, protected) — проще вызывать public-функцию, чем protected-функцию.
- Происхождение (библиотека, класс) — лучше вызывать функцию из тех же библиотеки или класса, в которых определена целевая функция.
- Число параметров — проще вызывать функции с меньшим числом параметров.

Затем последовательно для каждой найденной функции строится дерево вывода соответствующего графа зависимостей исходных данных. Поиск заканчивается, когда будет найдена функция, для которой это дерево удалось успешно построить.

3.3.3. Присваивание значения

Если рассматриваемый узел графа зависит от какого-нибудь специализированного типа (или виртуального специализированного типа)

имеющего значение атрибута **SpecValue,** то построение дерева вывода будет осуществляться в направлении этого специализированного типа.

3.4. Поиск корректного дерева вывода в графе зависимостей исходных данных

Поиск дерева вывода производится рекурсивно. Начальным узлом служит узел, соответствующий целевой функции. На каждом шаге рекурсии для текущего узла графа зависимостей определяется направление дальнейшего построения дерева вывода, которое порождает множество узлов графа для следующего шага рекурсии. Для избегания бесконечной рекурсии, возникающей из-за существующих циклов в графе зависимостей необходимо помечать использованные узлы графа и далее при анализе графа их не использовать. Корректный выход из рекурсии осуществляется по достижении листьев, для которых известно конкретное значение, с помощью которого они могут быть проинициализированы. Если в некоторой ветви рекурсии не удается проинициализировать тип, то она считается некорректной и происходит выход из этой ветви с ошибкой. Это приводит к выбору другого пути в графе зависимостей на более высоком уровне рекурсии и, в целом, к перебору возможных путей в нем.

Во время поиска пути инициализации могут встретиться специализированные типы. В этих случаях путь инициализации фиксирован и определяется соответствующим специализированным типом.

4. Генерация программы на основе дерева вывода

4.1. Приведение родственных типов данных друг к другу

При генерации программного кода на основе ребра, связывающего узлы, отвечающие за родственные типы данных, используется приведение типов. Приведение одного типа к другому родственному типу осуществляется на основе уровней вложенности указателей этих типов.

- Приведение типа Type1 с меньшим уровнем указателя к типу Type2 с большим уровнем указателя осуществляется посредством создания массива из элементов типа Type1, затем массива массивов из элементов Type1 и так далее до получения нужного уровня вложенности указателей типа Type2.

- Приведение типа Type1 с большим уровнем указателя к типу Type2 с меньшим уровнем указателя осуществляется посредством последовательного разыменования переменной типа Type1 до получения нужного уровня вложенности указателей типа Type2.

4.2. Получение экземпляра типа на основе вызова функции

Программный код на основе ребра, ведущего от некоторого типа к функции, включает вызов этой функции и присвоение ее возвращаемого значения экземпляру соответствующего типа.

4.3. Генерация программного кода на основе специализированного типа и специальных ограничений для функции

При генерации программного кода на основе ребра, связывающего любой узел дерева вывода с узлом, отвечающим за специализированный тип необходимо правильно интерпретировать атрибуты этого специализированного типа.

Сначала в атрибутах **InitCode, FinalCode, SpecValue** и **NormalResult** специализированного типа осуществляется замена специальных конструкций:

- ***$0*** — идентификатор целевого параметра. Вместо него подставляется экземпляр специализированного типа;
- ***$obj*** — идентификатор объекта. Вместо него подставляется экземпляр класса целевого метода.

Данные конструкции позволяют описать в специализированном типе действия над параметром, независимо от имени переменной этого параметра. Содержимое атрибутов **InitCode, FinalCode** располагается до и после вызова функции имеющей параметр соответствующего специализированного типа. Значение **SpecValue** присваивается экземпляру специализированного типа.

Значение атрибута **NormalResult** представляет собой логическое выражение, которое должно быть истинным в случае правильной работы функции. Из него формируется код, проверяющий правильность результата функции и сигнализирующий об ошибке в случае нарушения требования:

```
if (Not NormalResult)
{
        //Сообщение об ошибке
}
```

В атрибуте **SpecCode** специализированного типа содержится определение вспомогательных функций, которые используются в других атрибутах этого специализированного типа. В их телах иногда требуется инициализировать вспомогательные данные, для чего могут быть использованы следующие конструкции:

- ***$(имя типа)*** — указание проинициализировать и подставить переменную указанного типа:
- ***$[имя функции (или метода)]*** — указание проинициализировать параметры и подставить вызов указанной функции (или метода)

Рассмотрим некоторый специализированный тип, который служит для инициализации параметра типа `QListBoxItem` — элемента в списке `QListBox`. Примером атрибута **SpecCode** может послужить следующий код, содержащий определение вспомогательной функции `create_QListBoxItem()`, инициализирующей тип `QListBoxItem` посредством вызова метода `QListBox::firstItem` для которого уже внесены необходимые специализированные типы:

```
QListBoxItem* create_QListBoxItem()
{
    return $[QListBox::firstItem() const];
}
```

Специализированный тип для объекта метода `QListBox::firstItem` указывает провести инициализацию параметра с помощью вызова вспомогательной функции `create_QListBox` для получения непустого списка типа `QListBox`. Ниже приведена часть программы, полученная с помощью этого специализированного типа:

```
QListBox* create_QListBox()
{
    QListBox* Obj = new QListBox(NULL, "ListBox", 0);
    for(int i=0;i<10;i+=1)
    {
        Obj->insertItem("XXX", i);
    }
    Obj->setColumnMode(QListBox::FitToHeight);
    Obj->setRowMode(QListBox::FitToHeight);
    Obj->setMultiSelection(true);
    Obj->setVariableHeight(true);
    Obj->setVariableWidth(true);
    Obj->setCurrentItem(1);
    Obj->selectAll(true);
    return Obj;
}
QListBoxItem* create_QListBoxItem()
{
    // PARAMETER INITIALIZATION
    QListBox * Obj = create_QListBox();
    return Obj->firstItem();
}
```

Специализированные типы расширяют описания параметров функций и возвращаемых значений. Кроме этого есть ограничения, привязанные непосредственно к самим функциям — необходимый код в самом начале и конце теста (соответственно преамбула (**Preamble**) и постамбула (**Finalization**)). Преамбула и постамбула бывают двух типов — непосредственно относящиеся к конкретной функции (например, инициализация необходимого окружения или удаление созданного функцией

файла) и относящиеся ко всем функциям некоторой системы (создание и запуск приложения в библиотеке Qt).

5. Пример теста работоспособности

```
//*********************************//
// SHALLOW-TEST FOR LSB INTERFACE:
// QMutex::locked()
// GENERATED AT: Thu Sep 13 12:49:03 2007
// CPU: x86_64
//*********************************//

#include <qmutex.h>
#include <qtimer.h>
#include <qapplication.h>
#include <iostream>

int main()
{
    //PREAMBLE
    #ifdef Q_WS_X11
    bool useGUI = getenv( "DISPLAY" ) != 0;
    #else
    bool useGUI = TRUE;
    #endif
    int app_argc = 0;
    char* app_argv[0];
    QApplication* app = new QApplication(app_argc, app_argv, useGUI);
    QTimer::singleShot(100, app, SLOT(quit()));

    //PARAMETER INITIALIZATION
    QMutex *Obj = new QMutex(true);

    //INIT CODE
    Obj->lock();
    //TARGET INTERFACE
    bool
    Return = Obj->locked();
    //CHECK RETURN VALUE
    if(!(Return == true))
    {
        std::cout << "CONSTRAINT FOR RETURN FAILED: '$0 == true'" << std::endl;
        std::cout << "INTERFACE RETURN VALUE: " << Return << std::endl;
    }

    //FINAL CODE
    Obj->unlock();

    //FINALIZATION
    app->exec();

    return 0;
}
```

Рис. 4. Пример теста работоспособности для метода QMutex::locked

Пример теста работоспособности для метода `QMutex::locked()` из библиотеки Qt3 изображен на Рис. 4. Объект класса `QMutex` для вызова этого метода создается автоматически при помощи вызова конструктора `QMutex::QMutex(bool)`. Для класса `QMutex` задан специализированный тип, содержащий атрибуты **InitCode**, **NormalResult, FinalCode**, включающие соответственно блокировку (вызов метода `lock()`), проверку блокировки (результат вызова метода `locked()` должен быть равен true) и разблокировку

(вызов метода `unlock()`) экземпляра этого класса. Специальные ограничения на функцию **Preamble** и **Finalization** содержат создание и запуск приложения (приложение в библиотеке Qt3 — объект класса `QApplication`). В целом этот тест вызывает целевой метод с корректными параметрами и проверяет правильность его работы.

6. Заключение

Рассмотренный алгоритм генерации тестов работоспособности позволяет автоматически в течение нескольких часов получать десятки тысяч тестов. Таким образом, получена возможность быстрого создания тестов для любых уже имеющихся и новых библиотек в LSB. Дальнейшее улучшение алгоритма генерации тестов наряду с разработкой структуры расширений базы данных LSB приведет к повышению степени автоматизации создания тестов и повышению их качества.

Литература

[1] Р. Зыбин, В. Кулямин, А. Пономаренко, В. Рубанов, Е. Чернов. Технология Azov автоматизации массового создания тестов работоспособности. Опубликовано в этом же сборнике.

[2] Р. Зыбин, А. Пономаренко, В. Рубанов, Е. Чернов. Расширение описаний сигнатур операций для автоматической генерации тестов. Опубликовано в этом же сборнике.

[3] http://www.linuxbase.org.

[4] http://www.linux-foundation.org/en/LSB.