Расширение описаний сигнатур операций для автоматической генерации тестов

Р. С. Зыбин, А. В. Пономаренко, В. В. Рубанов, Е. С. Чернов
{phoenix, susanin, vrub, ches}@ispras.ru

Аннотация. В статье рассматривается задача расширения стандартной информации, извлекаемой из сигнатур программных интерфейсов (операций), для обеспечения возможности автоматической генерации тестов, вызывающих цепочки таких операций с корректными параметрами и в корректном окружении. Эта задача актуальна при тестировании интерфейсов систем с большим числом операций (больше 1000) в условиях ограниченных ресурсов на создание тестов. Для описания расширенных данных вводятся так называемые *специализированные типы*, уточняющие семантику типов объектов, возвращаемых значений и параметров. С их помощью можно дополнить исходное описание сигнатур операций, задав недостающую для эффективной генерации тестов информацию. Рассматриваются различные виды и свойства специализированных типов. Отдельно обсуждаются аспекты, ускоряющие ввод необходимых дополнительных данных для групп операций. Приводится пример реализации хранения расширенной информации об операциях в базе данных стандарта Linux Standard Base.

1. Введение

При тестировании программных систем, состоящих из большого числа интерфейсных операций (API) в условиях ограниченных ресурсов на создание тестов, необходимо использование автоматизированных методов и инструментов, позволяющих быстро и в большом количестве создавать тесты, проверяющие по крайней мере минимальную функциональность. Сами тесты при этом могут не проверять полную корректность соответствующих операций, но могут выявить случаи разрушения системы, а при удачном прохождении теста — гарантировать ее минимальную работоспособность.

Чтобы создать хоть и простейший, но корректный тест автоматически, необходимо иметь формальное описание правил вызова тестируемой операции. Сигнатуру операции (имя, типы принимаемых параметров и возвращаемого значения) можно полностью автоматически получить разными способами: из заголовочных или бинарных файлов или из существующих баз данных. Однако в большинстве случаев типы параметров и возвращаемых

значений в сигнатуре слишком общие. Для примера рассмотрим функцию *arcsin(double)*. Сигнатура позволяет вызывать ее на любом параметре типа *double*. Однако из математических особенностей этой функции следует, что значение фактического параметра по абсолютному значению не должно по модулю превосходить 1. Иначе возникнет исключительная ситуация, и такой вызов нельзя будет считать нормальным (в рамках тестирования минимальной функциональности). Поэтому для обеспечения корректного формирования значения параметра при вызове такой операции необходимо задать или формальное ограничение (x >= –1 && x <= 1) или просто конкретное значение, удовлетворяющее этому ограничению (например, x = 0,5). Для сложных типов данных задание правил конструирования корректного значения может приобретать более сложный вид, например для конструирования некоторых параметров необходимо вызывать другие функции или даже целые цепочки операций. Заметим, что, кроме описания свойств параметров, для целей генерации тестов следует описать еще и условие на возвращаемое значение и на объект (в случае, когда целевая операция — метод класса).

В данной работе рассматриваются необходимые расширения содержащейся в сигнатурах стандартной информации, которые позволяют полностью автоматически сформировать цепочку инициализации корректного окружения и параметров для вызова целевой операции и проверить базовую корректность ее выполнения. Центральным элементом этой расширенной информации являются *специализированные типы*, которые содержат в себе уточнение семантики стандартных типов объектов, параметров и возвращаемых значений и позволяют автоматически конструировать соответствующие корректные значения. Один специализированный тип может привязываться к различным параметрам и возвращаемым значениям различных операций, то есть переиспользоваться.

Информация из сигнатур, дополненная специализированными типами, привязанными к конкретным объектам, параметрам и возвращаемым значениям определенного набора операций позволяет полностью автоматически строить тесты работоспособности для этих операций. Однако сами специализированные типы необходимо описывать и привязывать к операциям вручную. Поэтому очень важно обеспечить эффективное создание новых и переиспользование уже созданных специализированных типов для разных операций. В связи с этим вводятся определенные механизмы для специализированных типов, позволяющие уменьшить количество создаваемых типов и увеличить скорость их привязки к операциям.

Статья состоит из четырех разделов. В первом разделе вводятся термины для обозначения атрибутов (свойств) специализированных типов. Во втором разделе рассматриваются основные атрибуты, необходимые для автоматической генерации тестов, а в третьем — свойства и механизмы, позволяющие ускорить создание тестов для большого числа операций. В

четвертой части описывается конкретная реализация механизма хранения расширенной информации об операциях и специализированных типах на примере расширения базы данных стандарта LSB [1,2].

2. Специализированные типы

Для задания специализированного типа, нужно перечислить значения всех его атрибутов. Ниже приведено краткое описание каждого из них.

- *Название* (**name**) — в нем отражена основная цель создания специализированного типа.
- *Исходный тип даных* (**type**) — тип данных, который специализированный тип уточняет.
- *Базовый тип* (**base type**) — специализированный тип, от которого данный специализированный тип унаследован.
- *Вид* (**kind**) — специализированный тип может быть нескольких видов: обычный, единожды используемый, тип по умолчанию для параметров, тип по умолчанию для возвращаемого значения.
- *Значение* (**value**) — им инициализируется параметр, использующий данный специализированный тип.
- *Ограничение* (**constraint**) — хранит условие на возвращаемое значение.
- *Инициализирующий и финализирующий код* (**init/final code**) — программный код, который необходимо вставить соответственно до и после вызова операции.
- *Вспомогательный код* (**auxiliary code**) — программный код, который оформляется в виде набора функций и используется для инициализации объектов. Кроме того, в нем имеется возможность переиспользовать уже созданные специализированные типы.
- *Прокси-значения* (**proxy-value**) — используются в случае зависимости параметров друг от друга. В этом случае один специализированный тип уточняет сразу несколько параметров. Все зависимости хранятся в прокси-значениях, каждое из которых используется одним параметром и может иметь ссылку на другие параметры.

Атрибуты *Название* и *Исходный тип данных* отличают специализированные типы друг от друга, т.е. для одного типа данных не может быть два разных специализированных типа с одинаковым названием.

Атрибут *Базовый тип* определяет отношение наследования между специализированными типами. Никаких других отношений между ними не предусмотрено.

Для описания атрибутов *Значение*, *Ограничение*, *Инициализирующий, Финализирующий и вспомогательный код* должен использоваться язык

программирования. В данной статье в качестве примера используется язык C++.

Как уже отмечалось, для задания специализированного типа нужно перечислить значения всех его атрибутов. Далее в статье, при определении конкретного типа, будут перечисляться все его непустые атрибуты. Исключение составляет его название, которое, как правило, указываться не будет, т.к. не является существенным.

Каждый из атрибутов будет более подробно описан в следующих разделах.

3. Главные атрибуты специализированного типа

В данном разделе описаны атрибуты специализированного типа (в дополнение к описанным выше атрибутам *Название*, *Исходный* и *Базовый тип*), позволяющие хранить необходимую для автоматического создания теста информацию.

3.1. Значение специализированного типа

Наиболее простым способом выполнить ограничения на входное значение параметра является явное указание значения, которое ему будет присвоено. Именно оно хранится в атрибуте "З*начение"* специализированного типа.

Приведем несколько примеров специализированных типов с допустимыми значениями:

1) value = *32*
2) value = *"Test string"*
3) value = *true*.

В сгенерированной программе этим специализированным типам соответствуют строки:

```
Par_N = 32;
Par_N = "Test string";
Par_N = true;
```

Где `N` — номер параметра, который использует специализированный тип с соответствующим значением.

Кроме того, значением может быть функция, определенная во вспомогательном коде (см. раздел 3.4) специализированного типа, например: value = `create_param()`. Ей соответствует строка:

```
Par_N = create_param();
```

Если для инициализации параметра нужно знать имя переменной объекта, то к ней можно обратиться с помощью специальной конструкции: *$obj*. В сгенерированной программе *$obj* заменится именем переменной объекта. Например, для значения `$obj.first()` будет создан следующий код:

```
SomeClass Obj;
```

```
…
OtherClass Par_N = Obj.first();
…
```

Обычно для инициализации встроенных типов — **`int`**, **`float`**, **`bool`**, **`const char*`** и т.д. — указывается конкретное значение (*10*, *5.25*, *false*, *"some string"*), а для инициализации объектов — либо конструктор, либо функция, определенная во вспомогательном коде специализированного типа.

Как правило, для параметра существует не единственное значение, с которым можно вызывать операцию, поэтому у одного специализированного типа может существовать несколько значений. Это приведет к созданию нескольких тестов для одной операции.

3.2. Ограничения специализированного типа

Кроме входных значений для параметра, специализированный тип может еще хранить некоторые ограничения на параметр. В качестве такого ограничения выступает логическое выражение, которое является истинным при правильной работе целевой операции. Как и для значения специализированного типа, для описания ограничений используются специальные конструкции:

- *$obj* — заменяется именем переменной объекта,
- *$0* — заменяется именем переменной параметра, для которого задан этот специализированный тип.

Вот несколько примеров такого вида ограничения: "`$0 == 1`", "`$0 != NULL`", "`$obj.isEmpty() == false`". В сгенерированных программах эти ограничения будут использоваться оператором условия для определения правильности работы интерфейса.

```
SomeClass Obj;
…
Obj.someMethod(Par_1, Par_2, …);
if (!(Par_1 == 1)) {
/* вывод сообщения о нарушении ограничения $0 == 1 */
}
if (!(Par_2 != NULL)) {
/* вывод сообщения о нарушении ограничения $0 != NULL
*/
}
if (!(Obj.isEmpty() == false)) {
/* вывод сообщения о нарушении ограничения
   $obj.isEmpty() == false */
}
```

В данном примере специализированный тип с первым и вторым ограничением был использован первым (Par_1) и вторым (Par_2) параметром операции, а

специализированный тип с последним ограничением — на объект. Если результат одного из логических выражений будет "ложь", то будет выдано сообщение о нарушении соответствующего ограничения.

3.3. Инициализирующий и финализирующий код

Инициализирующий код соответствует программному коду, который будет вставлен до вызова целевого интерфейса, а финализирующий — после. Также как и для ограничений, для их описания используются *$0* и *$obj*. Например, в *инициализирующем коде* можно заполнить список 10-ю элементами:

```
for (int i = 0; i < 10; i++) {
   $0.append(i);
}
```

Тогда в сгенерированной программе будет следующий код:

```
…
SomeClass Obj;
// Init code
for (int i = 0; i < 10; i++) {
   Par_1.append(i);
}
// Call of target interface
Obj.someMethod(Par_1,…);
…
```

В *финализирующем коде* можно, например, закрыть файл:

```
close($0);
```

Соответствующий сгенерированный код:

```
…
Par_1 = fopen("test.cpp", "r+");
Obj.someMethod(Par_1);
// Final code
close(Par_1);
…
```

3.4. Вспомогательный код

Набор дополнительных функций, которые можно затем использовать в других атрибутах специализированного типа (включая *Значение*), оформляется в атрибуте *Вспомогательный код*. При генерации теста, определения таких функций помещаются в начале файла с исходными кодами теста, в котором происходит вызов целевых операций, использующих данный тип. При этом, в отличие от инициализирующего кода, вспомогательный код не дублируется для различных параметров в рамках одного теста. Кроме того, в нем возможно использование дополнительных конструкций, указывающих генератору

проинициализировать переменную определенного типа или вызвать операцию со всеми автоматически проинициализированными корректными параметрами:

- *$(type)* — указание генератору создать переменную типа '*type*';
- *$[function]* — указание генератору проинициализировать и вызвать соответствующую операцию.

Эти конструкции позволяют существенно снизить время создания специализированного типа и количество ошибок, т.к. уменьшается объем кода, который должен написать разработчик. Кроме того, не нужно отвлекаться на то, чтобы узнать, как правильно инициализировать определенный тип данных или с какими параметрами запустить операцию. Не думать об этом позволяет переиспользование информации, занесенной ранее для этих операций или типов данных. Кроме того, указание таких ссылок, а не конкретного программного кода, позволяет менять способ инициализации операции или класса только в одном месте, а не во всех специализированных типах, которые их используют.

Конструкции *$0* и *$obj* здесь уже использовать нельзя, т.к. вспомогательный код не привязан к конкретному параметру. Однако при необходимости можно получить доступ к необходимым параметрам, передав их в качестве параметров определяемой во вспомогательном коде функции во время ее вызова в значении специализированного типа или в инициализирующем коде.

Часто для корректной инициализации объекта нужно вызвать несколько его set-методов. Они могут определять как простые свойства объекта (активный/неактивный, изменяемый/неизменяемый в размерах и т.д.), так и сложные, требующие инициализации других объектов (цвет, шрифт, иконка, курсор и т.д.). Например, можно написать следующий вспомогательный код и для значения соответствующего специализированного типа использовать просто строку `create_SomeClass()`:

```
SomeClass* create_SomeClass() {
   SomeClass* Obj = new SomeClass();
   Obj->setEnabled(true);
   Obj->setFont( $(QFont *) );
   Obj->setIconSet( $(QIconSet) );
   return Obj;
}
```

Здесь в методах `setFont` и `setIconSet` использовалась конструкция, указывающая генератору проинициализировать корректные значения типов `QFont*` и `QIconSet`.

Если созданный специализированный тип задать для объекта класса '`SomeClass`', то получим следующую программу:


```
static const char * const XPM[]={"16 15 8 1", "a c
#cec6bd"};

SomeClass* create_SomeClass(SomeClass* Obj) {
   Obj->setEnabled(true);
   Obj->setFont(new QFont("Times", 10, Bold));
   QPixmap Par_1_1(XPM);
   QIconSet Par_1(Par_1_1);
   Obj->setIconSet(Par_1);
   return Obj;
};

int main() {
   SomeClass* Obj = create_SomeClass();
                // Call of target interface
   …
}
```

Конструкция `$(QFont *)` развернулась в строку "`QFont("Times", 10, Bold)`", представляющую собой вызов конструктора класса `QFont`, для которого параметры были уточнены специализированными типами со значениями соответственно *"Times"*, *10* и "*Bold*". Конструкция `$(QIconSet)` развернулась в код из нескольких строчек, создающих объект `QPixmap` с помощью глобальной переменной `XPM`, определенной во вспомогательном коде специализированного типа, наложенного на параметр конструктора класса `QPixmap`. При написании этого кода вручную пришлось бы разбираться, как можно проинициализировать объекты классов `QFont`, `QIconSet` и типы, от которых они зависят (в данном случае `QPixmap`). Указания генератору создать объекты этих классов автоматически помогают сократить затраты времени и усилий на создание тестов для зависящих от них операций и избежать возникающих при этом ошибок.

3.5. Прокси-значения для комплексных специализированных типов

Обычно специализированный тип несет в себе дополнительное описание лишь одного параметра. При этом использующую только такие типы программу-тест можно разбить на независимые блоки кода, каждый из которых инициализирует свой параметр. Однако на практике существуют зависимости описаний параметров друг от друга, т.е. в строках кода для инициализации одного параметра необходимо использовать значение другого параметра. Типичным примером такого рода зависимости является ситуация, когда один параметр является строкой, а другой равен длине этой строки. В этом случае описание первого и второго параметра нельзя разделить на независимые строки кода:

```
Par_1 = "Some String";
Par_2 = strlen( Par_1 );
```

Для разрешения таких ситуаций вводятся *комплексные специализированные типы*. Фактически, они содержат в себе описание для нескольких параметров. В расширенной сигнатуре операции зависимые параметры объединяются в один, имеющий соответствующий комплексный специализированный тип (см. Рис. 1). Все зависимости между исходными параметрами остаются только внутри специализированного типа, а зависимостей между параметрами в расширенном описании операции нет. При этом конечная программа представляет собой объединение независимых описаний различных групп параметров, где специализированным типам соответствуют отдельные группы. При этом некоторые из них описывают зависимости между параметрами, как и блок кода, приведенный выше.

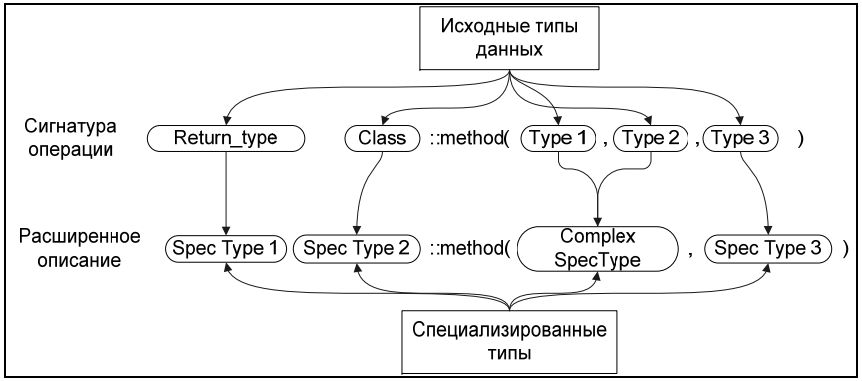

Рис. 1. Расширение сигнатуры операции

Чтобы иметь возможность объединять параметры одним описанием, специализированный тип имеет атрибуты "*Прокси-значение*". Количество этих значений равно количеству объединяемых параметров. Каждое прокси-значение хранит строку инициализации для соответствующего параметра.

В примере, описанном выше, будет создан комплексный специализированный тип, имеющий два прокси-значения: для первого параметра — "`$1 = "Some string"`", а для второго — "`strlen($1)`". При их описании была использована специальная конструкция — *$1*. Параметру, которому соответствует значение "`$1 = "Some string"`", будет присвоено значение "*Some string*", а *$1* будет везде заменяться именем переменной этого параметра. Поэтому вместо второго параметра, которому соответствует значение "`strlen($1)`", в программе будет строка "`Par_2 = strlen( Par_1 )`".

Вместо *$1* можно использовать другие конструкции, но в любом случае при описании комплексных типов должен использоваться механизм указания ссылок на другие параметры. В данном случае *$1* выступает в виде ссылки на первый параметр. Кроме того, могут быть использованы конструкции, которые, так же как и в обычном специализированном типе, заменяются именем самого параметра и именем переменной объекта (например, *$0* и *$obj* соответственно).

4. Средства упрощения создания тестов

Главным преимуществом автоматического создания простейших тестов являются небольшие затраты труда на один тест. Наибольший вклад в общую трудоемкость создания тестов вносят ручные действия, а именно: создание специализированных типов и расширение описаний операций (привязка специализированных типов к параметрам операции). Эти действия выполняет разработчик с помощью специальных инструментов редактирования информации о специализированных типах, имеющих графический интерфейс. Чтобы снизить усилия на обработку каждой операции, в специализированные типы были введены некоторые механизмы, позволяющие как снизить количество создаваемых типов, так и облегчить взаимодействие разработчика с графическим интерфейсом. В данном разделе приведено их подробное описание.

4.1. Наследование специализированных типов

При создании специализированного типа на основе некоторого исходного может возникнуть потребность в использовании некоторых атрибутов другого, уже созданного специализированного типа для родственного исходного типа. Повторно используемыми атрибутами могут являться значения, ограничения, инициализирующий, финализирующий и вспомогательный код. Чтобы была возможность воспользоваться атрибутами другого специализированного типа, был введен механизм наследования. При наследовании в атрибуте "*Базовый тип*" типа-потомка указывается тип-родитель. При использовании типа-потомка отсутствующие у него атрибуты берутся у типа-родителя.

Таким образом, специализированный тип можно создать достаточно быстро — просто указать его родителя, не указывая при этом никаких других его свойств. В этом случае все атрибуты будут браться у типа-родителя. Такое наследование полезно при создании специализированных типов для встроенных типов данных (**`int`**, **`char`**, **`long`**, **`void*`** и т.д.) для проверки возвращаемого значения. Например, чтобы иметь специализированный тип для проверки на `NULL`, нужно создать специализированный тип, основанный на **`void*`**, в ограничении которого написать условие "`$0 != NULL`", а другие специализированные типы ненулевых указателей можно от него наследовать.

Кроме простого наследования с заимствованием всех атрибутов у типа-родителя, можно также использовать более сложный вид наследования — с переопределением некоторых атрибутов в типе-потомке. Такое наследование типично для инициализации объектов, к примеру, когда значение и ограничения специализированного типа-родителя нужно сохранить, а инициализирующий код — немного изменить.

Такие механизмы как множественное (от нескольких специализированных типов) и многоуровневое (от потомка) наследования также могут быть реализованы, однако потребность в них возникает крайне редко, а их существование может привести к достаточно запутанной зависимости между специализированными типами.

4.2. Неявное наследование специализированных типов

При расширении описания операции обычный тип параметра заменяется на специализированный. В других операциях параметр такого же типа может быть заменен этим же специализированным типом. То, какой тип данных может быть заменен специализированным типом, указано у последнего в атрибуте "*исходный тип данных*". Однако, формально, типы **`int`**, **`int*`**, **`const int&`** и т.д. являются разными. Поэтому для инициализации параметров этих типов, например, единицей, следовало бы создать специализированные типы для каждого параметра. При этом, хотя потребуются значительные усилия для их создания, их смысловое содержание будет одинаковым: все должны инициализировать свой параметр единицей.

Одним из решений этой проблемы является использование механизма наследования. Например, можно создать специализированный тип для **`int`**, а для **`const int&`** создать его наследника. В этом случае создание нового типа займет меньше времени (т.к. основные атрибуты наследуются от родителя), но все равно потребуется создавать новый тип. Кроме того, для типа **`int*`** уже нельзя просто наследовать значение специализированного типа для **`int`**, т.к. у первого строка инициализации "**`new int`**`(1)`", а у второго — "`1`".

Чтобы избежать описанной проблемы, нужно иметь возможность использовать специализированный тип для параметров, типы которых являются производными от того же типа, на котором основан исходный тип этого специализированного типа. Тогда для параметра типа **`int*`** можно будет указать специализированный тип, основанный на **`int`**. Тип переменной, генерируемой при инициализации параметра, при этом будет совпадать с исходным типом специализированного типа. При использовании этой переменной в качестве параметра операции, она будет (при необходимости) приведена к нужному типу. Например, если для параметра типа **`int*`** указать специализированный тип со значением "*1*" и основанным на **`int`**, то будет создан следующий код:


```
...
int Par_1 = 1;
someFunction(&Par_1);
...
```

Такие действия можно предпринимать внутри классов эквивалентности типов, являющихся производными с помощью модификаторов **`const`**, `*` или `&`, а также с помощью операции **`typedef`**, от одного и того же типа. Данные любого типа можно привести к любому другому типу в рамках его класса эквивалентности.

Специализированный тип может использоваться вместо любого типа из класса эквивалентности его исходного типа. Это можно интерпретировать как разновидность наследования: при несовпадении типа параметра и исходного типа данных специализированного типа, происходит создание нового специализированного типа, основанного на типе параметра и наследующего этот специализированный тип. В действительности создавать новый тип не нужно — он генерируется автоматически, но из-за схожести с процессом наследования данный механизм называется неявным наследованием.

4.3. Единожды используемые специализированные типы

При расширении описания операции разработчику приходится выбирать нужный специализированный тип из списка созданных на основе типа данного параметра или результата операции. Часто этот список бывает настолько велик, что трудно найти необходимый тип. При этом в списке могут встречаться типы, каждый из которых используется только одной операцией, из-за того, что сильно привязан к ее специфике. С большой долей вероятности они больше никогда использоваться не будут. Поэтому их присутствие в списке возможных специализированных типов излишне и приводит к перегруженности этого списка. Такие специализированные типы называются *единожды используемыми* или *одноразовыми*. Это свойство указывается в атрибуте "*Вид*" этих типов.

Как ясно из его названия, одноразовый специализированный тип можно использовать только один раз. Он не появляется в списке специализированных типов, построенных для некоторого исходного.

Следует отметить, что точно определить будет ли использоваться где-нибудь еще создаваемый специализированный тип невозможно. Но, в крайнем случае, придется создать новый тип или изменить вид старого (если знать о его существовании).

В качестве примера одноразового специализированного типа можно привести тип для возвращаемого значения метода `className()` для любого класса из библиотеки Qt3 [3], например для `QPushButton`. Данный метод должен вернуть название класса, т.е. строку "`QPushButton`". В ограничении у соответствующего специализированного типа будет указано логическое

выражение: “`QString($0) == "QPushButton"`”. Исходный тип данных — **`char`**`*`. В данном случае и в случаях, подобных этому, специализированный тип должен быть одноразовым. Иначе в списке допустимых типов для *char* будет большое количество типов, проверяющих названия классов, хотя они нигде больше использоваться не будут.

4.4. Специализированные типы «по умолчанию»

Если в расширенном описании для параметра не указан специализированный тип, то работа по инициализации параметра полностью лежит на генераторе — программе, которая по расширенному описанию операции создает для нее тест. В зависимости от ситуации генератор может проинициализировать параметр конкретным значением (для простых типов), конструктором или вызовом другого интерфейса с подходящим возвращаемым значением. Но иногда такая инициализация без использования специализированных типов приводит к нежелательному результату. В этом случае нужно создать (или выбрать из уже созданных) подходящий специализированный тип. Как правило, для классов существует выделенный специализированный тип, который позволяет создать объект этого класса и задать для него некоторые общие свойства. Такой тип используется разработчиком как специализированный тип “по умолчанию”. Т.е. всегда, когда не нужно инициализировать объект каким-то особым образом, для него устанавливается этот специализированный тип. В таком случае удобно указать генератору, чтобы он при отсутствии у параметра специализированного типа считал, что его нужно проинициализировать с помощью именно этого специализированного типа. Для этого, нужный тип следует задать как *тип по умолчанию для параметров*. Это свойство указывается в атрибуте “*Вид*” соответствующего специализированного типа.

Аналогичная ситуация и для специализированных типов, проверяющих возвращаемые значения. Такой тип можно определить как *тип по умолчанию для возвращаемых значений* некоторого исходного типа. В случае если на возвращаемое значение какой-либо операции не наложено никаких ограничений, но существует тип по умолчанию для возвращаемых значений такого типа, генератор воспользуется именно им для проверки правильности возвращенного результата. Обычно такие специализированные типы проверяют общие свойства, например: “`$0.isValid() == true`”, “`$0.isEmpty == false`”, “`$0.isNull() == false`” и т.д. Такие проверки вынуждают подбирать такие параметры вызова операции, чтобы эти условия были выполнены, или указывать обычные специализированные типы, проверяющие обратные условия (“`$0.isValid() == false`”, “`$0.isEmpty == true`”, “`$0.isNull() == true`” и т.д).

В целом специализированные типы “по умолчанию” позволяют с одной стороны снизить усилия на уточнение описания операции, а с другой стороны более аккуратно нацелить тесты на нормальные сценарии использования

операций с помощью задания корректных путей инициализации объектов “по умолчанию” и автоматической вставки проверок возвращаемых значений.

4.5. Методы именования специализированных типов

Чтобы среди списка специализированных типов было удобно искать необходимый в данный момент, нужно следовать некоторым правилам при выборе названий этих типов. Иначе по названию сложно понять функции конкретного специализированного типа, и вместо повторного использования одного из имеющихся типов в большинстве ситуаций создаются новые.

В рамках нашей работы были приняты следующие принципы построения названий специализированных типов.

- *Содержательность.*
  Из названия должно быть ясно, чему равно значение, что проверяется в ограничении или какой метод объекта вызывается в инициализирующем коде.
- *Краткость.*
  Название специализированного типа не должно содержать избыточной информации. Примером такой информации может служить название исходного типа данных. Кроме того, в названии не должны быть отражены несущественные свойства уточняющего типа (базовый тип, вид и т.д.).

В зависимости от специфики исходного типа существует несколько правил для названия специализированных типов.

- Для встроенных типов (**`int`**, **`bool`**, **`float`**, **`char*`** и т.д.), как правило, существует много специализированных, которые непосредственно задают значение для соответствующих параметров. Поэтому названия этих специализированных типов совпадают с их единственным значением. Например: *1, –10, “Test String”, true*.

- Названия специализированных типов для перечислений строятся аналогично. Единственное отличие в том, что в название нужно включать пространства имен, т.к. оно не всегда совпадает с именем исходного типа соответствующего специализированного типа. Например: *Qt::AlignHCenter, QGL::SingleBuffer, QTextEdit::AutoAll*.

- Для классов специализированных типов, как правило, не много, но большинство содержат вспомогательный код, смысл которого может быть сразу неясен. Поэтому названия таких специализированных типов должны его раскрывать. В этом случае оно состоит из слова обозначающего действие, направленное на этот объект (*Create*, *Fill*, *Call* и т.д.), и некоторой информации, поясняющей это действие. Например: *Create, Create_Simple, Create_qRgb(15, 30, 200), Create_Filled_With_3_elements, Call_begin()*.

- Если в специализированном типе есть проверка на возвращаемое значение, то его название начинается с “*R_*” (что является сокращением от “*Return*”). После “*R_”* следует проверяемое ограничение или значение, равенство которому проверяется**.** В случае необходимости до “*R_*” указываются исходные данные. Пример: *R_Null, R_33, R_NotEmpty, true_R_true* (значит, подали *true* и вернуться должно *true*).

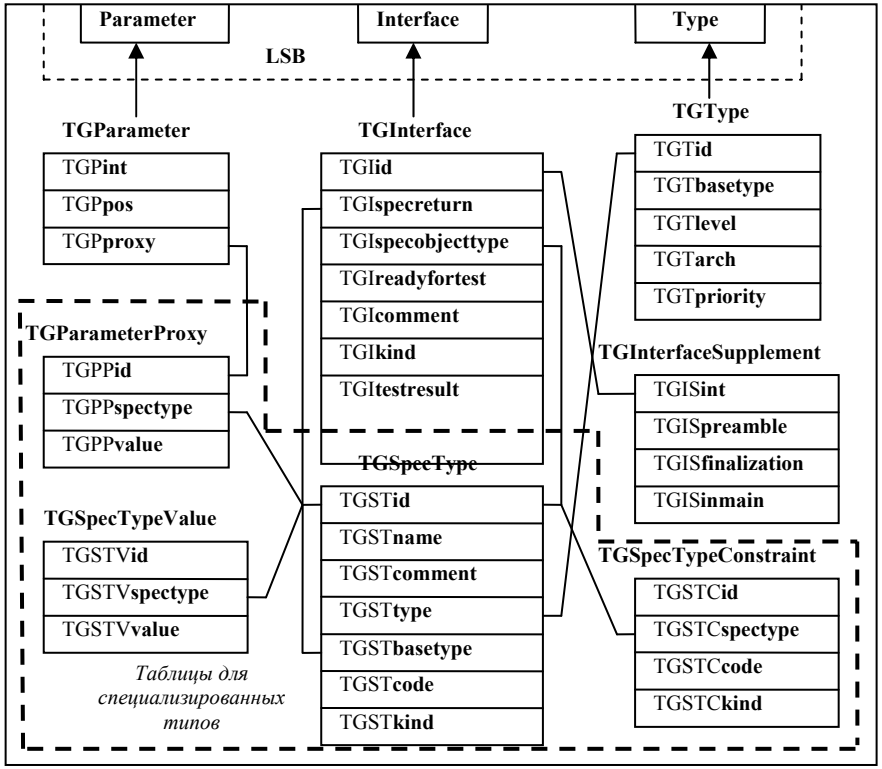

Рис. 2. Расширение базы данных LSB

5. Расширение базы данных LSB

Описанный в данной работе механизм специализированных типов был реализован в рамках проекта Linux Foundation в ИСП РАН. За основу была взята база данных LSB, содержащая необходимое описание сигнатур операций и типов данных. Она была расширена таблицами, содержащими специализированные типы и некоторую вспомогательную информацию,

необходимую для автоматического создания тестов. На Рис. 2 приведена схема этого расширения.

Расширение базы данных LSB включает в себя 8 таблиц, название которых начинается на TG (что значит Test Generation). Из них 4 таблицы хранят информацию о специализированных типах: *TGSpecType*, *TGSpecTypeValue*, *TGSpecTypeConstraint*, *TGParameterProxy* (на схеме эти таблицы выделены жирным пунктиром). Таблицы *TGParameter* и *TGInterface* служат для связи таблиц уточняющих типов с таблицами *Parameter* и *Interface* из LSB, *TGType* хранит информацию об исходных базовых типах для каждого типа данных, а *TGInterfaceSupplement* хранит код, создающий корректное окружение для вызова соответствующего интерфейса (операции). Более подробно эти таблицы описаны в разделе 5.3.

5.1. Необходимые таблицы из базы данных LSB

База данных LSB включает в себя большое число таблиц (примерно 50), но для создания тестов важны только следующие три.

- Таблица *Interface* содержит информацию об операциях стандартизованных библиотек: название операции, тип возвращаемого значения (ссылка на таблицу *Type*), содержащая ее библиотека, заголовочный файл, в котором она декларирована. Кроме того, указывается является ли операция методом класса и какого, а также свойства этого метода: конструктор ли он или деструктор, его доступность (public, private, protected) и контекст (static или не-static).
- Таблица *Parameter* содержит информацию о параметрах каждой операции: тип параметра (ссылка на таблицу *Type*) и номер параметра в сигнатуре операции.
- Таблица *Type* содержит информацию о типах: название, вид (intrinsic, struct, class, typedef, union и т.д.), заголовочный файл, в котором он определен. Эта таблица связана с таблицами *Interface* и *Parameter*.

5.2. Таблицы для специализированных типов

Специализированные типы хранятся в 4 таблицах:

1. ***TGSpecType*** — основная таблица. Она хранит основные свойства специализированного типа:
   - *TGSTid* — содержит уникальный идентификатор специализированного типа. По этому полю происходит связь других таблиц с таблицей *TGSpecType*.
   - *TGSTname* — название специализированного типа.
   - *TGSTtype* — исходный тип данных (ссылка на таблицу *Type*).
   - *TGSTbasetype* — содержит идентификатор “родителя” для данного специализированного типа (ссылку на таблицу *TGSpecType*). С помощью этого поля реализуется механизм наследования специализированных типов.

- *TGSTkind* — вид специализированного типа: *normal* (обычный), *once-only* (используемый один раз), *common for parameter* (“по умолчанию” для параметров), *common for return* (“по умолчанию” для возвращаемых значений).
- *TGSTcode* — вспомогательный код.
- TGSTcomment — содержит текстовое описание типа.

2. ***TGSpecTypeValue*** — таблица для значений специализированных типов. Главным полем является *TGSTVvalue*, содержащее значение для специализированного типа. Для одного специализированного типа может существовать несколько значений.

3. ***TGSpecTypeConstraint*** — таблица для ограничений специализированного типа. Поле TGSTCcode содержит само ограничение, а поле *TGSTCkind* — его вид: “*Normal Result*”, “*Init Code*” — инициализирующий код, “*Final Code*” — финализирующий код.

4. ***TGParameterProxy*** — служит для связи таблицы специализированных типов *TGSpecType* с таблицей параметров *TGParameter*. С таблицей *TGSpecType* она связана через поле *TGPPspectype*, которое хранит идентификатор специализированного типа, а с таблицей *TGParameter* — через поле *TGPPid*, на которое ссылается поле *TGPproxy* из таблицы *TGParameter*. При создании связи с параметром для обычного специализированного типа, таблица *TGParameterProxy* не играет никакой роли: для одного типа создается одна запись в этой таблице, на которую указывает поле *TGPproxy* из таблицы *TGParameter*. Но ее роль становиться ключевой в процессе создания комплексного типа. Если он создается для нескольких параметров, то в таблице *TGParameterProxy* появляется несколько записей, соответствующих одному специализированному типу (имеющих одинаковые значения в полях *TGPPspectype*). Причем каждая запись в поле *TGPPvalue* содержит прокси-значения, которыми будут проинициализированы соответствующие параметры.

5.3. Дополнительные таблицы

Кроме таблиц, хранящих данные для специализированных типов, в расширении базы данных LSB есть еще 4 таблицы.

- *TGParameter* — служит для связи таблицы *TGParameterProxy* с таблицей *Parameter*. Через поля *TGPint* (идентификатор интерфейса) и *TGPpos* (номер параметра в интерфейсе) она связана с таблицей *Parameter*. Поле *TGPproxy* хранит ссылку на таблицу *TGParameterProxy*. Фактически, чтобы задать для параметра специализированный тип, нужно указать в поле *TGPproxy* ссылку на прокси значение соответствующего типа.

- *TGInterface* — через поле *TGIid* связано с таблицей *Interface*. Поле *TGIspecreturn* и *TGIspecobjecttype* содержат ссылки на специализированные типы для возвращаемого значения и объекта соответственно. Остальные поля используются в процессе создания тестов. Поле *TGIreadyfortest* указывает готовность данной операции к тестированию. Поле *TGItestresult* содержит результат последнего запуска теста для данной операции. Результат может принимать следующие значения:
  - *Unknown* — неизвестен
  - *Success* — тест создался, откомпилировался и запустился без ошибок
  - *Generation failed* — произошла ошибка во время генерации теста
  - *Compilation failed* — произошла ошибка во время компиляции теста
  - *Execution falied* — произошла ошибка во время работы теста
  - Requirenment failed — было нарушено условие на возвращаемое значение

  Поле *TGIkind* определяет можно ли для текущей операции создавать тест. Его значения:
  - *Suitable for test* — годна для тестирования
  - *No data* — не хватает данных в базе данных для создания теста
  - *Not documented* — операция не имеет документации
  - Complex test — нельзя создать примитивный тест.

  Поле *TGIcomment* хранит текстовое описание проблем, из-за которых создать тест для данного интерфейса не удалось.

- *TGType* — для каждого типа данных содержит базовый тип (*TGTbasetype*), от которого данный является производным с помощью модификаторов `const`, `*`, `&`, количество модификаторов `*`, примененных при определении данного типа (*TGTlevel*), зависимость от архитектуры (*TGTarch* — некоторые типы на разных архитектурах являются производными от различных типов) и приоритет (*TGTpriority*). Приоритет важен для классов, операции которых тестируются. Он определяет порядок уточнения сигнатур операций классов. Приоритет класса, в качестве типов параметров операций которого используются другие классы, должен быть не выше приоритета этих классов. Чем выше приоритет, тем раньше нужно обработать соответствующий класс.

- *TGInterfaceSupplement* — содержит вспомогательную информацию для создания теста для некоторой операции.

- *TGISpreamble* — содержит код, который вставляется до вызова целевой операции.
- *TGISfinalization* — содержит код, который вставляется после вызова целевой операции.
- *TGISinmain* — указывает, нужно ли вызывать целевой интерфейс в функции `main()` или в отдельной, специально созданной функции (данное поле важно при тестировании некоторых библиотек, например Qt [3,4]).

6. Заключение

В данной работе была рассмотрена задача определения и представления информации, необходимой дополнительно к сигнатурам операций для автоматической генерации простых тестов, вызывающих эти операции с корректными параметрами и в корректном окружении и проверяющих отсутствие грубых ошибок исполнения. Эту информацию предложено представлять в виде так называемых *специализированных типов*, которые уточняют семантику параметров и возвращаемых значений тестируемых операций, позволяя автоматически строить инициализацию данных и окружения, необходимых для вызова этих операций. Были предложены также различные механизмы, позволяющие уменьшать количество создаваемых специализированных типов в рамках работы с группой связанных функций, что существенно снижает трудоемкость задания расширенной информации для таких групп.

Показана реализация хранения расширенной информации о специализированных типах и их привязка к данным об операциях на примере расширения базы данных стандарта LSB, реализованные в проекте LSB Infrastructure [5]. На основе этой реализации была задана расширенная информация для почти 10000 операций библиотеки Qt3 [3]. При этом было создано около 1600 специализированных типов, а производительность создания тестов, с учетом их отладки, составила в среднем 70 операций в день на человека.

Литература

[1] http://www.linuxbase.org.
[2] http://www.linux-foundation.org/navigator/commons/welcome.php.
[3] http://doc.trolltech.com/3.3/index.html.
[4] http://doc.trolltech.com/4.2/index.html.
[5] http://ispras.linux-foundation.org/.