Опыт применения технологии Azov для тестирования библиотеки Qt3

Р. С. Зыбин
phoenix@ispras.ru

Аннотация. В данной статье рассматривается задача адаптации технологии Azov для построения тестового набора, проверяющего работоспособность интерфейса библиотеки Qt3 для разработки приложений с графическим пользовательским интерфейсом. Приводится уточнение базовой методики, которое позволяет формировать корректные тесты с учетом специфики языка C++ и дополнительных возможностей тестируемой библиотеки. Вводятся расширения технологии, позволяющие ускорить работу над созданием тестового набора. Полученная методика показывает высокую эффективность при разработке простейших тестов работоспособности для сложных интерфейсов, содержащих большое количество методов и функций. Отдельно обсуждаются достоинства и недостатки технологии, выявленные в процессе ее реализации, а также указываются возможные направления ее дальнейшего развития.

1. Введение

Тестирование работы программы в различных ситуациях остается одним из самых широко используемых способов демонстрации ее корректности, особенно для достаточно сложного приложения. Но все существующие методики создания тестов, обеспечивающие их полноту в соответствии с некоторыми естественными критериями, требуют весьма значительных трудозатрат.

Однако в ряде случаев, например, при тестировании же объемных интерфейсов, содержащих тысячи операций, полное и тщательное тестирование обходится слишком дорого, а иногда и вовсе не требуется для всех элементов интерфейса системы. Вместо этого проводится тестирование работоспособности, то есть проверяется, что все функции системы устойчиво работают хотя бы на простейших сценариях использования. Уже затем, для наиболее критической части интерфейса разрабатываются детальные тесты.

Именно для решения таких типов задач предназначена технология Azov [1], созданная в 2007 году в Институте системного программирования. Технология нацелена на разработку тестов работоспособности, вызывающих тестируемые операции в правильном окружении с какими-нибудь

допустимыми значениями параметров, характеризующими простейшие сценарии использования этих операций. При этом она позволяет существенно автоматизировать создание теста.

Область применимости технологии включает те случаи, когда информация об интерфейсе доступна в хорошо структурированном, годном для автоматической обработки виде. При этом трудозатраты на создание одного теста существенно меньше, если тестируемый интерфейс содержит большое количество операций.

В частности технология Azov была использована при создании тестового набора для бинарных операций библиотеки Qt3, предназначенной для разработки приложений с графическим интерфейсом [4] и входящей в стандарт *Linux Standard Base* [2,3] (LSB). Стандарт включает в себя около 10000 методов и функций библиотеки и хранит информацию об их синтаксисе в своей базе данных.

Тестирование библиотек, написанных на языке C++, в частности Qt3 [4], является достаточно трудоемкой задачей. Поведение метода может зависеть не только от передаваемых ему параметров и глобального состояния системы, но и от состояния объекта, которому он принадлежит. Сама же библиотека графического интерфейса содержит дополнительные механизмы, которые также оказывают существенное влияние на работу методов.

Поэтому базовая методика, предлагаемая технологией Azov, должна быть дополнена набором средств, позволяющих формировать работоспособные тесты на языке C++ во всевозможных ситуациях, возникающих при тестировании элементов интерфейса, и учитывающих особенности библиотеки Qt [3], в частности механизм передачи сообщений.

2. Методика построения корректных тестов

Основной задачей рассматриваемой технологии является построение корректного теста работоспособности для каждой операции, входящей в тестируемый интерфейс.

Под корректным понимается такой тест, который обеспечивает правильную последовательность инициализации параметров целевой операции, инициализацию параметров среды исполнения, содержит непосредственный вызов целевой операции, а также освобождает по завершении задействованные ресурсы и возвращает систему в исходное состояние. Следует также отметить, что в тесте должно проверятся, выполнился ли вызов тестируемой операции успешно или наблюдается отклонение от ожидаемого поведения.

Вопросы, касающиеся инициализации и финализации (освобождение захваченных тестом ресурсов) тестовых данных относятся скорее к технологии в целом, и освещены в статье [1]. Здесь же речь пойдет об

особенностях реализации общего подхода технологии Azov при использовании языка C++ и для тестирования библиотеки Qt3.

Описанные ниже особенности обеспечиваются функциональностью инструмента-компоновщика тестов, генерирующего тесты по базе данных с уточненной информацией об интерфейсных операциях [1,5], и разработчику, как правило, не приходится о них беспокоиться.

2.1. Конструкторы и деструкторы

В языке C++ каждому конструктору на программном уровне соответствуют два различных конструктора на бинарном уровне: *in-charge* и *not-in-charge*, а каждому деструктору – два или три бинарных: *in-charge*, *not-in-charge* и, дополнительно для виртуальных классов, – *in-charge deleting*. Поскольку стандарт LSB описывает поведение системы на бинарном уровне, то тестированию подлежат все вышеперечисленные конструкторы и деструкторы.

In-charge конструкторы и деструкторы вызываются при непосредственной работе с объектом класса. *In-charge deleting* деструктор используется при удалении объекта виртуального класса из общей памяти. Таким образом, для конструктора и деструктора невиртуального класса компоновщик составляет тест, создающий объект либо в стековой, либо в общей памяти, а затем удаляющий его. В случае виртуального класса область памяти, в которой будет создаваться объект, зависит от тестируемого деструктора.

Not-in-charge конструкторы и деструкторы используются неявно при работе с объектом какого-либо класса, наследующего данному. Поэтому в тесте описывается наследник, и путем создания и уничтожения объекта полученного типа опосредованно выполняется вызов этих конструкторов и деструкторов.

2.2. Абстрактные классы

При тестировании ряда методов возникает необходимость построения объекта абстрактного класса. Следует, однако, уточнить, что непосредственно такой объект необходим только в том случае, когда тестируется его метод. В остальных же случаях можно, как правило, обойтись каким-либо объектом наследующего класса.

В первом случае необходимо провести дополнительную работу по определению чисто виртуальных методов. Реализация функциональности каждого метода является трудоемкой задачей, решения которой при проведении тестирования работоспособности хотелось бы избежать. Поэтому компоновщик тестов автоматически генерирует код, в котором определяется класс-наследник абстрактного класса, имеющий те же методы, что и родитель. Чисто виртуальные методы реализованы как заглушки, возвращающие какое-либо значение необходимого типа данных. Это значение получается по общим правилам генерации тестов, то есть может быть либо взято из

предопределенного пользователем множества, либо сконструировано компоновщиком.

Впрочем, в ряде случаев такой подход оказывается неприемлемым. Например, если заглушка используется каким-либо другим методом, вызываемым в процессе тестирования. В подобных ситуациях, если это явилось причиной падения теста, необходимо вручную реализовывать чисто виртуальные методы, основываясь, как правило, на исходных кодах одного из существующих наследников тестируемого класса.

Дополнительно, стоит отметить, что язык C++ допускает реализацию по умолчанию абстрактного метода, которая может быть использована при описании наследника. Однако в библиотеке Qt3 такая возможность не используется.

В тех же случаях, когда объект абстрактного класса выступает в качестве параметра, возможны различные варианты его инициализации. Во-первых, можно воспользоваться описанным выше способом и позволить компоновщику сконструировать объект. Во-вторых, можно доопределить абстрактный класс вручную. Однако более предпочтительным является использование объекта какого-либо класса, наследующего данному. Также могут встретиться методы, возвращающие указатель на нужный абстрактный класс. Разумеется, в действительности они возвращают указатель на объект наследующего класса. Результат работы таких методов также можно использовать в качестве значения параметра.

Аналогично, при тестировании невиртуальных методов можно не доопределять абстрактный класс, а воспользоваться экземпляром одного из наследников тестируемого класса.

In-charge конструкторы и деструкторы абстрактного класса протестировать невозможно, поскольку они предназначены для построения или разрушения объекта именно этого класса.

2.3. Защищенные методы

Использование защищенных методов также требует некоторых дополнительных построений. Область видимости таких методов ограничена методами класса, которому принадлежит защищенный метод, или его наследников.

При построении теста компоновщик генерирует описание класса, наследующего классу, содержащему защищенный метод. В нем определяется дополнительный общедоступный метод, являющийся оберткой защищенного. Параметры обоих методов совпадают, так что везде в тесте можно обращаться к построенному методу сгенерированного класса, подразумевая вызов требуемого защищенного метода.

Из этого правила имеется исключение, а именно случай, когда метод имеет в качестве одного из своих параметров объект защищенного типа данных. В рамках Qt, например, встречаются защищенные перечисления (protected

enum). В такой ситуации цепочка инициализации объекта нужного типа генерируется компоновщиком в теле метода-обертки, а сам этот общедоступный метод имеет на один параметр меньше.

Следует также добавить, что при таком подходе защищенные *in-charge* конструкторы и деструкторы не могут быть протестированы, поскольку они предназначены для работы с объектом исходного класса, а вызвать их можно только из объекта наследующего класса.

2.4. Методы, не входящие в программный интерфейс приложения

Определяя набор интерфейсных операций на бинарном уровне, стандарт LSB в некоторых случаях включает в себя операции, не являющиеся частью программного интерфейса (Application Programming Interface, API) описываемых им библиотек. Однако, при тестировании работоспособности необходимо, по возможности, протестировать даже подобные скрытые операции.

В рамках Qt3 существует ряд классов, предназначенных исключительно для использования внутри самой этой библиотеки. Описания таких классов находятся либо в заголовочных файлах, предназначенных только для сборки библиотеки, либо и вовсе в исходных кодах.

Одним из возможных подходов к тестированию методов таких классов является построение теста для метода, входящего в API, в процессе работы которого вызывается скрытый метод. Однако данный способ имеет большое количество недостатков. Во-первых, требуется весьма трудоемкое исследование того, какие методы могут быть использованы и какие значения их параметров приводят к вызову тестируемого метода. Во-вторых, отсутствует непосредственный контроль над параметрами тестового воздействия, а опосредованный контроль может быть слишком труден или даже вовсе невозможен. И, наконец, в-третьих, достаточно сложно определить, отработал ли тестируемый метод должным образом.

Учитывая все сказанное, на практике используется существенно более простое решение. Найденное описание класса выносится в отдельный заголовочный файл, который поставляется вместе с тестом. Вообще говоря, вынесение описания в отдельный файл необходимо только для классов, использующих предоставляемые Qt расширения языка C++, поскольку прекомпилятор *Meta Object Compiler*, который генерирует на их основе определения на чистом C++, обрабатывает только файлы заголовочного типа. Однако в целях стандартизации процесса подготовки необходимых для создания теста данных различий между классами, определенными на чистом языке, и классами, использующими расширения, не делается. Теперь, при подключении этого заголовочного файла к тесту, компилятор будет считать исследуемые методы частью API, и к ним можно обращаться напрямую, как и в общем случае. Данный подход также требует некоторого количества ручной работы по

поиску объявления класса, но ее объем не идет ни в какое сравнение с предыдущим вариантом.

Также в состав LSB входит множество так называемых *thunk* методов, которые конструируются компилятором и служат для обращения к виртуальным методам, декларированным в классах-предках. В силу их сугубо вспомогательной природы и отсутствия документации по ним такие операции считаются не подлежащими тестированию и исключаются из дальнейшего рассмотрения.

2.5. Среда исполнения тестируемого метода

Одним из ключевых компонентов приложения, написанного с использованием библиотеки Qt, является объект класса `QApplication`. Он содержит основной цикл обработки сообщений и служит для глобальной инициализации и финализации программы, в частности, если требуется, обеспечивает приложению доступ к графической подсистеме, задавая активный дисплей, визуальный и цветовой контексты. Также класс `QApplication` позволяет обращаться к таким параметрам системы как шрифты, палитра, интервал двойного нажатия, и параметрам, переданным приложению. Каждое приложение с графическим интерфейсом, использующее Qt, должно иметь лишь один объект этого класса.

Система Qt не позволяет создавать объекты классов, осуществляющих вывод какого-либо изображения, до тех пор, пока не проинициализирован объект `QApplication`. Существует также ряд методов, функциональность которых проявляется только после того, как программа начнет обрабатывать события, происходящие в ней самой и в операционной системе.

Поэтому компоновщик добавляет в начало каждого теста конструктор класса `QApplication`, а в конце теста вызывает метод *exec* полученного объекта, который запускает цикл обработки сообщений данного приложения.

Из этого правила существует несколько исключений. Поскольку в приложении может существовать лишь один объект класса `QApplication`, то при тестировании его конструкторов и деструкторов попытка создать дополнительный объект приведет к падению теста. Класс `QEventLoop` описывает основной цикл обработки сообщений, поэтому его объект должен быть сконструирован до объекта `QApplication`. Также, статический метод самого класса `QApplication setColorSpec()` влияет на инициализацию графической подсистемы и должен вызываться до начала ее работы, а значит до того, как будет вызван конструктор класса `QApplication`.

Разумеется, многие классы, не затрагивающие непосредственно графический интерфейс, могут работать и без привлечения объекта `QApplication`, поэтому дополнительная инициализация и вход в цикл обработки сообщений для их тестирования излишни. Однако библиотека Qt преимущественно используется для создания приложений, имеющих графический интерфейс,

поэтому подобная избыточность, напротив, приближает среду, в которой вызывается тестируемая операция, к реальной.

2.6. Сигналы и слоты

В библиотеке Qt передача сообщений между объектами приложения осуществляется посредством механизма сигналов и слотов. Вызов метода-сигнала объекта отправляет в основной цикл обработки сообщение, содержащее параметры этого сигнала, которое перехватывается и обрабатывается методом-слотом другого или того же самого объекта. Канал связи между методами определяется макросом `connect`.

Функциональность сигнала заключается в передаче слоту параметров через сообщение, неявно включая указатель на передающий объект, который может быть получен при помощи вызова `QObject::sender()` в теле слота. Именно ее и нужно проверять в процессе тестирования. В тесте формируется дополнительный объект, имеющий метод-слот с такими же, как и у сигнала, параметрами. Реализация этого метода проверяет значения пришедших параметров на эквивалентность посланным, а также неравенство нулю указателя на передающий сообщение объект, и выполняет выход из приложения. Таким образом, если сообщение не дошло до слота, то приложение не выйдет из основного цикла и будет завершено аварийно.

Слот представляет собой обычный метод, работающий с переданными ему параметрами, но может дополнительно обращаться к объекту, пославшему сообщение, посредством глобальной переменной. Поэтому не все реализации методов-слотов будут работать вне контекста передачи сообщения. Для тестируемого слота подбирается существующий в системе подходящий сигнал с таким же, как и у слота, набором параметров. В тесте конструируется объект, имеющий этот метод-сигнал, который при помощи макроса связывается со слотом. Тестовое воздействие производится путем вызова сигнала с проинициализированными параметрами.

2.7. Завершение теста и отложенное выполнение целевого воздействия

В подавляющем большинстве случаев приложение, осуществляющее обработку сообщений, завершает свою работу по получении некоторого внешнего сигнала. Например, приложение, имеющее графический интерфейс, в отсутствие исключительных ситуаций работает до тех пор, пока пользователь явно не даст команду на его закрытие.

Симуляция такого воздействия выходит за рамки тестирования работоспособности, поэтому завершение работы осуществляется тестом самостоятельно. При помощи сигнала `singleShot` класса `QTimer` через определенное время после запуска приложения вызывается слот quit класса `QApplication`, что и приводит к выходу из цикла обработки сообщений. Тесты запускаются параллельно, поэтому значительных задержек из-за

относительно большого времени жизни каждого из них в самом процессе тестирования не возникает.

Таким образом, если он завершился самостоятельно в отведенные временные рамки, и при этом не возникло ошибок, то тест считается прошедшим. Если же он не завершился в нужное время, то тест уничтожается загрузчиком, и считается, что тест не прошел.

Объекты классов-наследников `QDialog` имеют свой собственный локальный цикл обработки сообщений. Если локальный цикл получает управление до того, как будет осуществлен вход в основной цикл, то сигнал о завершении работы, испускаемый самим приложением, не будет обработан. Возникает необходимость воздействия на диалоговое окно со стороны внешнего источника. В дополнение, некоторые методы не могут работать в основном режиме до тех пор, пока не запущен механизм обработки сообщений приложения. Поэтому возникает необходимость тестировать ряд операций уже после того, как управление передано основному циклу.

Все содержимое теста, необходимое для вызова такой целевой операции, переносится в метод-слот дополнительно сгенерированного класса. С помощью сигнала `singleShot` этот слот вызывается через некоторое время после запуска приложения, но до его завершения.

3. Дополнительное обеспечение корректности тестового набора

Описанные в предыдущем разделе техники, дополнительные к базовому подходу технологии Azov, позволяют формировать простейшие тесты работоспособности для методов классов библиотеки Qt. Однако, несмотря на то, что процесс генерации тестов полностью автоматизирован, из-за очень большого объема работ неизбежно возникают несоответствия в уточняющей информации, ошибочно внесенные разработчиками, которые могут привести к ошибкам в тестовом наборе. Например, нужный специализированный тип может отсутствовать или вместо нужного может быть указан другой.

Разработчик, разумеется, просматривает и запускает тесты, над которыми он работает в данный момент, на предмет ошибок, но обнаружить их таким способом удается далеко не во всех случаях. В частности, одна из особенностей системы генерации состоит в том, что полный целевой тестовый набор генерируется каждый раз заново, и в зависимости от того, какие уточнения типов были внесены в базу данных, содержимое уже проверенных тестов может меняться.

Возникает необходимость в инструментах, позволяющих выполнять верификацию корректности сгенерированных тестов в указанном в предыдущем разделе смысле, причем в автоматическом режиме.

3.1. Проверки, добавляемые автоматически

Компоновщик тестов может гарантировать только синтаксическую корректность составляемого им кода, основываясь при этом на знании сигнатур операций. Для построения параметров, которые должны приводить к сценарию нормального использования целевой операции, этой информации недостаточно, и разработчик должен пополнять ее, внося соответствующие уточнения в виде специализированных типов. Если же при автоматической инициализации параметра был использован неподходящий специализированный тип или нужный специализированный тип не был описан на момент сборки, то полученный тест может не достигать поставленных целей.

Полностью исключить ошибочные ситуации автоматически невозможно, однако проверка промежуточных условий корректности везде, где это возможно, повышает уверенность в корректности теста в целом.

Требование, проверка которого вставляется в тесты компоновщиком по умолчанию, заключается в неравенстве нулю указателей на объекты, расположенные в общей памяти. Фактически, все возвращаемые методами при инициализации параметров указатели на объекты проходят такую проверку.

Более конкретные, и потому возможно более полезные, требования задаются разработчиками для каждого типа данных в отдельности и оформляются в виде так называемых общих специализированных типов. В качестве примера таких требований можно привести проверку свойств `isNull()` и `isValid()`, которые имеются у большого числа классов и характеризуют полноценность объекта.

Недостатком этого подхода является необходимость применения дополнительного специализированного типа там, где автоматически внесенное ограничение может быть нарушено из разумных соображений.

3.2. Контроль корректности наложенных ограничений

Дополнительные проверки, описанные в разделе 3.1, позволяют выявлять ошибки только в тех последовательностях инициализации данных, которые формируются самим компоновщиком. Однако уточнения, вносимые разработчиком, также могут быть некорректными. Обычно, код, генерируемый на основе специализированных типов, синтаксически верен и выполняет возложенные на него функции, поэтому основным источником ошибок является указание неверного специализированного типа для параметра операции.

Проверка наложенных ограничений выполняется статически при помощи дополнительного инструмента, реализованного в инструменте разработчика, который просматривает базу данных с уточненной информацией о тестируемых операциях на предмет потенциально опасных случаев. Дополнительные возможности обеспечиваются особыми правилами для

наименований специализированных типов, которые позволяют классифицировать их по назначению, например, тип, проверяющий возвращаемое значение и имеющий префикс *R_*, не должен уточнять неизменяемый параметр метода.

В частности специализированный тип, указанный для возвращаемого операцией значения, должен содержать некоторую проверку этого значения, а также иметь соответствующее название. Обратно, уточнение для параметров, которые не могут быть изменены в результате вызова целевой операции, не должно содержать проверки возвращаемого значения.

Не характерно наличие специализированных типов, уточняющих объекты, для статических методов, конструкторов и деструкторов классов, хотя иногда они необходимы для того, чтобы правильно проинициализировать среду исполнения теста. Также проверяется наличие общих специализированных типов, то есть таких, которые используются по умолчанию, для простейших типов данных, таких как **`int`**, **`bool`**, **`char`** и т.д., поскольку это приводит к массовым нарушениям ранее заданных ограничений на возвращаемые значения методов. Специализированные типы для типов более сложной организации чаще описывают правильную последовательность инициализации объекта, а не конкретное его значение, поэтому для них такая проверка не требуется.

Дополнительно имеется возможность поиска дублирующихся или очень похожих специализированных типов, у которых совпадают все поля кроме названия и быть может исходного типа, а также типов, для которых не описано никаких ограничений. Наличие таких типов иногда допустимо, поскольку они позволяют изменять тип конструируемого объекта. Это позволяет подавать в качестве параметра объект типа наследника вместо родителя, при этом, не определяя вручную инициализирующую последовательность.

Для защиты системы от случайных ошибок вводится понятие одноразовых специализированных типов. Такой тип, как правило, содержит уникальную проверку или инициализацию, применимую только к одной определенной операции и более нигде не используемую. Как следует из его названия, он может использоваться лишь однажды, и система не позволяет другим разработчиком применить его по ошибке, попутно облегчая им поиск более подходящих специализированных типов. Аналогично, одноразовыми специализированными типами не может пользоваться и компоновщик при инициализации параметров других операции.

4. Оптимизация разработки тестов

Ключевым достоинством технологии Azov является возможность достичь чрезвычайно высокой производительности при разработке тестового набора. Затраты на создание одного теста уменьшаются при разработке тестов для

интерфейсов большого размера (свыше 1000 операций). Основные трудозатраты приходятся на задачи, выполняемые разработчиком вручную, а именно изучение стандарта, формирование необходимых специализированных типов и отладка полученных тестов.

Учитывая огромное количество подлежащих тестированию операций, даже небольшое уменьшение времени разработки теста в среднем приводит к значительному выигрышу для всего тестового набора в целом. Поэтому среда разработки должна иметь гибкий и удобный интерфейс, а также обеспечивать широкие возможности переиспользования уже созданных специализированных типов данных.

4.1. Приведение типов и наследование

Специализированный тип фактически является композицией исходного типа данных и некоторого набора ограничений, то есть, по сути, состоит из достаточно независимых компонент. Эту особенность можно использовать для гибкого формирования новых специализированных типов из уже имеющихся в системе.

В текущей реализации специализированный тип имеет шесть атрибутов: исходный тип, к которому применяются ограничения, значение типа, блоки дополнительной инициализации и финализации параметра, блок проверки требований и блок вспомогательного кода. Доступ к объекту осуществляется через шаблон, разворачиваемый впоследствии компоновщиком, поэтому все атрибуты практически независимы, за исключением того, что при работе с параметром предполагается, что его тип соответствует исходному типу данных.

В системе реализовано однократное наследование специализированных типов. Каждому типу может быть сопоставлен один родитель. Если тип-наследник не содержит описания какого-либо из блоков, то используется описание этого блока у его непосредственного предка. Однако если последний в свою очередь заимствует соответствующее описание у своего предка, то в результате блок наследника окажется пустым. При таком подходе создавать цепочки наследования не имеет практического смысла, и в процессе разработки это, как правило, и не требуется. С другой стороны весьма полезно иметь некий базовый специализированный тип, обеспечивающий основную инициализацию, а уже на его основе строить типы, проверяющие требования. Тогда, при необходимости изменить инициализирующий код, это нужно будет сделать только в одном месте.

Существующий механизм полезно в дальнейшем дополнить наследованием от двух родителей с одинаковым исходным типом одновременно. От одного родителя можно будет взять инициализирующий код, от второго – проверку возвращаемого значения, а в самом наследнике определить специфичную для тестируемой операции инициализацию и финализацию. Это позволит несколько сократить время разработки теста, а также изменять меньшее

количество специализированных типов при необходимости внесения поправок.

Специализированные типы для производных типов, таких как указатели, ссылки, синонимы (typedef), константы (const), а также конструкции более сложной структуры, можно создавать автоматически с помощью механизма неявного наследования. Производные одного типа образуют класс эквивалентности в том смысле, что ограничения специализированного типа для одного из них могут быть применены к любому другому типу из этого множества. Дополнительные поправки в цепочку инициализации параметра автоматически вносятся компоновщиком при сборке теста. С точки зрения пользователя системы это выглядит так, как будто для всех параметров, имеющих типы из одного класса эквивалентности, существуют специализированные типы одинакового назначения.

В качестве направления дальнейшего развития этого механизма можно предложить заимствование иерархии наследования самого языка C++. То есть, например, специализированный тип, уточняющий объект класса-наследника, может использоваться при инициализации параметра, имеющего тип класса-родителя.

Также полезно ввести неявное наследование для произвольных определяемых пользователем групп классов. В частности, целесообразно переиспользовать специализированные типы данных для классов, имеющих похожие методы, например, классов-наследников одного и того же родителя, немного по-разному реализующих сходную функциональность.

4.2. Удобство и функциональность интерфейса

Инструмент разработчика тестов в первую очередь предназначен для доступа к базе данных с дополнительной информацией о тестируемых операциях, а именно позволяет создавать и редактировать специализированные типы, просматривать состояние работ в целом и для каждой операции в частности, а также собирать статистические данные. Скорость разработки тестового набора существенно зависит от удобства этого инструмента и его функций, позволяющих сократить время на выполнение рутинных задач.

Написание теста можно существенно упростить, если предоставить разработчику легкий доступ к справочной и отладочной информации. Поэтому каждая операция сопровождается ссылкой на ее описание в стандарте. Инструмент также позволяет просматривать сгенерированный компоновщиком код, запускать тесты на целевой машине и получать подробную информацию об ошибках, возникших на этапах генерации, компиляции и исполнения.

Возможны ситуации, когда несколько операций реализуют сходную функциональность и при этом имеют одинаковую сигнатуру, так что набор специализированных типов, которыми необходимо уточнить параметры, для них один и тот же. Разумно определить уточнения только для одной из таких

операции, а затем автоматически их дублировать. Для этой цели существует инструмент импортирования, который позволяет производить поиск таких случаев сразу для всех методов класса и выбирать источник копирования из списка возможных кандидатов. Дополнительно, с помощью него можно автоматически создавать копии существующих специализированных типов с изменением типов уточняемых ими параметров.

Быстрому поиску подходящих для подстановки специализированных типов служит система их названий. Название должно нести в себе краткую информацию об ограничениях, содержащихся в типе, например, содержать начальное значение параметра, если таковое имеется, или префикс *R_*, если выполняется проверка возвращаемого значения. Общеупотребительные типы принято называть с помощью имен, начинающихся с *Create*, в этом случае инструмент импортирования в одном из режимов работы дублирует их автоматически.

Ряд специализированных типов несет в себе уточнения, имеющие смысл только для одной определенной операции. Для того чтобы облегчить поиск подходящих типов, а также предотвратить их ошибочное переиспользование, такие специализированные типы отмечаются как одноразовые и скрываются из списка доступных. Возможна и обратная ситуация, когда требуется указать один и тот же специализированный тип для всех вхождений какого-либо типа данных, за исключением единичных случаев. Такой тип объявляется общим и используется при генерации тестов по умолчанию, если явно не указан другой.

При создании инициализирующей последовательности вызовов для специализированного типа ее параметры не обязательно конкретизировать. В этом случае там, где это нужно, указывается символическая конструкция, на место которой компоновщик подставляет правильно построенный объект, возможно используя при этом какой-нибудь специализированный тип. В некоторых случаях процесс описания специализированного типа можно упростить, используя шаблоны кода для часто встречающихся случаев. Например, применяется вставка вызовов всех методов, проставляющих атрибуты объекта, названия которых в Qt начинаются на *set*.

4.3. Разбиение на группы и порядок выполнения работ

В процессе разработки пользователь может столкнуться с необходимостью проинициализировать параметр типа, для которого еще никто не составил подходящих специализированных типов. В этих случаях разработчику требуется отвлечься на время от текущей задачи, изучить соответствующий раздел стандарта и описать нужные уточнения для типа данных или подождать, пока другой разработчик сделает это, что зачастую занимает весьма значительное время. При разработке тестов для большого числа операции постоянное отвлечение на другие задачи сильно замедляет создание

тестов, поскольку разработчику придется позже восстанавливать свои знания об отложенной на время операции.

Для того чтобы минимизировать время на переключение контекста в процессе разработки, используется инструмент построения расписания работ. Множество операций, для которых создается тестовый набор, разбивается на группы в соответствии с их назначением, что позволяет разработчикам изучать один и тот же раздел стандарта сразу для целой группы функций или классов. Затем для каждой группы и для каждой операции внутри группы определяется приоритет, согласно которому следует вести разработку. Применительно к C++ удобнее оперировать не в терминах отдельных операций, а в терминах целых классов.

Внутри группы каждому классу ставится в соответствие вес, равный разности между количеством классов этой же группы, имеющих методы, в которых этот класс выступает в качестве параметра, и количеством классов группы, которые используются в его методах. Чем больше вес класс, тем раньше следует создать уточняющего его специализированные типы. На практике это означает, что более высокий приоритет имеют низкоуровневые и вспомогательные классы, которые затем используются более сложными.

На уровне групп веса рассчитываются таким же образом, с учетом всех входящих в группу классов. Однако здесь приоритеты имеют более общий смысл и отражают то, насколько часто в тестируемом интерфейсе используется та или иная подсистема.

5. Достоинства и недостатки подхода

Описанная в предыдущих разделах реализация системы разработки тестов, основанная на технологии Azov, была применена при создании тестового набора для интерфейса библиотеки Qt3. Тестированию подлежало около 10000 операций, поведение которых описывается стандартом LSB. При этом практически во всех случаях удалось построить работоспособные простейшие тесты. На основе полученного опыта можно сделать выводы о преимуществах и недостатках технологии в целом и данной ее реализации.

Технология построения простейших тестов работоспособности действительно позволяет создавать тестовые наборы, решающие поставленную задачу для интерфейсов большого размера. Эффективность ее применения возрастает в тех случаях, когда интерфейс содержит небольшое количество существенно различных типов данных по сравнению с общим числом операций в нем. Это можно увидеть на примере подсистемы работы с базами данных *QSql*, которая содержит 447 операций из LSB (501 всего), 20 собственных классов и 7 общеупотребительных типов данных. При этом работающие тесты можно построить даже при отсутствии документации на часть операций, что для библиотеки Qt более чем актуально.

В результате расширения базовой методики технологии Azov простейшие тесты удалось создать для приблизительно 99,5% объема входящей в LSB части интерфейса Qt. Лишь около 35 операций не получилось протестировать по причине простоты тестов или критической нехватки документации. При этом в процессе описания ошибок проявилась неожиданная сложность. Дело в том, что при отсутствии сколько-нибудь подробной документации чрезвычайно трудно отличить случаи неверного использования операции от случаев обнаружения настоящих ошибок в ее реализации. Фактически для принятия решения в этих случаях приходится изучать исходный код операций.

В существующем виде технология не предполагает реализацию зависимости между параметрами операции и возвращаемым ею значением. Поэтому в общем случае допустимы только проверки общего вида (например, метод должен вернуть неотрицательное число), которые далеко не всегда можно указать. Однако в целях повышения аккуратности тестового набора в тех случаях, когда указаны конкретные значения параметров, влияющие на результат, при помощи специализированного типа проверяется равенство результата операции конкретному правильному значению.

Построение автоматических сценариев путем варьирования значений параметров операций может усложнить проверку возвращаемых значений. Вероятно, основные возможности для улучшения тестов заключаются в задании некоторого множества значений для каждого специализированного типа и построении отображений между исходными данными одних и проверками других.

Что касается среды разработки тестов, то она получилась достаточно удобной и простой в использовании. Ее изучение сводится к прочтению нескольких страниц руководства и небольшой самостоятельной практике.

К наиболее существенным недостаткам интерфейса относится отсутствие возможности выполнить какие-либо действия между вызовом целевой операции и проверкой возвращенного ею значения. Такие случаи возникают, когда перед получением результата надо произвести дополнительные действия над объектом целевого воздействия. Приходится добавлять такие вызовы непосредственно в условие проверки правильности результата.

Из-за особенностей механизма неявного наследования в некоторых случаях приходится создавать специализированные типы, в которых тип фактически инициализируемых данных не соответствует уточняемому типу. Подобная ситуация возникает, например, когда требуется использовать в качестве параметра объект класса-наследника вместо базового класса, указанного в сигнатуре операции. При этом корректное приведение типов моделируется компоновщиком неявно на основе эвристики, что запутывает разработчика и усложняет процесс генерации теста.

В том случае, когда для нормальной работы операции необходима явная зависимость между ее параметрами, технология Azov предлагает

использование комплексных специализированных типов, которые совместно уточняют сразу несколько параметров. Было реализовано сразу два способа задания таких конструкций. Во-первых, можно явным образом создать особый тип, уточняющий типы сразу нескольких параметров операции. Однако получающаяся конструкция выглядит слишком искусственно, поскольку зависимости между параметрами записываются чаще всего при помощи глобальной функции или переменной. Второй вариант – объединить два обычных специализированных типа путем указания в них ссылок друг на друга Недостатком в данном случае неочевидная зависимость между типами, поскольку используются ссылки на номер параметра операции.

Возникает потребность обобщить модель специализированных типов с учетом особенностей разных языков программирования, по меньшей мере, реализовать более полное явное и неявное наследование, а также формализовать и улучшить использование ссылочных отношений, указывающих как на номер параметра, так и на другой тип данных.

6. Заключение

В статье описываются техники, позволяющие адаптировать общий подход технологии Azov для создания тестов работоспособности для классов языка C++ и библиотеки Qt3. Большинство из таких техник могут быть использованы и в других системах.

На их основе была реализована система генерации тестов, позволившая с приемлемыми усилиями построить тестовый набор для 10000 методов и функций из библиотеки Qt3, описанных в стандарте LSB. При этом, несмотря на неполноту документации, производительность разработки тестов составила в среднем 70 операций в день на человека.

При помощи простейших тестов работоспособности удалось обнаружить порядка 10 ошибок в реализации методов. В процессе разработки тестового набора удалось обнаружить и исправить достаточно большое количество ошибок в базе данных LSB, а также пополнить ее недостающей информацией.

Таким образом, показана применимость на практике и высокая эффективность технологии Azov, пополненной набором описанных в данной работе техник.

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] http://www.linuxbase.org.
[3] http://www.linux-foundation.org/navigator/commons/welcome.php.
[4] http://doc.trolltech.com/3.3/index.html.
[5] А. Пономаренко, Е. Чернов. Алгоритм генерации тестов работоспособности на основе расширенной базы данных LSB. Опубликовано в этом же сборнике.