Технология Azov автоматизации массового создания тестов работоспособности

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

Аннотация. В статье рассказывается о технологии Azov, предназначенной для массового создания тестов работоспособности сложных программных систем. Данная технология основана на использовании базы данных со структурированной информацией об операциях тестируемой системы и методики пополнения этой информации за счет уточнения ограничений на типы параметров и результатов операций. Представленные результаты практической апробации технологии Azov демонстрируют ее высокую эффективность при разработке простейших тестов для систем с большим количеством функций.

1. Введение

Большая сложность используемых сегодня программных систем и важность решаемых ими задач требуют аккуратной и систематической проверки их корректности в смысле соответствия требованиям и стандартам. Для такой проверки чаще всего используется тестирование — наблюдение за работой системы в ряде специально создаваемых ситуаций и анализ правильности ее работы в каждой такой ситуации с учетом всех существенных аспектов ее поведения.

Большинство имеющихся на сегодняшний день методов тестирования требует серьезных затрат для обеспечения некоторых гарантий адекватности или полноты проводимой проверки. То есть, важно, чтобы оценка корректности системы, вынесенная на основе небольшого количества тестовых экспериментов, была верна и по отношению к ее работе во всех возможных ситуациях, которых для практически важных систем бесконечно много. Обычно для этого проводится классификация ситуаций, которые могут возникнуть при работе системы, и организуются тесты на каждый выделенный вид ситуаций, в ходе которых выполняются все необходимые проверки.

При автоматизации тестирования чаще всего автоматизируется лишь выполнение тестов. Автоматизация их создания связана с необходимостью формализации как критериев правильности работы тестируемой системы, так

и правил классификации тестовых ситуаций и техник формирования входных данных как представителей получаемых классов ситуаций. Поскольку для подавляющего большинства систем все эти критерии и правила заданы неформально, такая автоматизация может потребовать существенных дополнительных затрат на их полную формализацию.

В ряде ситуаций такие затраты неоправданны, так как результаты тестирования не должны демонстрировать высокую надежность и полноту проверки выделенных видов тестовых ситуаций. Таково, например, *тестирование работоспособности*, при котором проверяется только, что основные функции системы выполняются более-менее правильно, т.е. система не разрушается и возвращает результаты, проходящие простейшие проверки на корректность (полная проверка при этом не выполняется). При тестировании работоспособности библиотек проверяют каждую библиотечную операцию (каждый общедоступный метод каждого класса) на наиболее простом сценарии ее использования, проверяя выполнение базовых ограничений на результат работы. Скажем, при тестировании работоспособности реализации функции $sin(x)$ можно вызвать ее со значением параметра 1.0, убедиться, что никаких исключений не возникло, и проверить, что результат лежит на отрезке [–1, 1]. Понятно, что таким образом проверяется только, что данная реализация не делает что-то уж совсем неправильное.

Тестирование работоспособности используется, чтобы убедиться в том, что тестируемая система устойчиво работает в простейших сценариях использования. Оно часто проводится для проверки корректности очередной сборки системы, до выполнения более систематического и аккуратного тестирования, поскольку последнее требует значительно больше времени, но бессмысленно, если новая версия тестируемой системы не в состоянии справиться даже с простейшими задачами. Например, аккуратное тестирование функций интерфейса POSIX, управляющих работой потоков, требует разработки достаточно сложных сценариев, в ходе которых создаются многочисленные потоки с различными характеристиками, и в рамках некоторых из этих потоков выполняются какие-то специфические действия. Но использовать такие тесты бессмысленно и неэффективно, если сама функция создания потока содержит ошибку, делающую невозможным создание нового потока при ее обычном использовании. Затраты времени на выполнение сложных тестов и выяснение природы обнаруженной ошибки при этом будут слишком велики, а простейший тест, проверяющий, что новый поток действительно создается и выполняет хотя бы одну простейшую операцию, позволит существенно сократить эти затраты.

Таким образом, тестирование работоспособности позволяет экономить усилия, затрачиваемые на поиск и локализацию достаточно грубых ошибок в крупных и сложных программных системах.

Как можно автоматизировать создание тестов работоспособности? Казалось бы, что такая автоматизация потребует формализации существенной части требований и критериев выбора тестов как сценариев «нормальной» работы проверяемых операций, для чего необходимо потратить довольно много усилий. Например, функция, предназначенная для открытия файлов, в ходе теста на самом деле должна открывать некоторый файл, а не ограничиваться выдачей кода ошибки. В то же время качество тестов работоспособности довольно низкое, а в результате аналогичных усилий, потраченных на разработку традиционных тестовых вариантов, можно получить более строгие и полные тесты без проведения формализации.

Такая оценка в общем случае верна, но иногда возникают дополнительные факторы, позволяющие по новому взглянуть на возможность и необходимость автоматизации тестирования работоспособности. Первый такой фактор — размер и сложность тестируемой системы, определяемые на основании общего количества интерфейсных операций и сложности реализуемых ими функций. Если операций очень много, разработка тестов для них всех вручную становится слишком трудоемкой. Например, в стандарте POSIX 2004 года описано 1123 функции [1], многие из которых решают весьма сложные задачи, а в библиотеке Qt для разработки приложений с графическим интерфейсом пользователя [2], входящей в стандарт Linux Standard Base [3], насчитывается около 10000 общедоступных методов классов и глобальных функций.

Другой фактор, который может сделать автоматизацию создания тестов работоспособности более перспективной, — это наличие достаточно полной и хорошо структурированной информации об интерфейсах тестируемой системы. Если такая информация уже есть, она может быть использована для автоматической генерации элементов тестов или их заготовок.

Оба этих фактора — большой размер и наличие базы данных с информацией об интерфейсных операциях — имеются у стандарта Linux Standard Base [3] (LSB), включающего несколько отдельных стандартов (POSIX [1], ISO C [4], Filesystem Hierarchy Standard [5] и пр.) и библиотек (Xlib [6], OpenGL [7], GTK+ [8], Qt [2]). В целом в LSB версии 3.1 входит более 30000 функций и методов. Для автоматизации работ по поддержке в актуальном состоянии текста стандарта и набора инструментов для выполнения различных проверок синтаксическая информация обо всех входящих в стандарт интерфейсах помещена в единую базу данных. Это позволило использовать новый подход для автоматизации создания тестов работоспособности для LSB.

2. Технология автоматизации создания тестов работоспособности

Для тех случаев, когда интерфейс системы очень велик (состоит из тысяч различных операций) и большая часть информации об элементах этого

интерфейса хранится в хорошо структурированном, подходящем для автоматической обработки виде, можно предложить достаточно эффективную технологию автоматизации построения тестов работоспособности, позволяющую создавать их массово, в большом количестве. Такая технология, названная Azov, была разработана в 2007 году в Институте системного программирования РАН.

Эта технология существенно использует базу данных с информацией об операциях тестируемой системы и предполагает некоторое уточнение этой информации, в основном сводящееся к уточнению (специализации) типов параметров операций и их результатов. Такое уточнение выполняется вручную и позволяет в дальнейшем полностью автоматически построить корректные входные данные для каждой операции и проверить некоторые свойства ее результата.

Технология Azov включает следующие элементы.

- Методика уточнения данных об интерфейсных операциях.
- База данных с расширенной информацией о тестируемых операциях.
- Инструменты, использующиеся для внесения дополнительной информации в базу данных операций.
- Компоновщик тестов работоспособности, автоматически генерирующий набор тестов для выделенного множества операций по имеющейся в базе информации.

Основная идея технологии следующая: информация о типах параметров и результатов тестируемых операций уточняется таким образом, чтобы позволить сгенерировать значения параметров для простейших сценариев нормального использования этих операций и выполнить некоторые (далеко не полные) проверки корректности их результатов. Поскольку одни и те же типы данных в больших системах используются многократно, массовая генерация тестов может привести к существенному снижению трудозатрат на каждый тест.

2.1. Исходные данные и ожидаемые результаты

Исходными данными для выполнения работ по описываемой технологии являются база данных, содержащая структурированную информацию об операциях тестируемой системы, и документация на эти операции. Предполагается, что все операции являются функциями языка C или методами классов C++.

Структурированность информации об операциях означает, что в этой базе типы параметров и результатов операций должны присутствовать как отдельные сущности, связанные по ссылкам с соответствующими операциями. Более точные требования к исходной информации см. в разделе *2.4.1. Структура базы данных*.

Документация на тестируемую систему должна содержать достаточно информации, чтобы выявить основные сценарии работы всех интерфейсных операций, включая источники данных для аргументов вызова операции и базовые ограничения на возможные результаты при ее правильной работе.

Результатом работы по предлагаемой технологии является набор тестов работоспособности для всех операций тестируемой системы. Для каждой из них в этот набор входит тест, вызывающий операцию в рамках одного из сценариев ее нормальной работы, не приводящей при корректной работе системы к сбоям, исключительным ситуациям или возврату кода ошибки. Этот тест также проверяет некоторые ограничения на результат такого вызова. Все аргументы вызова должны быть построены корректным образом, и, при необходимости, должны быть сделаны предварительные вызовы других операций, инициализирующие необходимые внутренние данные системы, а также итоговое освобождение захваченных ресурсов.

Информация в базе данных об интерфейсных операциях и связанных с ними типах пополняется так, чтобы стала возможной автоматическая генерация таких тестов.

2.2. Организация работ по технологии Azov

Создаваемые по описываемой технологии тесты представляют собой тестовые варианты, т.е. программы, в рамках которых последовательно выполняются вспомогательные действия по подготовке системы к работе, инициализируются значения параметров для вызова тестируемой операции, выполняется этот вызов и производится финализация системы, т.е. освобождение ресурсов, которые должны быть освобождены по окончании работы. Кроме того, проверяются базовые ограничения на результаты всех выполняемых в тесте вызовов операций.

Разработка тестов в рамках технологии Azov организована следующим образом.

- ***Разбиение набора операций на функциональные группы.***
  Интерфейс системы делится на группы операций, работающих с одними и теми же внутренними данными, и предоставляющих полный набор действий над ними. Это необходимо, прежде всего, для разбиения работ на достаточно независимые части, каждую из которых может выполнять отдельный разработчик.

- ***Уточнение информации об интерфейсных операциях в базе данных.***
  На этом этапе разработчики анализируют документацию на операции выделенных им групп, определяя условия их нормальной работы и ограничения на их результаты при такой работе. Дополнительная выявляемая при этом информация заносится в базу данных в виде специализированных типов параметров и результатов операций,

возможных значений этих типов, а также в виде действий, необходимых для инициализации или уничтожения данных такого типа или же для инициализации или финализации внутренних данных системы, необходимых для нормальной работы операций.
В ходе работы применяется методика уточнения данных, представленная ниже.
Для заполнения базы данных используются вспомогательные инструменты, например, использующие Web-интерфейс и позволяющие осуществлять навигацию по базе данных, поиск в ней разнообразной дополнительной информации и редактирование уточняемых данных.
Перед проведением уточнения необходимо упорядочить тестируемые операции и типы данных, связанные с ними так, чтобы операции, имеющие более сложные типы параметров, шли позже тех, которые имеют простые параметры и могут быть использованы для получения данных более сложных типов. При этом уточнение выполняется от простых операций к более сложным без необходимости часто переключаться на анализ других операций и может естественным образом многократно использовать на поздних этапах информацию, выявленную на ранних.

- ***Контроль качества проведенного уточнения.***
  Введенная информация должна пройти проверку на корректность. Такая проверка выполняется как за счет ее просмотра и дополнительного анализа, проводимого другими разработчиками, так и с помощью отладки сгенерированных тестов — они должны компилироваться и собираться, а все проблемы, возникающие при их выполнении должны быть следствием ошибок в тестируемой системе, а не в тестах.

- ***Генерация тестов.***
  Итоговый набор тестов работоспособности генерируется при помощи компоновщика тестов на основе информации из пополненной базы данных.

- ***Выполнение тестов.***
  Сгенерированный набор тестов может представлять собой одну программу или набор программ на языке C или C++. В последнем случае они выполняются в пакетном режиме.

- ***Анализ результатов тестирования.***
  По итогам каждого теста выдается информация либо о его успешном выполнении, либо о нарушении одного из проверяемых ограничений, либо о разрушении тестируемой системы во время его работы. В случае какого-либо нарушения обнаруженная ошибка анализируется разработчиком, и по итогам этого анализа либо фиксируется как

ошибка в тестируемой системе, либо исправляется как ошибка в тесте, после чего тест выполняется снова.

Рис. 1. Схема выполнения работ по технологии Azov.

2.3. Методическая основа технологии

Методическая основа технологии Azov включает технику уточнения информации об интерфейсных операциях и типах их параметров и результатов в базе данных, а также процедуру автоматической компоновки теста на основе уточненной информации.

Уточнение информации об интерфейсных операциях и типах их параметров сводится к следующим действиям.

- ***Уточнение (специализация) типов.***
    - Если при нормальном вызове операции в качестве ее аргумента может быть использовано только значение из

определенного множества, для соответствующего параметра определяется специализированный тип-перечисление, значениями которого являются элементы этого множества.

- Если при нормальном вызове операции в качестве ее аргумента (или объекта вызова, если эта операция является методом класса) можно использовать лишь значение, являющееся результатом другой операции, то определяется специализированный тип, который одновременно указывается как тип данного аргумента первой операции и как тип результата второй операции.
- Если при нормальном вызове операции в качестве ее аргумента (или объекта вызова, если эта операция является методом класса) можно использовать лишь значение, для которого предварительно были вызваны некоторые другие операции, для соответствующего параметра определяется специализированный тип, с которым связывается программный код инициализации его значения с помощью необходимых операций.
- В тех случаях, когда использование аргумента (или объекта вызова) предполагает последующее освобождение ресурсов, с его специализированным типом связывается код финнализации его значения, выполняющий это освобождение.
- Для описания действий по инициализации и финализации отдельных объектов данного типа иногда требуется ввести вспомогательные операции, которые должны быть один раз определены в каждом тесте, где используется такой тип. Такой код также оформляется как дополнительный атрибут специализированного типа.
- При уточнении типов параметров иногда несколько параметров объединяются в один абстрактный объект, разные элементы которого используются в качестве их значений, и определяется специализированный тип такого составного объекта.
  Например, если параметрами операции является указатель на начало строки типа `char*` и ее длина, можно определить специализированный тип «строка». Вместо первого параметра тогда нужно задавать указатель на первый элемент строки, а вместо второго — результат применения функции `strlen()` к этому указателю.
  При создании таких комплексных специализированных типов

определяется код для получения значений отдельных параметров из комплексного объекта.

- Если при нормальной работе операции ее результат всегда удовлетворяет некоторым ограничениям, например, возвращается непустой список или возвращается целое число, большее 0, для ее результата определяется специализированный тип, связанный с соответствующим ограничением.
- При уточнении типов связи между операцией и исходным типом ее параметра или результата в базе данных дополняются аналогичными связями с соответствующим уточненным типом.
- Каждый раз при необходимости введения специализированного типа разработчики анализируют уже имеющиеся специализированные типы, чтобы по возможности использовать повторно уже имеющееся определение типа с нужным набором ограничений.

- ***Определение инициализации и финализации для операций.***
  Если для нормальной работы операции необходимо предварительно выполнить некоторые действия для инициализации внутренних данных системы, и/или провести их финализацию после вызова, то соответствующий вспомогательный код инициализации и финализации привязывается к данной операции.

- ***Определение значений типов параметров.***
  Из возможных типов параметров (в том числе специализированных) выбрасываются примитивные или производные от других типов (указатели, ссылки и пр.) и такие типы, значения которых можно получить только в результате вызовов специальных операций или конструкторов. Кроме того, выбрасываются те типы, любое значение которых может быть использовано как значение параметра этого типа при нормальном вызове произвольной операции с таким параметром. Для каждого из оставшихся типов определяется некоторое значение, которое используется в качестве значения параметров соответствующего типа при вызове операций. Код для получения этого значения заносится в базу данных.

После проведенного уточнения можно применить достаточно простую стратегию компоновки теста, которая позволяет по внесенной в базу данных уточняющей информации автоматически построить тесты работоспособности для всех интерфейсов.

Основная процедура построения теста для заданного интерфейса выглядит так.

- В начало теста вставляется код инициализации для работы данной операции.

- Затем строятся значения всех ее параметров. Для каждого типа аргумента вычисляется значение его порождающего типа (базового типа для указателей, ссылок и пр.), которое затем преобразуется в значение аргумента.
  - Если значение типа определено явно, используется это значение.
  - Если для вычисления значения нужно вызвать другую операцию, конструктор или выполнить инициализирующий код, вставляется вызов этой операции или соответствующий код. При этом значения параметров вызываемых в этом коде операций либо фиксированы, либо вычисляются по аналогичной процедуре.
    Определения вспомогательных операций, необходимых для построения значений, при этом вставляются в начало теста.
  - Значения других типов строятся автоматически. Для примитивных типов (числа, символы, строки) используются некоторые простые генераторы значений. Значения производных типов — указателей, ссылок, массивов и пр. — строятся из значений их базовых типов. Для перечислений используется первое возможное значение. Объекты структурных типов строятся по их полям, причем поля заполняются по этой же процедуре.

- Вставляется вызов тестируемой операции с построенными аргументами.

- Затем вставляется код финализации для всех построенных значений, для которых это необходимо.

- В конце вставляется код финализации после работы тестируемой операции.

- Для всех вызовов операций, использованных в коде, вставляются проверки ограничений на их результаты, указанные в соответствующих специализированных типах. Кроме того, для всех указателей, которые возникают при вызовах и используются в дальнейшем, проверяется их неравенство `NULL`.

Несколько более сложные действия выполняются при построении тестов для защищенных методов классов, которые нельзя вызвать из произвольного места. В этом случае генерируется класс, наследующий класс, в котором определен тестируемый метод. В генерируемом классе определяется общедоступный метод, являющийся оберткой унаследованного защищенного

метода. В рамках построенного теста вызывается именно этот общедоступный метод.

Дополнительная работа необходима для построения значения типа, являющегося абстрактным, неполностью определенным классом. В этом случае, если нет возможности использовать объект одного из наследников этого абстрактного класса, такой класс-наследник генерируется. В нем все абстрактные (чисто виртуальные) методы определяются простейшим образом, и в качестве значения нужного типа используется объект этого класса-наследника.

2.4. Пример построения тестов по технологии Azov

В данном разделе ряд элементов методики построения тестов в рамках описываемой технологии проиллюстрированы на примере функций работы с хранителем экрана (screen saver) в библиотеке Xlib, входящей в LSB.

Всего в этой библиотеке 5 таких функций.

- `int XSetScreenSaver(Display*, int, int, int, int)` — устанавливает режим работы хранителя экрана для заданного дисплея.
- `int XGetScreenSaver(Display*, int*, int*, int*, int*)` — возвращает текущие параметры работы хранителя экрана для данного дисплея, значения возвращаются по параметрам-указателям, соответствующим параметрам предыдущей функции.
- `int XForceScreenSaver(Display*, int)` — активирует или дезактивирует хранитель экрана для заданного дисплея в зависимости от указанного второго параметра.
- `int XActivateScreenSaver(Display*)` — активирует хранитель экрана для заданного дисплея.
- `int XResetScreenSaver(Display*)` — дезактивирует хранитель экрана для заданного дисплея.

Документация [6] на функции библиотеки Xlib дает следующее уточнение интерфейса.

- Второй и третий параметры функции `XSetScreenSaver()` являются интервалами времени в секундах, определяющими режим работы хранителя экрана. Можно ввести для них тип *`TimeIntervalInSeconds`*, определив для него возможное корректное значение 1. Второй и третий параметры `XGetScreenSaver()` являются указателями на этот же тип.

- Четвертый и пятый параметры `XSetScreenSaver()`, а также второй параметр `XForceScreenSaver()` имеют на самом деле перечислимые типы, определяющие возможные режимы работы или активации/дезактивации хранителя экрана. Для них можно ввести

типы, соответственно, *`BlankingMode`*, *`ExposuresMode`* и *`ForceMode`*. Корректные значения для них четко определены в тексте стандарта — это, соответственно, {`DontPreferBlanking`, `PreferBlanking`, `DefaultBlanking`}, {`DontAllowExposures`, `AllowExposures`, `DefaultExposures`} и {`Active`, `Reset`}. Четвертый и пятый параметры `XGetScreenSaver()` являются указателями на типы *`BlankingMode`* и *`ExposuresMode`*.

- Возвращаемое всеми функциями значение является кодом ответа, который может сигнализировать о каких-либо проблемах, при этом возвращается значение `BadValue`. Тип результата в режиме нормальной работы этих функций можно уточнить, назвав его *`XScreenSaverResult`* и определив в качестве базового ограничения для его значений неравенство константе `BadValue`.

Анализ возможности получения значения типа `Display*` дает следующие результаты.

- Всего в LSB упоминается 18 функций, возвращающих значение типа `Display*`, и две функции, возвращающие ссылку на значение типа `Display`, из которой можно построить нужный указатель.
  6 из этих функций находятся в рамках библиотеки Xlib: `XDisplayOfIM()`, `XDisplayOfOM()`, `XDisplayOfScreen()`, `XOpenDisplay()`, `XcmsDisplayOfCCC()`, `XkbOpenDisplay()`. Остальные функции находятся в других библиотеках — X Toolkit, GTK, Open GL и Qt. Для простейших тестов предпочтительно использовать функции той же библиотеки, поэтому далее анализируются только первые 6 функций.

- Из 6 выбранных функций 4 не могут быть использованы, потому что сами косвенно требуют уже иметь некоторое значение типа `Display*`.
  Например, `XDisplayOfIM()` имеет параметр типа `XIM`, для получения значения которого есть только 2 функции — `XOpenIM()` и `XIMOfIC()`. Первая требует на вход `Display*`, вторая — `XIC`, который, в свою очередь, может быть создан только функцией `XCreateIC()`, которая снова требует значения `XIM`; `XDisplayOfScreen()` требует параметра типа `Screen*`, который в Xlib может быть получен только из `XDefaultScreenOfDisplay()` и `XScreenOfDisplay()`, а они обе требуют параметра типа `Display*`.
  Описание функции `XkbOpenDisplay()` отсутствует в документации на Xlib.

Остается только функция `XOpenDisplay()`, которая может быть использована, поскольку требует только параметра типа **`const char*`**, могущего принимать значение `NULL` при штатном использовании этой функции.

Таким образом, тесты для рассмотренных функций могут быть построены следующим образом.

- В качестве значений параметра `Display*` используется результат вызова `XOpenDisplay()` с аргументом `NULL`.
- В качестве значений интервала времени в секундах используется 1.
- В качестве значений специализированных типов *`BlankingMode`*, *`ExposuresMode`* и *`ForceMode`* используются, соответственно, значения `DontPreferBlanking`, `DontAllowExposures` и `Active`.
- Результаты работы всех функций проверяются на равенство `BadValue`, если результат оказывается равен этому значению, выдается сообщение об ошибке.

Кроме того, корректность результатов, возвращаемых по указателям функцией `XGetScreenSaver()` через ее 4-й и 5-й параметры, можно проверять сравнением их с возможными значениями специализированных перечислимых типов *`BlankingMode`* и *`ExposuresMode`*. На Рис. 2 представлена схема получения значений параметров и ограничений на результаты для функций из рассмотренного примера. Чтобы можно было разместить ее на странице, на этой схеме опущен специализированный тип *`ExposuresMode`*, тип указателей на его значения и связи обоих типов.

Данный пример показывает также, что основной источник повышения производительности создания тестов в рамках описываемой технологии — многократное использование специализированных типов. Один раз уточнив тип параметра функции `XOpenDisplay()`, мы можем получать значения типа `Display*`, необходимые для многих функций из библиотеки Xlib. В данном примере функций немного, а количество параметров с различным смыслом у них достаточно велико, поэтому снижения трудоемкости создания тестов ожидать не приходится. Если же реализовать многократное использование одних и тех же типов для параметров большого числа функций, становится возможным существенное повышение производительности.

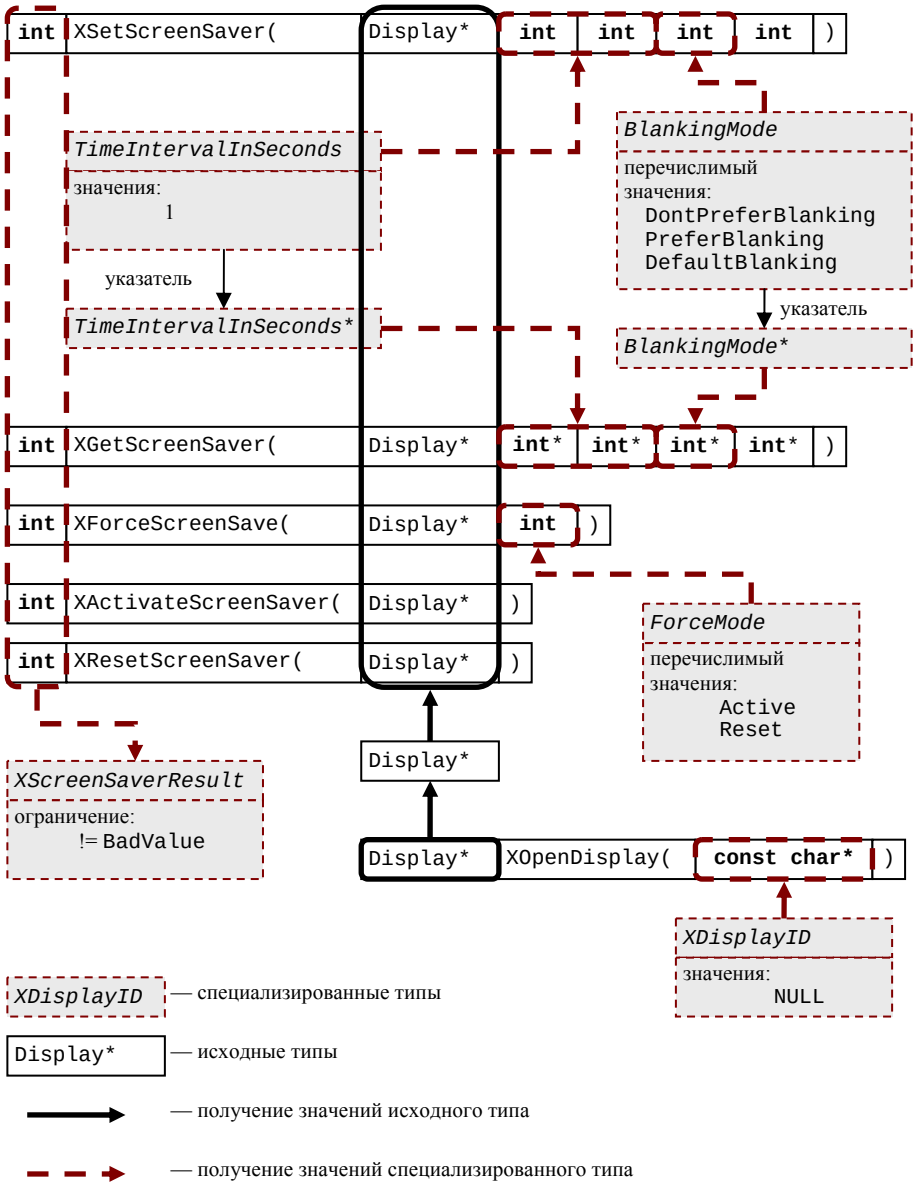

Рис. 2. Схема получения значений параметров и ограничений на результаты для функций работы с хранителем экрана (не показан тип ExposuresMode).

2.5. Инструментальная поддержка

Работа по описываемой технологии поддерживается с помощью следующих инструментов.

- База данных с уточненной информацией о тестируемых операциях.
- Инструмент для редактирования информации в базе.
- Компоновщик тестов, строящий тесты работоспособности по этой информации.

Для редактирования уточненной информации о тестируемых операциях и используемых ими типах данных используется специально разработанный инструмент, имеющий Web-интерфейс и позволяющий как вносить информацию в дополнительные таблицы базы данных, так и выполнять ряд запросов по поиску данных во всех базе и переходы по ссылкам между ее записями.

Этот инструмент применяется для создания специализированных типов, внесения дополнительных данных о тестируемых операциях, установления связей между тестируемыми операциями и уже созданными специализированными типами. В частности, он позволяет находить все типы, уточняющие некоторый заданный тип, и тем самым помогает разработчикам повторно использовать уже имеющиеся специализированные типы. Этот же инструмент используется при контроле качества для проверки корректности проведенного уточнения.

Для автоматической генерации тестов используется компоновщик тестов, реализующий процедуру построения теста, описанную в разделе *2.3. Методическая основа технологии*.

Схема основных таблиц базы данных, используемой для хранения как исходной, так и уточненной информации о тестируемых операциях, приведена на Рис. 3.

На этом рисунке таблицы исходной базы данных показаны в виде прямоугольников, очерченных тонкими линиями, а таблицы, содержащие уточненную информацию — в виде прямоугольников с жирными границами. Таблицы с уточненной информацией имеют имена, начинающиеся на префикс TG (Test Generation).

Из исходной базы данных используется информация о тестируемых операциях, их параметрах и типах параметров и результатов. При этом в базе должны быть следующие поля и связи.

- Для каждой тестируемой операции:
  - ее имя;
  - ссылка на тип результата (если этот тип не равен **`void`**);
  - ссылка на класс, если эта операция — метод класса;

  - для методов классов — флаги, указывающие, что это статический, чисто виртуальный, конструктор, деструктор или обычный метод, а также доступность метода (**`public`**, **`protected`**).

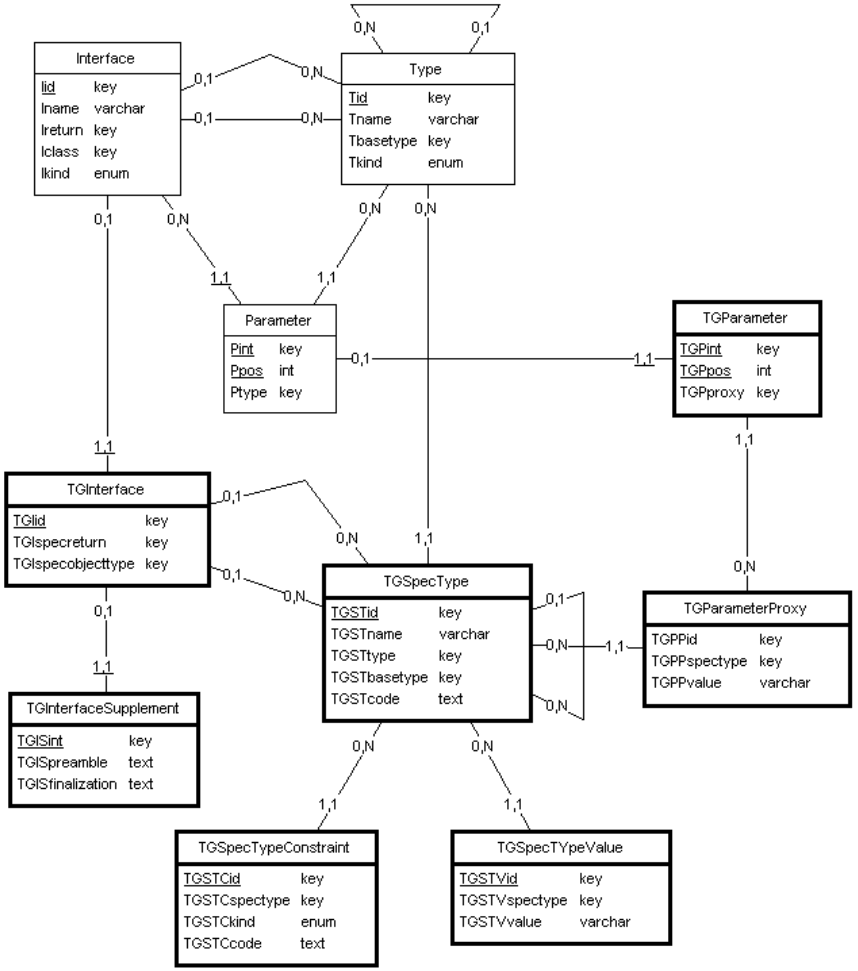

Рис. 3. Основные таблицы базы данных с информацией о тестируемых операциях.

- Для каждого параметра одной из тестируемых операций:
  - ссылка на операцию;
  - номер параметра;

- ссылка на тип параметра.

- Для каждого типа:
  - его имя;
  - разновидность типа — примитивный ли это тип, перечисление, класс, структура, объединение, шаблон, экземпляр шаблона, указатель, алиас (определенный с помощью `typedef`), константный тип (тип `const T` для некоторого другого `T`), тип ссылки, функциональный указатель и др.;
  - ссылка на базовый тип для указателей, ссылочных типов, константных типов и других аналогичных производных типов.

Дополнительные таблицы базы данных содержат следующую информацию.

- В ходе уточнения создается набор специализированных типов, данные которых должны удовлетворять дополнительным ограничениям. Такие типы описываются в таблице TGSpecType. Для каждого специализированного типа указывается следующее.
  - *Имя* — оно не используется в каком-либо коде и нужно только для удобства ссылок на данный тип, в частности, для поддержки возможности использовать один и тот же специализированный тип для параметров и результатов нескольких операций.
  - *Ссылка на исходный тип*, который уточняется данным типом. Например, если для параметра, обозначающего интервал времени в секундах в тестируемой системе используется тип int, можно определить специализированный тип «временной интервал, измеряемый в секундах», для которого исходным типом будет int.
  - *Ссылка на базовый специализированный тип* для указателей, ссылочных типов, константных типов и других аналогичных производных типов.
  - Программный код, определяющий вспомогательные функции для работы с данным специализированным типом. Этот код вставляется в начало каждого теста, в котором фигурируют какие-либо данные этого типа.

  Ограничения на специализированный тип описаны в таблице TGSpecTypeConstraint. Каждая запись об ограничении имеет ссылку на соответствующий тип. Ограничения бывают трех видов, вид ограничения заносится в поле TGSTCkind.

  - InitCode — набор действий, которые нужно выполнить над объектом или значением исходного, не специализированного

    типа, чтобы получить объект или значение специализированного типа.
  - FinalCode — набор действий, которые нужно выполнить, чтобы освободить ресурсы, занятые при создании объекта или значения специализированного типа.
  - NormalResult — код, который нужно выполнить, чтобы проверить корректность возвращенного какой-либо операцией значения данного специализированного типа.

  Возможные значения специализированных типов описаны в таблице TGSpecTypeValue. Каждая запись о значении содержит код, позволяющий построить это значение.

- При уточнении типов может быть уточнен тип результата операции или тип объекта, в котором эта операция вызывается как метод. Для указания этих данных об операциях используется таблица TGInterface и ее поля TGIspecreturn и TGIspecobjecttype.
  Кроме того использование операции в нормальном режиме может потребовать некоторых действий по инициализации внутренних данных системы и захвату ресурсов до вызова, а также действий по освобождению этих ресурсов после вызова. Код этих действий заносится в поля TGISpreamble и TGISfinalization дополнительной таблицы TGInterfaceSupplement, вместе с ссылкой на уточняемую операцию.

- Для поддержки комплексных специализированных типов связь параметра с его специализированным типом опосредуется двумя промежуточными таблицами — таблицей TGParameter, содержащей ссылки на исходные параметры и на получаемые таким образом комплексы, и таблицей комплексов TGParameterProxy. Для каждого комплекса указывается набор параметров, из которых он состоит, ссылка на специализированный тип комплекса и код для получения значений отдельных параметров.

3. Использование предложенной технологии на практике

Технология Azov была применена для построения тестов работоспособности библиотеки Qt версии 3 [9], предназначенной для разработки переносимых приложений с графическим интерфейсом пользователя и входящей в состав стандарта Linux Standard Base (LSB, [3]).

Стандарт LSB включает 10873 доступных для проверки (public или protected) методов, конструкторов и деструкторов классов Qt 3. Информация о них в структурированном виде представлена в базе данных стандарта, находящейся

в открытом доступе [10], как и информация обо всех операциях, входящих в этот стандарт.

Однако, часть данных, необходимых для построения корректных тестов, например, сигнатуры чисто виртуальных методов классов, отсутствует в этой базе данных. Эта информация была добавлена в базу при проведении уточнения ее данных.

Операции были разбиты на группы по классам, методами которых они являются. Поскольку классов Qt тоже довольно много (около 400), они также были разделены на несколько групп в соответствии с их основной функциональностью.

В ходе уточнения было определено 1665 специализированных типов, и для 36 операций были добавлены описания действий по инициализации и финализации. Ряд показателей использования специализированных типов приведен в Табл. 1. Около половины специализированных типов используются повторно, а некоторые — очень много раз. Из таблицы видно также, что примерно в 50% случаев параметры и объекты вызова генерируются автоматически, без использования явно указанных значений специализированных типов.

	Количество		**Исходные типы**
Максимальное количество использований специализированного типа		513	bool (специализированный тип — тип параметров, принимающих значение true)
Количество специализированных типов, использованных	400 и более раз	3	bool, int
	200-399 раз	5	bool, int, char*, QWidget*
	100-199 раз	16	—
	10-99 раз	225	—
	2-9 раз	556	—
Общее количество специализированных типов		1665	—
Количество использований специализированных типов как типов параметров или объектов вызова		11503	—
Количество использований всех типов как типов параметров или объектов вызова		22757	—

Таблица 1. Показатели использования специализированных типов.

Разработка тестов для Qt вместе с разработкой инструментов, поддерживающих технологию Azov, силами 3-х человек заняла около 4-х месяцев. При этом в начале проекта значительная часть усилий тратилась на разработку и отладку инструментов. На конечной фазе проекта, когда инструменты были уже отлажены, каждый разработчик в день создавал тесты для 80-100 функций, с учетом затрат времени на анализ документации, уточнение данных, генерацию, компиляцию и отладку получаемых тестов. Это существенно больше, чем 3-8 функций в день, обрабатываемых при традиционной ручной разработке тестовых вариантов. Причиной такого повышения производительности являются, прежде всего, широкое и многократное использование специализированных типов и его инструментальная поддержка при проведении уточнения данных о тестируемых операциях.

В результате были получены тесты для 10803 функций и методов из 10873. 70 методов (0.6%) не было протестировано по одной из следующих причин.

- Методы класса QSessionManager не могут быть вызваны обычным образом, поскольку библиотека Qt 3 не позволяет как-либо создать объект этого класса. Такой объект может использоваться только в рамках переопределяемого метода финализации приложения, который вызывается средой Qt3 автоматически в конце работы приложения.
- Некоторые конструкторы и деструкторы, предназначенные для вызовов только для объектов самого этого класса, а не его наследников, определены для абстрактных классов. Выполнить вызов такого конструктора или деструктора средствами языка C++ не удается.
- Несколько методов классов Qt 3 попали в стандарт LSB, хотя не предназначены для использования извне библиотеки.
- Для ряда методов отсутствует документация и не удалось подобрать набор значений параметров, для которых такой метод выполнялся бы без разрушения процесса.

При выполнении полученных тестов на одной из реализаций Qt 3 было выявлено около 10 различных ошибок в самой реализации библиотеки, несмотря на то, что все тесты представляют собой простейшие сценарии работы с методами ее классов.

Достаточно успешное применение технологии Azov в описанном проекте показывает, что она вполне годится для быстрого создания тестов работоспособности больших промышленных программных систем.

4. Сопоставление с другими подходами к автоматизации создания тестов

Поскольку основной целью разработки технологии Azov являлось массовое создание тестов API для достаточно больших программных систем, использование автоматической генерации тестов в ней является необходимостью. В большинстве подходов к автоматизации тестирования такая генерация отсутствует, поэтому далее Azov рассматриваются в контексте только тех методов, где либо тестовые данные, либо последовательность тестовых воздействий, либо оба этих элемента действительно строятся автоматически на основе некоторой информации.

Такие методы можно классифицировать по источникам данных для построения тестовых воздействий или их последовательностей и источникам данных для выполнения проверок корректности в тестах (или тестовых оракулов). При этом многие методы и инструменты попадают сразу в несколько классов, поскольку используют сочетание различных источников для генерации тестов.

Можно выделить следующие источники для построения тестовых данных, тестовых воздействий и их последовательностей.

- Структура тестируемой системы.
  - Синтаксис сигнатур тестируемых операций и/или разбиение входных данных на области.
  - Структура типов данных.
  - Связи по типам — использование одних и тех же типов данных для параметров и результатов тестируемых операций и др.
  - Структура кода (прежде всего, потока управления) операций.
- Описание функциональности тестируемых операций.
  - Ограничения, связанные с типами данных.
  - Ограничения, связанные с каждой операцией отдельно от остальных.
  - Ограничения, связывающие поведение нескольких операций.

Для выполняемых в тестах проверок исходной информацией всегда служит либо описание функциональности в одном из перечисленных выше видов, либо предположения о том, что корректное поведение системы не должно включать создание исключительных ситуаций, посылку сигналов, разрушение процессов и другие специфические действия, обычно (но не всегда!) сигнализирующих об ошибках.

<table>
<tr><td>Источник данных для проверки
Источник тестовых воздействий</td><td>Нет проверок</td><td>Требование отсутствия исключений и серьезных сбоев</td><td>Ограни-чения типов данных</td><td>Ограни-чения на одну операцию</td><td>Совмест-ные огра-ничения для нескольких операций</td></tr>
<tr><td>Синтаксис сигнатур и/или разбиение входных данных</td><td>Методы и инструменты генерации тестов на основе пок-рывающих наборов [11]</td><td></td><td></td><td></td><td></td></tr>
<tr><td rowspan="2">Структура типов</td><td>OTK/Pinery [12,13]</td><td colspan="4">Инструменты Parasoft (Jtest, C++test, .TEST [14])
Ограничения в виде утверждений, постусловий и инвариантов</td></tr>
<tr><td></td><td>JCrasher [15]</td><td></td><td></td><td></td></tr>
<tr><td rowspan="2">Связи по типам</td><td></td><td colspan="3" rowspan="2">Azov</td><td rowspan="2"></td><td rowspan="2"></td></tr>
<tr><td colspan="2">Методы генерации структурных тестов на основе генетических алгоритмов [16-18]</td></tr>
<tr><td rowspan="2">Структура потока управления</td><td></td><td>CodeScroll API Tester, Suresoft Technologies [19]</td><td rowspan="2"></td><td rowspan="2"></td><td rowspan="2"></td></tr>
<tr><td colspan="2">Методы генерации структурных тестов с использованием разрешения ограничений (constraint solving) [20,21]</td></tr>
<tr><td>Ограничения типов данных</td><td>Pinery [13]</td><td colspan="2" rowspan="2">Azov
Ограничения в виде инвариантов данных типа и действий по корректному построению этих данных</td><td rowspan="2">T-VEC Test Vector Generation System [22]
Ограничения в виде постусловий и инвариантов</td><td></td></tr>
<tr><td>Ограничения на одну операцию</td><td rowspan="3"></td><td></td></tr>
<tr><td rowspan="2">Совместные ограничения для нескольких операций</td><td></td><td></td><td rowspan="2">Lurette [23]</td><td></td></tr>
<tr><td rowspan="2"></td><td colspan="2">Методы и инструменты генерации тестов на основе автоматных моделей, включая Stateflow, Statecharts, SDL, LOTOS, Lustre или специализированные нотации [24,25]</td><td></td></tr>
<tr><td colspan="3">UniTESK [26]</td></tr>
</table>

Таблица 2. Классификация подходов к автоматическому построению тестов.

Классификация имеющихся на рынке инструментов, а также ряда исследовательских методов, использующих автоматическую генерацию

тестов, на основе выполняемых проверок и видов данных, используемых для построения тестов, показана в Табл. 2. Технология Azov в этой таблице попадает сразу в несколько клеток, поскольку сочетает при генерации тестов использование структуры типов, связей по типам, действий по инициализации данных типа и (иногда) специфических действий по инициализации системы перед использованием определенной операции. Для проверок используются ограничения типов результатов и фактор отсутствия сбоев при вызове тестируемой операции.

Заметим, что кроме технологии Azov связи по типам используются при генерации тестов только рядом методов построения структурных тестов на основе генетических алгоритмов [19-21].

5. Дальнейшее развитие технологии

Дальнейшее развитие технологии Azov возможно по нескольким направлениям.

- ***Расширение возможностей повторного использования.***
  В данный момент специализированные типы представляют собой практически единственный вид элементов технологии, которые можно использовать многократно. Для расширения таких возможностей можно определить механизмы, позволяющие повторно использовать ограничения типов, позволив таким образом создавать типы, уточняющие другие специализированные типы и наследующие все их ограничения.

- ***Увеличение качества и полноты тестов.***
  Тестирование работоспособности предполагает однократный вызов каждой из тестируемых операций и проверку только некоторых базовых ограничений. Однако предложенную технологию достаточно просто расширить так, чтобы создаваемые тесты обладали большей полнотой и более аккуратно проверяли тестируемую систему.

  - *Добавление дополнительных ограничений.*
    В настоящий момент проверяются ограничения, применимые только при одном из простейших сценариев использования. Можно ввести более сложные проверки за счет введения дополнительных ограничений и условий на входные параметры, при которых эти ограничения должны проверяться.

  - *Использование нескольких наборов параметров.*
    Для более тщательного тестирования можно использовать не по одному значению каждого параметра, а строить различные их наборы, перебирая некоторое множество значений для каждого параметра. Множества значений типов можно указывать как простым перечислением этих значений в базе,

так и определением некоторых шаблонов построения таких значений, в свою очередь имеющих некоторые параметры. Во втором случае для построения конкретного значения по шаблону с параметрами можно использовать процедуру выбора значений параметров, реализуемую компоновщиком тестов.

  - *Добавление различных сценариев использования.*
    При тестировании работоспособности операции вызываются в наиболее простых сценариях их использования. Однако даже часто возникающих сценариев использования операции бывает несколько, иногда для граничных значений параметров операция должна демонстрировать специфическое поведение. Кроме того, большинство операций должно правильно реагировать на некорректно заданные параметры.
    Для добавления поддержки различных сценариев использования необходимо уметь задавать набор сценариев или режимов работы операции и для каждого режима определять соответствующие специализированные типы параметров и специфические ограничения на результаты работы.

  - *Построение тестовых последовательностей.*
    Для операций, поведение которых зависит от внутреннего состояния системы, можно увеличить качество и количество создаваемых тестов за счет автоматического добавления перед вызовом операции обращений к другим операциям, изменяющих те внутренние данные, от которых зависит работа этой операции.

- ***Создание тестов совместимости.***
  Описанная технология может использоваться для создания тестов совместимости нескольких библиотек, если объекты вызова и аргументы вызова тестируемых операций из одной библиотеки в тестах получать как результаты вызовов операций из других библиотек.

- ***Использование представленной методики уточнения при быстрой разработке программного обеспечения.***
  Методику уточнения информации об операциях можно использовать при разработке программного обеспечения не только для создания тестов. Поскольку база данных с исходной информацией об операциях системы может быть достаточно просто получена как побочный результат компиляции, эта методика может быть основой контроля качества, анализа и уточнения проектных решений, создания проектной документации при быстрой разработке программных систем.
  Работа может быть построена на основе ежедневных сборок: база

данных с информацией о системе, собранной в предыдущий вечер наутро поступает для уточнения и анализа. Выявленная при таком анализе за день информация о дополнительных ограничениях на типы параметров и результатов далее согласовывается с проектировщиками и может использоваться для уточнения и исправления ошибок проектирования и для создания проектной документации. При этом, однако, важно иметь специальные механизмы распознавания и интеграции изменений базы по сравнению с предыдущей версией.

6. Заключение

Автоматизация создания тестов обычно основана на формализации большого количества правил и критериев, которые управляют ручной разработкой тестов, оставаясь несформулированными явно. Такая формализация чаще всего требует серьезных трудозатрат, но окупается за счет полноты и качества получаемых в результате тестов. Однако возможно автоматизировать и создание гораздо менее аккуратных тестов, проверяющих только базовую работоспособность системы.

В данной статье представлена технология Azov автоматизации разработки тестов работоспособности, созданная в Институт системного программирования РАН. Для ее применимости достаточно иметь базу данных, где информация об операциях системы представлена в хорошо структурированном виде, и документацию с описанием базовой функциональности этих операций. Можно заметить, что любой компилятор в принципе способен выдать такую базу данных в качестве побочного результата своей работы.

Апробация технологии на библиотеке Qt для разработки приложений с графическим пользовательским интерфейсом, включающей более 10000 операций, показала, что технология вполне успешно и достаточно эффективно справляется с возложенными на нее задачами. Хотя инструменты, поддерживающие работу по описываемой технологии, создавались и дорабатывались непосредственно в ходе этого проекта, были достигнуты весьма высокие показатели производительности.

Технология Azov обладает также серьезным потенциалом для дальнейшего развития и может быть использована как для создания более качественных тестов, так и для совсем других видов тестирования, например, тестирования совместимости и возможности взаимодействия различных программных систем.

Литература

[1] IEEE 1003.1-2004. Information Technology — Portable Operating System Interface (POSIX). New York: IEEE, 2004.

[2] http://doc.trolltech.com/4.2/index.html.
[3] http://www.linuxbase.org.
[4] ISO/IEC 9899:1999. Programming Languages — C. Geneve: ISO, 1999.
[5] http://www.pathname.com/fhs/.
[6] Xlib — C Language X Interface. X Consortium Standard. http://refspecs.freestandards.org/X11/xlib.pdf.
[7] http://www.opengl.org.
[8] http://www.gtk.org.
[9] http://doc.trolltech.com/3.3/index.html.
[10] http://www.linux-foundation.org/navigator/commons/welcome.php.
[11] http://www.pairwise.org/tools.asp.
[12] С. В. Зеленов, С. А. Зеленова, А. С. Косачев, А. К. Петренко. Генерация тестов для компиляторов и других текстовых процессоров. Программирование, 29(2):59–69, 2003.
[13] А. В. Демаков, С. В. Зеленов, С. А. Зеленова. Генерация тестовых данных сложной структуры с учетом контекстных ограничений. Труды ИСП РАН, 9:83–96, 2006.
[14] http://www.parasoft.com/jsp/products.jsp.
[15] C. Csallner, Y. Smaragdakis. JCrasher: and Automatic Robustness Tester for Java. Software — Practice & Experience, 34(11):1025–1050, 2004.
[16] R. Ferguson, B. Korel. The Chaning Approach for Software Test Data Generation. ACM Transactions on Software Engineering Methodology, 5(1):63–86, 1996.
[17] R. P. Pargas, M. J. Harrold, R. Peck. Test-data Generation Using Genetic Algorithms. Software Testing. Verification & Reliability, 9(4):263–282, 1999.
[18] A. Seesing, H.-G. Gross. A Genetic Programming Approach to Automated Test Generation for Object-Oriented Software. Intl. Trans. on System Science and Applications, 1(2):127–134, 2006.
[19] http://www.suresofttech.com/eng/main/product/api.asp.
[20] B. Korel. Automated Test Data Generation for Programs with Procedures. In Proc. of ISSTA 1996, pp. 209–215.
[21] A. Gotlieb, B. Botella, M. Rueher. Automatic Test Data Generation Using Constraint Solving Techniques, ACM SIGSOFT Software Engineering Notes, 23(2):53–62, March 1998.
[22] http://www.t-vec.com/solutions/tvec.php.
[23] P. Raymond, D. Weber, X. Nicollin, N. Halbwachs. Automatic testing of reactive systems. In Proc. of 19-th IEEE Real-Time Systems Symposium, 1998.
[24] A. Hartman. Model Based Test Generation Tools. 2002. http://www.agedis.de/documents/ModelBasedTestGenerationTools.pdf.
[25] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, A. Pretschner, eds. Model-Based Testing of Reactive Systems. Advanced Lectures. LNCS 3472, Springer, 2005.
[26] В. В. Кулямин, А. К. Петренко, А. С. Косачев, И. Б. Бурдонов. Подход UniTesK к разработке тестов. Программирование, 29(6):25–43, 2003.