Автоматизация массового создания тестов работоспособности

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

Аннотация. В данной статье рассказывается о технологии массового создания тестов работоспособности сложных программных систем, разработанной в Институте системного программирования РАН и названной Azov. Технология Azov основана на использовании базы данных со структурированной информацией об интерфейсных операциях тестируемой системы и методике пополнения этой информации за счет уточнения ограничений на типы параметров и результатов операций. Представленные результаты практической апробации технологии демонстрируют ее высокую эффективность при разработке тестов работоспособности для систем с большим количеством функций.

1. Введение

В современном мире как общее количество задач, решение которых возлагается на программные системы, так и их важность и ответственность все время возрастают. В связи с этим, с одной стороны, повышается сложность программного обеспечения (ПО), а с другой — риски, связанные с неизбежно возникающими при создании систем такой сложности ошибками. Единственным средством минимизации этих рисков является проведение на всех этапах создания или внесения изменений в ПО аккуратных и систематических проверок его корректности в смысле соответствия требованиям. Для такой проверки на последних этапах разработки чаще всего используется тестирование — наблюдение за работой системы и анализ ее правильности в ряде специально создаваемых ситуаций с учетом всех существенных аспектов ее поведения.

Имеющиеся на сегодняшний день методы тестирования требуют серьезных затрат для обеспечения некоторых гарантий полноты проводимой проверки. При возрастании сложности системы эти затраты возрастают нелинейно. Автоматизация создания тестов обычно не сильно снижает их, поскольку она связана с необходимостью формализации требований к тестируемой системе, исходно заданных неформально, и классификации тестовых ситуаций для нее.

Однако иногда, когда результаты тестирования и не должны демонстрировать высокую надежность и полноту, эти затраты неоправданны. Например, при *тестировании работоспособности* проверяется только то, что система не разрушается и возвращает результаты, проходящие простейшие проверки на корректность (полная проверка при этом не выполняется). Оно используется, чтобы убедиться в том, что все функции системы устойчиво работают в простейших ситуациях, до того, как подвергнуть ее более тщательной и систематической проверке, которая требует значительно больше усилий, но бессмысленна, если система не в состоянии справиться даже с простейшими задачами. Такое тестирование экономит усилия на поиск и локализацию достаточно грубых ошибок в сложных программных системах.

Обычно тесты работоспособности разрабатываются вручную, но иногда огромное количество интерфейсных операций тестируемой системы и наличие полной и хорошо структурированной информации о них подталкивают к автоматизации их создания. Если операций очень много, традиционная разработка тестов для них становится слишком трудоемкой. В то же время, информация об их синтаксисе может быть использована для автоматической генерации заготовок тестов.

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 поддержки в актуальном состоянии текста стандарта и набора инструментов для выполнения различных проверок синтаксическая информация обо всех входящих в стандарт интерфейсах помещена в единую, хорошо структурированную базу данных. Все это позволяет использовать новый подход для создания тестов работоспособности LSB.

2. Технология автоматизации создания тестов работоспособности

Для тех случаев, когда интерфейс системы состоит из тысяч операций и информация об элементах этого интерфейса хранится в хорошо структурированном, подходящем для автоматической обработки виде, можно предложить достаточно эффективную технологию автоматизации построения тестов работоспособности, основанных на требованиях, позволяющую создавать их массово, в большом количестве. Такая технология, названная Azov, разработана в 2007 году в Институте системного программирования РАН.

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

Технология 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 так, 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 полученный тест.

2.1. Исходные данные и ожидаемые результаты

Исходными данными для выполнения работ по описываемой технологии являются база данных, содержащая структурированную синтаксическую информацию об операциях тестируемой системы (их сигнатуры), и документация на эти операции. На текущий момент предполагается, что все операции являются функциями языка C или методами классов C++. Изменение базового языка программирования в рамках императивных языков требует обычно лишь замены модуля печати выходной информации компоновщика тестов. Если в этом языке возможны способы передачи параметров и результатов операций, существенно отличающиеся от используемых в C/C++, может потребоваться некоторая модификация структуры используемой базы данных и алгоритмов генерации тестов.

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 *2.5. Инструментальная поддержка*.

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

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 простейших сценариев ее нормальной работы, не приводящих при корректной работе системы к сбоям, исключительным ситуациям или возврату кода ошибки. Тест также проверяет базовые 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 проведено итоговое освобождение захваченных ресурсов.

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 разбита на следующие действия.

- ***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 выполняет отдельный разработчик.

- ***Уточнение информации об интерфейсных операциях в базе данных.***
  Разработчики анализируют документацию на операции выделенных им групп, определяя условия их нормальной работы и действующие при этом ограничения на их результаты. Выявляемые ограничения заносятся в базу данных в виде специализированных типов параметров и результатов операций. Каждый такой тип может иметь список возможных значений или действий, необходимых для инициализации или уничтожения данных такого типа. Если для нормальной работы операции необходимо правильно инициализировать не только ее аргументы, но и глобальные данные системы, для нее указываются соответствующие процедуры инициализации или финализации этих данных. В ходе работы применяется методика уточнения данных, представленная ниже. Для заполнения базы данных используются вспомогательные инструменты, использующие Web-интерфейс и позволяющие осуществлять навигацию по базе данных, поиск в ней разнообразной дополнительной информации и редактирование уточняемых данных. Перед проведением уточнения тестируемые операции и типы данных, связанные с ними, упорядочиваются 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. Уточнение выполняется, начиная с простых операций, и постепенно переходит к более сложным. При этом нет необходимости часто переключаться на анализ других операций, а информация, выявленная на ранних этапах, может естественным образом многократно использоваться на поздних.

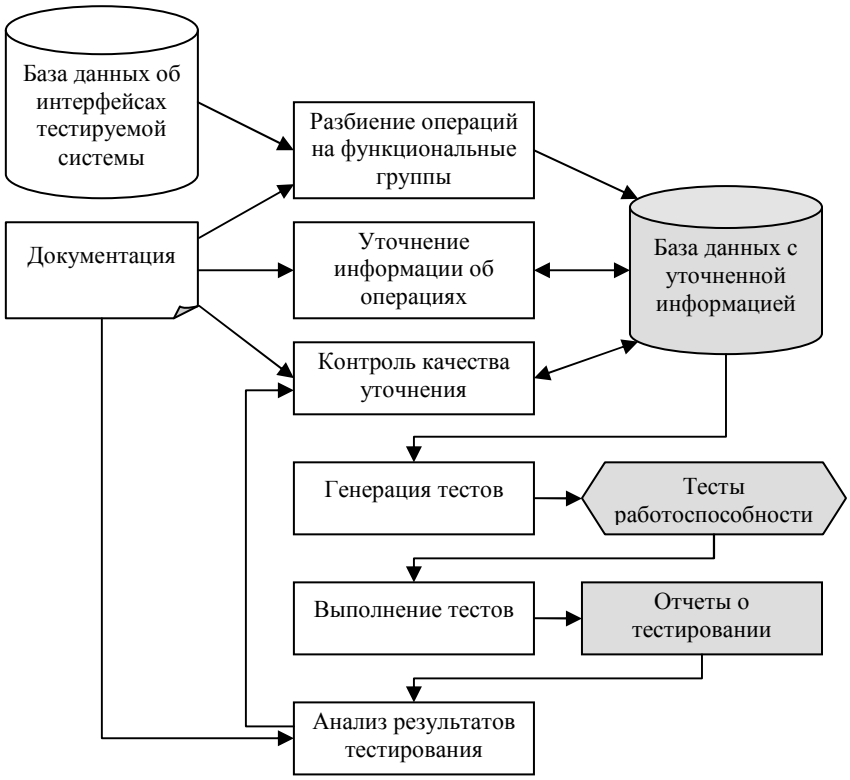

Рисунок 1. Схема выполнения работ по технологии Azov.

- ***Контроль качества проведенного уточнения.***
  Введенная информация проходит проверку на корректность, выполняемую за счет ее просмотра и дополнительного анализа другими разработчиками, а также за счет отладки сгенерированных тестов — они должны успешно компилироваться и собираться, а все проблемы, возникающие при их выполнении должны быть только следствием ошибок в тестируемой системе.

- ***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. Обнаруженная проблема анализируется разработчиком, и по итогам этого анализа либо фиксируется как ошибка в тестируемой системе, либо приводит к исправлению некоторой информации в базе, инициировавшей ошибку в тесте, после чего тест выполняется снова.

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) 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 этих операций.
  - В тех случаях, когда использование аргумента (или объекта вызова) предполагает последующее освобождение ресурсов, с его специализированным типом связывается код финализации 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 значений параметров, и определяется специализированный тип такого составного объекта.
    Например, если параметрами операции являются указатель на начало строки типа `char*` и ее длина, можно определить специализированный тип «строка». Тогда первым аргументом становится указатель на первый элемент строки, а вторым — результат применения к нему функции `strlen()`.
    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.

- Указанные в базе данных связи между операцией и исходным типом ее параметра или результата при уточнении типа дополняются аналогичными связями с соответствующим уточненным типом.
- Каждый раз при необходимости введения специализированного типа сначала анализируются уже имеющиеся типы, чтобы по возможности использовать повторно тип с нужным набором ограничений.

- ***Определение инициализации и финализации для операций.***
  Если для нормальной работы операции необходимо предварительно выполнить некоторые действия по 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 (включая специализированные) выбрасываются примитивные или производные от других типов (указатели, ссылки и пр.), такие типы, значения которых можно получить только в результате вызовов специальных операций или конструкторов, и 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 в базе данных явно, оно и используется.
  - 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 вычисляются рекурсивно, по общей процедуре построения значений параметров.
    Определения вспомогательных операций, необходимых для построения нужных значений, 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 рекурсивно, по этой же процедуре.
- Вставляется вызов тестируемой операции с построенными аргументами.
- При необходимости вставляется код финализации построенных значений.
- 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 создается объект класса-наследника и в нем вызывается общедоступный метод-обертка.

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 автоматически. 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 методика построения тестов в рамках описываемой технологии проиллюстрирована 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 дисплея.
- `int XResetScreenSaver(Display*)` — дезактивирует хранитель экрана для 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 этого типа, из которой можно построить 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 строятся 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 (опущен тип *ExposuresMode*, тип указателей на его значения и связи обоих типов).

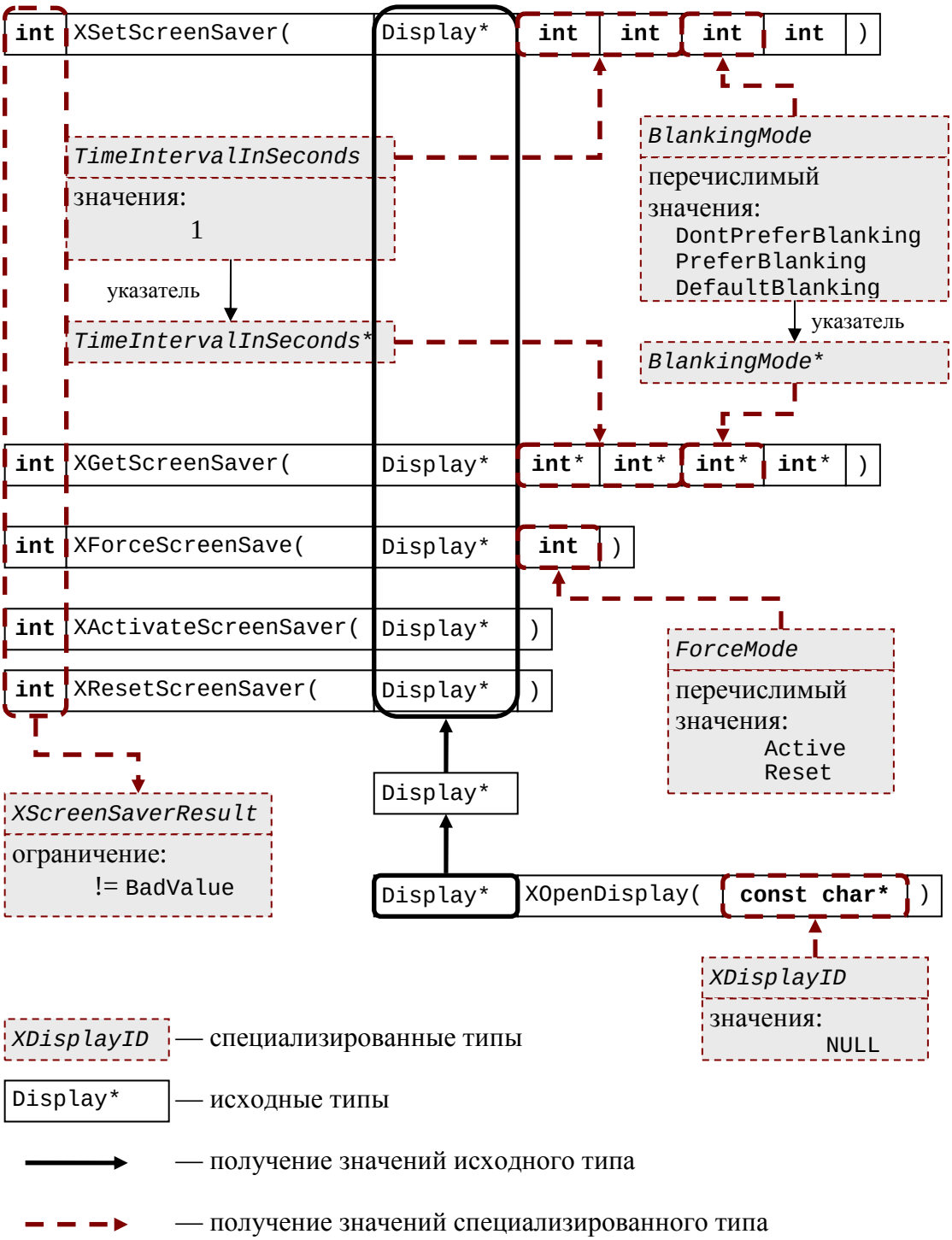

Рисунок 2. Схема получения значений параметров и ограничений на результаты для функций работы с хранителем экрана (не показан тип ExposuresMode).

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 нет. Если же реализовать многократное использование одних и тех же типов для параметров большого числа функций, становится возможным существенное повышение производительности.

2.5. Инструментальная поддержка

Работа по описываемой технологии поддерживается с помощью следующих инструментов.

- Базы данных с уточненной информацией о тестируемых операциях.
- Инструмента для редактирования информации в базе, имеющего Web-интерфейс. Он позволяет вносить информацию в дополнительные таблицы базы данных и выполнять ряд запросов по поиску данных во всех базе и переходы по ссылкам между ее записями. В частности, он позволяет находить все типы, уточняющие некоторый заданный тип, и тем самым помогает разработчикам повторно использовать уже имеющиеся специализированные типы.
- Компоновщика тестов, строящего тесты по информации базы данных, используя процедуру, описанную в разделе *Методическая основа технологии*.

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

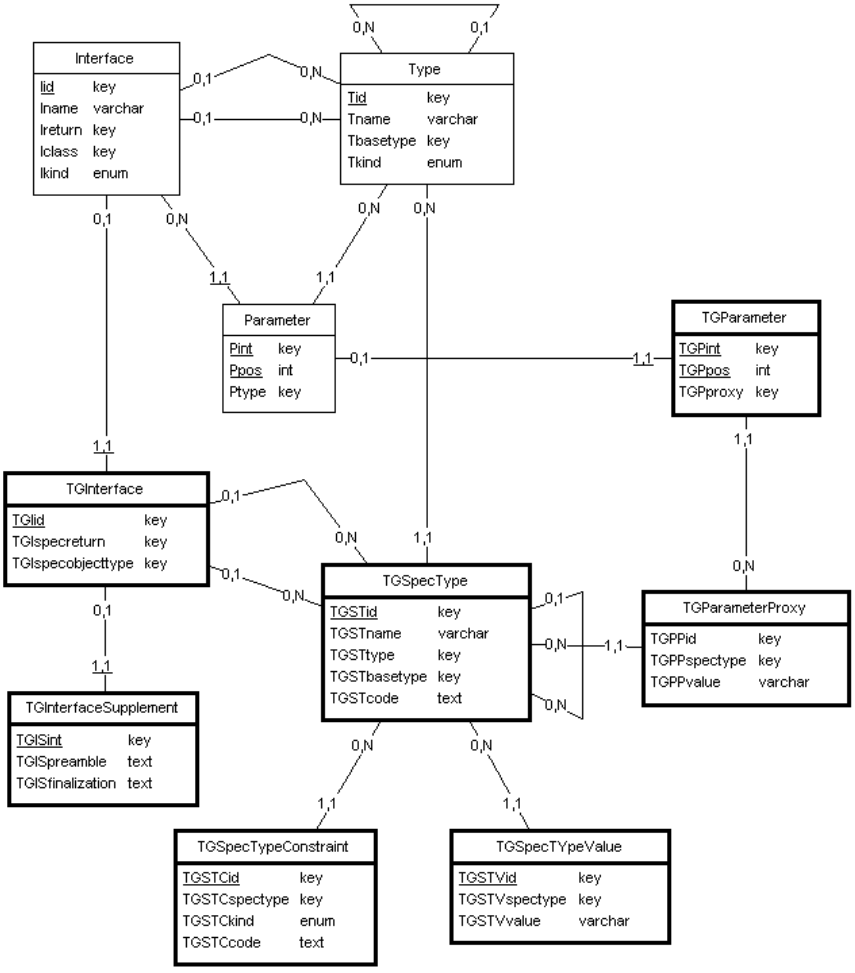

Рисунок 3. Основные таблицы базы данных с информацией о тестируемых операциях.

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. Информация о типах и операциях включает их имена, различные описатели (**`public`**, **`protected`**, **`static`**), разновидность

типа — примитивный ли это тип, перечисление, класс, структура, объединение, шаблон, экземпляр шаблона, указатель, и пр., ссылку на базовый тип для указателей, ссылочных и других производных типов.

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 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 при проведении уточнения.

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

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.

		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 конечном этапе, 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.

В результате были получены тесты для 10803 функций и методов из 10873. Лишь 70 методов (0.6%) не было протестировано по разным причинам: отсутствие документации, случайное попадание в стандарт внутренних методов библиотеки, невозможность вызова такого метода в языке C++ и пр.

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 проверяют простейшие сценарии работы методов ее классов.

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

В настоящее время по этой технологии создаются тесты для библиотеки Qt 4 [2], а также планируется использовать ее для разработки тестов нескольких других библиотек, входящих в стандарт LSB и не снабженных тестовыми наборами.

4. Сопоставление с другими подходами к автоматизации создания тестов

Основной целью разработки технологии Azov являлось массовое создание тестов для достаточно больших программных систем, и использование автоматической генерации тестов в ней является необходимостью. Поэтому в приводимом здесь обзоре рассматриваются только те методы, где либо тестовые данные, либо последовательность тестовых воздействий, либо оба этих элемента действительно строятся автоматически на основе некоторой информации. Кроме того, рассматриваемая технология предназначена для построения тестов программного интерфейса (API), и сопоставлять ее разумно с такими же подходами, поскольку генерация тестов для других видов интерфейсов (графический пользовательский интерфейс, передача сообщений, событийный интерфейс и пр.) имеет свои особенности.

Описанные в доступной литературе методы автоматического построения тестов в соответствии с основными используемыми техниками можно разделить на следующие классы.

- *Методы генерации тестов на основе покрывающих наборов.*
  В основе этих методов лежат комбинаторные техники построения тестов. Каждый тест является комбинацией значений нескольких параметров или факторов, а каждый фактор может принимать конечное множество значений. Тестовый набор строится так, чтобы в рамках минимального числа тестов использовать все возможные комбинации пар, троек и т.д. значений факторов. Эта задача эквивалентна построению комбинаторной схемы — покрывающего набора с определенными параметрами. Достаточно полный обзор техник построения покрывающих наборов и возможностей их использования в тестировании можно найти в [11,12]. Большинство доступных инструментов, использующих такие техники, представлено на сайте [13]. Наиболее известны из них AETG [14,15], TestCover [16,17], AllPairs [18], Jenny [19], Intelligent Test Case Handler [20] и PICT [21]. Первые два инструмента наиболее зрелые, имеют широкие возможности конфигурации и позволяют получать несколько более компактные тестовые наборы.
  Подобные методы используют для построения теста только синтаксическую информацию об интерфейсе тестируемой операции и определяемые пользователем конечные множества значений параметров. Никаких проверок не делается — предполагается, что результаты тестов будут оцениваться либо человеком, либо с помощью отдельных инструментов.

Для массовой генерации тестов работоспособности такие методы не подходят, поскольку для каждой операции необходим фактически только один тест, при построении которого покрывающие наборы бесполезны.

- *Методы генерации тестовых данных на основе моделирования их структуры и постепенного построения сложных объектов из значений примитивных типов.* Такие методы используют описание структуры данных в некоторой нотации для построения тестовых данных, имеющих такую структуру. При этом данные простых типов — числа, символы, строки, даты и пр. — либо выбираются из некоторых заранее заданных множеств, либо генерируются с помощью простых, часто вероятностных алгоритмов. Из этих значений с помощью различных стратегий комбинирования постепенно составляются все более и более сложные объекты. Такие методы могут учитывать простые ограничения, на значения одного или нескольких элементов, например, совпадение значений определенных полей. Для удовлетворения ограничений может использоваться частичная или полная фильтрация — генерация и отбрасывание результатов, не удовлетворяющих ограничениям. Однако такой подход становится неэффективными при возрастании сложности ограничений — в этих случаях обычно используют методы следующей группы.
  Инструментов, реализующих техники такого рода, довольно много. Большинство из них привязано к определенному формату результатов, т.е. генерирует либо заполнение реляционной базы данных, либо XML-документы, либо тексты на языке с заданной на входе грамматикой. Подавляющее большинство инструментов, генерирующих заполнение баз данных, являются коммерческими (см. [22-26]), есть небольшое количество исследовательских [27-29], которые могут учитывать не только структуру данных, но и структуру запросов приложений. XML-генераторы являются, в основном, исследовательскими или свободными инструментами [30-34]. Первые работы в области генерации сложных данных [35-37] имели дело с генерацией текстов, подходящих под заданную грамматику. Несколько более современных инструментов [38,39] продолжают эту линию. Есть также ряд инструментов и каркасов для создания генераторов тестовых данных, способных давать результаты в различных форматах. К ним относятся разработанные в ИСП РАН инструменты OTK [40] и Pinery [41].
  Для таких методов необходимо описание структуры данных в каком-либо виде (BNF, DDL, DTD, XML Schema, Relax NG и пр.) и задание ограничений на размеры и структуру генерируемых объектов. Проверка корректности работы тестов обычно возлагается на человека или другие инструменты.
  Для генерации тестов работоспособности такого рода инструменты могут быть использованы, если интерфейсные операции тестируемой системы принимают в качестве входных данных достаточно сложные объекты.
  В рамках технологии Azov тоже используется эта техника генерации объектов сложных типов, однако предпочтительный способ их получения — использование конструкторов или других операций тестируемой системы, которые возвращают объекты таких типов. Это позволяет в большинстве случаев не заботиться о корректности построения такого объекта, если его составные части должны быть связаны специфическими ограничениями.

- *Методы генерации тестовых данных при помощи разрешения ограничений.* В рамках таких подходов ограничения, которыми связаны элементы тестовых данных, разрешаются без использования фильтрации. Результат такого решения — конкретные тестовые данные для одного вызова.
  Эти методы построения тестов используют и при структурном, и при

функциональном тестировании. В обоих случаях ограничения являются условиями достижения определенной ситуации, но в структурном случае она описывается в терминах структуры тестируемого кода, а в функциональном — в терминах проверяемой функциональности.

Используемые техники разрешения ограничений могут существенно отличаться. Это могут быть техники прямого решения возникающих систем ограничений или логического программирования [42], символическое выполнение программ [43], методы поиска, отталкивающиеся от случайно сгенерированных стартовых данных, чтобы подобрать данные, подходящие под ограничения [44,45], и др.

Среди инструментов, использующих такие методы, есть несколько коммерческих [46-48]. Инструменты компании Parasoft [46] Jtest, C++Test, .TEST наиболее широко известны. Они используют для построения тестовых данных как их структуру, так и структуру кода тестируемых методов. Для проверки используются предусловия и постусловия методов и инварианты типов данных, занесенные пользователем в виде комментариев в исходный код. Исключительная ситуация во время тестирования также рассматривается как ошибка. Инструмент T-VEC [47] использует и для генерации тестов, и для проверки их выполнения постусловия операций. В инструментах SureSoft [48] генерация тестов выполняется на основе структуры кода, а правильность их выполнения оценивается на основе отсутствия исключений и сбоев тестируемой программы. Исследовательские инструменты TestEra [49] и Korat [50] используют написанные пользователем инварианты и технику разрешения ограничений для генерации объектов сложных типов данных в разнообразных состояниях.

Использование подобных подходов для генерации тестов работоспособности в принципе возможно, но очень неэффективно. Для тестов работоспособности, основанных на требованиях, использовать методы построения структурных тестов нельзя, а формализация соответствующих требований, достаточная для применения имеющихся инструментов, требует слишком много усилий.

- *Методы динамической генерации структурных тестов.*

  Термин «динамическая генерация тестов» используют для таких методов, в рамках которых объекты и значения, создаваемые в тестируемой программе в ходе тестирования используются для построения новых и новых тестов [51]. При этом генерация направляется на достижение покрытия определенных ситуаций и состояний программы, для чего используются методы поиска, например, генетические алгоритмы [52,53] или поиск экстремума (hill climbing) [54,55].

  Тесты, построенные такими методами, являются не отдельными вызовами, как для методов предыдущих типов, а последовательностями вызовов. Таким образом, появляется возможность проверять работу тестируемой системы во многих состояниях, и в то же время в качестве параметров вызовов можно использовать объекты, находящиеся в разных состояниях.

  Технология Azov активно использует технику «динамической генерации» тестов, строя цепочки вызовов, обеспечивающих нормальное функционирование тестируемых операций.

  За последние 5-7 лет появилось достаточно много исследовательских инструментов этого типа, многие из которых применимы и к достаточно сложным промышленным программным системам. Поскольку в этих инструментах часто используется сложная комбинация из различных техник анализа тестируемой программы и генерации тестов, их можно назвать *синтетическими*.

- Инструменты JCrasher [56], Check-n-Crash [57] и DSDCrasher [58,59] разработаны в университете Орегона. JCrasher, исторически первый из этой группы инструментов, генерирует тесты для Java-программ, используя случайные данные простых типов, несколько эвристик нацеливания на вероятные ошибки, синтаксис операций и структуру данных. Выполняемые им проверки сводятся к отсутствию исключений и сбоев. В инструменте Check-n-Crash добавлен дополнительный первый этап статического анализа кода тестируемых операций с помощью ESC/Java [60], чтобы получить наборы ограничений, соответствующих различным путям, которые затем разрешаются частично прямыми методами, частично за счет небольших модификаций случайных тестов. Инструмент DSDCrasher добавляет еще одну предварительную фазу, на которой тестируемая программа выполняется на множестве случайных сценариев, и с помощью инструмента Daikon [61] выявляются возможные инварианты программы. Затем они используются для отсеивания некорректных сценариев тестирования, которые приводят к ошибкам не в силу ошибочной работы программы, а из-за неправильного ее использования.
- Те же техники, что и JCrasher, использует AutoTest [62] при генерации тестов для программ на Eiffel. Для проверки, однако, используется не только исключительные ситуации, но и инварианты и постусловия, определенные пользователем.
- В MIT разработаны инструменты Elcat [63] и Randoop [64], использующие случайную генерацию совместно с техниками сокращения множества анализируемых состояний и некоторыми эвристиками попадания в новые ситуации (в Randoop используются возникающие с определенной вероятностью длинные последовательности вызовов одних и тех же методов). Elcat также может использовать Daikon для отбрасывания некорректных сценариев тестирования. Randoop использовался при генерации тестов для достаточно больших библиотек (части JDK и библиотек .NET, размером около 700 тыс. строк) и продемонстрировал в этих условиях способность строить тесты, выявляющие серьезные ошибки.
- Символическое выполнение тестируемой программы для выявления ограничений на входные данные, приводящие к возможно ошибочному поведению, и перебора возможных состояний используется в инструментах Rostra [65] и Symstra [66]. Другой пример использования символического выполнения — построения тестов с использованием инструмента Java Path Finder [67], способного симулировать работу Java-машины. Параллельное символическое выполнение и выполнение с конкретными данными, соответствующими символическим ограничениям, используется в инструментах CUTE и jCUTE [68]. Случайная генерация тестов, направляемая эвристиками обнаружения ошибок и сокращением пространства состояний за счет символического выполнения, используется в инструментах DART [69] и Unit Meister [70].

Для построения тестов работоспособности такие подходы вполне пригодны, однако получаемые с их помощью тесты жестко привязаны к коду конкретной версии ПО.

- *Методы генерации тестов на основе формальных моделей.*
  В основе этих методов лежит использование формальной модели поведения

тестируемого ПО. Чаще всего эта модель является автоматом (finite state machine, FSM) или системой помеченных переходов (labeled transition system, LTS). Однако есть и методы построения тестов, основанные на логических моделях ПО и ситуациях, возникающих при их дедуктивном анализе (theorem proving) [71,72].

Методы создания тестов на основе автоматных моделей бывают двух видов.

- Методы построения тестов, основанные непосредственно на покрытии структуры автоматных моделей. Таких методов достаточно много, их обзор и обзоры инструментов на их основе можно найти в [73,74]. К этому типу относится и технология UniTESK [75], созданная в ИСП РАН.
- Инструменты и техники построения тестов, использующие проверку моделей (model checking) для генерации сценариев, приводящих тестируемую систему в специфические состояния [76-79].

Для автоматизации построения тестов работоспособности такие методы слишком неэффективны, поскольку требуют разработки формальных моделей поведения тестируемой системы.

5. Заключение

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

В данной статье представлена созданная в Институте системного программирования РАН технология Azov автоматизации массового создания тестов работоспособности, основанная на уточнении информации о типах параметров и результатов интерфейсных операций и использовании этой информации при построении тестовых данных. Технология применима для систем, имеющих достаточно большой интерфейс (>500 операций), документацию с описанием базовой функциональности интерфейсных операций и базу данных с хорошо структурированной синтаксической информацией о них. Можно заметить, что любой компилятор в принципе способен создать 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 технологии, создавались и дорабатывались непосредственно в ходе этого проекта, были достигнуты высокие показатели производительности.

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] C. J. Colbourn. Combinatorial aspects of covering arrays. Le Matematiche (Catania), 58:121–167, 2004.
[12] A. Hartman, L. Raskin. Problems and algorithms for covering arrays. Discrete Math., 284(1-3):149–156, Jul. 2004.
[13] http://www.pairwise.org/tools.asp.
[14] http://aetgweb.argreenhouse.com/.
[15] D. M. Cohen, S. R. Dalal, A. Kajla, G. C. Patton. The Automatic Efficient Test Generator (AETG) System. Proc. of 5-th International Symposium on Software Reliability Engineering (ISSRE), Monterey, California, November 1994.
[16] http://www.testcover.com/.
[17] G. Sherwood. Effective Testing of Factor Combinations. Proc. of 3-rd International Conference on Software Testing, Analysis & Review, Washington, DC, USA, May 1994.
[18] http://www.satisfice.com/tools.shtml.
[19] http://burtleburtle.net/bob/math/jenny.html.
[20] http://alphaworks.ibm.com/tech/whitch.
[21] http://download.microsoft.com/download/f/5/5/f55484df-8494-48fa-8dbd-8c6f76cc014b/pict33.msi.
[22] http://www.turbodata.ca/.
[23] http://www.sqlmanager.net/products.
[24] http://www.sqledit.com/dg/.
[25] http://www.datatect.com/.
[26] http://www.forsql.com/.
[27] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, E. J. Weyuker. AGENDA: a test generator for relational database applications. Technical Report TR-CIS-2002-04, Polytechnic University, 2002.
[28] C. Binnig, D. Kossmann, E. Lo. Testing database applications. Proc. of the 2006 ACM SIGMOD International Conference, Chicago, IL, USA, pp. 739-741, ACM, 2006.
[29] N. Bruno, S. Chaudhuri. Flexible database generators. Proc. of 31-st International Conference on Very Large Databases, pp. 1097-1107, Trondheim, Norway, 2005.
[30] D. Barbosa, A. Mendelzon. Declarative generation of synthetic XML data. Software: Practice & Experience, 36(10):1051-1079, August 2006.
[31] R. Lämmel, W. Schulte. Controllable combinatorial coverage in grammar-based testing. In Proc. of TESTCOM 2006, LNCS 3964:19-38, Springer, 2006.
[32] http://www.alphaworks.ibm.com/tech/xmlgenerator.
[33] http://xml-xig.sourceforge.net/.
[34] http://iwm.uni-koblenz.de/datagen/.
[35] P. Purdom. A sentence generator for testing parsers. BIT, 12(3):366–375, 1972.
[36] A. Celentano, S. Crespi Reghezzi, P. Della Vigna, C. Ghezzi, G. Granata, F. Savoretti. Compiler Testing using a Sentence Generator. Software: Practice and Experience, 10:897–918, 1980.
[37] P. Maurer. Generating test data with enhanced context-free grammars. IEEE Software, 7(4):50–56, 1990.
[38] С.В. Зеленов, С.А. Зеленова. Генерация позитивных и негативных тестов парсеров. Программирование, 31(6):25–40, 2005.
[39] http://www.mmsindia.com/JSynTest.html.
[40] 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.

[41] 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.
[42] A. Gotlieb, B. Botella, M. Rueher. Automatic test data generation using constraint solving techniques. ACM SIGSOFT Software Engineering Notes, 23(2):53-62, 1998.
[43] R. DeMillo, A. Offutt. Constraint-based automatic test data generation. IEEE Trans. on Software Engineering, 17(9):900-910, 1991.
[44] B. Korel. Automated Test Data Generation. IEEE Trans. on Software Engineering, 16(8):870-879, 1990.
[45] N. Gupta, A. P. Mathur, M. L. Soffa. Automated test data generation using an iterative relaxation method. ACM SIGSOFT Software Engineering Notes, 23(6):231-244, 1998.
[46] http://www.parasoft.com/jsp/products.jsp.
[47] http://www.t-vec.com/solutions/tvec.php.
[48] http://www.suresofttech.com/eng/main/product/api.asp.
[49] D. Marinov, S. Khurshid. TestEra: A novel framework for automated testing of Java programs. Proc. of 16-th IEEE International Conference on Automated Software Engineering, pp. 22–31, 2001.
[50] C. Boyapati, S. Khurshid, D. Marinov. Korat: automated testing based on Java predicates. Proc. of International Symposium on Software Testing and Analysis, pp. 123–133, 2002.
[51] B. Korel. A dynamic approach of automated test data generation. Proc of Conference on Software Maintenance, San Diego, CA, 1990, pp. 311-317.
[52] R. P. Pargas, M. J. Harrold, R. Peck. Test-data Generation Using Genetic Algorithms. Software Testing. Verification & Reliability, 9(4):263–282, 1999.
[53] 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.
[54] B. Korel. Automated Test Data Generation for Programs with Procedures. In Proc. of ISSTA 1996, pp. 209–215.
[55] R. Ferguson, B. Korel. The Chaning Approach for Software Test Data Generation. ACM Transactions on Software Engineering Methodology, 5(1):63–86, 1996.
[56] C. Csallner, Y. Smaragdakis. JCrasher: and Automatic Robustness Tester for Java. Software — Practice & Experience, 34(11):1025–1050, 2004.
[57] C. Csallner, Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. Proc. of 27-th International Conference on Software Engineering (ICSE), pp. 422–431, ACM, May 2005.
[58] C. Csallner, Y. Smaragdakis. DSD-Crasher: A hybrid analysis tool for bug finding. Proc. of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pp. 245–254. ACM, July 2006.
[59] Y. Smaragdakis, C. Csallner. Combining Static and Dynamic Reasoning for Bug Detection. Proc. Of TAP 2007, LNCS 4454, pp. 1-16, Springer, 2007.
[60] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata. Extended static checking for Java. Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 193–205, 2002.
[61] M. D. Ernst, J. Cockrell, W. G. Griswold, D. Notkin. Dynamically discovering likely program invariants to support program evolution. IEEE Trans. on Software Engineering, 27(2):99-123, February 2001.
[62] B. Meyer, I. Ciupa, A. Leitner, L. Liu. Automatic testing of object-oriented software. Proc. of 33-rd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM). Springer, Jan. 2007.
[63] C. Pacheco, M. D. Ernst. Eclat: Automatic generation and classification of test inputs. Proc. of ECOOP, pp. 504-527, July 2005.

[64] C. Pacheco, S. K. Lahiri, M. D. Ernst, T. Ball. Feedback-Directed Random Test Generation. Proc. of ICSE 2007, pp. 75-84.
[65] T. Xie, D. Marinov, D. Notkin. Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests. Proc. of 19-th IEEE International Conference on Automated Software Engineering (ASE 2004), Linz, Austria, pp. 196-205, September 2004.
[66] T. Xie, D. Marinov, W. Schulte, D. Notkin. Symstra: A Framework for Generating Object-Oriented Unit Tests using Symbolic Execution. Proc. of 11-th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005), Edinburgh, UK, pp. 365-381, April 2005.
[67] C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. Pasareanu, G. Rosu, K. Sen, W. Visser, R. Washington. Combining Test Case Generation and Runtime Verification. Theoretical Computer Science (TCS), 336(2-3):209-234, May 2005.
[68] K. Sen, G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. Proc. of Computer Aided Verification, pp.419-423, August 2006.
[69] P. Godefroid, N. Klarlund, K. Sen: DART: directed automated random testing. Proc. of ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, pp. 213-223, 2005.
[70] N. Tillmann, W. Schulte. Parameterized Unit Tests with Unit Meister. ACM SIGSOFT Software Engineering Notes, 30(5):241-244, September 2005.
[71] G. Yorsh, T. Ball, M. Sagiv. Testing, abstraction, theorem proving: better together! Proc. of International Symposium on Software Testing and Analysis 2006, Portland, Maine, USA, pp. 145-156, ACM, 2006.
[72] A. D. Brucker, B. Wolf. Interactive Testing with HOL-TestGen. Proc. of FATES 2006, LNCS 3997:87-102, Springer, 2006.
[73] A. Hartman. Model Based Test Generation Tools. 2002. http://www.agedis.de/documents/ModelBasedTestGenerationTools.pdf.
[74] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, A. Pretschner, eds. Model-Based Testing of Reactive Systems. Advanced Lectures. LNCS 3472, Springer, 2005.
[75] В. В. Кулямин, А. К. Петренко, А. С. Косачев, И. Б. Бурдонов. Подход UniTesK к разработке тестов. Программирование, 29(6):25–43, 2003.
[76] P. Ammann, P. E. Black. Abstracting Formal Specifications to Generate Software Tests via Model Checking. NIST-IR 6405 (extended version), 1999.
[77] A. Gargantini, C. Heitmeyer. Using Model Checking to Generate Test from Requirements Specifications. Proc. of Joint 7-th European Software Engineering Conf. and 7-th ACM SIGSOFT Intern. Symposium on Foundations of Software Engineering (ESEC/FSE99), Toulouse, France, September 1999.
[78] G. Devaraj, M. P. E. Heimdahl, D. Liang. Coverage-Directed Test Generation with Model Checkers: Challenges and Opportunities. Proc. of 29-th Annual International Computer Software and Applications Conference (COMPSAC'05), Vol. 1, pp. 455-462, 2005.
[79] D. Beyer, T. A. Henzinger, R. Jhala, R. Majumdar. The Software Model Checker Blast: Applications to Software Engineering. International Journal on Software Tools for Technology Transfer (STTT), 9(5-6):505-525, 2007.