Генерация оптимизированных для ручного выполнения сценариев тестирования приложений с графическим интерфейсом пользователя

А. В. Баранцев, С. В. Грошев, В. А. Омельченко
{barancev,sgroshev,vitaliy}@ispras.ru

Аннотация. В статье описывается решение задачи построения последовательностей действий пользователя, оптимизированных для ручного выполнения, на основе модели в виде диаграммы состояний и переходов. Сценарии для ручного выполнения необходимы, во-первых, когда графический интерфейс является единственной возможностью взаимодействия с приложением, а его реализация не предусматривает или делает экономически невыгодным программную эмуляцию воздействий через него. Во-вторых, тестовые наборы для ручного выполнения могут быть необходимы для оценки практичности графического интерфейса пользователя, для проверки его соответствия выбранным стандартам и для приемочного пользовательского тестирования.

1. Введение

Графический интерфейс пользователя (ГИП) давно и прочно занял доминирующее положение среди способов взаимодействия с прикладными программами (приложениями) общего назначения. Взаимодействие пользователей с офисными и корпоративными приложениями в подавляющем большинстве случаев заключается в совершении некоторых последовательностей действий с графическим интерфейсом. Поэтому для обеспечения качества современных программных продуктов важной задачей является проверка корректности внешнего поведения приложений при взаимодействии пользователя с ГИП. Такая проверка предполагает выполнение различных сценариев, состоящих из последовательностей воздействий на приложение через ГИП, с проверкой промежуточных и конечного результатов. Выполнение таких последовательностей вручную представляет из себя рутинную ресурсоемкую деятельность, особенно при необходимости неоднократного воспроизведения одних и тех же сценариев, например, при регрессионном тестировании. Поэтому существующие

инструменты автоматизации тестирования приложений с ГИП прежде всего нацелены на автоматизацию выполнения тестов: воспроизведение заранее заданных (как правило, вручную) сценариев взаимодействия с приложением через ГИП.

Для автоматизации выполнения тестов необходимо иметь возможность через какой-то интерфейс подавать стимулы, то есть воздействовать на тестируемую программу, и принимать реакции, то есть наблюдать за тем, что делает программа в ответ на эти воздействия. Для программ с ГИП подача стимулов на логическом уровне означает эмуляцию действий пользователя, таких как нажатие кнопок, ввод данных в поля ввода, выбор элементов из списков и так далее. На физическом уровне ввод данных требует от инструмента возможности эмулировать нажатие кнопок на клавиатуре, а нажатие кнопок и выбор элементов из списка могут быть выполнены как с клавиатуры, так и эмуляцией действий с мышью – перемещение и нажатие кнопок манипулятора. Для наблюдения реакций необходимо распознавать элементы ГИП, то есть определять наличие тех или иных элементов и получать значения их атрибутов. Эмуляция действий с мышью также требует умения распознавать элементы ГИП, поскольку координаты объекта на экране являются частным случаем этих атрибутов.

Во многих случаях для распознавания элементов ГИП и получения их атрибутов есть возможность использовать специальные программные интерфейсы такие, как, например, интерфейсы графических библиотек Swing/AWT или SWT/RCP/Eclipse на платформе Java [1], Windows Presentation Framework для приложений на платформе .NET [2], COM-интерфейсы для программ в Microsoft Word или Microsoft Excel [3].

Однако имеется ряд программ, для которых распознавание элементов ГИП невозможно в силу технологических причин, так как используемая для создания ГИП библиотека не реализует ни один из механизмов доступа к атрибутам объектов. Такие программы позволяют пользователям разработать собственные формы, но используемые при этом элементы ГИП не позволяют получить доступ к их атрибутам. В результате вся созданная пользователем форма распознаётся как один большой элемент (например, типа RichTextBox для IBM Lotus Notes), а находящиеся внутри него элементы распознать не удаётся. Иногда в таких случаях возможно решить проблему автоматизации подачи стимулов с помощью с помощью внедрения внутрь тестируемой программы агентов, которые обеспечивают доступ к требуемой информации об элементах ГИП (например, перекомпиляция Delphi-приложений как Open Application для тестирования с помощью инструмента AutomatedQA TestComplete [4]).

В остальных случаях единственным способом автоматизации подачи стимулов через ГИП является запись и последующее воспроизведение взаимодействий пользователя с интерфейсом в терминах экранных координат и нажатия кнопок мыши и клавиатуры. Основные недостатки такого способа в

том, что он позволяет создавать набор регрессионных тестов только для практически законченного и уже функционирующего приложения, и любые изменения не только в функциональности, но и в дизайне ГИП, как правило, требуют переработки тестов. Иногда при такой записи можно использовать определение координат элементов ГИП методом поиска и сравнения фрагментов снимков экрана или поиска распознанного со снимков экрана текста [5], что несколько снижает зависимость тестов от внешнего расположения элементов, но не позволяет сделать тесты независимыми даже от незначительных изменений дизайна графического интерфейса, при которых инструмент не сможет сопоставить функционально одинаковые элементы в новой и старой версиях интерфейса. Также снизить зависимость тестов от обновления внешнего вида ГИП иногда позволяет полный отказ от использования мыши и управление тестируемой программой только с помощью клавиатуры, но не во всех приложениях есть возможность доступа ко всем элементам ГИП при помощи клавиатуры.

В таких условиях автоматизация тестирования становится экономически невыгодна. Если в обычных условиях, когда распознавание элементов ГИП не вызывает затруднений, расходы на автоматизацию превышают расходы на один цикл ручного тестирования в 3 раза [6], то усложнение создания и особенно сопровождения тестов, связанное с трудностями распознавания элементов ГИП, по нашим оценкам может увеличить эту цифру ещё в 2-3 раза.

Кроме того, в случае приложений с ГИП к значимому для пользователя поведению кроме собственно функциональности приложения относится так же внешний вид интерфейса с точки зрения практичности, соответствия некоторому заданному стандарту, эстетичности и т. д. Автоматизированные тесты, как правило, не предназначены для оценки такого рода характеристик качества. В них обычно программируется только небольшая часть наиболее важных с точки зрения функциональности проверок. Большая часть дефектов, связанных с визуальным представлением ГИП и практичностью, не обнаруживается автоматизированными тестами, например, «выползание» текста за границы, «наползание» элементов ГИП друг на друга, неудобное расположение элементов интерфейса, приводящее к частым ошибкам пользователей и т. п. Поэтому, независимо от наличия или отсутствия автоматизированных тестов, необходимо также иметь тестовый набор для ручного выполнения. При этом он должен обеспечивать заданный уровень полноты тестирования и быть достаточно прост в сопровождении.

Отдельный тест из такого ручного набора представляет собой сценарий, состоящий из последовательности действий и проверок, которые должен выполнить тестировщик. В целом тестовый набор должен обеспечивать некоторую полноту тестирования. Поскольку человек более подвержен ошибкам, чем компьютер, сценарии не должны быть слишком длинными, так как в результате ошибки тест придётся выполнять сначала. В свою очередь, «измельчение» тестов приводит к тому, что некоторые действия приходится

выполнять неоднократно, в каждом тесте заново, что также увеличивает общую трудоёмкость. В то же время, нужно стремиться сократить общую трудоёмкость всех тестов, то есть не количество тестов, а суммарную длину последовательностей действий, из которых состоят отдельные тесты, сохраняя при этом приемлемую полноту тестирования. Поэтому при поиске решения необходимо найти способ сохранить баланс между этими противоречивыми требованиями.

В статье рассматривается подход, при котором тесты для ручного выполнения генерируются автоматически из некоторой модели, описывающей поведение приложения при взаимодействии с ним пользователя. Такой подход гарантирует полноту покрытия по построению (способ генерации зависит от выбранного критерия полноты) и дает возможность быстро перегенерировать тесты в случае изменения модели (в связи с изменением функциональности приложения или дизайна ГИП).

В целях проверки предложенного подхода создана прототипная реализация инструмента генерации тестовых последовательностей для проверки интерфейса приложений, реализованных в системе документооборота на основе Lotus Notes. Платформа Lotus Notes включает в себя интегрированную среду разработки, предоставляющую разработчикам средства для разработки приложений с ГИП, поддерживающих управление документами и данными, а так же рабочими потоками внутри организации. В среде Lotus Notes отсутствуют возможности программного доступа к элементам ГИП. Типичным сценарием взаимодействия с приложением Lotus Notes является последовательность воздействий на некоторый документ, который в результате этих воздействий может переходить в новые состояния. В качестве формализованной модели поведения таких приложений удобно использовать диаграммы состояний и переходов, так как, с одной стороны, при помощи таких диаграмм достаточно естественно описываются состояния документа и переходы между ними при воздействиях пользователя, а с другой стороны, эти диаграммы достаточно формальны для автоматической обработки с целью генерации на их основе тестовых сценариев с заданными свойствами.

2. Модель приложения

Документ в системе документооборота может находиться в различных состояниях. Переход между состояниями осуществляется при помощи воздействий на документ через ГИП. В системе документооборота определены роли пользователей, при этом воздействия делятся на 3 класса в зависимости от того, кем они могут осуществляться:

1. определённым набором ролей;
2. любым пользователем системы, независимо от его роли;
3. не пользователями системы (например, тайм-аут).

Предполагается, что при тестировании мы имеем возможность входить в систему документооборота в любой определённой в её спецификации роли, проверять возможность или невозможность осуществления воздействий данной ролью, а также осуществлять над документом любое описанное в спецификации воздействие. В противном случае соответствующие роли и воздействия исключаются из используемой для тестирования модели документа.

Входными данными задачи тестирования документооборота являются список ролей пользователей системы и описание жизненного цикла (ЖЦ) документа. Описание ЖЦ представляется в виде графа состояний Конечного Автомата (КА), начальное состояние которого соответствует началу ЖЦ, переходы из начального состояния – различным способам создания документа данного типа, конечное состояние (их может быть несколько) – завершению ЖЦ, прочие состояния – промежуточным состояниям документа, а переходы – воздействиям на документ. Если воздействие относится к первому классу, то соответствующий переход КА помечен списком ролей, которым разрешено выполнять соответствующее воздействие.

В используемой нами модели также предусмотрена возможность пометить некоторые воздействия (желательно, но не обязательно оставляющие документ в том же состоянии), которые мы считаем надёжными или по каким-то другим причинам не хотим тестировать. Например, редактирование текста документа, которое доступно (возможно, не всем ролям) практически в любом состоянии документа и не переводит притом его в другое состояние. Для таких воздействий мы проверяем возможность их осуществления, но не выполняем их в ходе теста.

Используя такую модель, мы получаем возможность использовать для тестирования систем документооборота методы тестирования КА, адаптированные к особенностям предметной области. В выбранном нами методе тестирования каждый тест описывает ЖЦ некоторого документа от создания до уничтожения и, соответственно, моделируется некоторым маршрутом по графу состояний модельного КА, ведущим из начального его состояния в конечное.

3. Тестовые проверки

В первую очередь тестируется живучесть системы документооборота и её способность выполнять заявленные воздействия. Проверяется, что система не завершает аварийно работу при выполнении осуществляемых над документом воздействий.

Поскольку спецификация ЖЦ документа представлена в виде графа состояний, при тестировании мы проверяем изоморфизм наблюдаемого графа состояний документа модельному графу. Для этого в каждом состоянии мы проверяем список доступных в нём воздействий (соответствующих переходам

из состояния), а после выполнения каждого тестового воздействия проверяем, соответствует ли полученное состояние документа ожидаемому. Расхождение между ожидаемым и полученным состоянием считается ошибкой и требует исправления тестируемой системы или пересмотра спецификации.

Согласно используемой модели некоторые воздействия могут быть выполнены только определёнными ролями; для тестирования этого требования мы проверяем соответствие между ролями и доступными им воздействиями. Обнаруженная в ходе тестирования невозможность осуществить некоторой ролью воздействие, которое описано в спецификации как возможное для неё, равно как и наоборот, считается ошибкой.

Другие проверки в данную модель не входят, но могут быть добавлены при ручной доработке сгенерированного по ней тестового набора.

4. Критерии тестового покрытия

Элементарными событиями будем называть события следующих видов:

- Попадание документа в заданное состояние.
- Применение к документу в заданном состоянии заданного тестового воздействия.

Поскольку нам необходимо протестировать все состояния и воздействия, возможные в ЖЦ документа, то естественным образом возникает *первый критерий тестового покрытия*: *достижение всех элементарных событий*.

Каждое элементарное событие принадлежит некоторому тесту, моделирующему ЖЦ одного документа. События внутри теста линейно упорядочены.

Событие `B` будем считается *наступившим после* события `A`, если оба эти события произошли внутри одного теста, и событие `B` произошло в нём позже события `A`. Событие `B` будем считать наступившим *не после* события `A`, если в рамках теста, которому оно принадлежит, ему не предшествовало событие `A`.

В состоянии документа возможны скрытые составляющие, которые влияют на поведение системы. Они могут изменяться при попадании документа в некоторые критичные состояния или при применении к нему некоторых критичных воздействий. Например, если документ был удалён, а потом восстановлен, это может отразиться на его дальнейшем поведении. Элементарные события, которые предположительно могут влиять таким образом на поведение, будем называть *существенными в истории событиями*.

Для тестирования таких возможных аномалий поведения введем второй критерий тестового покрытия: достижение каждого элементарного события отдельно после и не после каждого события, существенного в истории. В общем случае все существенные в истории события могут соответствовать различным взаимно независимым скрытым элементам состояния документа, и различные возможные комбинации наличия/отсутствия в истории документа

таких событий соответствуют потенциально различным состояниям. Однако, чтобы избежать комбинаторного взрыва состояний и излишней сложности тестирования, существенные в истории события рассматриваются независимо друг от друга, и покрытие возможных их комбинаций не учитывается.

Для особых случаев введем *третий критерий тестового покрытия*: разработчик тестов имеет возможность на основании дополнительных знаний о системе дополнительно отмечать интересующие его пары элементарных событий и требовать покрытия одного события в паре после и/или не после другого.

Элементы всех трёх критериев покрытия могут быть представлены как достижение некоторого элементарного события `B` после или не после некоторого элементарного события А. События такого вида мы будем называть *тестовыми ситуациями*. При этом первый критерий покрытия порождает все возможные тестовые ситуации вида *«B после A»*, где в качестве элементарного события `A` выступает начальное состояние графа, а в качестве элементарного события `B` – все определённые в графе состояний документа элементарные события. Второй критерий покрытия порождает все возможные тестовые ситуации видов *«B после A»* и *«B не после A»*, где в качестве `A` выступают существенные в истории элементарные события, а в качестве `B` – определённые в графе состояний документа элементарные события. Элементы третьего критерия покрытия переводятся в тестовые ситуации очевидным образом.

При разработке тестов документооборота для оценки полноты тестового покрытия используется составной критерий достижения всех тестовых ситуаций, построенных для модели ЖЦ документа по вышеописанным правилам из трех вышеописанных критериев. В дальнейшем под критерием покрытия имеется ввиду только этот критерий покрытия.

5. Оптимизация тестового набора

Нами выделены следующие критерии оптимизации, которые необходимо учитывать при создании тестов для ручного тестирования, в порядке понижения их приоритета:

1. Максимизация тестового покрытия.
2. Минимизация суммарной трудоёмкости тестового набора.
3. Ограничение трудоёмкости отдельных тестов: поскольку тесты исполняются не машиной, а человеком, то при превышении тестом некоторого порога сложности сильно возрастает вероятность ошибки тестировщика, в результате которой этот тест придётся исполнять заново.

Эти критерии оптимизации противоречат друг другу: так, увеличение полноты тестового покрытия вызывает увеличение трудоёмкости отдельных тестов и тестового набора в целом, а ограничение длины отдельных тестов обычно приводит к увеличению количества таких тестов и, как следствие, суммарной сложности тестового набора, так как зачастую в различных тестах одного набора приходится заново выполнять одни и те же действия.

Исходя из приоритетов поставленной задачи тестирования, была поставлена задача многокритериальной оптимизации набора тестов, для которой выбрана следующая стратегия решения:

1. Фиксируем некое разумное (с учётом размера и структуры графа состояний документа) ограничение на длину отдельных тестов. Длиной теста здесь и далее считается количество применяемых в нём тестовых воздействий.
2. Строим тестовый набор с минимальной суммарной длиной тестов, удовлетворяющий следующим ограничениям:
   2.1. Каждый тест в составе тестового набора имеет длину не выше заданной.
   2.2. Тестовый набор должен обеспечивать покрытие всех возможных при ограничении 2.1 тестовых ситуаций.
3. Если некоторые тестовые ситуации могут быть покрыты только тестами длины большей, чем заданное в п.1 ограничение, то строим тесты минимальной суммарной длины, покрывающие все такие тестовые ситуации, и добавляем их к построенному тестовому набору.
   3.1. В этом случае проверяем, не покрывают ли добавленные на шаге 3 длинные тесты сразу все тестовые ситуации, покрываемые какими-то из ранее построенных коротких тестов, и при обнаружении таких «лишних» коротких тестов удаляем их из тестового набора.

6. Алгоритм построения тестового набора

Разработанный алгоритм сначала проверяет полученный на вход граф состояний документа согласно следующим ограничениям целостности:

- Граф должен содержать ровно одно начальное состояние и как минимум одно конечное состояние, достижимое из начального. При нарушении этого условия выдаётся сообщение об ошибке, и алгоритм прекращает работу.
- Граф не должен содержать состояний, недостижимых из начального или из которых не достижимо ни одно конечное состояние. При обнаружении таких состояний выдаётся предупреждение, и они удаляются из графа вместе со всеми инцидентными им переходами.

После проверки входного графа состояний документа генерируются все возможные на нём тесты. Каждый тест моделируется маршрутом по графу,

ведущим из начального состояния в одно из конечных. Тесты генерируются без ограничения длины, но со следующим ограничением: рассматриваются только такие маршруты, которые проходят через каждое состояние не более 2 раз. Такое ограничение позволяет разумно ограничить количество тестов, из которых будет в дальнейшем конструироваться тестовый набор, гарантируя притом покрытие любого элементарного события (включая петлевые дуги, а также такие дуги и вершины, которые недостижимы на путях без циклов), но не гарантирует покрытие всех возможных упорядоченных пар событий. Применение данного алгоритма на графах реально используемых документов, даже с очень сложными критериями покрытия, показало, что такое ограничение оставляет непокрытыми только тестовые ситуации настолько нетривиальные, что ими в большинстве случаев можно пренебречь. При завершении работы инструмент выдаёт вместе с построенных тестовым набором список непокрытых тестовых ситуаций для дальнейшего анализа и, при необходимости, ручной доработки тестового набора.

Далее строится полный набор тестовых ситуаций, которые необходимо покрыть, согласно правилам, описанным в главе «Критерии тестового покрытия», и из него удаляются тестовые ситуации некоторых особых видов, в частности:

- Ситуации вида *«Начальное состояние после любого события»*, как заведомо недостижимые.
- Ситуации вида *«Операция создания документа после любого события, кроме начального состояния»*, как заведомо недостижимые.
- Ситуации вида *«Любое событие не после начального состояния»*, как заведомо недостижимые.
- Ситуации вида *«Переход после своего начального состояния»*, как заведомо дублирующие ситуации *«Переход после начального состояния документа»*, так как одна ситуация не может быть достигнута при не достигнутой другой.
- Ситуации вида *«Переход не после своего начального состояния»*, как заведомо недостижимые.

Для каждого построенного на предыдущем шаге теста вычисляется множество тестовых ситуаций, покрываемых им. Все построенные тестовые ситуации считаются при дальнейшей оптимизации тестового набора равноценными.

Далее из построенного множества путей строится тестовый набор минимальной сложности, покрывающий все возможные тестовые ситуации, с помощью нижеописанного эвристического алгоритма.

В записи алгоритма используются следующие условные обозначения:

`|S|` - мощность множества `S`
`Len(t)` - длина теста `t`, измеряемая в количестве тестовых воздействий
`{}` - пустое множество

`<>` - пустой упорядоченный список

Фиксируем ограничение длины тестов `M`, которое является параметром данного алгоритма. Полагаем множество непокрытых тестовых ситуаций `C:=` полный набор построенных ранее тестовых ситуаций, множество доступных тестов `T:=` полный набор ранее построенных тестов, а тестовый набор `R:=<>`.

Тестовый набор строится в две итерации: на первой рассматриваются только тесты с длиной, не превышающей `M`, а на второй рассматриваются все оставшиеся тесты.

На каждой итерации применяется следующий «жадный» алгоритм:

1. Для каждого теста `t` из `T` вычисляем `Cov(t):=` множество тестовых ситуаций из `C`, покрываемых им. Если `Cov(t)={}`, то удаляем `t` из `T`.
2. Перебираем в произвольном порядке все тесты `t` из всего множества `T` на второй итерации, и только те тесты, длина которых не превышает `M`, на первой. Если перебираемое множество пусто, то завершаем текущую итерацию. Ищем среди перебираемых тестов *лучший* следующим образом:
   - 2.1. Первый встреченный в ходе перебора тест объявляем *лучшим*.
   - 2.2. Каждый следующий тест в ходе перебора сравниваем с текущим *лучшим* по следующим критериям сравнения, по порядку:
     - 2.2.1. если $\mathrm{Len}(t_1) \le M$ и $\mathrm{Len}(t2) > M$, то t_1 лучше;
     - 2.2.2. если тесты t_1 и t_2 оба укладываются в ограничения по длине и при этом $|\mathrm{Cov}(t_1)| > |\mathrm{Cov}(t_2)|$, то t_1 лучше;
     - 2.2.3. если $|\mathrm{Cov}(t_1)| \ge |\mathrm{Cov}(t_2)|$ и $\mathrm{Len}(t_1) < \mathrm{Len}(t_2)$, то t_1 лучше;
     - 2.2.4. если $|\mathrm{Cov}(t_1)| > |\mathrm{Cov}(t_2)|$ и $\mathrm{Len}(t_1) \le \mathrm{Len}(t_2)$, то t_1 лучше;
     - 2.2.5. если оба теста t_1 и t_2 не укладываются в ограничения по длине и $(\mathrm{Len}(t_1 - \mathrm{Len}(t2))*P < (|\mathrm{Cov}(t_1)|-|\mathrm{Cov}(t_2)|)$, где `P` – параметр алгоритма, то t_1 лучше.
   - 2.3. Если проверки 2.2.1–2.2.5 показывают, что проверяемый тест лучше предыдущего *лучшего*, то объявляем *лучшим* его, и далее сравниваем другие перебираемые тесты уже с ним. Если проверяемый тест признан хуже или ни одна проверка не выносит вердикт, что один тест лучше другого, то продолжаем считать *лучшим* тот же тест, что и раньше.

3. Добавляем найденный *лучший* тест t в конец списка R и удаляем из множества непокрытых тестовых ситуаций C все элементы множества Cov(t).
4. Переходим к шагу 1.

Правило сравнения 2.2.1 гарантирует, что в первую очередь все возможные тестовые ситуации будут покрываться тестами с длиной, не превышающей максимальную.

Правило 2.2.2 выбирает тесты с максимально возможным (в пределах ограничения длины) тестовым покрытием – в большинстве случаев эта эвристика позволяет уменьшить суммарную длину тестов в наборе за счёт уменьшения их количества.

Правила 2.2.3 и 3.2.4 отдают предпочтение тестам, улучшающим покрытие или длину и не ухудшающим притом другую из этих двух характеристик.

Согласно правилу 2.2.5 из двух тестов, превышающих максимальную длину, один предпочтительнее другого, если увеличение длины компенсируется (с учётом весового параметра P) увеличением покрытия.

Поскольку алгоритм эвристический, и в нём используется перебор по неупорядоченному множеству, в общем случае результаты его работы не детерминированы: при разных запусках на одних и тех же данных возможна генерация различных тестовых наборов; однако в большинстве случаев сгенерированные таким образом наборы имеют одинаковые метрики.

Для настройки алгоритма используются параметры M и P.

M: задаёт максимальную длину тестов генерируемого набора, превышение которой допускается только для покрытия недостижимых иными способами тестовых ситуаций. Допустимые значения: целое положительное число или $+\infty$.

P: задаёт вес превышения максимальной длины тестов по отношению к повышению покрытия. При больших значениях параметра алгоритм в первую очередь выбирает из тестов, превышающих максимальную длину и добавляющих хотя бы какое-то тестовое покрытие, самые короткие, что позволяет уменьшить максимальную длину тестов, но может приводить к существенному увеличению суммарной сложности тестового набора; при малых значениях алгоритм в первую очередь выбирает тесты, покрывающие большее количество тестовых ситуаций, что позволяет минимизировать суммарную сложность тестового набора за счёт добавления в него очень длинных тестов. Допустимые значения: неотрицательные числа (можно нецелые).

Оптимальные значения обоих параметров зависят от размера и структуры графа состояний документа, от требований к тестовому покрытию и от возможностей тестировщиков исполнять длинные тесты. В ходе апробации алгоритма установлено, что в большинстве случаев оптимальное значение для

параметра M немного превосходит длину самого длинного возможного в графе маршрута без циклов, ведущего из начального состояния в конечное, а для параметра P лежит в диапазоне [0; 2].

7. Оптимизация и балансировка тестового набора

Далее производится фильтрация и сортировка построенного набора тестов R; при этом используется упорядоченность тестового набора по времени добавления в него тестов.

Для фильтрации вычисляется полный набор тестовых ситуаций, покрываемых каждым тестом из построенного тестового набора, и для каждого теста проверяется, добавляет ли он какое-либо тестовое покрытие к суммарному покрытию тестов, добавленных в набор после него (напоминаем, что при добавлении «поздних» тестов в набор учитывалось только то покрытие, которое они добавляли к уже имеющемуся на момент добавления); если все тестовые ситуации, покрываемые тестом, покрываются объединением более поздних тестов, то он удаляется из построенного набора.

Отфильтрованный таким образом набор тестов сортируется по длине: самые короткие (по количеству тестовых воздействий) тесты ставятся в начало набора, а самые длинные – в его конец; это позволяет в дальнейшем сбалансировать количество проверок, выполняемых в различных тестах набора.

8. Генерация готового к исполнению тестового набора

Основным результатом работы инструмента является набор тестов, в котором каждый тест представляет собой пошаговую инструкцию для тестировщика. Каждая такая инструкция представляет собой последовательный список, состоящий из воздействий на документ и проверок, которые надо выполнить.

Инструмент генерирует проверки следующих видов:

- Проверка успешности выполнения воздействия.
- Проверка состояния документа. Проверка считается успешной, если состояние документа соответствует ожидаемому.
- Проверка списка воздействий, доступных для определённой роли. Проверка считается успешной, если список доступных для данной роли воздействий совпадает с указанным списком. Список воздействий может быть пустым.
- Проверка списка воздействий, доступных для всех остальных ролей, включая не описанные в используемой для генерации тестов модели документа.

Если в ходе исполнения теста какая-то из проверок не прошла успешно, то мы считаем, что тест обнаружил ошибку.

Для генерации инструкции тестировщику в построенном, отфильтрованном и отсортированном тестовом наборе для каждого теста строится последовательность происходящих в нём элементарных событий: воздействий и состояний. Далее для каждого события определяются тестовые ситуации, добавляемые им к множеству тестовых ситуаций, покрытых в рамках тестового набора ранее, то есть, предыдущими тестами из набора, а также предыдущими элементарными событиями данного теста.

Для каждого выполняемого в ходе теста воздействия генерируется указание тестировщику выполнить его (с указанием ролей, которыми можно его выполнять, или указанием, что роль может быть любой) и проверить успешность выполнения.

Для достигнутых в ходе теста состояний проверки генерируются в зависимости от того, добавляет ли достижение данного состояния покрытие новых тестовых ситуаций: если попадание в состояние (с учётом предыстории элементарных событий данного теста) не даёт нового покрытия, то генерируется только указание проверить полученное состояние документа; если тестовое покрытие увеличилось (то есть, мы впервые попали в данное состояние или впервые попали в него с такими существенными элементами предыстории), то дополнительно генерируются проверки воздействий, доступных различным ролям.

Для каждой роли «правильное» множество доступных воздействий состоит из:

1. Всех доступных согласно спецификации в текущем состоянии воздействий, не помеченных списком допустимых ролей. В частности, сюда относятся воздействия, которые выполняются не пользователями системы: например, тайм-аут можно выполнить, войдя в систему под любой ролью или даже вообще не входя в неё.
2. Всех доступных воздействий, помеченных списком допустимых ролей, таких что в их список входит данная роль.

Проверки генерируются для каждой роли, встречающейся в пункте 2.

Также генерируется проверка воздействий, доступных всем остальным ролям – в соответствующий список входят все доступные в текущем состоянии воздействия, не помеченные списком допустимых ролей.

Поскольку проверки возможных воздействий генерируются только при достижении новых в рамках тестового набора элементов покрытия, сортировка тестов по длине позволяет сбалансировать сложность их исполнения: первые тесты набора коротки, но на каждом шаге дают новые элементы покрытия, а следовательно, и проверки; последние тесты набора длинны, но новые элементы покрытия в них возникают лишь иногда. Разумеется, после того, как тестовый набор сгенерирован и сбалансирован,

составляющие его тесты можно без изменений исполнять в произвольном порядке, не теряя при этом тестового покрытия.

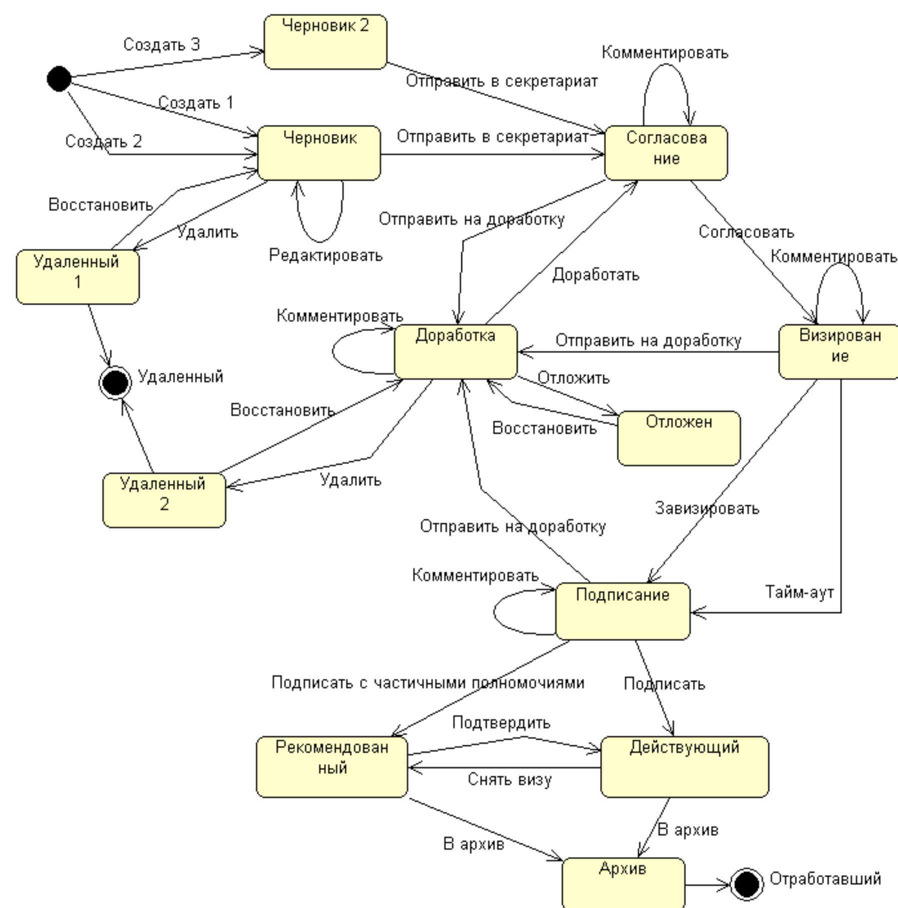

Рис. 1. Пример графа приложения системы документооборота

9. Пример применения

Инструмент реализован на языке Java версии 1.5 и предоставляет программный интерфейс, позволяющий сконструировать граф состояний документа, разметить в нём переходы списками ролей, пометить некоторые переходы как не тестируемые, задать критерии покрытия и параметры работы

главного алгоритма, после чего запустить генерацию тестов. Ввод данных может осуществляться специально написанной для данного графа программой на языке Java (которую можно считать текстовой формой представления графа) или специально разработанными конвертерами, которые будут обходить граф в некотором другом представлении и вызывать соответствующие методы данного интерфейса.

Граф представленный на Рис. 1 был создан на основе анализа требований к реально существующему приложению системы документооборота, после чего дополнительно усложнён для получения нетривиальных путей (например, добавлено состояние «Отложен», недостижимое на путях без циклов). Почти в каждом состоянии документа возможны также его редактирование и комментирование, причём оба этих воздействия не меняют состояние документа с точки зрения данной автоматной модели. Для сокращения трудоёмкости тестирования в большинстве состояний их решено не тестировать, а учесть в списке не тестируемых воздействий, для которых при тестировании только проверяется возможность их выполнения; в результате на рисунке фигурируют только те петлевые дуги этих двух видов, которые решено тестировать дополнительно. Чтобы не загромождать излишне приведённый рисунок, на нём также не отображены списки ролей, которыми помечены переходы графа.

В результате применения генератора тестов к приведённому на Рис. 1 графу с параметром `M=12` (параметр `P` в данном случае несущественен, так как все сгенерированные тесты укладываются в ограничение по длине) и без дополнительных требований к покрытию он выдаёт тестовый набор следующего вида (приводится с сокращениями):

```
Test #0:
[+] <Начать тест>
[+] Действие: Создать 2; роль: [Автор]
[ ] Проверка: действие завершено успешно
[ ] Проверка: состояние документа = Черновик
[ ] Проверка: доступные действия:
 [ ] Действия, доступные для роли Автор:
  [ ] Отправить в секретариат
  [ ] Редактировать
  [ ] Удалить
  [ ] <Других нет>
 [ ] Действия, доступные для всех остальных: <пусто>
[+] Действие: Удалить; роль: [Автор]
[ ] Проверка: действие завершено успешно
[ ] Проверка: состояние документа = Удаленный 1
[ ] Проверка: доступные действия:
 [ ] Действия, доступные для роли Автор:
  [ ] Восстановить
  [ ] <Завершить тест>
  [ ] <Других нет>
 [ ] Действия, доступные для всех остальных:
```



```
  [ ] <Завершить тест>
  [ ] <Других нет>
[+] Действие: <Завершить тест>; роль: <любая>
[ ] Проверка: действие завершено успешно
[ ] Проверка: жизненный цикл документа завершен
------------------------
Test #1:
[+] <Начать тест>
[+] Действие: Создать 2; роль: [Автор]
[ ] Проверка: действие завершено успешно
[ ] Проверка: состояние документа = Черновик
[+] Действие: Отправить в секретариат; роль: [Автор]
[ ] Проверка: действие завершено успешно
[ ] Проверка: состояние документа = Согласование
[ ] Проверка: доступные действия:
 [ ] Действия, доступные для роли Автор:
  [ ] Редактировать
  [ ] Комментировать
  [ ] <Других нет>
 [ ] Действия, доступные для роли Визирующий:
  [ ] Комментировать
  [ ] Согласовать
  [ ] <Других нет>
 [ ] Действия, доступные для роли Подписывающий:
  ...
[ ] Действия, доступные для роли Секретарь:
  [ ] Редактировать
  [ ] Отправить на доработку
  [ ] Комментировать
  ...
 [ ] Действия, доступные для всех остальных: <пусто>
[+] Действие: Согласовать; роль: [Визирующий,
Подписывающий, Секретарь]
[ ] Проверка: действие завершено успешно
[ ] Проверка: состояние документа = Визирование
[ ] Проверка: доступные действия:
...
[+] Действие: Отправить на доработку; роль: [Визирующий,
Секретарь]
[ ] Проверка: действие завершено успешно
[ ] Проверка: состояние документа = Доработка
[ ] Проверка: доступные действия:
...
[+] Действие: Удалить; роль: [Автор]
[ ] Проверка: действие завершено успешно
[ ] Проверка: состояние документа = Удаленный 2
[ ] Проверка: доступные действия:
...
[+] Действие: Восстановить; роль: [Автор]
[ ] Проверка: действие завершено успешно
```

```
[ ] Проверка: состояние документа = Доработка
[+] Действие: Удалить; роль: [Автор]
[ ] Проверка: действие завершено успешно
[ ] Проверка: состояние документа = Удаленный 2
[+] Действие: <Завершить тест>; роль: <любая>
[ ] Проверка: действие завершено успешно
[ ] Проверка: жизненный цикл документа завершен
------------------------
Test #2:
...
```

Также выдаются метрики построенного тестового набора и список непокрытых тестовых ситуаций:

```
Total: 5 pathes, sum length = 45 transitions
Maximal path length = 12
Uncovered elements: <empty>
```

Ручной анализ приведённого графа ЖЦ документа и сгенерированного для него тестового набора показывает, что тестовый набор меньшей длины, покрывающий все переходы, невозможен.

Тестовый набор может быть распечатан в таком виде и передан тестировщику. Значок «`[ ]`» служит местом, куда тестировщик может ставить отметки об успешном выполнении проверок.

10. Заключение

Предложенный в данной работе подход не уменьшает и не увеличивает количество работы на этапе создания тестов по сравнению с ручным проектированием тестов, поскольку ручное проектирование в любом случае предполагает создание некоторой модели, может быть чуть менее формализованной, для обеспечения полноты тестирования. В то же время этот подход имеет ряд дополнительных достоинств:

- гарантированная по построению полнота тестового набора относительно выбранного критерия;
- минимизация размера тестового набора;
- простота сопровождения (перегенерации) тестового набора при внесении изменений в функциональность тестируемого приложения или дизайн его ГИП.

Сгенерированные сценарии могут использоваться как для ручного выполнения тестов (функционального тестирования, пользовательского приемочного тестирования, тестирования практичности), так и для создания набора регрессионных автоматизированных тестов. Но здесь есть серьёзная проблема. Когда в модель вносятся изменения, вызванные изменениями в реализации, новые сгенерированные тестовые сценарии могут существенно

отличаться от тех, которые были сгенерированы по предыдущей модели. Это означает, что автоматизированные тесты придётся создавать заново.

Для преодоления этого ограничения планируется разработать механизм генерации непосредственно исполнимого кода тестовых сценариев. Подобный подход генерации исполнимых тестовых сценариев для ГИП по диаграмме состояний и переходов используется, например, в [7]. Но все известные нам решения предполагают интеграцию с уже существующими инструментами автоматизации, то есть неявно накладывают требование на то, чтобы элементы ГИП были распознаваемы и была возможность получения их атрибутов.

К сожалению, как мы уже отмечали в начале статьи, требование доступности элементов ГИП выполняется не всегда. Чтобы работать с такими интерфейсами, необходимо помимо логической модели поведения в виде диаграммы состояний и переходов построить также физическую модель ГИП. Используя сочетание этих двух моделей можно достичь следующего уровня автоматизации создания и выполнения тестов. Это является предметом дальнейших исследований и развития описанного в данной статье подхода к разработке тестов.

Литература

[1] Test-driven GUI development with FEST Try a functional approach to testing Swing GUIs in test-driven environments, By Alex Ruiz, JavaWorld.com, 07/17/07, http://www.javaworld.com/javaworld/jw-07-2007/jw-07-fest.html

[2] TestApi — a library of Test APIs, http://www.codeplex.com/TestApi

[3] Web UI Automation with Windows PowerShell, Dr. James McCaffrey, http://msdn.microsoft.com/en-us/magazine/cc337896.aspx

[4] Automated Software Testing with TestComplete 6, http://www.automatedqa.com/products/testcomplete/index.asp

[5] QA Automation Testing Tools Eggplant Functional Tester (FT), http://www.testplant.com/products/eggplant_functional_tester

[6] Cost Benefits Analysis of Test Automation, Douglas Hoffman, Software Quality Methods, LLC, 1999

[7] Автоматическая генерация тестов для графического пользовательского интерфейса по UML диаграммам действий, Калинов А.Я., Косачёв А.С., Посыпкин М.А., Соколов А.А., Труды Института Системного Программирования РАН, 2005