Автоматическая генерация тестов для графического пользовательского интерфейса по UML–диаграммам действий

А.Я. Калинов, А.С. Косачёв, М.А. Посыпкин, А.А. Соколов

Аннотация. В статье излагается метод автоматической генерации набора тестов для графического интерфейса пользователя, моделируемого детерминированным конечным автоматом с помощью UML–диаграмм действий. Метод заключается в построении обхода графа состояний системы с применением неизбыточного алгоритма обхода [1, 2] и компиляции построенного обхода в тестовый набор.

1. Введение

Практически все программные системы предусматривают интерфейс с оператором. Практически всегда этот интерфейс – графический (GUI – Graphical User's Interface). Соответственно, актуальна и задача тестирования создаваемого графического интерфейса.

Вообще говоря, задача тестирования создаваемых программ возникла практически одновременно с самими программами [1]. Известно, что эта задача очень трудоёмка как в смысле усилий по созданию достаточного количества тестов (отвечающих заданному критерию тестового покрытия), так и в смысле времени прогона всех этих тестов. Поэтому решение этой задачи стараются автоматизировать (в обоих смыслах).

Для решения задачи тестирования программ с программным интерфейсом (API – Application Program Interface: вызовы методов или процедур, пересылки сообщений) известны подходы – методы и инструменты, – хорошо зарекомендовавшие себя в индустрии создания программного обеспечения. Основа этих подходов следующая: создается формальная спецификация программы, и по этой спецификации генерируются как сами тесты, так и тестовые оракулы – программы, проверяющие правильность поведения тестируемой программы. Спецификации как набор требований к создаваемой программе существовали всегда, Ключевым словом здесь является формальная спецификация. Формальная спецификация – это спецификация в форме, допускающей её формальные же преобразования и обработку компьютером. Это позволяет анализировать набор требований с точки зрения их полноты,

непротиворечивости и т.п. Для задачи автоматизации тестирования эта формальная запись должна также обеспечивать возможность описания формальной связи между понятиями, используемыми в спецификации, и сущностями языка реализации программы.

В наиболее распространенных подходах используются специализированные языки формальных спецификаций, но существуют определенные трудности внедрения таких подходов в индустрию. Другой подход заключается в расширении обычных языков программирования (или других, привычных для пользователей языков) средствами спецификаций. Такой подход используется в методологии UniTesK [15]. В данной статье описана попытка применения такого подхода к автоматизации тестирования графических интерфейсов.

2. Обзор имеющихся средств автоматического тестирования графического интерфейса

Самым распространённым способом тестирования графического интерфейса является ручное тестирование. Естественно, у такого подхода имеются серьёзные недостатки, и самый главный из них – большая трудоёмкость, которая часто становится критичной при организации регрессионного тестирования. Другим недостатком такого подхода является отсутствие гарантий выполнения каждого доступного воздействия из каждого возможного состояния системы. Таким образом, приходится полагаться на однозначность реакции системы на одинаковые воздействия из разных состояний, то есть считать эти состояния эквивалентными для данного воздействия.

Другим способом организации тестирования является автоматическое тестирование [2]. Существуют два раскрытия этого термина:

- Ручное написание тестов и их автоматический прогон;
- Автоматическая генерация тестов.

Из наиболее доступных инструментов для автоматизированного тестирования можно выделить следующие пять:

- WinRunner;
- QA Run;
- Silk Test;
- Visual Test;
- Robot.

Перечисленные инструменты применяются для автоматизированного тестирования, в том числе при тестировании GUI. В своей статье [3] Рэй Робинсон приводит сравнительную таблицу (см. таб. 1) этих инструментов по способности удовлетворять заданные потребности пользователя. При этом рассматриваются различные свойства этих инструментов:

- Запись и проигрывание тестов;
- Возможности Web-тестирования;
- Возможности базы тестов и функции для работы с ней;

- Объектное представление;
- Сравнение рисунков;
- Обработка ошибочных ситуаций;
- Сравнение свойств объектов;
- Возможности расширения языка;
- И т.п.

(По 5-бальной шкале, наивысшая оценка – 1)

	Record & Playback	Web Testing	Database tests	Data functions	Object Mapping	Image testing	Test/Error recovery	Object Name Map	Object Identity Tool	Extensible Language	Environment support	Integration	Cost	Ease of use	Support	Object Tests
WinRunner	2	1	1	2	1	1	2	1	2	2	1	1	3	2	1	1
QA Run	1	2	1	2	1	1	2	2	1	2	2	1	2	2	2	1
Silk Test	1	2	1	2	1	1	1	1	2	1	2	3	3	3	2	1
Visual Test	3	3	4	3	2	2	2	4	1	2	3	2	1	3	2	2
Robot	1	2	1	1	1	1	2	4	1	1	2	1	2	1	2	1

Таблица 1.

Кроме самых распространённых, имеется много других инструментов для автоматического прогона тестов, созданных вручную. Но эти инструменты не предоставляют возможности автоматической генерации тестов (исключая процесс записи теста).

В основе всех этих инструментов лежат средства формального описания тестов (разного рода скриптовые языки). Тесты пишутся вручную. Данный подход имеет существенные недостатки. Один из них состоит, как и при полностью ручном подходе, в недостаточной систематичности. Другим серьёзным недостатком является трудность при организации регрессионного тестирования. При изменении тестируемой системы возникает необходимость переписывания (редактирования) большого количества тестов. То есть, например, система изменилась в одном месте, а тесты приходится модифицировать в несравненно большем количестве мест. Причём если некоторые такие проблемы можно обойти созданием набора вызываемых функций, которые также можно изменять в одном месте, то другие (например, изменение в логике программы) так просто не решаются. Кроме того, остаётся проблема дублирования одних и тех же действий (написание одного и того же тестового кода) при совершении одного и того же события из разных состояний. Такой метод сменил ручной прогон тестов на ручное написание самих тестов для дальнейшего автоматического прогона, но само их написание по-прежнему остаётся трудоёмким занятием.

Доступные инструменты для автоматической генерации тестов для GUI нам не известны. Поэтому мы в первую очередь сконцентрировали свои усилия именно на автоматизации генерации тестов для GUI по формальной спецификации. Причем при разработке средств формальной спецификации GUI мы старались максимально использовать существующие, привычные для пользователей методы и инструменты.

3. Описание подхода

Правильность функционирования системы определяется соответствием реального поведения системы эталонному поведению. Для того чтобы качественно определять это соответствие, нужно уметь формализовать эталонное поведение системы. Распространённым способом описания поведения системы является описание с помощью диаграмм UML (Unified Modeling Language) [4]. Стандарт UML предлагает использование трех видов диаграмм для описания графического интерфейса системы:

- Диаграммы сценариев использования (Use Case);
- Диаграммы конечных автоматов (State Chart);
- Диаграммы действий (Activity).

С помощью UML/Use Case diagram можно описывать на высоком уровне наборы сценариев использования, поддерживаемых системой [5]. Данный подход имеет ряд преимуществ и недостатков по отношению к другим подходам, но существенным с точки зрения автоматизации генерации тестов является недостаточная формальная строгость описания.

Широко распространено использование UML/State Chart diagram для спецификации поведения системы, и такой подход очень удобен с точки зрения генерации тестов. Но составление спецификации поведения современных систем с помощью конечных автоматов есть очень трудоёмкое занятие, так как число состояний системы очень велико. С ростом функциональности системы спецификация становится всё менее и менее наглядной.

Перечисленные недостатки этих подходов к описанию поведения системы преодолеваются с помощью диаграмм действий (Activity). С одной стороны, нотация UML/Activity diagram является более строгой, чем у сценариев использования, а с другой стороны, предоставляет более широкие возможности по сравнению с диаграммами конечных автоматов. Следующие причины определяют пригодность диаграмм действий для моделирования графического интерфейса:

1. Графический интерфейс пользователя (GUI) в большинстве случаев удобно представить в виде конечного автомата, так как поведение системы зависит от состояния, в котором она находится, и воздействия пользователя (и, может быть, других внешних воздействий).
2. GUI имеет иерархичную структуру. Действительно, модальные диалоги (те, которые перехватывают управление) удобно моделировать с помощью отдельной диаграммы состояний (для верхней диаграммы это поддиаграммы), инкапсулируя тем самым их функциональность.

3. Диаграмма действий позволяет специфицировать систему таким образом, при котором отсутствует дублирование одинаковых событий из разных состояний. Достигается это использованием на диаграмме управляющих элементов, из которых возможны тестовые воздействия, а не самих состояний.
4. Спецификация диаграммами действий удобна для восприятия человеком.

Другими словами, спецификация графического интерфейса с помощью диаграмм действий является достаточно естественным способом описания требований к графическому интерфейсу. Причём, при таком подходе сохраняется вся сила формальных спецификаций – как показывает опыт, большая часть ошибок выявляется именно на этапе составления спецификации системы.

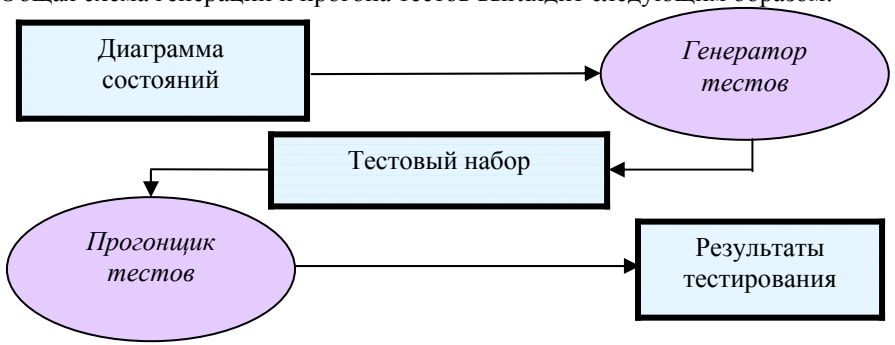
Итак, перейдём к детальному рассмотрению предлагаемого подхода.

Для создания прототипа работающей версии данного подхода использовался инструмент Rational Rose [6]. Он использовался в первую очередь для спецификации графического интерфейса пользователя при помощи диаграмм действий UML.

Для прогона сгенерированных по диаграмме состояний тестов использовался инструмент Rational Robot [6]. Из возможностей инструмента в работе мы использовали следующие:

1. Возможность выполнять тестовые воздействия, соответствующие переходам между состояниями в спецификации.
2. Возможность проверять соответствие свойств объектов реальной системы и эталонных свойств, содержащихся в спецификации. Из тех возможностей, которые доступны с помощью этого инструмента, используется проверка следующих свойств объектов:
   - Наличие и состояние окон (заголовок, активность, доступность, статус);
   - Наличие и состояние таких объектов, как PushButton, CheckBox, RadioButton, List, Tree и др. (текст, доступность, размер);
   - Значение буфера обмена;
   - Наличие в оперативной памяти запущенных процессов;
   - Существование файлов.

Общая схема генерации и прогона тестов выглядит следующим образом:



Рассмотрим ее компоненты более подробно.

3.1. Диаграмма состояний

Диаграмма состояний – это формальная спецификация всех состояний системы и возможных переходов между ними с помощью диаграмм действий. Но во избежание так называемого «взрыва состояний» был использован метод расширенных конечных автоматов (EFSM – Extended Finite State Machine). При этом выделяется множество так называемых управляющих состояний и множество переменных. Реальные же состояния системы описываются управляющими состояниями и наборами значений переменных. Каждому переходу приписывается (кроме обычных стимулов и реакций) предикат от значений переменных и действия по изменению значения переменных. EFSM позволяют существенно сократить размер описания, сохраняя практически всю информацию о системе.

На диаграммах имеются объекты типа State(Normal). Это управляющие элементы, или управляющие состояния. Переменные были введены как раз затем, чтобы не дублировать одно и тоже действие из разных состояний, а изобразить его один раз из одного управляющего состояния. Таким образом, каждому управляющему состоянию на диаграмме соответствует множество реальных состояний системы. То есть мы можем написать:

```
Состояние = <UML-State, набор значений переменных>.
```

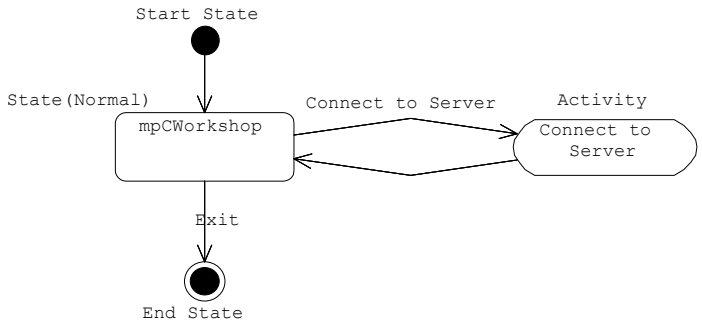
(где UML-State есть управляющее состояние).

Соответственно, переходы между управляющими состояниями обретают предусловия (guard condition), определяющие, из каких конкретно состояний, соответствующих данному управляющему состоянию, возможен переход, и действия (action), определяющие, в какое конкретно состояние ведёт этот переход. Предусловия и действия записываются на простом языке, в котором есть следующие конструкции:

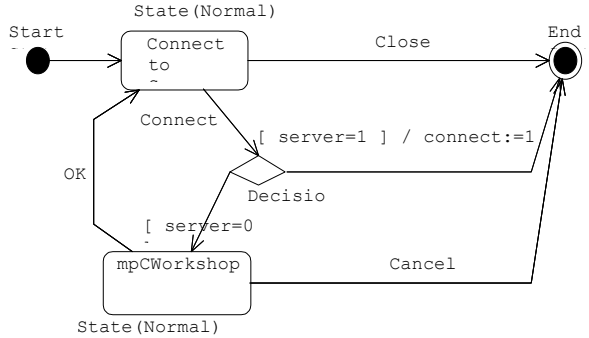
1. конструкция ::= список предусловий | список действий
2. список предусловий ::= предусловие | список предусловий, предусловие
3. список действий ::= действие | список действий, действие
4. предусловие ::= переменная = выражение
5. действие ::= переменная := выражение
6. выражение ::= переменная | константа
7. переменная ::= символ{символ}
8. константа ::= цифра{цифра}
9. символ ::= буква | цифра

Такой язык был выбран из-за простоты интерпретации, что упростило написание интерпретатора. Но, как оказалось, перечисленных возможностей вполне достаточно для описания большинства возможностей графического интерфейса.

Ниже приведён пример диаграммы состояний с двумя переменными и одной поддиаграммой.



Поддиаграмма главной диаграммы, связанная с объектом типа Activity "Connect to Server":



В этом примере используются две переменных – ‘server’ и ‘connect’. Значение переменной ‘server’ определяет состояние сервера (значение «1» означает, что сервер запущен, значение «0» означает, что сервер не запущен), а переменная 'connect' определяет наличие соединения с сервером (значение «1» означает, что есть соединение с сервером, а значение «0» означает, что соединения с сервером нет). Соответственно, интерпретировать эту диаграмму следует так:

1. Переход по событию ‘Connect to Server’ из управляющего состояния ‘mpCWorkshop’ означает переход в поддиаграмму, соответствующую объекту типа Activity ‘Connect to Server’, и дальнейший переход к состоянию ‘Connect to Server’.
2. Переход по событию ‘Close’ из управляющего состояния ‘Connect to Server’ ведёт в конечное управляющее состояние на этой поддиаграмме, что означает переход из Activity ‘Connect to Server’ на главной диаграмме в управляющее состояние mpCWorkshop.
3. Переход по событию ‘Connect’ из управляющего состояния ‘Connect to Server’ ведёт в объект типа Decision, в котором происходит разветвление в зависимости от значения переменной ‘server’.

   a) Если сервер запущен (‘server=1’), то устанавливается соединение (‘connect:=1’), и, как и в случае 2, переход в конечное управляющее состояние этой поддиаграммы ведёт в управляющее состояние mpCWorkshop главной диаграммы.
   b) Если сервер не запущен (‘server=1’), то переход ведёт в управляющее состояние mpCWorkshop этой поддиаграммы (это модальный диалог-предупреждение о том, что соединение не может быть установлено). Из этого управляющего состояния возможны два перехода – по событию ‘OK’ переход ведёт в управляющее состояние ‘Connect to Server’, а по событию ‘Cancel’, как и в случае 2, переход ведёт в управляющее состояние mpCWorkshop главной диаграммы.
4. Переход по событию ‘Exit’ из управляющего состояния ‘mpCWorkshop’ ведёт в конечное управляющее состояние главной диаграммы, что означает переход к начальному управляющему состоянию этой диаграммы и дальнейший переход в состояние mpCWorkshop.

Итак, можно составить набор правил, определяющих переходы между состояниями:

- Переход между состояниями осуществляется по событию и может состоять из нескольких переходов между управляющими элементами на диаграмме, к каковым относятся объекты типа State(Normal), Start State, End State, Activity, Decision, а также переход в поддиаграмму или возвращение из неё.
- Событие есть непустое имя перехода.
- Событие есть у всех переходов, начинающихся в управляющих элементах типа State(Normal), и только у них.
- Переход из управляющего состояния типа State(Normal) происходит по событию.
- Переход из управляющего состояния типа Start State, End State, Activity, Decision происходит без события.
- Переход между состояниями начинается и заканчивается только в управляющих элементах типа State(Normal).
- Переход между состояниями атомарный.
- Каждый переход между управляющими элементами может иметь предусловия и действия. Наличие пустого предусловия по умолчанию означает его истинность. Пустое действие означает отсутствие изменения значений переменных.
- Каждый переход может иметь несколько предусловий, разделённых оператором «,».
- Каждый переход может иметь несколько действий, разделённых оператором «,».

- Переходы детерминированы, то есть не может существовать двух переходов с одинаковым событием (или оба без события), начинающихся в одном управляющем состоянии и обладающих предусловиями, которые допускают одновременное выполнение.
- На каждой диаграмме должны существовать начальное и конечное (не обязательно одно) управляющие состояния (на главной диаграмме конечное состояние может не существовать).
- Поддиаграммы есть у объектов типа Activity, и только у них.
- Если переход из какого-либо управляющего элемента ведёт в управляющий элемент типа Activity, это означает переход к начальному управляющему элементу поддиаграммы, соответствующей элементу типа Activity, и дальнейший переход без события.
- Если переход из какого-либо управляющего элемента ведёт в конечный управляющий элемент какой-либо поддиаграммы, то это означает переход к управляющему элементу типа Activity, которому соответствует эта поддиаграмма, и дальнейший переход из этого элемента без события.
- Если переход из какого-либо управляющего элемента ведёт в управляющий элемент типа Decision, это означает продолжение перехода от этого элемента без события.

Дальше диаграмма, составленная по таким правилам, подаётся на вход генератору тестов.

Кроме этого, с каждым объектом на диаграмме (модель, диаграммы, управляющие элементы, переходы) связаны требования (спецификации) к данному объекту. Эти требования размещаются в поле Documentation, предусмотренное Rational Rose для каждого объекта на диаграмме. Эти требования записываются в виде инструкций на языке SQABasic, которые выполняет Rational Robot. В большинстве случаев с переходами связаны инструкции, означающие какие-то действия пользователя, а с управляющими элементами связаны инструкции, означающие требования к данному состоянию, которые может проверить Rational Robot. По сути, данный набор инструкций реализует оракул этого состояния.

3.2. Генератор тестов

Генератор строит по диаграмме состояний набор тестов. Условием окончания работы генератора является выполнение тестового покрытия. В данном случае в качестве покрытия было выбрано условие прохода по всем рёбрам графа состояний, то есть выполнения каждого доступного тестового воздействия из каждого достижимого состояния.

Структурно генератор тестов состоит из следующих компонентов:

- ✓ Интерпретатор;
- ✓ Итератор;
- ✓ Обходчик;

- ✓ Компилятор.

Интерпретатор осуществляет переход между состояниями в соответствии с введёнными ранее правилами. В случае успешного перехода он останавливается и возвращает указатель на тот управляющий элемент, в котором он остановился, в случае неудачи он останавливается и выводит сообщение о возникшей ошибке (обычно это несоответствие введённым правилам). Для замыкания диаграммы вводится дополнительное условие: если переход из какого-либо управляющего элемента ведёт в конечный управляющий элемент главной диаграммы, то это означает переход к её начальному управляющему элементу и дальнейший переход без события.

Связь интерпретатора с обходчиком выполняет **итератор** переходов. В соответствии с выбранным алгоритмом обхода итератор возвращает событие, переход по которому ещё не был совершён из данного состояния. Если все переходы были совершены, итератор возвращает пустое событие.

Обходчик есть ключевой компонент генератора тестов. Чтобы построить тестовый набор, он строит обход ориентированного графа состояний, проходя по всем дугам графа. Алгоритм его работы взят из [7, 8, 9]. Прибегнуть к специальному алгоритму обхода ориентированного графа заставило то обстоятельство, что граф состояний скрыт – на диаграмме действий изображён граф, состоящий из управляющих элементов. Это значит, что заранее неизвестны все возможные состояния графа, поэтому мы и воспользовались неизбыточным алгоритмом обхода ориентированного графа. В процессе работы обходчик вызывает итератор переходов для получения событий, соответствующих не пройденным переходам, и интерпретатор событий, последовательность которых формируется в соответствии с алгоритмом. Обходчик останавливается на любом графе, вынося вердикт, совершен обход или нет. В том случае, если нет, то выводится отладочная информация о причинах, не позволивших его совершить, чтобы можно было быстро их устранить (обычно это либо несоответствие правилам, либо тупиковое состояние).

Компилятор (здесь термин “компиляция” подразумевается в его “не программистском” смысле – сборка). Как уже говорилось, с каждым объектом на диаграмме (модель, диаграммы, управляющие элементы, переходы) связан некоторый набор инструкций на языке SQABasic, которые выполняет Rational Robot. В большинстве случаев с переходами связаны инструкции, означающие какие-то действия пользователя, а с управляющими элементами связаны инструкции, означающие оракул этого состояния. В процессе обхода графа состояний пройденный путь компилируется в тестовые файлы, содержащие инструкции для Rational Robot. Компиляция происходит путём записи информации, связанной с каждым пройденным объектом. Далее приведен фрагмент вызова функции записи документации.

Автоматическая генерация тестов по диаграммам действий имеет следующие преимущества перед остальными подходами к тестированию графического интерфейса:

- Спецификация автоматически интерпретируется (тем самым она проверяется и компилируется в набор тестов).
- Если какая-то функциональность системы изменилась, то диаграмму состояний достаточно изменить в соответствующем месте, и затем сгенерировать новый тестовый набор. Фактически, это снимает большую часть проблем, возникающих при организации регрессионного тестирования.
- Гарантия тестового покрытия. Эта гарантия даётся соответствующим алгоритмом обхода графа состояний.

Генератор тестов есть программа (script), написанная на языке ‘Rational Rose Scripting language’(расширение к языку Summit BasicScriptLanguage). В Rational Rose есть встроенный интерпретатор инструкций, написанных на этом языке, посредством которого можно обращаться ко всем объектам модели (диаграммы состояний).

3.3. Тестовый набор

В процессе построения обхода генератор тестов компилирует набор тестов – инструкции на языке SQABasic. Эти инструкции есть чередование тестовых воздействий и оракула свойств объектов, соответствующих данному состоянию.

Для того, чтобы продолжать тестирование, когда один тест не прошёл, в генератор тестов встроена возможность выбора – генерировать один большой тест или набор атомарных тестов. Атомарный тест – тот, который не требует приведения системы в состояние, отличное от начального состояния.

В связи с наличием ограничения инструмента прогона тестов на тестовую длину, в тесты после каждой законченной инструкции вставляется строка разреза. Во время прогона по этим строкам осуществляется разрез теста в случае, если его длина превышает допустимое ограничение. После прохождения части теста до строки разреза продолжается выполнение теста с первой инструкции, следующей за строкой разреза. Нарезку и сам прогон тестов осуществляет прогонщик тестов.

3.4. Прогонщик тестов

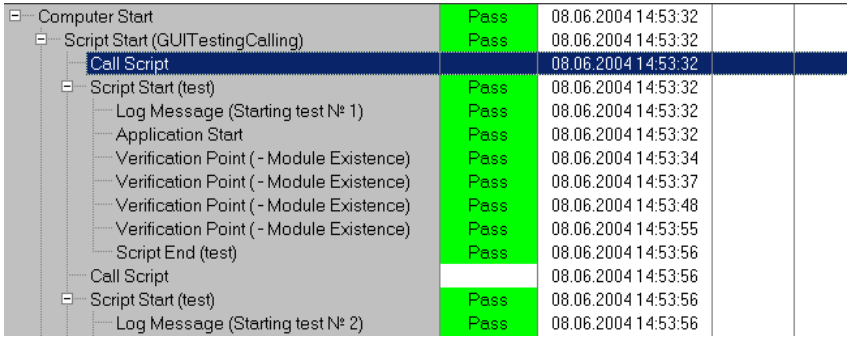
В качестве прогонщика тестов мы используем Rational Robot, который выполняет сгенерированные наборы инструкций. В случае удачного выполнения всех инструкций выносится вердикт – тест прошёл. В противном случае, если на каком-то этапе выполнения теста поведение системы не соответствует требованиям, Robot прекращает его выполнение, вынося соответствующий вердикт – тест не прошёл.

3.5. Анализ и отображение результатов

Когда Rational Robot закончит выполнение тестов, результаты тестирования будут отображены в наглядном виде в Rational TestManager[6]. Это программа

для управления отчётами о тестировании, и в случае ошибочной ситуации выдаётся достаточная информация об ошибке для её нахождения и исправления.

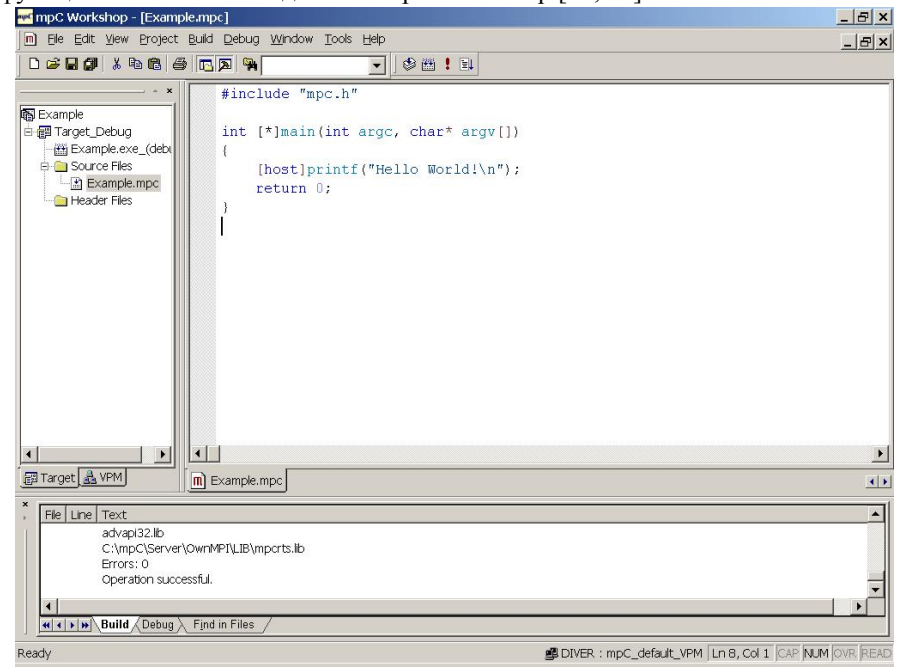
Типичный screenshot программы Rational TestManager представлен ниже на рисунке.



4. Апробация

4.1. Поставленная задача

Разработка и реализация методов автоматизированной генерации набора функциональных тестов для GUI mpC Workshop [10, 11].

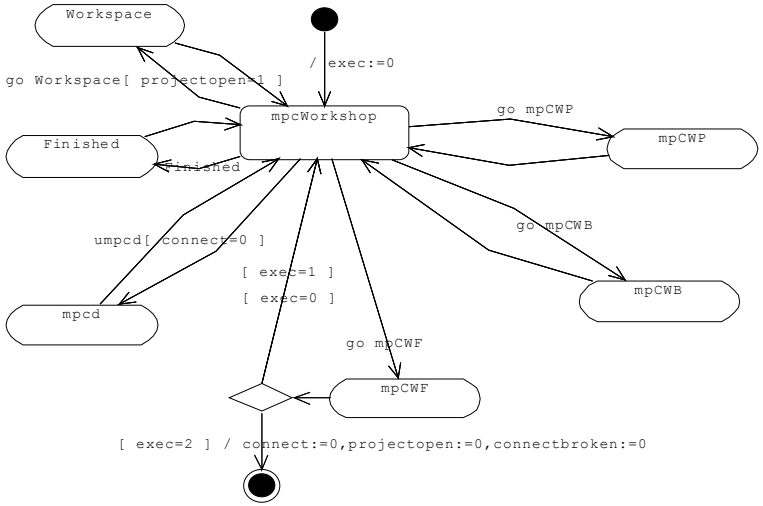


mpC Workshop – это среда для разработки приложений для параллельных вычислений на языке mpC [12, 13, 14] с функциональностью графического

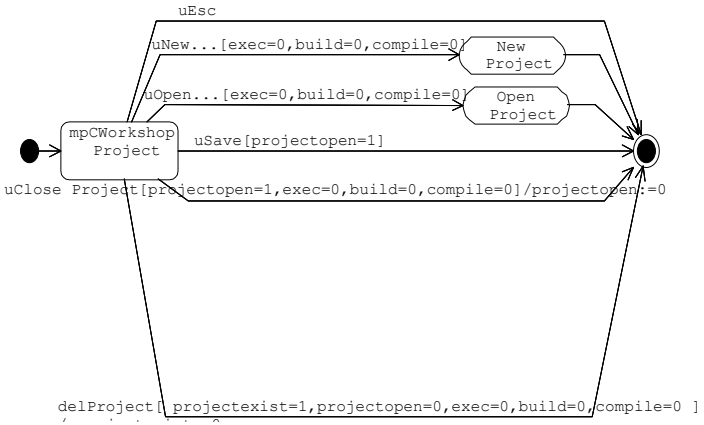
интерфейса, типичной для такого типа систем. Выше приведён типичный screenshot mpCWorkshop.

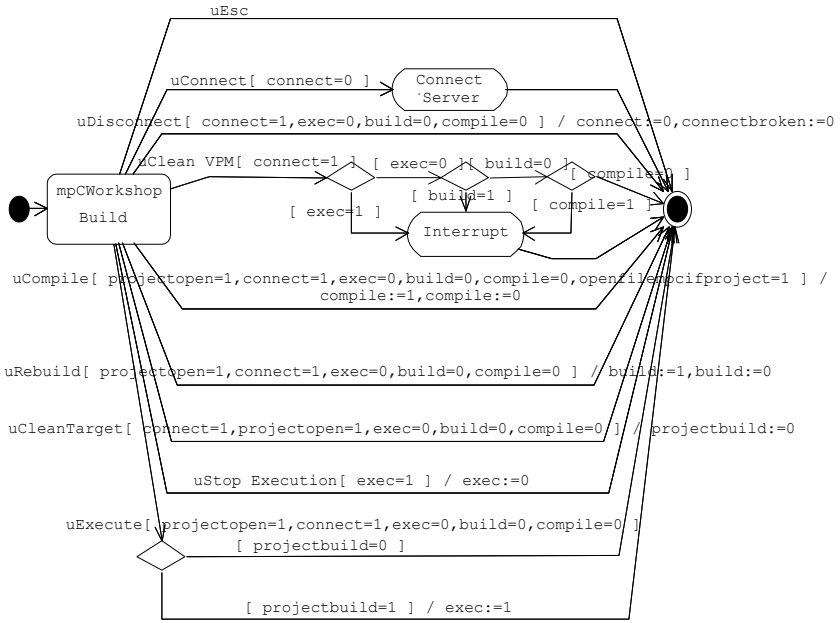
Для тестирования базовой функциональности была составлена базовая диаграмма состояний и переходов mpC Workshop. Ниже приведены фрагменты этой диаграммы (включая поддиаграммы).

Главная диаграмма выглядит следующим образом:

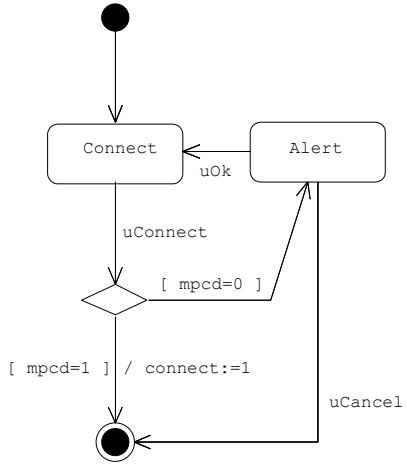


Такие объекты типа Activity, как “mpCWP”, “mpCWB”, “mpCWF”, “mpcd”, “Finished”, “Workspace” означают наличие под собой поддиаграмм. Ниже приведены поддиаграммы, соответствующие объектам “mpCWP” и “mpCWB”.





С Activity “Connect to Server” связана следующая поддиаграмма:



Как видно из рисунков, на этих поддиаграммах тоже есть поддиаграммы. Фактически, каждый объект типа Activity скрывает под собой часть функциональности системы. Под “mpCWB” содержится описание функциональности, связанной с работой с сервером, под “mpCWP” содержится

описание функциональности, связанной с работой с проектом, под “mpCWF” содержится описание функциональности, связанной с работой с файлами, и т.д.

В этой базовой диаграмме используется 12 переменных с начальными значениями, приведёнными ниже:

- connect, 0
- connectbroken, 0
- mpcd, 0
- exec, 2
- build, 0
- compile, 0
- projectopen, 0
- projectbuild, 0
- projectexist, 0
- projectpathenter, 0
- projectpathcorrect, 0
- openfilempcifproject, 0

Практически с каждым объектом на диаграмме связаны тестовые инструкции на языке SQABasicScript. Ниже в таблице приведены примеры этих инструкций, связанных с управляющими состояниями и переходами:

MpCWorsk-hop	`Window SetContext, "Caption={mpC Worksho*}; State=Enabled", "Activate=1; Status=MAXIMIZED" Result=ModuleVP (Exists, "Name=mpcworkshop.exe", "ExpectedResult = PASS")`
go mpCWP	`InputKeys "%p"`
Connect	`Window SetContext, "Caption={mpC Worksho*}; State=Disabled", "Activate=0; Status=MAXIMIZED" Window SetContext, "Caption=Connect to server; State=Enabled", "Activate=1" PushButton Middle_Click, "Text=Start server;State=Enabled" PushButton Middle_Click,"Text=Close;State=Enabled" PushButton Middle_Click, "Text=Advanced >>;State=Enabled" PushButton Middle_Click, "Text=Connect;State=Enabled" CheckBox Middle_Click, "Text=Once" CheckBox Middle_Click, "Text=Hidden" ListView Click, "Text=List1;\;ItemText=DIVER", ""`
uConnect	`PushButton Click, "Text=Connect;State=Enabled"`

И так далее...

Генератор тестов для данной диаграммы генерирует тесты за время порядка 1-й минуты. Суммарный размер выходных тестов 1541024 байта (1,5 Мб).

На данной диаграмме изображено 14 управляющих состояний, тогда как в процессе генерации тестов воспроизводится 258 реальных состояний системы (то есть управляющих состояний и значений переменных).

Ниже приведен начальный кусок сгенерированного тестового кода. Видно, что тестовые воздействия перемежаются с оракулом заданных свойств состояний. Также вставляется строка: `'------cutting line------`.

Она вставляется после каждого законченного набора тестовых инструкций, связанных с одним объектом диаграммы. Эта строка требуется для осмысленного разрезания сгенерированного скрипта, так как его размер примерно равен 1,5 Мб, а у прогонщика тестов Rational Robot есть ограничение на тест в размере 20 Кб. Таким образом, большой тест разрезается на маленькие кусочки по линиям разреза для дальнейшего последовательного прогона.

```
Sub Main
' copyright by ISP RAS 2004
' implemented by A.Sokolov
SQALogMessage sqaPass, "Starting test № 0", ""
' Initially Recorded: 07.06.2004 20:20:53
' Class № 0 with name .
' Attributes: connect = 0, connectbroken = 0, mpcd = 0, exec =
2, build = 0, compile = 0, projectopen = 0, projectbuild = 0,
projectexist = 0, projectpathenter = 0, projectpathcorrect = 0,
openfilempcifproject = 0, IDState = 0,
workDirectory$ =
"E:\Alexey\Work\mpCGUITesting\mpCGUITestingProjects"
'------cutting line------
StartApplication "mpcWorkshop"
'------cutting line------
 ...
```

Прогон сгенерированного теста длится около 50-ти минут. Скорость ввода символов и нажатия на объекты системы ограничена возможностями инструмента, но, тем не менее, она велика, и человеку практически невозможно совершать действия с такой скоростью.

4.2. Практические результаты

В ходе выполнения работы был обнаружен ряд ошибок в графическом интерфейсе mpC Workshop. Наибольшее количество ошибок было найдено на этапе составления спецификации системы. Это характерная черта использованного метода составления спецификации – по реальной работе системы. То есть на момент формализации графического интерфейса системы продукт mpC Workshop был уже разработан и реализован, таким образом сам процесс составления спецификации был уже своеобразным тестированием системы. На этом этапе было найдено порядка 10-15 ошибок в системе.

После того, как была составлена спецификация основной функциональности тестируемой системы, был сгенерирован тестовый набор, ей соответствующий. После его прогона была выявлена одна трудноуловимая ошибка.

4.3. Обработка ошибочной ситуации

В результате применения предложенного подхода была поднята некоторая проблема. А именно, как продолжать прогон тестов после его останова во время ошибочной ситуации. На данный момент существует несколько вариантов её разрешения:

- Ошибка исправлена разработчиками в разумные сроки.
- Сознательно специфицируется именно ошибочное поведение системы.
- Данная функциональность вообще не тестируется (например, у соответствующего перехода ставится невыполнимое предусловие).
- Тестирование продолжается со следующего теста. При этом нужно привести систему в состояние, соответствующее начальному состоянию следующего теста.

5. Развитие подхода

Предложенный подход генерации набора тестов, на наш взгляд, дальше можно развивать в следующих направлениях:

- Внедрение асинхронности. Диаграммы действия UML позволяют моделировать асинхронное поведение системы.
- Использование других инструментов для специфицирования системы и прогона тестов.
- Внедрение зависимости инструкций, соответствующих объекту на диаграмме, от набора значений переменных текущего состояния (либо печатать в трассу сами действия, либо подставлять значения переменных в соответствующие поля инструкций).
- Использование более полной проверки состояний объектов системы с помощью Verification Point – данная возможность используется сейчас слабо.
- Использование инструкций, связанных с диаграммой, во всех управляющих элементах этой диаграммы для сокращения их дублирования.
- Использование функций от набора значений переменных в управляющих элементах диаграммы для проверки состояния объектов системы.
- Использование не самих наиболее часто встречающихся функций, а только их вызова – для сокращения дублирования тестового кода.
- Предоставление тестеру, составляющему спецификацию системы, некоторого набора шаблонов, позволяющих быстро сгенерировать соответствующую диаграмму вместе с соответствующими тестовыми инструкциями.

6. Заключение

Предложенная в настоящей работе методика автоматизации генерации тестов для GUI была разработана и апробирована с августа 2003 г. по май 2004 г. в ходе выполнения проекта по тестированию mpC Workshop. Исследование мето-дов автоматизированной генерации тестов для GUI сейчас активно развивается.

Литература

1. С.Канер, Д.Фолк, Е.К.Нгуен, Тестирование программного обеспечения.
2. Э.Дастин, Д.Рэшка, Д.Пол, Автоматизированное тестирование программного обеспечения.
3. Ray Robinson, AUTOMATION TEST TOOLS, Date Created: 1st March 2001, Last Updated: 11th Sept 2001. http://tester.com.ua/test_types/docs/automation_tools.doc
4. UML. http://www.omg.org/technology/documents/formal/uml.htm
5. С.Г.Грошев, Тестирование на основе сценариев, дипломная работа. МГУ, ВМиК.
6. P.Kruchten. The Rational Unified Process. An introduction. //Rational Suite documentation. http://www.interface.ru/rational/rup01_t.htm
7. И.Б.Бурдонов, А.С.Косачёв, В.В.Кулямин, «Неизбыточные алгоритмы обхода ориентированных графов. Детерминированный случай.» "Программирование", -2003, №6–стр 59-69
8. И.Б.Бурдонов, А.С.Косачёв, В.В.Кулямин, А.Хорошилов, Тестирование конечных автоматов. Отчет ИСП РАН.
9. http://www.ispras.ru/~RedVerst/
10. A.Kalinov, K.Karganov, V.Khatzkevich, K.Khorenko, I.Ledovskih, D.Morozov, and S.Savchenko, The Presentation of Information in mpC Workshop Parallel Debugger, Proceedings of PaCT-2003, 2003.
11. http://mpcw.ispras.ru/
12. A.Lastovetsky, D.Arapov, A.Kalinov, and I.Ledovskih, "A Parallel Language and Its Programming System for Heterogeneous Networks", Concurrency: Practice and Experience, 12(13), 2000, pp.1317-1343.
13. A.Lastovetsky, Parallel Computing on Heterogeneous Networks. John Wiley & Sons, 423 pages, 2003, ISBN: 0-471-22982-2.
14. http://www.ispras.ru/~mpc/
15. И.Б.Бурдонов, А.С.Косачёв, В.В.Кулямин, А.К.Петренко «Подход UniTesK к разработке тестов», "Программирование", -2003, №6–стр 25-43