Тестирование компонентов, взаимодействующих посредством удаленного вызова методов[1]

В.С. Мутилин
mutilin@ispras.ru

Аннотация. В статье описывается разработанный автором метод тестирования компонентов, взаимодействующих посредством удаленного вызова методов. Метод позволяет гарантировать, что будут проверены все различные чередования вызовов методов в системе, приводящие к различным результатам. В работе выделены ограничения, при которых такой перебор различных порядков вызовов методов гарантирует корректность системы. Показано, что этим ограничениям удовлетворяют системы, разработанные по технологии Enterprise JavaBeans. В отличие от методов проверки моделей (model checking), в предложенном методе перебор осуществляется не для всей системы целиком, а для отдельных тестовых воздействий, что позволяет существенно сократить область перебора.

1. Введение

Для разработки сложных современных программных систем широко применяется компонентный подход, в котором система состоит из набора *компонентов*. Согласно [1], такой компонент представляет собой структурную единицу системы, обладающую четко определенным *интерфейсом*, который описывает ее зависимости от окружения. Интерфейс состоит из *интерфейсных методов*, посредством вызова которых с компонентом взаимодействуют другие компоненты и окружение.

В данной работе мы будем рассматривать системы, в которых компоненты взаимодействуют посредством *удаленного вызова методов*, например Java Remote Method Invocation (Java RMI) [2]. Удаленный вызов отличается от обычного тем, что данные вызова передаются по сети, позволяя, тем самым, взаимодействовать между собой компонентам, находящимся на разных компьютерах. Система, построенная на основе такого взаимодействия, является распределенной системой [3] с присущими таким системам сложностями при разработке и тестировании.

[1] Работа частично поддержана грантом РФФИ 05-01-00999.

Традиционно целью тестирования программной системы является проверка того, что ее реальное поведение соответствует ожидаемому поведению. Для осуществления такой проверки требуется некоторым образом воздействовать на систему, подавая входные данные в различных состояниях системы. Соответственно, первой задачей тестирования является генерация таких воздействий. После осуществления воздействия система выдает результаты и переходит в некоторое состояние. Вторая задача – оценить корректность выдаваемых результатов, убедиться, что они соответствуют ожидаемым результатам. В данной работе корректность результатов формулируется в виде формальной спецификации системы на основе подхода Design-By-Contract [4], при котором спецификация системы описывается в виде пред- и постусловий методов, а также инвариантов. Третья задача – оценка полноты тестирования. Решение этой задачи позволяет ответить на вопрос: достаточно ли сгенерированных тестовых воздействий для завершения тестирования?

Для решения этих трех задач для распределенных систем требуется учет ряда особенностей. Воздействия на систему требуется производить не только последовательно, но и *асинхронно*, т.е. подавать их из различных потоков, процессов или с разных машин. Результаты, выдаваемые системой после подачи воздействия, зависят от чередований событий внутри нее. В системах, взаимодействующих посредством удаленного вызова методов, такими событиями являются удаленные вызовы методов. Результат может зависеть от порядка вызова этих методов. Соответственно, метод оценки корректности должен обеспечивать проверку результатов для любых возможных чередований событий. Кроме того, для исчерпывающего тестирования необходимо проверить результаты для различных чередований, т.е. уже недостаточно проверить одно из возможных чередований. Эта особенность должна быть учтена при оценке полноты тестирования.

Особо важную роль в распределенных системах играет задача перебора различных чередований событий. В силу того, что появление того или иного порядка событий зависит от многих факторов, таких как планирование процессов в операционной системе, управлять которыми разработчик системы не имеет возможности, нет гарантии того, что данный порядок появится в процессе воздействия на систему. Перебор только части возможных чередований оставляет возможность проявления ошибки на непроверенных чередованиях. Ошибки такого рода крайне тяжело выявлять и исправлять; даже если такая ошибка обнаружена, ее трудно повторить, так как вероятность появления порядка, на котором проявляется ошибка, может быть крайне мала.

Целью данной работы является построение метода тестирования, позволяющего гарантировать, что будут проверены все различные чередования событий, приводящие к разным результатам.

Задача перебора различных чередований событий традиционно решается в *методах проверки моделей* (*model checking*), которые позволяют найти

последовательность событий, нарушающих данную темпоральную формулу или доказать, что такой последовательности не существует.

Методы проверки моделей работают с *замкнутой* системой, т.е. системой, которая не принимает входных воздействий и не выдает реакций. Для верификации других систем, например, предоставляющих программный интерфейс, требуется предоставить окружение, взаимодействующее с системой. От размеров окружения напрямую зависит количество состояний, получающихся в процессе поиска. С другой стороны, от окружения зависит качество верификации, и далеко не всегда можно ограничиться простым окружением.

Методы проверки моделей обладают рядом ограничений:

1. Взрыв количества состояний;
2. Требование функции сравнения состояний;
3. Требование отката.

Взрыв состояний возникает из-за большого количества состояний в программных системах. Кроме того, если система не является замкнутой, то к взрыву часто приводит попытка проанализировать систему с *наиболее общим окружением*, которое воздействует на систему произвольным образом. Известны методы целенаправленной генерации окружений, например, [5], однако данные методы не применимы для генерации окружений для распределенных систем.

Для сокращения пространства перебора широко используются методы *редукции частичных порядков* [6]. Эти методы основаны на понятии *зависимости* событий. Зависимости можно выявить до выполнения системы или при выполнении одного из зависимых событий. Эксперименты, проведенные в [6], показывают, что такие методы позволяют значительно сократить пространство перебора.

Классические методы проверки моделей для осуществления поиска требуют наличия функции сравнения состояния. Для того чтобы сравнивать состояния, требуется хранить информацию об этих состояниях. *Поиск без сохранения состояний* (*stateless search*) [7] позволяет осуществлять поиск для систем со сложными состояниями, которые сложно сохранять, а также для систем с большим количеством состояний. Однако поиск накладывает ограничения на систему – требуется, чтобы в ее пространстве состояний не было циклов.

Широко используемой возможностью в методах поиска является возможность отката (возврата) в предшествующее состояние. Такая возможность может обеспечиваться за счет механизмов отката в реализации (например, с использованием специальной виртуальной машины [8]) или с помощью перевыполнения, т.е. сброса системы в начальное состояние и повторного выполнения. Мы сформулируем задачу для случая, когда в реализации нет механизма отката и нет возможности сброса в начальное состояние. Мы будем

предполагать, что вместо отката алгоритму поиска предоставляется возможность возврата в одно из состояний, которое принадлежит некоторому множеству, называемому *обобщенным состоянием*.

Мы предполагаем, что последовательность воздействий на систему строится на основе *обхода графа состояний* конечного автомата [9,10]. Состояния автомата являются множествами исходных состояний системы и называются обобщенными состояниями. Множества выбираются, как разбиения состояний системы на классы эквивалентности. Возможность обобщенного отката реализуется за счет требования *сильной связности* графа состояний, необходимого для построения обхода графа. Таким образом, обобщенный откат – это путь в графе состояний, ведущий в требуемое обобщенное состояние.

В следующем разделе описан пример, на котором будет продемонстрирован метод тестирования. В разделе «Алгоритм поиска» описан алгоритм, позволяющий перебирать различные порядки вызовов методов. Далее следует описание архитектуры, необходимой для его работы.

2. Пример

2.1. Описание системы

Рассмотрим пример системы, построенной из компонентов, которые взаимодействуют посредством удаленного вызова методов. Система написана на основе Enterprise Java Beans 3.0 [11]. Система содержит один *сеансовый компонент с состоянием* (*stateful session bean*) *CompanyManagerBean*, реализующий удаленный интерфейс *CompanyManager*, и один класс *сущность* (*entity*) *Company* (Рис. 1).

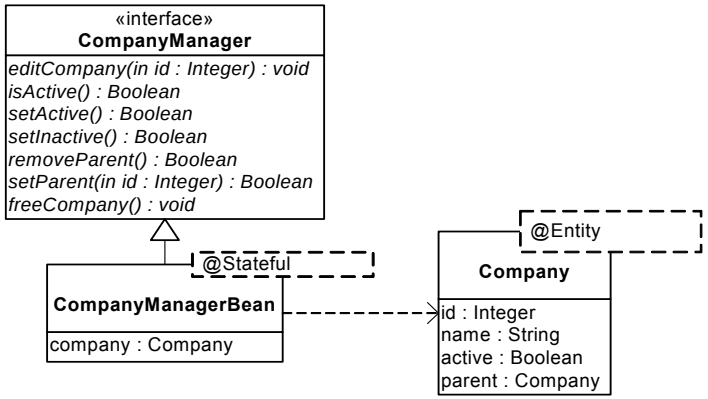

Рис. 1. Система управления компаниями

Компонент *CompanyManagerBean* предоставляет интерфейс работы с иерархиями компаний. У каждой компании имеются уникальный идентификатор, имя, могут иметься активный/неактивный статус и родительскую компанию. Через компонент *CompanyManagerBean* можно редактировать компании. Клиент запрашивает экземпляр компонента, затем устанавливает компанию, которую собирается редактировать (*editCompany*), блокируя доступ к данной компании другим экземплярам, редактирует компанию, изменяя статус компании (*setActive, setInactive*) и родительскую компанию (*setParent, removeParent*), затем освобождает компанию (*freeCompany*), делая ее доступной для редактирования другим экземплярам.

На иерархию компаний накладываются следующие ограничения:

1. Компания с активным статусом не может иметь родительскую компанию с неактивным статусом;
2. Иерархия компаний более четырех уровней (глубины) недопустима.

Таким образом, данные ограничения запрещают иерархии, показанные на Рис. 2: (а) – активная компания C_2 имеет неактивного родителя C_1, (б) – иерархия компаний имеет более четырех уровней.

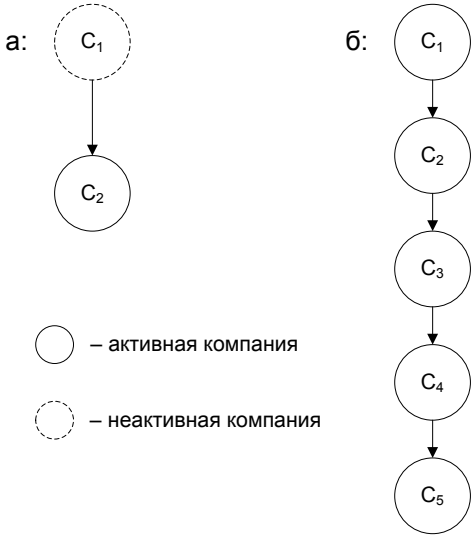

Рис. 2. Примеры запрещенных иерархий

Реализация методов компонента *CompanyManagerBean* состоит в предварительной проверке ограничений и, в случае успеха, изменении состояния компании и сохранении результатов изменения в базе данных. Для примера рассмотрим реализации методов *setInactive* и *setParent*. Метод *setInactive* устанавливает статус компании неактивным (Рис. 3).

Предварительно метод проверяет, что все дочерние компании имеют неактивный статус. Проверка производится во вспомогательном методе *isChildrenInactive*, в котором происходит поисковый запрос к базе данных с выбором всех дочерних компаний, т.е. компаний, у которых родительская компания совпадает с данной. Далее, если все дочерние компании неактивны, то возвращается *true*, статус компании меняется на неактивный, и изменения сохраняются в базу данных (*enityManager.merge*), иначе возвращается *false*, и статус компании не изменяется.

Обращения к базе данных производятся через специальный компонент *EntityManager*, который предоставляет методы добавления (*persist*), изменения (*merge*), поиска (*find*) записей в базе данных. Отметим, что для корректности результатов данной работы требуется, чтобы обращения компонентов к общим данным, хранящимся в базе данных, являлись атомарными, т.е. одновременно может выполняться запрос не более чем от одного компонента. В данном примере это требование обеспечивается механизмом транзакций EJB таким образом, что обращения к *EntityManager* включаются в *контекст транзакций*.

```
public boolean setInactive() {
        if (isChildrenInactive()) {
                company.setActive(false);
                entityManager.merge(company);
                return true;
        } else {
                return false;
```

Рис. 3. Реализация метода setInactive

```
public boolean setParent(int id) {
        Company newParent = entityManager.find(Company.class, id);
        if (newParent == null) throw new NoResultException();

        if (newParent.getActive()
                        || !company.getActive()) {
                company.setParent(newParent);
                entityManager.merge(company);
                return true;
        } else {
                return false;
        }
}
```

Рис. 4. Реализация метода setParent

Метод *setParent* устанавливает родительскую компанию (Рис. 4). Требуется, чтобы существовала компания с идентификатором указанным в качестве аргумента вызова этого метода. Метод проверяет, что для активной компании нельзя установить неактивного родителя. Т.е. если родитель активный или компания неактивная, то устанавливается новый родитель, и изменения сохраняются в базе данных; иначе изменения не происходят.

Рассмотрим один из возможных тестов для компонента *CompanyManagerBean*. В тесте предполагается, что в начальном состоянии имеются две активные компании без родителей. Имена компаний C_1, C_2, а идентификаторы 1, 2 соответственно. Тест состоит из двух потоков, выполняющихся одновременно. Оба потока выполняют метод *testCase*, показанный на Рис. 5, с разными аргументамии. Первый выполняет метод с аргументами 0, 1, 2 а второй – с аргументами 1, 2. Таким образом, первый поток захватывает компанию C_1 на редактирование и, в случае успеха, вызывает метод *setParent* с аргументов – идентификатором компании C_2 (будем писать $C_1.setParent(C_2)$), затем освобождает компанию. Второй захватывает компанию C_2 и, в случае успеха, вызывает метод *setInactive* (будем писать $C_2.setInactive()$), затем освобождает компанию. Для создания экземпляра компонента *CompanyManagerBean* используется вспомогательный метод *getCompanyManager*.

```
public static void testCase(int num, int id, int parent)
throws Exception {
        CompanyManager manager = getCompanyManager();
        manager.editCompany(id);
        try {
                if (num==0) {
                        manager.setParent(parent);
                } else {
                        manager.setInactive();
                }
        } finally {
                manager.freeCompany();
        }
}
```

Рис. 5. Тестовый вариант

Результаты выполнения данного теста зависят от того, в какой последовательности выполнялись удаленные вызовы методов (рис. 6). Если сначала полностью выполнится первый поток, а затем второй, то в результате для компании C_1 будет установлен родитель C_2, а статус компании C_2 по-прежнему останется активным, так как метод *isChildrenInactive* обнаружит у

C_1 активного ребенка C_2. Если сначала выполнится второй поток, а затем первый, то в результате статус компании C_2 станет неактивным, но родитель для C_1 установлен не будет, так как в момент проверки статус родительской компании C_2 будет неактивным. Однако возможен еще и третий вариант, когда проверки в методах *setParent* и *setInactive* выполняются до того, как происходят изменения в базе данных, т.е. выполняются методы *entityManager.merge*. Например, пусть вызовы выполняются в следующей последовательности:

t1.editCompany; t1.setParent; s1.find; t2.editCompany; t2.setInactive; s2.find; s1.merge; s2.merge; t1.freeCompany; t2.freeCompany,

где префиксы *t1*, *t2* означают, что вызовы происходили из первого и второго потоков соответственно, а *s1*, *s2* – соответствующие экземпляры *CompanyManagerBean*. Результатом такого выполнения будет установка для C_1 родительской компании C_2, а для C_2 – установка неактивного статуса, что нарушает первое ограничение.

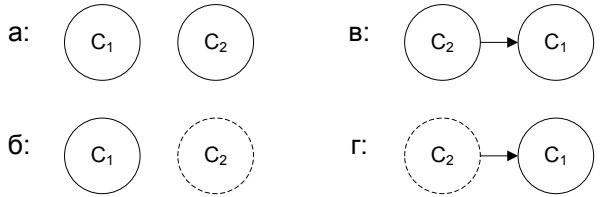

Рис. 6. Возможные результаты выполнения теста

Таким образом, на данном примере можно видеть, что результат работы системы существенным образом зависит от порядка вызовов удаленных методов. Для полноценной проверки системы необходимо проверить работу системы для различных порядков вызова методов.

2.2. Спецификация

Для решения задачи спецификации, генерации тестовых данных и оценки покрытия в качестве основы мы будем использовать технологию UniTESK [12,13]. В UniTESK проверка корректности производится на основе задания пред- и постусловий функций, а также инвариантов [4,14]. Дополнительно для обеспечения возможности спецификации параллельных и распределенных систем введено понятие *отложенной реакции* [15], с помощью которых удается моделировать внутренний недетерминизм в системе.

Для рассмотренного примера приведенные ограничения на систему могут быть сформулированы в виде инвариантов на состояние (Рис. 7). В *модельном состоянии theCompanies* хранятся описания компаний: статус компании и информация о родителях. Инвариант *Rule1_InactiveParent* проверяет первое ограничение, а *Rule2_Level4Hierachy* – второе.

```
invariant Rule1_InactiveParent() {
  for (Company company: theCcompanies) {
    if (company.isActive()) {
          if (company.parent!=null && !company.parent.isActive())
            return false;
    }
  }
  return true;
}

invariant Rule2_Level4Hierarchy() {
  for (Company company: theCompanies) {
          if (company.computeLevel()>4) return false;
  }
  return true;
}
```

Рис. 7. Спецификация ограничений

2.3. Генерация тестовой последовательности

Для генерации тестовой последовательности в UniTESK используется обход конечных автоматов [16,17]. Такой способ построения тестовой последовательности хорошо зарекомендовал себя на практике, многие исследователи отмечают высокое качество получаемых тестовых последовательностей [18,19].

В качестве состояний автомата выбирается модельное состояние или его *обобщение*. Обобщение необходимо для сокращения количества состояний и представляет собой разбиение состояний на классы эквивалентности, рассматриваемые как равные состояния автомата. Переходами автомата являются вызовы одного или более тестируемых методов.

Как показано в [20], одним из удачных обобщенных состояний для деревьев является мультимножество чисел детей. Чтобы получить дерево, в нашем примере мы добавим корневой узел, чтобы все компании без родителей были его детьми. Кроме того, мы разделим детей на активных и неактивных, т.е. обобщенным состоянием будет мультимножество пар (число активных детей, число неактивных детей). На Рис. 8 показаны два различных состояния с добавленным корневым узлом. Обоим состояниям соответствует одинаковое

обобщенное состояние {(0,0), (0,0), (1,0), (1,1)}. Такое обобщение позволяет значительно сократить количество состояний автомата и, тем самым, длину тестовой последовательности.

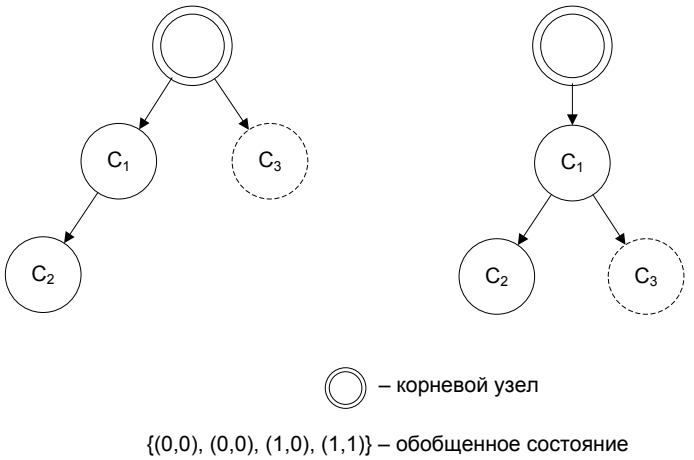

Рис. 8. Обобщение состояний

Рис. 9. Архитектура UniTESK

В процессе тестирования, используя заданное пользователем обобщенное состояние, специальный компонент тестовой системы – *обходчик* строит тестовую последовательность на основе обхода всех переходов автомата. Информация для построения переходов автомата задается в *тестовом сценарии* в виде *сценарных методов* (Рис. 9). В сценарном методе задаются тестовые воздействия на систему. Это могут быть как простые тестовые воздействия, такие как вызовы методов с различными аргументами, так и сложные, такие как порождение нескольких потоков и вызов последовательности методов. Примером сценарного метода является тестовый вариант на рис. 5 с заданной итерацией параметров *id* и *parent* по всем компаниям. Будем ссылаться на этот пример как на сценарный метод *scenExample*.

2.4. Поиск различных порядков

Итерация параметров в сценарном методе происходит с помощью *итерационных переменных*. Итерационые переменные отличаются от обычных переменных тем, что значения этих переменных вместе с именем сценарного метода определяют *стимул* автомата. Количество переходов по заданному стимулу в заданном обобщенном состоянии автомата зависит от результатов вызова сценарного метода с заданными значениями итерационных переменных. Если вызов сценарного метода приводит всегда к одному и тому же результату, то автомат имеет единственный переход по заданному стимулу; иначе переходов столько, сколько возможно различных результатов. Для того чтобы получить различные результаты вызова сценарного метода, требуется перебрать различные порядки событий в системе.

В рассмотренном примере сценарного метода *scenExample* с использованием теста *testCase* возможны три различных результата, соответствующих вызову этого метода с итерационными переменными *id=1*, *parent=2* в начальном состоянии {(0,0), (0,0), (0,2)}: {(0,0), (0,0), (1,1)}, {(0,0), (1,0), (1,0)}, {(0,0), (0,1), (1,0)}.

Рассмотрим следующий пример. Предположим, что для рассмотренного примера иерархии компаний написан сценарий, в котором в качестве обобщенного состояния выбрано описанное выше мультимножество. В сценарии, во-первых, заданы сценарные методы для методов интерфейса *CompanyManager*. Данные сценарные методы позволяют получить разнообразные состояния иерархии: широкие, длинные, с разными конфигурациями активных и неактивных компаний. Эти методы также обеспечивают сильную связность графа автомата. Во-вторых, предположим, что имеется сценарный метод *scenExample*, описанный на основе метода *testCase*. Предположим, что обходчик в ходе построения тестовой последовательности попадает в состояние, соответствующее обобщенному состоянию {(0,0), (0,0), (1,1)}, в котором компания C_1 – неактивная, а C_2 – активная. В этом состоянии обходчик выполняет переход, соответствующий

сценарному методу *scenExample* с итерационными переменными *id=1*, *parent=2*. На Рис. 10 показаны возможные пути выполнения без учета вызовов *editCompany* и *freeCompany*. Кроме того, на рисунке пунктиром показаны вызовы, перебора которых можно избежать за счет использования методов редукции частичных порядков. В соответствии с постановкой задачи метод поиска не имеет возможности отката в предыдущее состояние, поэтому каждый раз метод поиска проходит один из путей целиком и запоминает пройденное дерево. После этого обходчик продолжает обход автомата, и если метод сообщил, что в состоянии {(0,0), (0,0), (1,1)} есть неперебранные порядки, обходчик возвращается в это состояние. При этом от обходчика требуется лишь то, чтобы каждый переход был пройден столько раз, сколько это нужно методу поиска порядков. Никаких других изменений для обходчика не требуется.

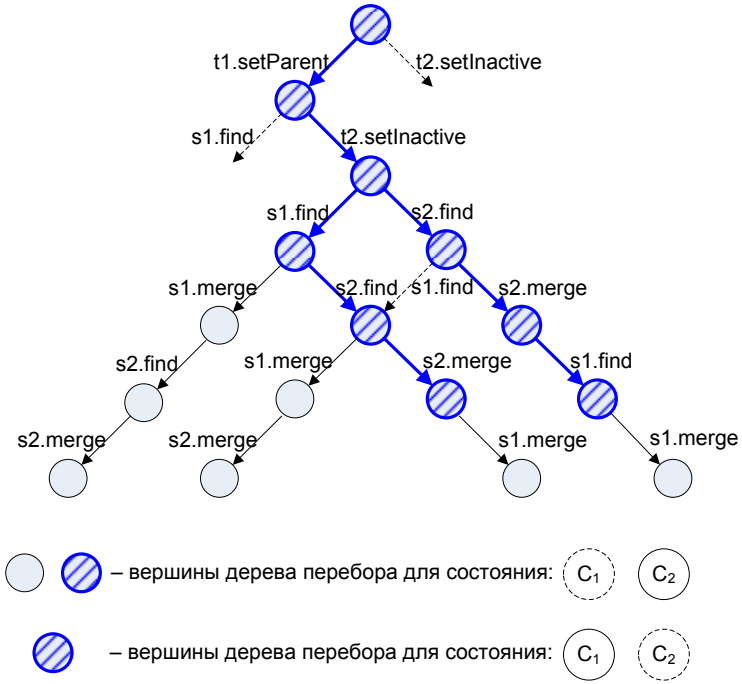

Рис. 10. Деревья перебора для обобщенного состояния {(0,0), (0,0), (1,1)}

Так как обобщенному состоянию {(0,0), (0,0), (1,1)} соответствует также состояние системы, в котором компания C_1 – активная, C_2 – неактивная, обобщенный откат может быть осуществлен в это состояние вместо первоначального состояния. Таким образом, поиск будет необходимо продолжить в этом состоянии. На Рис. 10 полужирными линиями показано соответствующее ему дерево различных порядков выполнения. Как можно

видеть, в этом дереве отсутствует часть порядков, которые присутствовали в предыдущем состоянии. От метода поиска требуется перебрать все возможные порядки, встречающиеся во всех состояниях, соответствующих одному обобщенному состоянию. В рассматриваемом примере множеством порядков, которые необходимо перебрать, является дерево вершин, выделенных жирными линиями.

3. Алгоритм поиска

В данном разделе описан алгоритм поиска различных порядков вызовов методов. Информацию о тестируемой системе алгоритм получает через *управляющего* (*Controller*). В начале каждого удаленного вызова метода выполнение этого вызова блокируется, и управление передается управляющему, который получает информацию о вызываемом методе в виде возможного *перехода* (*Transition*). Будем говорить, что система находится в *глобальном состоянии*, если в тестируемой системе не остается активных потоков, т.е. все потоки ожидают команды управляющего. Алгоритму поиска информация предоставляется через интерфейс управляющего (рис. 11). Для обнаружения достижения глобального состояния алгоритм вызывает метод управляющего *waitForGlobalState*. В глобальном состоянии управляющий выдает информацию о глобальном состоянии (метод *getStateInfo*), включающую в себя список возможных переходов (метод *getTransitionList*) и сигнал о завершении взаимодействий с тестируемой системой (метод *isEndState*). Алгоритм поиска может выполнить один из переходов, вызвав метод *applyTransition*.

```
public interface Controller {
   public static interface StateInfo {
      public List<Transition> getTransitionList();
      public boolean isEndState();
   }
   public void applyTransition(Transition t);
   public StateInfo getStateInfo();
   public void waitForGlobalState();
}
```

Рис. 11. Интерфейс управляющего

Как уже упоминалось ранее, в предлагаемом методе тестирования тестовая последовательность строится на основе обхода графа автомата. Переходы автомата представляют собой тестовые воздействия на тестируемую систему. Состояния – обобщенные состояния. Для построения тестовой последовательности от графа автомата требуется сильная связность, т.е. из любого состояния существует путь в любое другое.

Алгоритм поиска различных порядков выполняется для некоторого перехода в автомате в некотором обобщенном состоянии. Каждый раз при выполнении алгоритма система находится в некотором состоянии, соответствующем данному обобщенному состоянию. Для перехода в автомате и состояния системы определим понятие *дерева перебора*. Вершинами дерева перебора являются глобальные состояния, а дуги в дереве – это возможные переходы из соответствующих глобальных состояний.

Пусть заданы обобщенное состояние S, которому принадлежат состояния системы $s_1, \ldots, s_n$, и переход автомата a. Каждому состоянию s_i и переходу a соответствует дерево перебора t_i. Будем говорить, что дерево t_1 является *поддеревом* дерева t_2, если для любого пути p_1 из t_1 существует путь p_2 в t_2, такой что p_1 является префиксом p_2.

Задача алгоритма поиска – перебрать все пути дерева t, содержащего все пути p, такие что для каждого дерева t_i найдется такой путь p_i, что p является префиксом p_i, то есть $t = \{ p \mid$ для всех t_i существует p_i в t_i: p префикс $p_i \}$. Отметим, что t является поддеревом любого дерева t_i. Такое поддерево будем называть *максимальным поддеревом* деревьев $t_1, \ldots, t_n$.

После каждого отката алгоритм сохраняет дерево перебранных путей выполнения. На выходе у алгоритма максимальное поддерево деревьев $t_1, \ldots, t_n$, соответствующих состояниям $s_1, \ldots, s_n$ обобщенного состояния S.

Алгоритм управляющего выполняется в несколько этапов. На первом этапе вход алгоритма пуст, и алгоритм начинает выполняться в некотором состоянии. На каждом последующем этапе алгоритм получает на входе пройденное дерево переходов, включающее информацию о последнем пройденном пути. Выполнение на каждом последующем этапе может начинаться в другом состоянии, но принадлежащем тому же обобщенному состоянию.

На каждом этапе алгоритм пытается пройти новый путь, принадлежащий деревьям предыдущих этапов. Если же все пути уже пройдены, алгоритм проходит произвольный путь и сообщает о завершении поиска. В конце каждого этапа алгоритм выдает обновленное дерево путей, а также информацию о завершении поиска. Отметим, что алгоритм определяет окончание пути по информации от управляющего. В конце каждого перехода алгоритму известно, завершился ли путь. В тестовых наборах, разработанных по технологии UniTESK, завершение пути соответствует завершению выполнения сценарного метода.

3.1. Структуры данных алгоритма

Структуры данных, используемые алгоритмом, показаны на Рис. 12. Каждый переход *Transition* обязательно включает идентификатор. Дерево путей *Tree* ссылается на корневой узел *Node*, который включает упорядоченный список переходов *transitions*, индекс последнего выполненного перехода в этом списке *lastIndex* и список дочерних вершин *children*. В начале списка

transitions идут выполненные переходы до *lastIndex*, далее идут невыполненные переходы. Список *children* содержит дочерние вершины, соответствующие переходам *transitions*, в том же порядке, размер списка – *lastIndex+1*.

```
public class Tree {
   public Node rootNode =  new Node();
}
public class Node {
   List<Transition> transitions;
   int lastIndex = -1;
   List<Node> children;//size=lastIndex+1
   boolean isEndState = false;
}
public class Transition {
```

Рис. 12. Структуры данных алгоритма поиска

3.2. Этап работы алгоритма

В начале выполнения этапа у алгоритма имеется пройденное дерево переходов *Tree*. Текущий узел дерева – его корень. Алгоритм ожидает начального глобального состояния. После прихода в глобальное состояние алгоритм выполняет цикл, пока не достигнет конечного состояния. В цикле алгоритм получает список возможных переходов и выполняет действия в зависимости от режима работы.

В режиме поиска алгоритм пытается пройти по новому пути. Алгоритм находит пересечение списка переходов в текущем узле дерева и списка переходов для текущего глобального состояния. Таким образом, находится общее поддерево, т.е. остаются только те переходы, которые имеются для уже пройденного дерева и дерева перебора для конкретного состояния системы. Если в результате пересечения уже пройденный переход не был удален, и поддерево в следующем состоянии дерева по этому переходу имеет не пройденные переходы, то алгоритм проходит по ранее пройденному переходу. Текущая вершина – следующая вершина дерева по пройденному переходу. Иначе, если в текущей вершине имеются непройденные переходы, алгоритм проходит по новому переходу и текущая вершина – новая вершина в дереве. Иначе данное поддерево не имеет непройденных переходов, и тогда алгоритм завершает проход по произвольным переходам в режиме симуляции.

В режиме симуляции алгоритм проходит по произвольному пути. В текущем глобальном состоянии выбирается первый из возможных переходов.

В конце цикла алгоритм ожидает глобального состояния. По приходу в него он обновляет информацию о состоянии и переходит в начало цикла.

По завершению цикла, алгоритм возвращает *true*, если в дереве остались непройденные переходы, и *false* иначе.

4. Архитектура для EJB 3.0

Всю информацию, необходимую для выполнения перебора различных порядков, алгоритм поиска получает через интерфейс управляющего (*Controller*). Архитектура, реализующая методы этого интерфейса, показана на Рис. 13.

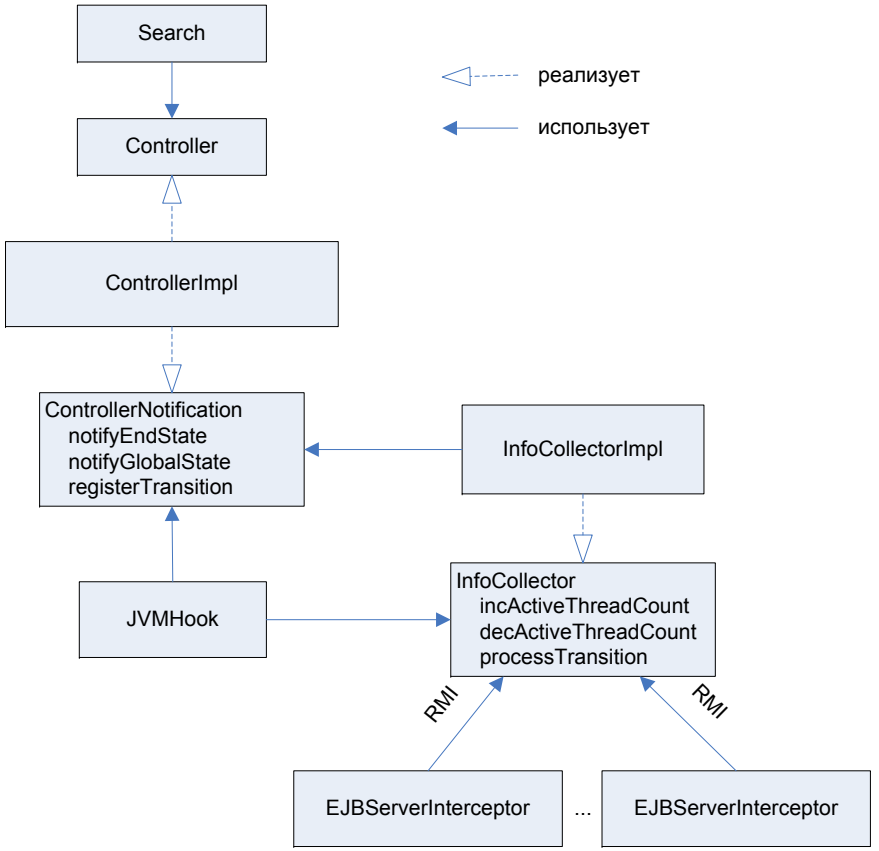

Рис. 13. Архитектура для EJB

Основные поставщики информации – это *EJBServerInterceptor* и *JVMHook*. *EJBServerInterceptor* работает на сервере приложений и перехватывает удаленные вызовы методов компонентов. Встраивание данного класса происходит с помощью механизма *перехватчиков (interceptor)*, определенных

в EJB 3.0 [21]. До выполнения метода компонента выполняются перехватчики, определенные для данного типа компонентов. Механизм перехватчиков используется для реализации таких служб, как управление транзакциями и безопасностью. Нам же от этого механизма требуется только блокировка удаленного вызова метода. *EJBServerInterceptor* через удаленный интерфейс *InfoCollector* сообщает информацию о вызываемом методе, создавая по описанию метода переход *Transition* (*processTransition*). Вызов при этом блокируется. *JVMHook* работает на той же JVM, на которой выполняется тест. Этот класс передает информацию о порождаемых в тесте потоках *incActiveThreadCount* и *decActiveThreadCount*, а также информацию о завершении теста *notifyEndState*. Информация о количестве порождаемых потоков необходима для обнаружения начального глобального состояния.

Компоненты, разработанные по технологии EJB, обладают рядом важных свойств:

1. В компонентах не создаются новые потоки. Если с системой работает *N* клиентов, то для блокировки всех взаимодействий в системе достаточно заблокировать *N* вызовов методов.
2. В каждый момент для экземпляра компонента может выполняться лишь один метод. В зависимости от настроек сервера приложений другие вызовы либо помещаются в очередь, либо завершаются с исключением. Это гарантирует, что данные экземпляра могут одновременно изменяться лишь одним потоком.

Кроме того, мы будем требовать выполнения двух дополнительных ограничений, которые не являются обязательными по спецификации EJB, однако выполняются для большинства систем:

3. Обращения к *EntityManager* происходят внутри транзакций. Этим обеспечивается атомарность обновления данных в базе данных.
4. Все обращения к разделяемой памяти сосредоточены в *EntityManger*, т.е. других обращений нет.

Последние два ограничения, по сути, требуют, чтобы все обращения к разделяемой памяти были атомарными и контролировались (перехватывались) управляющим.

Для систем, удовлетворяющих этим ограничениям, верно, что для любого конкретного состояния системы и воздействий на систему:

1. Существует единственное дерево перебора;
2. Пути дерева перебора описывают все возможные результаты выполнения.

Таким образом, гарантируется, что если для воздействий, осуществляемых в переходе автомата, система может работать некорректно (несоответствует

спецификации), то соответствующий некорректный результат будет обнаружен.

Корректность для всей системы в целом зависит от выбранного разработчиком тестов обобщенного состояния и тестовых воздействий, осуществляемых в сценарных методах. При выполнении ряда гипотез [12] можно гарантировать, что вся система корректна. В гипотезах требуется, чтобы система в обобщенных состояниях вела себя «похоже». Важным результатом данной работы является возможность утверждать корректность не только для последовательных, но и распределенных программ.

5. Заключение

В работе разработан метод тестирования систем, которые построены на основе компонентов, взаимодействующих с помощью удаленного вызова методов. Метод позволяет использовать удачные решения технологии UniTESK, такие как формулировка требований в виде формальных спецификаций и построение тестовой последовательности на основе обхода автоматов. Вместе с тем, метод гарантирует, что для всех тестовых воздействий, выполняемых в каждом переходе автомата, будут проверены все различные порядки вызовов методов. В работе показано, что для систем, построенных по технологии EJB 3.0, этого достаточно для получения всех возможных результатов выполнения. Тем самым, удается гарантировать корректную работу системы в соответствии со спецификацией вне зависимости от «погодных» условий, в которых работает система, таких как планирование процессов и потоков в операционной системе, а также задержки при сетевых взаимодействиях.

Метод тестирования был применен к нескольким системам. Были обнаружены проблемы, которые не проявляются или проявляются редко при запусках без перебора различных порядков. Благодаря тому, что одни и те же тесты можно использовать как с перебором, так и без него, отладку тестов можно проводить без перебора, а затем проводить более тщательное тестирование, используя возможность перебора. Такой подход в проведенных экспериментах позволил значительно сократить время разработки тестов.

Использование возможности перебора не для всей системы вместе с окружением целиком, а для отдельных тестовых воздействий позволило значительно сократить пространство перебора. Использование поиска без сохранения состояний позволяет сократить использование памяти. В проведенных экспериментах время работы тестов не превышало нескольких минут.

Литература

[1] Szyperski C. Component Software Beyond Object-Oriented Programming. Boston, MA: Addison-Wesley and ACM Press, 1998.
[2] Документация по Java RMI. http://java.sun.com/j2se/1.5.0/docs/guide/rmi/.

[3] Таненбаум Э., М. ван Стеен. Распределенные системы. Принципы и парадигмы. СПб.: Питер, 2003.
[4] Bertrand Meyer. Applying 'Design by Contract'. IEEE Computer, vol. 25, No. 10, October 1992, pp. 40–51.
[5] O. Tkachuk, M. B. Dwyer and C. S. Pasareanu. Automated Environment Generation for Software Model Checking. Proceedings of the Eighteenth IEEE International Conference on Automated Software Engineering, 2003.
[6] P. Godefroid. Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Secaucus, NJ, USA: Springer-Verlag, 1996.
[7] P. Godefroid. Model Checking for Programming Languages using VeriSoft. Proceedings of the 24th ACM Symposium on Principles of Programming Languages. ACM Press, pp. 174–186, January 1997.
[8] G. Brat, K. Havelund, S.-J. Park, and W. Visser. Model Checking Programs. In IEEE International Conference on Automated Software Engineering (ASE), pp. 3–12, September 2000.
[9] И.Б. Бурдонов, А.С. Косачев, В.В. Кулямин. Применение конечных автоматов для тестирования программ. Программирование, 26(2):61-73, 2000.
[10] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines – a survey. Proceedings of the IEEE, vol. 84, pp. 1090-1123, Berlin, August 1996.
[11] R. Monson-Haefel, B. Burke. Enterprise JavaBeans 3.0, Fifth Edition. Sebastopol, CA: O'Reilly, 2006
[12] В.В. Кулямин, А.К. Петренко, А.С. Косачев, И.Б. Бурдонов. Подход UniTESK к разработке тестов. Программирование, 29(6): 25–43, 2003.
[13] I. Bourdonov, A. Kossatchev, V. Kuliamin, and A. Petrenko. UniTesK Test Suite Architecture. Proceedings of FME 2002. LNCS 2391, pp. 77–88, Springer-Verlag, 2002.
[14] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. ACM SIGSOFT Software Engineering Notes, 31(3):1–38, March 2006.
[15] Хорошилов А.В. Спецификация и тестирование компонентов с асинхронным интерфейсом. Диссертация на соискание ученой степени кандидата физико-математических наук. Москва: ИСП РАН, 2006.
[16] И. Б. Бурдонов, А. С. Косачев, В. В. Кулямин. Неизбыточные алгоритмы обхода ориентированных графов. Детерминированный случай. Программирование, 29(5):59–69, 2003.
[17] И. Б. Бурдонов, А. С. Косачев, В. В. Кулямин. Неизбыточные алгоритмы обхода ориентированных графов. Недетерминированный случай. Программирование, 30(1):2–17, 2004.
[18] H. Robinson. Intelligent Test Automation. Software Testing and Quality Engineering, September/October 2000, pp. 24–32.
[19] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating Finite State Machines from Abstract State Machines. ISSTA 2002, International Symposium on Software Testing and Analysis, July 2002.
[20] В.С. Мутилин. Паттерны проектирования тестовых сценариев. Труды ИСП РАН, 9: 97–128, 2006.
[21] EJB 3.0 Specification, JSR 220 FR. http://java.sun.com/products/ejb/.