И.Б. Бурдонов, С.Г. Грошев, А.С. Камкин, А.С. Косачев, А.А. Сортов.

Параллельное тестирование больших автоматных моделей

Введение

Данная статья описывает расширение технологии тестирования UniTESK [1], использующее возможности вычислительных кластеров для параллельного тестирования сложных систем на соответствие формальным моделям требований к ним.

В технологии UniTESK тестируемая система (реализация) представляется в виде модельного конечного автомата (МКА), входные символы соответствуют тестовым стимулам, а выходные символы – выдаваемым тестируемой системой реакциям. Процесс тестирования заключается в прохождении маршрутов по графу состояний МКА (далее называемому «модельным графом» или просто «графом»), при этом на каждом переходе автоматически осуществляются проверки на соответствие наблюдаемого поведения (реакции) тестируемой системы заданным требованиям, которые описаны в виде спецификации. Вообще говоря, спецификация определяет класс удовлетворяющих ей реализаций, модельные графы которых являются различными подграфами графа спецификации. Кроме того, сам граф спецификации задается неявно в виде пред- и постусловий. Поэтому при тестировании граф реализации строится по мере его обхода; обход графа завершается после прохождения всех его дуг (когда обнаружено, что ни из одной известной вершины не выходит непройденная дуга) или обнаружения ошибки, не позволяющей продолжать тестирование.

Подробнее о способе тестирования с помощью конечных автоматов, используемом в технологии UniTESK можно прочитать в [1,3]. Такой способ тестирования даёт хорошее покрытие требований к тестируемой системе. Однако размер модельного графа может быть очень большим, даже после применения различных способов его сокращения [2]. В результате обход МКА на одной машине может длиться недопустимо долго.

Чтобы ускорить процесс тестирования, сохраняя при этом достоинства технологии UniTESK (в частности, автоматическое обеспечение полноты обхода модельного графа и автоматическую оценку достигнутого покрытия формальной модели тестируемой системы), предлагается использовать возможности, предоставляемые распределёнными вычислительными системами.

Используемый метод тестирования имеет следующие особенности:

1. Тестовая последовательность строится в виде единого маршрута по графу, проходящего все его дуги.
2. Граф не задан заранее, а строится динамически по мере его обхода.
3. Как правило, единственным способом перевода тестируемой системы в требуемое состояние является прохождение в графе маршрута, ведущего из текущего состояния в требуемое.

Эти особенности создают определённые сложности для распараллеливания, не позволяя распределять работу между узлами вычислительного кластера статически. Существуют методы (например, описанный в работе [4]), позволяющие записывать информацию о графе во время его обхода и использовать её в дальнейшем для статического построения кратких маршрутов, частично решая эти проблемы. Однако для таких методов необходим первоначальный обход графа, который также может занимать недопустимо долгое время; кроме того, такие статические методы неустойчивы к изменениям как спецификации, так и реализации, которые неизбежны в процессе разработки.

Для решения этих проблем была разработана система тестирования методом параллельного обхода графов, описанная в данной статье.

Требования к тестовой системе

Для возможности параллельной работы тестовой системы необходимо, чтобы структуры МКА и порождённого им модельного графа удовлетворяли следующим ограничениям:

(A) Модельный граф конечен и сильно связан.
(B) На всех узлах вычислительного кластера граф одинаков.
(C) Для любой вершины графа можно вычислить все стимулы, которыми помечены выходящие из неё дуги.
(D) МКА детерминирован в том смысле, что пресостояние и стимул любого перехода однозначно определяют его постсостояние. Это требование может быть ослаблено в зависимости от используемых алгоритмов обхода [5].

Выполнение вышеуказанных требований необходимо для *возможности* применения предлагаемого метода распределённого тестирования. Кроме того, для *целесообразности* его применения должно выполняться следующее требование:

(E) Обмены информацией о дугах с другими узлами кластера должны требовать существенно меньше времени, чем проходы дуг графа (включающие в себя применение тестового воздействия, его обработку тестируемой системой и оценку правильности полученных реакций) одним узлом.

Архитектура тестовой системы

Тестовая система состоит из множества процессов, выполняющихся на узлах вычислительного кластера. Процессы могут обмениваться сообщениями, но не имеют общих данных: каждый процесс имеет свою копию тестируемой системы и работает, вообще говоря, по своему алгоритму обхода графа. Процессы связаны односторонними сетевыми соединениями типа точка-точка (двусторонние соединения рассматриваются как пары односторонних соединений), образующими сильно связный ориентированный граф; по этим соединениям они обмениваются информацией об исследованной части графа. Предполагается, что соединения сохраняют порядок сообщений, не теряют их и не производят лишних сообщений.

Каждый процесс осуществляет генерацию своей тестовой последовательности и подачу тестовых воздействий на тестируемую систему. Процесс представляет собой систему, состоящую из нескольких взаимодействующих компонентов. Основными из них являются (см. Рис. 1):

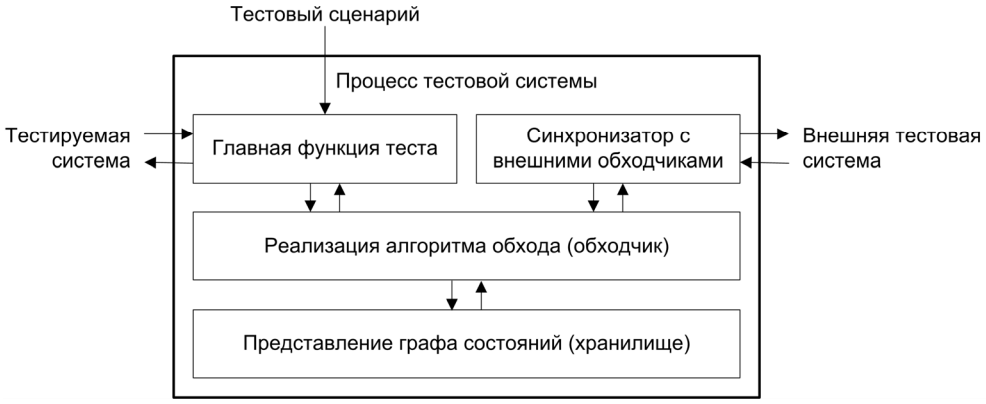

Рисунок 1. Устройство процесса тестовой системы

Реализация алгоритма обхода или *обходчик* – библиотечный компонент, реализующий некоторый неизбыточный алгоритм обхода графов [3]. Обходчик является пассивным компонентом, в том смысле, что он не подает воздействия на тестируемую систему. Основной задачей обходчика является вычисление маршрута из заданной вершины графа в какую-либо вершину, из которых выходят еще не пройденные дуги.

Все обходчики реализуют единый интерфейс, предоставляющий следующие функции:

- *Инициализировать обходчик*. Инициализирует хранилище, устанавливает параметры обхода и передает обходчику начальную вершину графа и список допустимых в ней стимулов.
- *Вычислить маршрут в графе и подаваемый стимул*. Вычисляет маршрут из указанной вершины графа в вершину, в которой есть еще не пройденные дуги, а также стимул одной из этих дуг. Если в графе отсутствуют непройденные дуги, функция сообщает о завершении тестирования. Если обнаружено нарушение условия сильной связности графа или других требований на граф, налагаемых обходчиком, функция сообщает об ошибке.
- *Добавить в граф пройденную дугу*. Добавляет дугу в хранилище (при этом указывается, получена ли эта дуга синхронизатором от другого процесса или пройдена локально). Вместе с дугой передается информация о числе стимулов, допустимых в конечной вершине дуги.
- *Получить список дуг, пройденных локально*. Возвращает список дуг, пройденных локальным обходчиком с момента последнего вызова этой функции (или начала работы процесса тестовой системы). Эта функция вызывается синхронизатором для получения дуг, которые нужно передать другим обходчикам.
- *Завершить работу обходчика*. Освобождает ресурсы, занятые во время обхода.

Для повышения эффективности параллельного обхода графа используется «развод» обходчиков по разным подграфам (идеальной является ситуация, когда разные обходчики работают с попарно не пересекающимися подграфами). Это особенно актуально в начале тестирования, поскольку все процессы стартуют из одной и той же начальной вершины, а информация о пройденных дугах распространяется между процессами с задержками. Развод может быть реализован разными способами. Один из наиболее простых подходов основан на передаче каждому обходчику при инициализации его идентификатора – числа I из множества $\{0, \ldots, N-1\}$, где N — число процессов тестовой системы. Каждый раз, когда обходчик сталкивается с выбором стимула из списка еще не поданных стимулов L, он выбирает тот из них, который расположен по индексу $I \bmod |L|$, где $|\cdot|$ — размер списка. Другим способом является случайный выбор стимула.

Представление графа состояний или *хранилище* – часть обходчика или независимая подсистема, отвечающая за хранение информации о пройденной части графа. Обновление хранилища осуществляется через интерфейс обходчика. Используемое представление данных может зависеть от выбранного алгоритма обхода и особенностей реализации, однако любое представление содержит следующую информацию:

- Известные вершины графа.
- Пройденные дуги, для которых известны начальная вершина, стимул перехода и конечная вершина.
- Стимулы, допустимые в известных вершинах.

Главная функция теста – активная часть тестовой системы, которая на основе тестового сценария (неизбыточного описания графа), используя обходчик (обновляя модельный граф и вычисляя пути в вершины с непройденными дугами), подает воздействия на тестируемую систему и проверяет полученные от неё реакции.

Синхронизатор – осуществляет обмен данными с аналогичными компонентами на других узлах и с обходчиком.

Протокол синхронизации

Каждый процесс регулярно выполняет процедуру синхронизации. Процедура синхронизации инициируется появлением входящих сообщений, или обновлениями в локальном хранилище или таймером. Алгоритм синхронизации следующий:

I. Синхронизатор принимает все входящие сетевые сообщения и запрашивает у обходчика *обновление* – множество новых дуг, пройденных *данным* процессом с момента последней синхронизации. Обозначим через A множество дуг, информация о которых содержится в принятых сообщениях, B – множество дуг, которые данный процесс уже посылал в сообщениях по выходящим связям, C – обновление.

II. В локальное хранилище (через обходчик) добавляется множество *новых* переходов $A \setminus (B \cup C)$, помеченных как полученные от других процессов.

III. По *всем* исходящим соединениям рассылается сообщение, содержащее множество переходов $(A \cup C) \setminus B$

IV. Обновляется множество посланных переходов: $B := B \cup A \cup C$.

Упорядочив каким-либо образом стимулы, мы в силу требований (B) и (C) можем перенумеровать для каждой вершины Z все допустимые в ней стимулы числами от **1** до $deg(Z)$, где $deg(Z)$ – количество таких стимулов, и далее рассматривать в качестве стимулов переходов только эти номера. Дуги передаются по сети в виде четвёрки $(X,Y,Z,deg(Z))$, где X – идентификатор начальной вершины дуги, Y – номер стимула, Z – идентификатор конечной вершины дуги. Таким образом получателям сообщений вместе с информацией о новой вершине графа становится также известен список допустимых в ней стимулов.

Несложно убедиться, что при таком протоколе синхронизации информация о каждой новой дуге, то есть пройденной хотя бы одним процессом, дуге графа передаётся по каждому соединению ровно 1 раз. Оценим время T, через которое информация о новой дуге станет известна всем процессам (избавляя их от необходимости проходить её самостоятельно). Пусть t – максимальное время от передачи информации о дуге от одного процесса к соседнему процессу в сети, включая время обработки сообщения в процессе. *Расстояние* от процесса p_1 до процесса p_2 – минимальная длина пути из p_1 в p_2 по графу межпроцессных соединений. Пусть d – диаметр графа межпроцессных соединений, то есть максимум из расстояний от процесса до процесса. Тогда $T=O(dt)$.

Отметим также, что используя этот факт, можно очищать множество B, удаляя из него дуги, которые уже были получены по каждой из входящих связей.

Координация

Процесс тестирования может управляться с использованием координатора. Координатор предоставляет пользователю возможность задать конфигурацию вычислительной среды: узлы, на которых запускаются процессы тестовой системы и связи между ними. В начале работы тестовой системы координатор запускает все процессы и передает им информацию об их входящих и выходящих соединениях. Кроме того, координатор позволяет управлять процессом тестирования и централизованно собирать информацию о состоянии процессов, найденных ими ошибках, достигнутом тестовом покрытии, количестве пройденных дуг графа и т.д.

При обнаружении ошибки дальнейшее поведение тестовой системы определяется уровнем критичности этой ошибки. Выделяются следующие уровни:

1. Локально устранимая ошибка – ошибка, которая не влияет на дальнейший обход графа. При обнаружении такой ошибки процесс оповещает координатора и продолжает процесс тестирования и обмен данными с другими процессами.
2. Локально неустранимая ошибка – ошибка, при которой дальнейший обход графа этим процессом невозможен. При обнаружении такой ошибки процесс передает информацию об этом координатору, прекращает тестирование, однако продолжает

обмен данными с другими процессами, сохраняя таким образом сильную связность графа межпроцессных соединений.

3. Глобально неустранимая ошибка – ошибка, при которой невозможен дальнейший обход графа всеми процессами. Процесс оповещает координатора, который передает всем остальным процессам указание о прекращении тестирования.

Завершение работы тестовой системы также возможно при получении соответствующей команды от пользователя. В этом случае координатор передает всем узлам указание завершить работу. Процессы могут сохранить свое состояние, чтобы была возможность в дальнейшем возобновить тестирование.

Сохранение состояния и восстановление после сбоев

Даже несмотря на ускорение тестирования за счёт распараллеливания полный обход модельного графа всё равно может занимать весьма продолжительное время. В случае если вычислительный кластер собран из рядовых дешёвых компонентов без применения технологий обеспечения отказоустойчивости системы в целом, его среднее время наработки до отказа одного из узлов уменьшается вместе с ростом числа узлов. В результате возникает необходимость сохранять результаты проделанной работы по обходу графа для последующего восстановления и продолжения обхода только неисследованной его части. Кроме того, даже при отсутствии сбоев может возникать потребность остановить работу вычислительного кластера без потери результатов работы, например, для проведения регламентных работ.

Рассмотрим два протокола такого сохранения и восстановления.

Протокол контрольных точек.

Построение контрольной точки. Время от времени (по команде координатора или по некому общему соглашению) все процессы приостанавливают обход графа, продолжая при этом обмены информацией, и ждут, когда каждый процесс завершит прохождение очередной дуги, а в сети перестанут циркулировать сообщения о пройденных дугах. Как показано выше, это произойдёт, как только информация о каждой пройденной дуге достигнет каждого процесса, после чего в хранилище каждого процесса будет содержаться информация об одном и том же наборе вершин и пройденных и непройденных дуг. После этого каждый процесс сохраняет информацию из своего хранилища на локальный диск, а когда сохранение успешно завершено всеми процессами, продолжает тестирование из той вершины модельного графа, в которой он находится в данный момент (или завершает работу, если поступила соответствующая команда от координатора).

Восстановление происходит следующим образом: после запуска всех процессов и установления между ними соединений каждый процесс загружает из локального файла состояние хранилища, сохранённое при последнем успешном построении контрольной точки, после чего начинает обход графа с его начальной вершины.

Протокол журнализации.

Несколько процессов при старте назначаются сохраняющими состояние (если не нужна защита от таких сбоев узла, при которых теряется содержимое его диска, то достаточно одного процесса). Синхронизаторы этих процессов каждый раз при рассылке по сети сообщений с множеством дуг одновременно записывают эти дуги в том же порядке в файл на локальном диске (*журнал*).

Восстановление происходит следующим образом: после запуска всех процессов и установления соединений один из процессов начинает последовательную загрузку дуг из журнала, в то время как все остальные начинают работать обычным образом. Во время загрузки этот процесс не осуществляет обход графа, но принимает и ретранслирует сообщения о новых дугах, полученные от других процессов. Загружаемые из журнала дуги также рассылаются по сети и добавляются в локальное хранилище вместе с полученными от

других процессов. Отметим, что выполнение требования (E) гарантирует, что загрузка и рассылка всем процессам сохранённых данных об известной части графа могут быть выполнены существенно быстрее, чем повторный её обход. После завершения загрузки из журнала этот процесс переходит к тестированию в обычном режиме, при этом он может дописать в конец своего журнала дуги, которые в нём отсутствовали, но были получены во время загрузки от других процессов, и в дальнейшем продолжать дописывать в конец журнала все новые дуги.

Заключение

В статье описано расширение технологии UniTESK средствами распараллеливания работы тестовых систем на распределенных вычислительных кластерах. Важной особенностью представленной работы является то, что распараллеливание осуществляется динамически, без использования информации о структуре графа. С точки зрения инженеров работать с распределенной тестовой системой не сложнее, чем с обычной тестовой системой, выполняемой на одном компьютере (дополнительные входные данные связаны со структурой межпроцессных соединений, но они задаются один раз, при конфигурации). Предлагаемый подход обеспечивает существенное сокращение времени выполнения тестов, а как следствие, уменьшение времени, затрачиваемого на обнаружение ошибок, и ускорение процесса проектирования в целом.

Дальнейшее направление работы мы видим в реализации механизмов, нацеленных на повышения гибкости и эффективности распараллеливания тестов. К ним относятся динамическая реконфигурация топологии сети (для сложных тестов может потребоваться ускорение за счет добавление новых вычислительных узлов в граф межпроцессных соединений) и поддержка систем с общей памятью, в частности, многоядерных процессоров (в этом случае возможна более эффективная реализация синхронизаторов, а также совместное использование одного хранилища несколькими процессами).

Литература:

1. В.В. Кулямин, А.К. Петренко, А.С. Косачев, И.Б. Бурдонов. Подход UniTesK к разработке тестов. // Программирование, 29(6), стр. 25-43, 2003
2. И.Б. Бурдонов, А.С. Косачев, В.В. Кулямин. Использование конечных автоматов для тестирования программ. // Программирование, 26(2), стр. 61-73, 2000
3. И.Б. Бурдонов, А.С. Косачев, В.В. Кулямин. Неизбыточные алгоритмы обхода ориентированных графов: детерминированный случай. // Программирование, 29(5), стр. 59-69, 2003
4. С.Г. Грошев. Локализация ошибок методом построения сокращенных трасс. // Программирование, 35(3), стр. 35-50, 2009
5. И.Б. Бурдонов, А.С. Косачев, В.В. Кулямин. Неизбыточные алгоритмы обхода ориентированных графов: недетерминированный случай. // Программирование №1(30), стр. 2-17, 2004