А. А. Усенок

ОБЗОР МЕТОДОВ АВТОМАТИЗИРОВАННОГО ОБНАРУЖЕНИЯ СБОЕВ В ПРОГРАММНОМ ОБЕСПЕЧЕНИИ

Введение

Значительный рост информатизации предприятий привел к увеличению сложности вычислительных систем в целом и программного обеспечения в частности. К автоматизированным системам, обеспечивающим непрерывные бизнес-процессы, предъявляются высокие требования по времени безотказной работы. Они должны оставаться в работоспособном состоянии даже в условиях программных и аппаратных сбоев, а кроме того, должны позволять осуществлять мониторинг, отладку и обновление некоторых своих подсистем без остановки работы.

При разработке ПО, отвечающего столь высоким требованиям, необходимо обратить внимание на два аспекта. Первый из них касается корректности спецификаций программы и соответствия фактической реализации этим спецификациям. Второй относится к проверкам во время выполнения, контролирующим правильную работу программы в реальных условиях, включающих аппаратные сбои, программные и человеческие ошибки.

Эта статья посвящена второму аспекту, а именно обзору методов автоматизированного обнаружения сбоев в процессе выполнения программ.

Сложность обнаружения сбоев в процессе выполнения состоит в том, что не все программные ошибки приводят к катастрофическим последствиям для системы, таким как отказ или выход из строя. Многие ошибки приводят к получению некорректных результатов, и потому обнаружить их гораздо сложнее. Кроме того, программные ошибки могут проявляться с большой задержкой, что также затрудняет их своевременное обнаружение и локализацию.

Метод зондирования

Метод зондирования является самым очевидным подходом к обнаружению сбоев в работе системы и заключается в мониторинге и периодической проверке функционирования различных компонентов системы. Как уже было отмечено, программные ошибки могут проявляться с задержкой, поэтому такой подход позволяет существенно снизить накладные расходы, поскольку не требует постоянного контроля над работой системы.

Идея одной из подобных методик, предложенной в [1], состоит в том, что зачастую для обнаружения сбоя нет необходимости проводить полноценное тестирование, а достаточно нескольких простых проверок. При этом важно правильно разделить систему на уровни и составные части, чтобы тестирование было максимально полным.

Функционирование системы зондирования может рассматриваться как взаимодействие двух базовых компонентов: диспетчера и анализатора (рис. 1).

Диспетчер отвечает за запуск тестов и сохранение информации о функционировании системы в буфер результатов тестирования. Исполняемая часть всех тестов хранится в библиотеке и запускается диспетчером после получения соответствующей команды из очереди. Тесты находятся в активном состоянии до получения результатов от целевых ресурсов либо до истечения установленного промежутка времени. Результаты выполнения тестов помещаются в буфер.

Анализатор выбирает результаты тестирования из буфера и разбирает их для получения информации о функционировании системы в целом. Для этого он использует модель системы, содержащую всю необходимую информацию о ее компонентах, которая может потребоваться при разборе результатов. На основании результатов анализа выдается оповещение о состоянии системы для администратора.

Запуск тестов производится по расписанию. Анализатор помещает соответствующую команду в очередь и ожидает результатов выполнения теста в буфере.

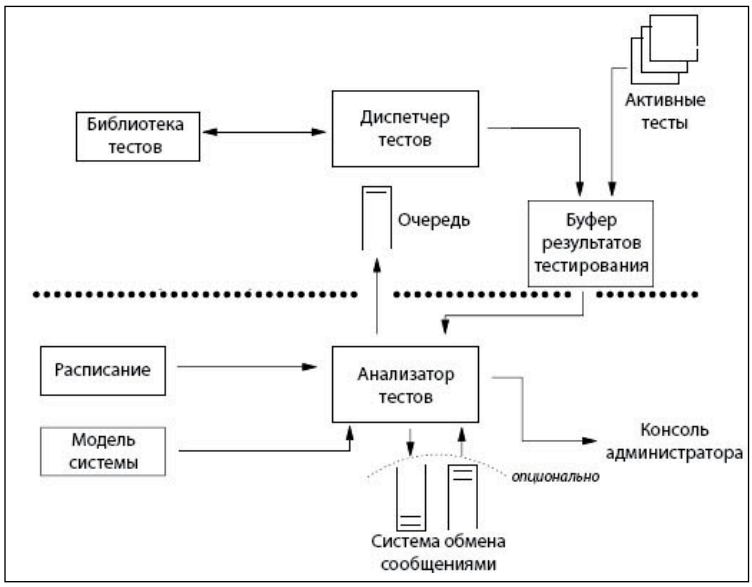

Рис. 1. Архитектура системы зондирования

Метод контроля времени выполнения

Метод был впервые предложен в 1993 г. в работе [2]. Он заключается в контроле программы во время выполнения на соответствие формализованной спецификации, составленной на этапе разработки. Спецификация формируется путем аннотирования программного кода конструкциями специального языка. На этапе компиляции эти конструкции преобразуются в контрольные функции, которые вставляются в исходную программу в тех местах, где соответствие спецификации может быть нарушено. Когда это происходит, генерируется оповещение.

При таком подходе становится возможным обнаруживать сбои в аппаратном обеспечении, ошибки в реализации компилятора, ядра операционной системы и прочем системном программном обеспечении, а кроме того, достигается высокая точность обнаружения и локализации ошибок.

Поскольку непосредственный контроль в процессе выполнения программы приводит к большим накладным расходам, исследователями было предложено осуществлять его параллельно, в отдельном потоке или процессе. Однако при подобном подходе у контролируемой программы появляется возможность продолжать работу после нарушения спецификации — такова плата за увеличение быстродействия методики на многопроцессорных системах. Тем не менее в ряде случаев это не представляет опасности.

Для решения этой проблемы в тех случаях, когда это необходимо, исследователями была предложена специальная конструкция для синхронизации выполнения основного и контролирующего потоков программы, которая позволяет приостановить выполнение до тех пор, пока все необходимые проверки не будут завершены.

Если контролируемая программа проходит новую контрольную точку в тот момент, когда еще не завершена обработка предыдущей, выполнение программы останавливается до тех пор, пока не будет завершена более ранняя проверка. Для предотвращения подобных блокировок предложено организовывать очереди проверок для отслеживания их очередности. При этом новая проверка просто добавляется в конец очереди, откуда ее извлечет контролирующий поток, предоставляя возможность основному потоку продолжать выполнение.

Существует три основных способа реагирования на обнаруженные ошибки:
- игнорирование — контрольный поток генерирует только оповещение об ошибке. Такая схема полезна в некритичных случаях, когда журнал событий в дальнейшем используется для ручного анализа программы;
- генерация исключений — в следующей контрольной точке будет сгенерировано специальное исключение, которое может быть обработано целевой программой. Такая схема может быть применена в том случае, когда необходимо обеспечить возможность автоматического восстановления после отказов;
- завершение программы — контрольный поток вынуждает программу аварийно завершиться в случае обнаружения ошибки. Такая схема может быть применена, если продолжение работы после отказа недопустимо.

Метод динамического дублирования

В последнее десятилетие произошел значительный рост быстродействия микропроцессоров, который во многом связан с достижениями в области миниатюризации электронных компонентов. Однако у этого подхода есть и обратная сторона — увеличение плотности компоновки неизбежно приводит к уменьшению помехоустойчивости и увеличению вероятности перемежающего отказа.

Для уменьшения влияния отказов такого рода на работоспособность системы применяются различные механизмы: коды обнаружения и коррекции ошибок, резервирование компонентов и т. п. Все эти механизмы требуют внесения изменений в аппаратное обеспечение системы.

Метод динамического дублирования [3] решает эту же задачу и является исключительно программным решением, не требующим модификации оборудования, что значительно упрощает процесс его внедрения. Кроме того, этот метод не требует повторной компиляции программ из исходного кода, что в ряде случаев может быть затруднительно или вовсе невозможно.

Суть метода заключается в дублировании каждой процессорной инструкции (кроме инструкций записи в память) с последующим сравнением результатов выполнения обеих инструкций. Поскольку сбой проявляется только в том случае, когда он влияет на выходные данные программы, то желательно отложить проверку до тех пор, пока не будет выполнена инструкция, которая может оказать влияние на эти данные. Разработчики метода предполагают, что данные, записываемые в память, могут повлиять на выходные данные, но откладывают проверку до сохранения или вывода этих данных. Это уменьшает количество ложных срабатываний, но приводит к невозможности обнаружения ошибок при работе с памятью.

Кроме того, существует ряд инструкций, которые невозможно просто продублировать, — такие инструкции должны обрабатываться отдельно. К примеру, метод не дублирует инструкции загрузки в регистр из памяти, а просто копирует загруженное значение из одного регистра в другой, поскольку две последовательные операции загрузки с одинаковым адресом не всегда имеют одинаковый результат. Это особенно актуально для многопоточных программ, где один из потоков может изменить данные в промежутке между исходной и повторной инструкциями чтения. Аналогичные проблемы возникают с инструкциями, работающими с системным таймером, для которых результаты всегда различны (например, RDTSC).

Дублирование инструкций происходит динамически в процессе работы программы. По результатам тестирования, проведенного разработчиками метода, среднее время выполнения программы при дублировании инструкций увеличивается примерно в 2,02 раза, а при добавлении проверок на совпадение результатов выполнения инструкций — в 3,77 раза.

Метод диспетчера

В случае, когда поведение системы может быть формализовано в виде спецификации, программные отказы могут быть обнаружены отдельным модулем, который контролирует

входные и выходные данные системы. Модуль сопоставляет эти данные и определяет, насколько это поведение соответствует спецификации (рис. 2). При этом не исключено, что такой модуль окажется сложнее системы, которую он контролирует. В работе [4] модуль, реализующий подобный функционал, именуется диспетчером (англ. *supervisor*).

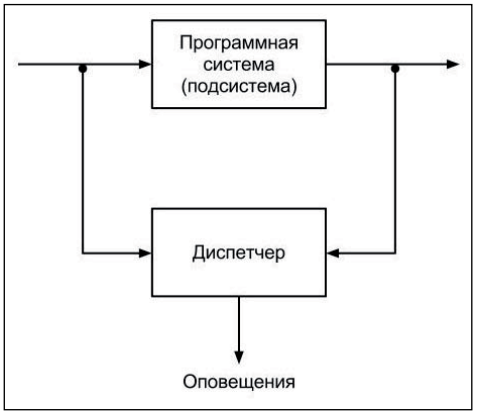

Рис. 2. Схема работы программного диспетчера

В общем виде, программный диспетчер состоит из пяти компонентов (рис. 3). Спецификация, содержащая требования к программной системе, преобразуется в поведенческую модель, которая может быть использована интерпретатором для генерации допустимых ответов. Полный набор сгенерированных ответов размещается в соответствующем буфере, в другой буфер помещаются полученные от системы ответы. Блок сопоставления сравнивает данные, содержащиеся в обоих буферах, и если совпадения отсутствуют — генерирует оповещение об ошибке.

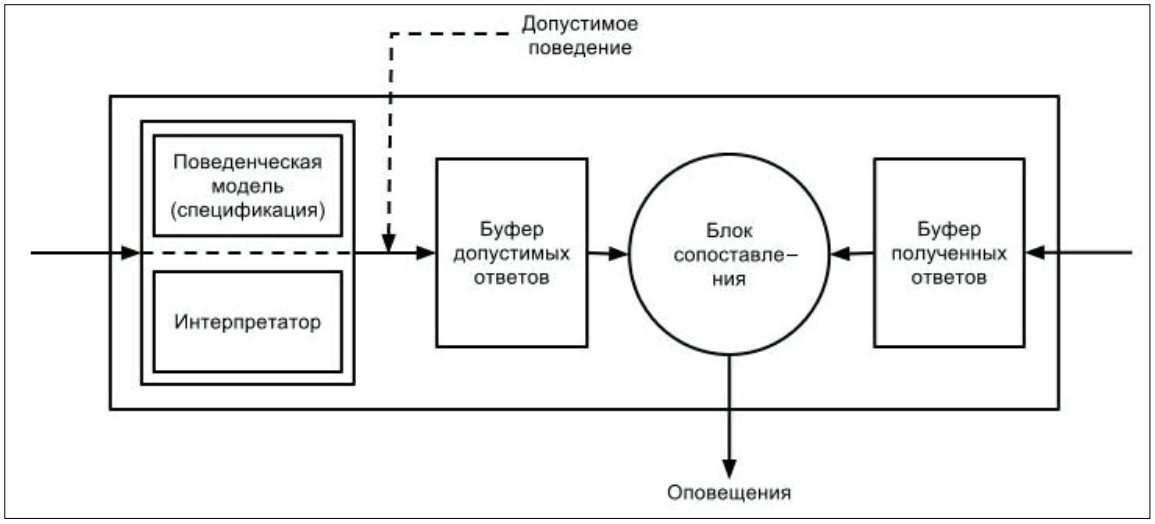

Рис. 3. Архитектура программного диспетчера

По способу организации своей работы диспетчеры делятся на два типа. Диспетчер первого типа после получения входных данных предварительно вычисляет все допустимые ответы в соответствии со спецификацией и текущим состоянием системы. После получения выходных данных диспетчер производит сопоставление с вычисленным набором ответов. Если он не обнаруживает соответствия (или если система не сгенерировала выходных данных), генерируется сообщение о сбое. Диспетчер второго типа ожидает получения выходных данных и проверяет их корректность по отношению ко всем полученным входным данным.

В простейшем случае диспетчер может отслеживать только некорректное завершение работы программы и осуществлять ее перезапуск, генерируя при этом соответствующее уведомление.

Диспетчер может работать в двух режимах: реального времени и с задержкой. Диспетчер реального времени обрабатывает входные и выходные данные, как только они получены, то есть синхронно с целевой системой. Во втором режиме диспетчер откладывает обработку этих событий на некоторый промежуток времени, добавляя их в очередь.

Основной сложностью при реализации диспетчера является неопределенность, которая зачастую присутствует в спецификации. Разработчики спецификаций могут предусмотреть несколько возможных вариантов работы системы в зависимости от прошлого и будущего состояний системы. Такие альтернативы в поведении могут быть как явными, так и неявными. Соответственно, диспетчер не должен делать никаких предположений о том, как эти неопределенности разрешаются в контролируемой системе.

Существует множество методов, во многом повторяющих принцип диспетчера [5, 6]. Идея контроля за выполнением программы и проверки входных и выходных данных на соответствие спецификации выглядит естественно, но, несмотря на это, не получила достаточного практического распространения.

Заключение

В статье было рассмотрено несколько методов обнаружения сбоев в программном обеспечении.

Метод зондирования позволяет лишь обнаружить последствия сбоя, но не позволяет его достаточно точно локализовать. При этом накладные расходы оказываются минимальными по сравнению с другими методами.

Метод контроля времени выполнения обеспечивает хорошую точность обнаружения и локализацию отказов. Его недостатком является необходимость дополнительной подготовки приложения на уровне исходных кодов для включения в него специальных конструкций, представляющих формализованную спецификацию системы. Кроме того, даже при параллельном выполнении основного и контролирующего потоков программы накладные расходы оказываются достаточно высоки.

Исходя из этого данный метод труднопременим к высокопроизводительным системам, находящимся в непрерывной эксплуатации, но весьма неплох на этапах разработки и отладки программы.

Метод динамического дублирования решает очень узкий класс задач, при этом не самым эффективным способом. Тем не менее он прост для реализации и не требует минимума дополнительных действий для внедрения.

Метод диспетчера отличается высокой скоростью работы и полной независимостью от целевой системы. Это обеспечивается возможностью работы диспетчера с задержкой. Недостатком данного метода является потенциальная сложность диспетчера, который не может делать никаких предположений об особенностях реализации конкретного приложения. Кроме того, при использовании диспетчера сложно или вовсе невозможно выбрать достаточные критерии завершения целевой программы в случае отказа.

Таким образом, каждый из рассмотренных методов имеет свои достоинства и недостатки, которые могут не отвечать требованиям в отдельных случаях — ни одно из решений не может считаться универсальным. Наиболее универсальный подход — анализ графа потока выполнения программы, который более сложен в реализации по сравнению с рассмотренными методами и выходит за рамки настоящей статьи. Исследования в этой области являются наиболее перспективным направлением развития в сфере автоматизированного обнаружения сбоев в программном обеспечении.

СПИСОК ЛИТЕРАТУРЫ:

1. *Chillarege R.* Self-testing software probe system for failure detection and diagnosis // Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research (стр. 10), IBM Press, Toronto. 1994.
2. *Sankar S., Mandal M.* Concurrent runtime monitoring of formally specified programs // Technical Report, Stanford University, Stanford. 1990.
3. *Reis G. A., August D. I., Mukherjee S. S., Cohn R.* Software fault detection using dynamic instrumentation // Proceedings of the Fourth Annual Boston Area Architecture Workshop (стр. 91–98), Boston. 2006.
4. *Savor T., Seviora R. E.* An approach to automatic detection of software failures in real-time systems // Proceedings of IEEE Real-Time Technology and Applications Symposium (стр. 136–146), IEEE Computer Society, Montreal. 1997.
5. *Richardson D. J., Aha S. L., and O'Malley T. O.* Specification-based test oracles for reactive systems // Proceedings of the 14th International Conference on Software Engineering (стр. 105–118), ACM, New York. 1992.
6. *Diaz M., Juanole G., Courtiat J.-P.* Observer – a concept for formal on-line validation of distributed systems // IEEE Transactions on Software Engineering (№12, том 20, стр. 900–913), IEEE Computer Society. 1994.