Шубинский И.Б.

МЕТОДЫ ОБЕСПЕЧЕНИЯ ФУНКЦИОНАЛЬНОЙ НАДЕЖНОСТИ ПРОГРАММ

В статье рассматриваются основные аспекты построения надежного программного обеспечения. Обсуждаются методы и способы предупреждения ошибок на основе защитного и многоверсионного программирования, средства пассивного и активного обнаружения ошибок, принципы и методы исправления ошибок на основе динамической избыточности и рестарта. Большое внимание в статье уделяется решению задач построения программ устойчивых к ошибкам с помощью методов отступления, методов изоляции ошибок и построения избыточных алгоритмов не критичных к различного рода нарушениям информационного процесса.

***Ключевые слова**: программа, ошибка, надежность программ, предупреждение ошибок, обнаружение ошибок, исправление ошибок, устойчивость к ошибкам, защитное программирование, многоверсионное программирование, рестарт, динамическая избыточность, алгоритмическая избыточность, изоляция ошибок.*

1. Введение

Функциональная надежность информационных систем в значительной мере зависит от надежности программных средств [1]. Все принципы и методы обеспечения функциональной надежности программ, в соответствии с их целью, можно разделить на четыре группы: ***предупреждение ошибок, обнаружение ошибок, исправление ошибок*** и ***обеспечение устойчивости к ошибкам*** *(рис. 1)*. К первой группе относятся принципы и методы, позволяющие минимизировать или вообще исключить ошибки. Методы второй группы сосредоточивают внимание на функциях самого программного обеспечения, помогающих выявлять ошибки. К третьей группе относятся функции программного обеспечения, предназначенные для исправления ошибок или их последствий. Устойчивость к ошибкам (четвертая группа) – это мера способности системы программного обеспечения продолжать функционирование при наличии ошибок.

2. Предупреждение ошибок

К этой группе относятся принципы и методы, цель которых – не допустить появления ошибок в готовой программе. Должно быть очевидно, что предупреждение ошибок – оптимальный путь к достижению надежности программного обеспечения. Лучший способ обеспечить надежность

Рис. 1. Основные аспекты построения архитектуры надежного программного обеспечения

– прежде всего не допустить возникновения ошибок. С этой целью широко применяют способы так называемого ***защитного (безопасного) программирования***, направленного на уменьшение вероятности ошибок в программах, а также ***многоверсионное программирование.***

Под ***защитным программированием*** понимают ограничение неправильного использования программных объектов. Другими словами, выдвигается требование проектировать и программировать таким образом, чтобы не только гарантировать ожидаемое использование программы в строгом соответствии со спецификациями, но и сделать невозможным ее неправильное использование. Например, при проектировании системы, в которой взаимодействуют много модулей, мы можем потребовать, чтобы какие-то взаимодействия между ними разрешались лишь в определенных ситуациях. Таким образом, вызов модулем А модуля В может быть разрешен всегда, в определенных ситуациях или же никогда, несмотря на то, что модули А и В находятся в таких отношениях, что вызов возможен.

Защитное программирование – технология предупреждения потенциальных ошибок путем проверки в каждом модуле множества допустимых условий. Во время программирования может быть использовано много технических приемов, чтобы проверить аномалии в управлении или в данных. Ниже перечислены некоторые из методов безопасного программирования:

- анализируются граничные значения переменных;
- проверяются размеры, тип и диапазон параметров процедуры ввода данных;
- параметры “только для доступ к ним чтения” и “считывание-запись” должны быть разделены, и должен быть проверен доступ к ним;
- символьные константы не должны быть доступны для записи;
- входные переменные и промежуточные (вспомогательные) переменные с физическим значением должны быть проверены на предмет достоверности;

- воздействие выходных переменных должно быть проверено, предпочтительно непосредственным наблюдением связанных с ними изменений состояния системы и т.д.

Простейший метод защитного программирования заключается в использовании специальных ловушек ошибок, рассчитанных на ошибки типа неправильного использования модулей. Разработчика программы не интересует, что будет делать пользователь после того, как получит сообщение о неправильном использовании модуля, но при этом он обязан спроектировать модуль так, чтобы ошибки пользователя не вызывали необратимых изменений в модуле. Таким образом, пользователь, пойманный на неправильном употреблении модуля, принимает корректирующие действия и снова вызывает модуль, не оставляя никаких следов ошибочных вызовов. Иначе говоря, речь идет о таком программировании, когда программный продукт очень трудно или невозможно использовать за пределами области действия его спецификации.

Многоверсионное программирование (N-версионное программирование). Цель: обнаружить и замаскировать остаточные ошибки проекта программного обеспечения в течение выполнения программ, чтобы предотвратить критически опасные отказы системы, и продолжать работу с высокой надежностью. В многоверсионном программировании данная спецификация программы реализуется по-разному N раз. Те же самые значения входных данных задаются N версиям, и результаты, произведенные N версиями, сравниваются. Если результат считается достоверным, то он передается компьютеру в качестве выходных данных. N версии могут выполняться параллельно на отдельных компьютерах, альтернативно все версии могут выполняться на одном компьютере. Выходные результаты подвергаются внутреннему голосованию. Различные стратегии голосования могут быть использованы на N версиях в зависимости от требований применения. Для критически важных информационных систем необходимо полное согласие всех N версий. Для других информационных систем может быть использована стратегия голосования путем простого большинства. Для случаев, где не имеется коллективного согласия, могут быть использованы вероятностные подходы, чтобы максимизировать шанс выбора правильного значения, например, взяв среднее значение, временно заморозив выходные данные до возвращения согласия, и т.д.

При дуальном программировании (если разрабатываются две версии программы) в случае обнаружения расхождения в результатах, необходимо определить по дополнительным критериям, какой результат правильный и отбросить другой результат. При N-версионном программировании правильный результат определяется по мажоритарному признаку, т.е. выбирается тот результат, который наблюдается в большинстве вариантов программы.

Рассмотренные способы резервирования требуют в 2 или N раз больше времени для вычислений и увеличение объема труда программистов во столько же раз. В связи с этим представляет интерес модифицированное дуальное программирование, при котором наряду с достаточно точной, но сложной основной программой используется менее точная, но простая резервная программа. Если при одинаковых исходных данных результаты работы программ отличаются на величину большую, чем допустимая погрешность, делается предположение, что отказала основная программа, как менее надежная, и в качестве правильного результата принимается результат работы резервной программы.

Гарантировать отсутствие ошибок, однако, невозможно никогда. Другие три группы методов опираются на предположение, что ошибки все-таки будут.

3. Обнаружение ошибок

Если предполагать, что в программном обеспечении какие-то ошибки все же будут, то лучшая (после предупреждения ошибок) стратегия – включить средства обнаружения ошибок в само программное обеспечение.

Большинство методов направлено по возможности на незамедлительное обнаружение сбоев. Немедленное обнаружение имеет два преимущества: можно минимизировать влияние ошибки и последующие затруднения для человека, которому придется извлекать информацию о ней, находить ее и исправлять.

Меры по обнаружению ошибок можно разбить на две подгруппы: *пассивные* попытки обнаружить симптомы ошибки в процессе «обычной» работы программного обеспечения и *активные* попытки программной системы периодически обследовать свое состояние в поисках признаков ошибок.

Пассивное обнаружение. Меры по обнаружению ошибок могут быть приняты на нескольких структурных уровнях программной системы. Здесь мы будем рассматривать уровень подсистем, или компонентов, т.е. нас будут интересовать меры по обнаружению симптомов ошибок, предпринимаемые при переходе от одного компонента к другому, а также внутри компонента. Все это, конечно, применимо также к отдельным модулям внутри компонента.

Разрабатывая эти меры, мы будем опираться на следующее.

1. *Взаимное недоверие.* Каждый из компонентов должен предполагать, что все другие содержат ошибки. Когда он получает какие-нибудь данные от другого компонента или из источника вне системы, он должен предполагать, что данные могут быть неправильными, и пытаться найти в них ошибки.

2. Немедленное обнаружение. Ошибки необходимо обнаружить как можно раньше. Это не только ограничивает наносимый ими ущерб, но и значительно упрощает задачу отладки.

3. *Избыточность.* Все средства обнаружения ошибок основаны на некоторой форме избыточности (явной или неявной).

Когда разрабатываются меры по обнаружению ошибок, важно принять согласованную стратегию для всей системы. Действия, предпринимаемые после обнаружения ошибки в программном обеспечении, должны быть единообразными для всех компонентов системы. Это ставит вопрос о том, какие именно действия следует предпринять, когда ошибка обнаружена. Наилучшее решение – немедленно завершить выполнение программы или (в случае операционной системы) перевести центральный процессор в состояние ожидания. С точки зрения предоставления человеку, отлаживающему программу, например системному программисту, самых благоприятных условий для диагностики ошибок немедленное завершение представляется наилучшей стратегией. Конечно, во многих системах подобная стратегия бывает нецелесообразной (например, может оказаться, что приостанавливать работу системы нельзя). В таком случае используется метод *регистрации ошибок*. Описание симптомов ошибки и «моментальный снимок» состояния системы сохраняются во внешнем файле, после чего система может продолжать работу. Этот файл позднее будет изучен обслуживающим персоналом.

Всегда, когда это возможно, лучше приостановить выполнение программы, чем регистрировать ошибки (либо обеспечить как дополнительную возможность работу системы в любом из этих режимов). Различие между этими методами проиллюстрируем на способах выявления причин возникающего иногда скрежета вашего автомобиля. Если автомеханик находится на заднем сиденье, то он может обследовать состояние машины в тот момент, когда скрежет возникает. Если вы выбираете метод регистрации ошибок, задача диагностики станет сложнее.

Активное обнаружение ошибок. Не все ошибки можно выявить пассивными методами, поскольку эти методы обнаруживают ошибку лишь тогда, когда ее симптомы подвергаются соответствующей проверке. Можно делать и дополнительные проверки, если спроектировать специальные программные средства для активного поиска признаков ошибок в системе. Такие средства называются *средствами активного обнаружения ошибок.*

Активные средства обнаружения ошибок обычно объединяются в *диагностический монитор:* параллельный процесс, который периодически анализирует состояние системы, с целью обнару-

жить ошибку. Большие программные системы, управляющие ресурсами, часто содержат ошибки, приводящие к потере ресурсов на длительное время. Например, управление памятью операционной системы сдает блоки памяти «в аренду» программам пользователей и другим частям операционной системы. Ошибка в этих самых «других частях» системы может иногда вести к неправильной работе блока управления памятью, занимающегося возвратом сданной ранее в аренду памяти, что вызывает медленное вырождение системы.

Диагностический монитор можно реализовать как периодически выполняемую задачу (например, она планируется на каждый час) либо как задачу с низким приоритетом, которая планируется для выполнения в то время, когда система переходит в состояние ожидания. Как и прежде, выполняемые монитором конкретные проверки зависят от специфики системы, но некоторые идеи будут понятны из примеров. Монитор может обследовать основную память, чтобы обнаружить блоки памяти, не выделенные ни одной из выполняемых задач и не включенные в системный список свободной памяти. Он может проверять также необычные ситуации: например, процесс не планировался для выполнения в течение некоторого разумного интервала времени. Монитор может осуществлять поиск «затерявшихся» внутри системы сообщений или операций ввода-вывода, которые необычно долгое время остаются незавершенными, участков памяти на диске, которые не помечены как выделенные и не включены в список свободной памяти, а также различного рода странностей в файлах данных.

Иногда желательно, чтобы в чрезвычайных обстоятельствах монитор выполнял диагностические тесты системы. Он может вызывать определенные системные функции, сравнивая их результат с заранее определенным и проверяя, насколько разумно время выполнения. Монитор может также периодически предъявлять системе «пустые» или «легкие» задания, чтобы убедиться, что система функционирует хотя бы самым примитивным образом.

4. Исправление ошибок

Следующий шаг – методы исправления ошибок. После того как ошибка обнаружена, либо она сама, либо ее последствия должны быть исправлены программным обеспечением. Исправление ошибок самой системой – плодотворный метод проектирования надежных систем аппаратного обеспечения. Некоторые устройства способны обнаружить неисправные компоненты и перейти к использованию идентичных запасных. Аналогичные методы неприменимы к программному обеспечению вследствие глубоких внутренних различий между сбоями аппаратуры и ошибками в программах. Если некоторый программный модуль содержит ошибку, идентичные «запасные» модули также будут содержать ту же ошибку.

Другой подход к исправлению связан с попытками восстановить разрушения, вызванные ошибками, например искажения записей в базе данных или управляющих таблицах системы. Польза от методов борьбы с искажениями ограничена, поскольку предполагается, что разработчик заранее предугадает несколько возможных типов искажений и предусмотрит программно реализуемые функции для их устранения. Это похоже на парадокс, поскольку, если знать заранее какие ошибки возникнут, можно было бы принять дополнительные меры по их предупреждению. Если методы ликвидации последствий сбоев не могут быть обобщены для работы со многими типами искажений, лучше всего направлять силы и средства на предупреждение ошибок. Вместо того, чтобы, разрабатывая операционную систему, оснащать ее средствами обнаружения и восстановления цепочки искаженных таблиц или управляющих блоков, следовало бы лучше спроектировать систему так, чтобы только один модуль имел доступ к этой цепочке, а затем настойчиво пытаться убедиться в правильности этого модуля.

5. Устойчивость к ошибкам

Основное допущение программирования, ***устойчивого к программным ошибкам*,** заключается в том, что как бы хорошо ни была спроектирована и реализована программа, в ней обязательно будет содержаться несколько остаточных ошибок. А раз так, то модули программы, которые могут дать сбой, должны иметь "резервный запас". С этой целью модуль проектируется в виде *блоков восстановления*. Каждый блок восстановления содержит пропускной тест и один или несколько вариантов реализации. Основной вариант инициируется при вызове блока восстановления, и когда его выполнение завершается, происходит проверка значения пропускного теста. Если он дает «истину», то считается, что выполнение блока восстановления успешно завершено. Если же тест дает «ложь», то инициируется другой вариант, за которым следует определение значения пропускного теста и т.д., и так до успешного выполнения блока восстановления. Если же ни один вариант не прошел пропускного теста, то блок восстановления рассматривается как ошибочный и начинается исполнение другого варианта вызываемого модуля. Применяется также и другой технический прием: написаны, часто независимо, несколько различных сегментов программы, каждый из которых предназначен для выполнения одной функции. Программа строится из этих сегментов. Первый сегмент, называемый первичным, выполняется первым. За ним следует приемочное испытание результата вычислений первого сегмента. Если испытание прошло успешно, тогда результат принимается и передается к последующим частям системы. Если испытание было неудачным, любые побочные эффекты первого сегмента сбрасываются, и выполняется второй сегмент, называемый первый альтернативный. За ним также следует приемочное испытание, результаты которого рассматриваются как в первом случае. Если необходимо, могут быть реализованы другие альтернативные приемы.

Приведенный способ парирования остаточных ошибок в программе путем проектирования модуля в виде блока восстановления требует порой неоправданных усилий, связанных с проектированием нескольких блоков восстановления, каждый из которых содержит пропускной тест и один или несколько вариантов реализации. В целях практического обеспечения функционирования программной системы при наличии в ней ошибок разработана группа методов, которая разбивается на четыре подгруппы: *динамическая избыточность, методы отступления, методы изоляции ошибок и построение алгоритмов нечувствительных (или не критичных) к различного рода нарушениям информационного процесса (использование алгоритмической избыточности).*

1. Истоки концепции ***динамической избыточности*** лежат в проектировании аппаратного обеспечения. Один из подходов к динамической избыточности – *мажоритарное резервирование* (метод голосования). Данные обрабатываются независимо несколькими идентичными устройствами, и результаты сравниваются. Если большинство устройств выработало одинаковый результат, этот результат и считается правильным. И опять, вследствие особой природы ошибок в программном обеспечении ошибка, имеющаяся в копии программного модуля, будет также присутствовать во всех других его копиях, поэтому идея голосования здесь, видимо, неприемлема. Предлагаемый иногда подход к решению этой проблемы состоит в том, чтобы иметь несколько неидентичных копий модуля. Это значит, что все копии выполняют одну и ту же функцию, но либо реализуют различные алгоритмы, либо созданы разными разработчиками. Этот подход бесперспективен по следующим причинам. Часто трудно получить существенно разные версии модуля, выполняющие одинаковые функции. Кроме того, возникает необходимость в дополнительном программном обеспечении для организации выполнения этих версий параллельно или последовательно и сравнения результатов. Это дополнительное программное обеспечение повышает уровень сложности системы, что, конечно, противоречит основной идее предупреждения ошибок – стремиться в первую очередь минимизировать сложность.

Второй подход к динамической избыточности – выполнять эти запасные копии только тогда, когда результаты, полученные с помощью основной копии, признаны неправильными. Если это происходит, система автоматически вызывает запасную копию. Если и ее результаты неправильны, вызывается другая запасная копия и т.д.

2. Вторая подгруппа методов обеспечения устойчивости к ошибкам называется ***методами отступления*** или сокращенного обслуживания. Эти методы приемлемы обычно лишь тогда, когда для системы программного обеспечения существенно важно корректно закончить работу. Например, если ошибка оказывается в системе, управляющей технологическими процессами, и в результате эта система выходит из строя, то может быть загружен и выполнен особый фрагмент программы, призванный подстраховать систему и обеспечить безаварийное завершение всех управляемых системой процессов. Аналогичные средства часто необходимы в операционных системах. Если операционная система обнаруживает, что вот-вот выйдет из строя, она может загрузить аварийный фрагмент, ответственный за оповещение пользователей у терминалов о предстоящем сбое и за сохранение всех критических для системы данных.

3. Третья подгруппа – ***методы изоляции ошибок***. Основная их идея – не дать последствиям ошибки выйти за пределы как можно меньшей части системы программного обеспечения, так, чтобы, если ошибка возникнет, не вся система оказалась неработоспособной; отключаются лишь отдельные функции в системе либо некоторые ее пользователи. Например, во многих операционных системах изолируются ошибки отдельных пользователей, так что сбой влияет лишь на некоторое подмножество пользователей, а система в целом продолжает функционировать. В телефонных переключательных системах для восстановления после ошибки, чтобы не рисковать выходом из строя всей системы, просто разрывают телефонную связь. Другие методы изоляции ошибок связаны с защитой каждой из программ в системе от ошибок других программ. Ошибка в прикладной программе, выполняемой под управлением операционной системы, должна оказывать влияние только на эту программу. Она не должна сказываться на операционной системе или других программах, функционирующих в этой системе.

В информационной системе изоляция программ является ключевым фактором, гарантирующим, что ошибки в программе одного пользователя не приведут к ошибкам в программах других пользователей или к полному выводу системы из строя. Основные правила изоляции ошибок в программах состоят в следующем:

1) Прикладная программа не должна иметь возможности непосредственно ссылаться на другую прикладную программу или данные в другой программе и изменять их.

2) Прикладная программа не должна иметь возможности непосредственно ссылаться на программы или данные операционной системы и изменять их. Связь между двумя программами (или программой и операционной системой) может быть разрешена только при условии использования четко определенных сопряжений и только в случае, когда обе программы дают согласие на эту связь.

3) Прикладные программы и их данные должны быть защищены от операционной системы до такой степени, чтобы ошибки в операционной системе не могли привести к случайному изменению прикладных программ или их данных.

4) Операционная система должна защищать все прикладные программы и данные от случайного их изменения операторами системы или обслуживающим персоналом.

5) Прикладные программы не должны иметь возможности ни остановить систему, ни вынудить ее изменить другую прикладную программу или ее данные.

6) Когда прикладная программа обращается к операционной системе, должна проверяться допустимость всех параметров. Прикладная программа не должна иметь возможности изменить эти параметры между моментами проверки и реального их использования операционной системой.

7) Никакие системные данные, непосредственно доступные прикладным программам, не должны влиять на функционирование операционной системы. Ошибка в прикладной программе, вследствие которой содержимое этой памяти может быть случайно изменено, приводит, в конце концов, к сбою системы.

8) Прикладные программы не должны иметь возможности в обход операционной системы прямо использовать управляемые ею аппаратные ресурсы. Прикладные программы не должны прямо вызывать компоненты операционной системы, предназначенные для использования только ее подсистемами.

9) Компоненты операционной системы должны быть изолированы друг от друга так, чтобы ошибка в одной из них не привела к изменению других компонентов или их данных.

10) Если операционная система обнаруживает ошибку в себе самой, она должна попытаться ограничить влияние этой ошибки одной прикладной программой и в крайнем случае прекратить выполнение только этой программы.

11) Операционная система должна давать прикладным программам возможность по требованию исправлять обнаруженные в них ошибки, а не безоговорочно прекращать их выполнение.

Реализация многих из этих принципов влияет на архитектуру лежащего в основе системы аппаратного обеспечения. Хотя в формулировке многих из них употребляются слова «операционная система», они применимы к любой программе (будь то операционная система, монитор телеобработки или подсистема управления файлами), которая занята обслуживанием других программ.

4. Четвертая подгруппа – ***введение алгоритмической избыточности***.

6. Заключение

В статье рассмотрены эффективные методы и способы построения архитектуры функционально надежных программных средств. Использован европейский опыт разработки программ для связанных с безопасностью систем [2], а также рекомендации стандарта [3]. Обеспечение надежности программных средств не ограничивается этапами разработки их спецификации и качественной архитектуры. Для решения ключевой задачи построения надежной информационной системы необходимо спроектировать надежное программное обеспечение, реализовать его, интегрировать с аппаратными средствами, обеспечить надежность программных средств в процессе аттестации, эксплуатации и сопровождения. Эти этапы построения надежной программы являются предметом дальнейшего обсуждения.

Литература

1. **Шубинский И.Б.** Функциональная надежность информационных систем (Методы анализа). М.: ООО «Журнал «Надежность», 2012, 296с.

2. BS EN 50128:2011. Программное обеспечение для систем управления и защиты на железных дорогах.

3. ГОСТ Р/МЭК 61508–3–2012. Функциональная безопасность систем электрических, электронных, программируемых электронных, связанных с безопасностью. Требования к программному обеспечению.