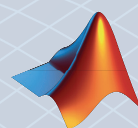
Всесторонний статический анализ с использованием продуктов Polyspace

Решение актуальных проблем при верификации встраиваемого ПО



MathWorks®

Введение

Верификация встраиваемого программного обеспечения является сложной задачей, бросающей дополнительные вызовы, учитывая сроки и возрастающую сложность встраиваемых систем. Большинство команд, разрабатывающих программное обеспечение, полагаются на ручные или не всесторонние методы выявления дефектов и ошибок при выполнении в программном обеспечении. Рассмотрение кода трудозатратно и часто не реализуется на практике для больших, сложных приложений. Динамическое тестирование (метод «белого ящика») требует от инженеров написания и выполнения огромного числа тестов. Когда тест не выполняется, требуется дополнительное время для поиска причины ошибки путем недетерминированного процесса отладки. Тестирование не является всесторонним и нельзя полагаться только на результаты тестирования при создании надежного программного обеспечения.

Инструменты статического анализа Polyspace предлагают иной подход. Осуществляется поиск дефектов во встраиваемом программном обеспечении и используются методы, основанные на доказательстве, такие, как абстрактная интерпретация, для доказательства того, что программное обеспечение является надежным. Polyspace Bug Finder™ идентифицирует ошибки при выполнении, проблемы с потоками данных и другие дефекты во встраиваемом программном обеспечении, написанном на С и С++. Используя статический анализ, Polyspace Bug Finder анализирует потоки данных, потоки управления и межпроцедурное взаимодействие в программе. Это позволяет выявлять и устранять программные дефекты на ранних стадиях процесса разработки.

Polyspace Code Prover™ обеспечивает гораздо более глубокий анализ, что позволяет доказывать отсутствие критических ошибок при выполнении в исходном С и С++ коде — таких, как переполнение, деление на ноль, доступ за пределы массива и других. Используются формальные методы — такие, как абстрактная интерпретация, для доказательства корректности кода. По результатам верификации Polyspace Code Prover отмечает каждую операцию цветом, указывающим, содержит ли эта операция ошибки при выполнении, является недостижимой или недоказанной.

Используя абстрактную интерпретацию, Polyspace Code Prover проводит автоматическую верификацию важных динамических свойств программ, включая доказательство присутствия или отсутствия ошибок при выполнении. Комбинация акцентированной точности рассмотрений кода и автоматизации позволяет осуществлять ранний поиск ошибок и доказывать робастность кода. Абстрактная интерпретация учитывает посредством верификации динамических свойств встраиваемых приложений все возможные варианты поведения системы и все возможные вариации входных данных, включая те, что приводят к ошибке в системе. Предоставленное доказательство корректности кода обеспечивает достаточную уверенность в надежности кода.

Используя инструменты поиска дефектов и доказательства кода, компании могут сократить расходы, при этом ускорив поставку надежных встраиваемых систем. Эта статья описывает, как Polyspace Bug Finder и Polyspace Code Prover, а также методика абстрактной интерпретации используются для преодоления ограничений традиционных методов верификации встраиваемого программного обеспечения.

Сложности при тестировании встраиваемого программного обеспечения

По мере удешевления процессорных ресурсов и памяти встраиваемое программное обеспечение, применяемое повсеместно, значительно усложняется. За последние несколько лет сильная потребность в завершенных многоцелевых программных приложениях привела к появлению больших, очень сложных встраиваемых систем. В некоторых отраслях количество встраиваемого кода удваивается каждые 18 месяцев.

Поскольку число и сложность этих приложений продолжает расти, то аспекты, связанные с разработкой систем повышенной надежности, критических к безопасности, требующих высокой надежности и робастности, начинают играть все более важную роль.

Давление со стороны рынка влияет на весь процесс разработки программного обеспечения. Разрабатываемое приложение может быть объектом постоянных изменений, включая модификацию требований, обновление спецификаций и изменения проекта.

Организации часто сталкиваются с конфликтующими бизнес-целями: поставлять встраиваемое программное обеспечение более высокого качества, при сокращении времени выхода на рынок. Эта проблема усугубляется возрастающей сложностью разрабатываемых приложений и частой нехваткой инженерных ресурсов.

Одно из решений заключается в повышении эффективности путем использования программных инструментов для проектирования, генерации кода и отладки кода. Эти инструменты позволяют командам разработчиков сделать больше за меньшее время. Однако инструменты для тестирования и отладки не успевают за возрастающим размером и сложностью встраиваемого программного обеспечения. В результате стоимость тестирования встраиваемой системы сегодня может составлять более половины всей стоимости разработки.

Ранняя верификация кода является эффективным способом снизить давление сроков сдачи и расходов на про-

ект. Ошибки, обнаруженные на поздних стадиях процесса разработки, сложнее отлаживать, поскольку их требуется отследить к источнику проблемы среди тысяч или миллионов строк кода, который обычно пишут не те, кто тестирует. Стоимость исправления проблем, обнаруженных на поздних стадиях тестирования, может быть от 10 до 20 раз выше[1] стоимости исправления той же ошибки во время кодирования.

Несмотря на то, что ранняя верификация кода предлагает очевидные преимущества, такой подход все еще является скорее исключением, чем правилом. Для многих команд это означает, что тестирование осуществляется ближе к срокам сдачи проекта, когда сроки поджимают сильнее всего. В таких условиях ошибки при выполнении, которые обычно считаются самыми сложными для нахождения, диагностики и исправления, могут быть пропущены.

Эти ошибки, вызванные арифметическими и другими аномалиями в коде, также известны как латентные, или скрытые, поскольку они не проявляются при нормальных режимах работы. Примеры таких ошибок включают в себя деление на ноль, доступ за пределы массива, некорректное разыменование указателя и чтение неинициализированных данных. Разработчики программного обеспечения обнаруживают, что 20–50% дефектов, обнаруженных в программе во время цикла поддержки, являются ошибками при выполнении.[2] Эти ошибки приводят к неопределенному поведению, некорректным вычислениям — таким, как целочисленное переполнение или останов процессора. Последствия всех этих ошибок оказывают непредсказуемое, иногда трагическое влияние на надежность системы.

Ограничения распространенных методов поиска ошибок при выполнении

Методы и инструменты, традиционно используемые для выявления и отладки ошибок при выполнении, полагаются на технологию, которой уже несколько десятков лет. Эти подходы делятся на две широкие категории: ручное рассмотрение кода и динамическое тестирование. Эти методики фокусируются на поиске некоторых ошибок, а не всех сразу. Они не способны провести гарантировать, что в программном обеспечении не осталось ошибок при выполнении.

Ручное рассмотрение кода может быть эффективным методом поиска ошибок при выполнении в относительно небольших приложениях размером 10000–30000 строк кода. Для систем большего размера ручное рассмотрение является трудозатратным процессом. Требуется, чтобы опытные инженеры рассмотрели участки исходного кода и сообщили об опасных или ошибочных конструкциях. Такие мероприятия являются сложными, не всесторонними, не повторяемыми и затратными. Доказательство отсутствия ошибок при выполнении — это сложная операция, которую нельзя осуществить посредством ручного рассмотрения кода.

Динамическое тестирование требует от инженеров написания и выполнения тестов. Для типичного встраиваемого приложения могут потребоваться многие тысячи тестовых векторов. После выполнения теста, инженер-тестировщик должен рассмотреть результаты и осуществить трассировку ошибок к причине их появления. Эта методика тестирования не претерпела существенных изменений за последние четыре десятилетия. Несмотря на то, что инженеры могут выполнять другие шаги, чтобы повысить вероятность обнаружения аномалии — такие, как добавление оснастки в код и покрытие кода тестами, динамическое тестирование основывается на методе проб и ошибок, и предоставляет только частичное покрытие всех возможных комбинаций значений, которые могут встретиться во время выполнения. Как и ручное рассмотрение кода, этот процесс является трудозатратным. Время, потраченное на создание тестов и ожидание разработки исполняемой системы, часто приводит к тому, что динамическое тестирование смещается в конец цикла разработки, когда ошибки наиболее дорого исправлять.

Сложности использования динамического тестирования для поиска ошибок при выполнении во встраиваемых приложениях

Стоимость тестирования растет экспоненциально размеру приложения. Поскольку число ошибок на заданное число строк является в среднем постоянной величиной — по консервативным оценками от 2 до 10 ошибок на 1000 строк кода — то шансы нахождения всех ошибок уменьшаются по мере увеличения общего числа строк в программном обеспечении. При увеличении размера исходного кода вдвое, усилия по тестированию должны быть в общем случае умножены на четыре для обеспечения того же уровня уверенности, что и раньше.

Динамическое тестирование ищет только симптомы ошибки, а не причину. Как результат, требуется потратить дополнительное время на отладку, чтобы воспроизвести каждую ошибку, а затем локализовать проблему. Трассировка причины аномалии в коде может быть исключительно долгим процессом, если ошибка была выяв-

---

[1] Basilli, V. and B. Boehm. "Software Defect Reduction Top 10 List." Computer 34 (1), January 2001.

[2] Sullivan, M. AND R. Chillarge. "Software Defects and Their Impact on System Availability." Proceedings of the 21th International Symposium on Fault-Tolerant Computing (FTCS-21), Montreal, 1991, 2-9. IEEE Press.

лена на поздних стадиях разработки. Трассировка аномалии, выявленной на стадии валидации, может занять 100 часов, в то время как ошибка, выявленная во время модульного тестирования, может быть локализована в течение одного часа или меньше.

Если некоторые мероприятия по тестированию можно автоматизировать, то отладку нельзя. Например, если каждые 1000 строк кода содержат от 2 до 10 ошибок, то приложение из 50000 строк будет содержать минимум 100 ошибок. Разработчику, тратящему в среднем 10 часов на отладку каждой ошибки, потребуется 1000 часов для отладки приложения.

Динамическое тестирование часто заставляет инженеров добавлять оснастку в код, чтобы проще отследить аномалии во время выполнения программы. Но добавление оснастки занимает время, увеличивает накладные расходы на выполнение, и может даже скрывать ошибки, такие, как нарушение памяти или конфликтующий доступ к общим данным. Методы, базирующиеся на добавлении оснастки в код, выявляют ошибки только в том случае, если выполняемые тесты приводят к аномалии.

Эффективность выявления ошибок при выполнении зависит от способности тестовых векторов проанализировать комбинации значений и условий, которые могут произойти во время выполнения. Тестовые вектора в общем случае покрывают только часть всех возможных комбинаций. Это оставляет большое число не протестированных комбинаций, включая те, что могут привести к ошибке при выполнении.

Поиск дефектов с использованием Polyspace Bug Finder на ранних стадиях разработки

Когда код разрабатывается, рекомендуется проверять его на дефекты на ранних стадиях процесса разработки. Разработчикам программного обеспечения требуется быстрый и эффективный метод идентификации дефектов в программном обеспечении. Polyspace Bug Finder является инструментом статического анализа кода, который используется для анализа частей кода или всего проекта встраиваемой системы. Polyspace Bug Finder использует быстрые методики статического анализа кода, включая формальные методы с низким уровнем ложных срабатываний, для точного выявления дефектов (численных, потоков данных, ошибок программирования и других) в исходном C или C++ коде.

Эти дефекты идентифицируются в исходном коде и приводится дополнительная информация, помогающая отследить причину дефекта и выявить его источник. Нарушения правил кодирования (MISRA C/C++ и JSF++) идентифицируются непосредственно в исходном коде, и приводится дополнительная информация о нарушенном правиле. Эта информация может использоваться для итеративной отладки и исправления кода на ранних стадиях процесса разработки. Polyspace Bug Finder поддерживает вызов из командной строки, предоставляет графический интерфейс пользователя и возможность использования с популярной средой разработки Eclipse™. Продукт может интегрироваться с системой сборки кода для автоматизации. Он также может быть интегрирован в окружения для автоматической генерации кода, такие, как Simulink® или TargetLink® или IBM® Rational® Rhapsody®.

Для каждого обнаруженного дефекта Polyspace Bug Finder предлагает детальную информацию о причине этого дефекта. Например, когда происходит целочисленное переполнение, Polyspace Bug Finder осуществляет трассировку всех строк кода, которые привели к этому переполнению. Разработчики программного обеспечения могут использовать эту информацию для того, чтобы определить, как лучше исправить этот дефект. Инженеры по качеству могут использовать эту информацию, чтобы классифицировать дефект и запланировать дальнейшие действия.

Использование Polyspace Code Prover для доказательства кода

Polyspace Code Prover использует проверенные и тщательные формальные методы абстрактной интерпретации. Эта техника заполняет пробел между традиционными методами статического анализа и динамического тестирования, осуществляя верификацию динамических свойств системы. Не выполняя саму программу, абстрактная интерпретация исследует все возможные последовательности выполнения, определяя, как и при каких условиях программа может провалиться.

Преимущества абстрактной интерпретации для доказательства кода

Абстрактная интерпретация является эффективным способом обеспечения поставки надежных встраиваемых систем. Эта техника, основанная на доказательстве, предлагает четыре преимущества: обеспечение надежности кода, увеличение эффективности, сокращение накладных расходов и упрощенную отладку.

Обеспечение надежности кода

При помощи тщательного рассмотрения кода абстрактная интерпретация не только позволяет обнаруживать ошибки при выполнении, но и также доказывать корректность кода. Это особенно важно для приложений, критических к безопасности, в которых ошибки в системе могут привести к катастрофе. Традиционные инструменты отладки настроены на выявление ошибок, но не осуществляют верификацию робастности остального кода. Аб-

Как работает абстрактная интерпретация

Абстрактная интерпретация основана на широком наборе математических теорем, которые определяют правила анализа сложных динамических систем — таких, как программное обеспечение. Вместо того, чтобы анализировать перечислимое множество состояний программы, абстрактная интерпретация представляет эти состояния в более общей форме и предоставляет правила для манипулирования этими состояниями. Абстрактная интерпретация не только создает математическую абстракцию, но также позволяет интерпретировать эту абстракцию.

Для создания математической абстракции состояний программы, абстрактная интерпретация тщательно анализирует все переменные в коде. Существенные вычислительные мощности, требуемые для такого анализа, не были раньше легкодоступны. Абстрактная интерпретация, будучи объединённой с сегодняшними сложными алгоритмами и возросшими вычислительными мощностями, является практическим решением сложных задач по тестированию.

Будучи примененной к выявлению ошибок при выполнении, абстрактная интерпретация осуществляет всестороннюю верификацию всех опасных операций и автоматически диагностирует каждую операцию как доказанную, проваленную, недостижимую или недоказанную. Инженеры могут использовать абстрактную интерпретацию для получения результатов до компиляции, самой ранней стадии тестирования. См. приложение B для дополнительной информации по применению абстрактной интерпретации и приложение C для примеров кода.

страктная интерпретация позволяет идентифицировать код, который никогда не приведет к ошибке в программе, таким образом, устраняя любую неопределенность в надежности программного обеспечения.

Увеличение эффективности

При помощи верификации динамики приложения абстрактная интерпретация позволяет разработчикам и тестировщикам идентифицировать секции кода в программе, которые не содержат ошибок при выполнении, и те, которые приведут к нарушению надежности. Поскольку ошибки могут быть найдены до компиляции кода, абстрактная интерпретация помогает командам существенно сэкономить время и расходы путем поиска ошибок тогда, когда их проще всего исправить.

Сокращение накладных расходов

Абстрактная интерпретация не требует выполнения программы, поэтому тщательные результаты производятся без создания и выполнения тестов. Нет необходимости осуществлять оснастку кода, а затем эту оснастку убирать перед поставкой программы. Также абстрактная интерпретация может быть реализована в текущих проектах без изменения существующих процессов разработки.

Упрощенная отладка

Абстрактная интерпретация упрощает отладку, поскольку напрямую идентифицируется источник каждой ошибки, а не только симптом этой ошибки. Устраняется необходимость тратить время на трассировку аномалий к их источнику, а также на попытки воспроизвести редкие дефекты. Абстрактная интерпретация является повторяемой и тщательной. Каждая операция в коде автоматически идентифицируется, анализируется и проверяется относительно всех возможных комбинаций входных значений.

Заключение

Polyspace Code Prover использует абстрактную интерпретацию для статического анализа при верификации кода. Совместно с Polyspace Bug Finder эти продукты предлагают сквозное решение для статического анализа на ранних стадиях разработки, которое охватывает поиск дефектов, проверку на стандарты кодирования и доказательство. Такой подход обеспечивает качество встраиваемого программного обеспечения, которое должно функционировать на высочайших уровнях надежности и безопасности.

Приложение А: Аналогия из физического мира

Инженер, которому требуется прогнозировать траекторию снаряда в полете, может пойти тремя путями:

1. Создать исчерпывающий список различных препятствий на пути снаряда, изучить их свойства и определить, как столкновение с каждым препятствием повлияет на траекторию. Этот подход не является практичным из-за большого числа и разнообразия препятствий, встречающихся по время полета. Даже если было бы возможно заранее знать обо всех условиях, которые могут возникнуть в любой момент времени — таких, как скорость ветра и облачность, они могут меняться с каждым новым полетом. Это означает, что требовалось бы проводить новый тщательный анализ перед каждым запуском.

2. Запустить много снарядов для получения эмпирических законов движения и связанных с ними погрешностей. Эти закономерности могут использоваться для оценки траекторий в рамках определенного доверительного интервала. Такой подход является дорогим и затратным по времени. Кроме того, всестороннее тестирование снаряда при всех возможных комбинациях условий невозможно.

3. Использовать законы физики и известные значения (сила гравитации, коэффициент сопротивления воздуха, начальная скорость и т.п.) для преобразования этой задачи в набор уравнений, которые можно решить, используя математические правила, либо формальные, либо численные. Этот подход производит решения для широкого набора условий, которые являются параметрами математической модели, позволяя инженеру прогнозировать поведение снаряда при разных условиях.

Абстрактная интерпретация аналогична математическому моделированию. На основании исходного кода рассчитываются динамические свойства данных — т.е. математические взаимосвязи переменных — и применяются к верификации определенных динамических свойств.

Приложение В: Применение абстрактной интерпретации

Для лучшего понимания того, как работает абстрактная интерпретация, представим программу P, в которой используются две переменные, X и Y. Осуществляется следующая операция:

X=X/(X–Y)

Для проверки этой программы на ошибки при выполнении, мы учитываем все возможные причины ошибки в этой операции:

- X и Y могут быть неинициализированы;
- При вычитании X–Y может произойти переполнение или исчезновение порядка;
- X и Y могут быть равны и привести к делению на ноль;
- При делении X/(X–Y) может произойти переполнение или исчезновение порядка.

Хотя все эти условия могут привести к ошибке при выполнении, в следующих шагах мы сфокусируемся на возможности деления на ноль.

Мы можем представить все возможные значения X и Y в программе P на диаграмме. Красная линия на рисунке 1 представляет собой набор значений (X, Y), которые приведут к делению на ноль.

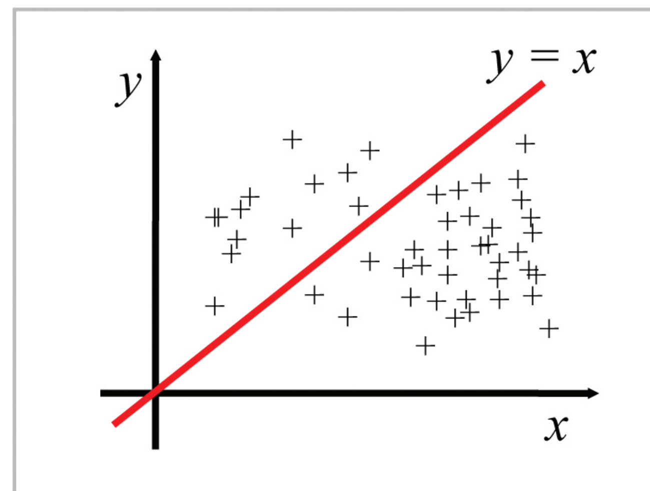


Рисунок 1. Все возможные значения X и Y в программе P.

Очевидный способ проверки деления на ноль заключается в оценке каждого состояния и определении того, попадает ли оно на красную линию. Это подход, который применяется традиционным тестированием типа «белый ящик», но он обладает фундаментальными ограничениями. Во-первых, число возможных состояний в реальном приложении обычно очень велико, поскольку используется много переменных. Потребовались бы годы для оценки всех возможных состояний, что делает всестороннюю проверку невозможной.

В отличие от метода перебора всех возможных состояний, абстрактная интерпретация устанавливает обобщенные правила для манипуляции целым набором состояний. Создается абстракция программы, используемая для доказательства свойств.

Один из примеров такой абстракции, который называется анализом типов, приведен на рисунке 2. Этот тип абстракции используется компиляторами, линковщиками и базовыми статическими анализаторами, применяющими правила пересечения типов. При анализе типов, мы проецируем набор точек на оси X и Y, получая минимальные и максимальные значения для X и Y, и рисуем соответствующий прямоугольник. Поскольку прямоугольник включает все возможные значения X и Y, то, если свойство доказано для прямоугольника, оно будет валидно для всей программы. В этом случае, нас интересует пересечение красной линии и прямоугольника. В случае если пересечение пустое, деление на ноль никогда не произойдет.

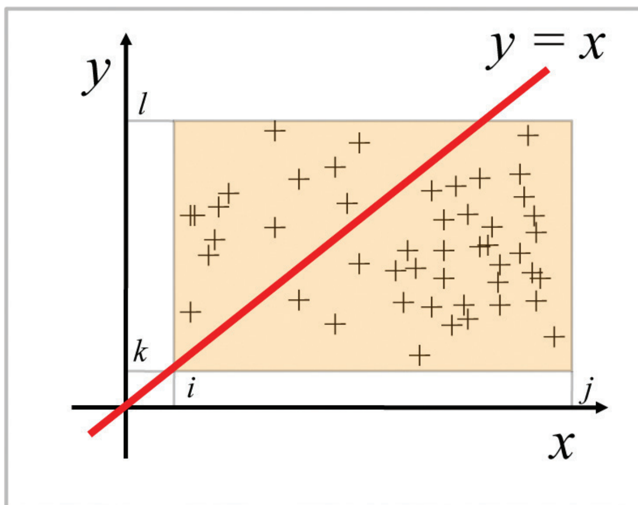


Рисунок 2. Анализ типов проецирует все значения X и Y на оси.

Основной недостаток анализа типов заключается в том, что прямоугольник содержит слишком много нереалистичных значений X и Y. Это приводит к плохим, неточным результатам и генерирует большое число предупреждений, которые обычно не рассматриваются, или инженер их вовсе отключает.

Вместо большого прямоугольника, абстрактная интерпретация устанавливает правила для построения более точных фигур, как на рисунке 3. Используются техники, основанные на целочисленных решетках, наборы полигонов (union of polyhedra) и базисы Грёбнера для представления взаимосвязей между данными (X и Y), которые принимают во внимание управляющие структуры (такие, как if-then-else, for-циклы, while-циклы и switch), межпроцедурные операции (вызовы функций) и анализ многозадачности.

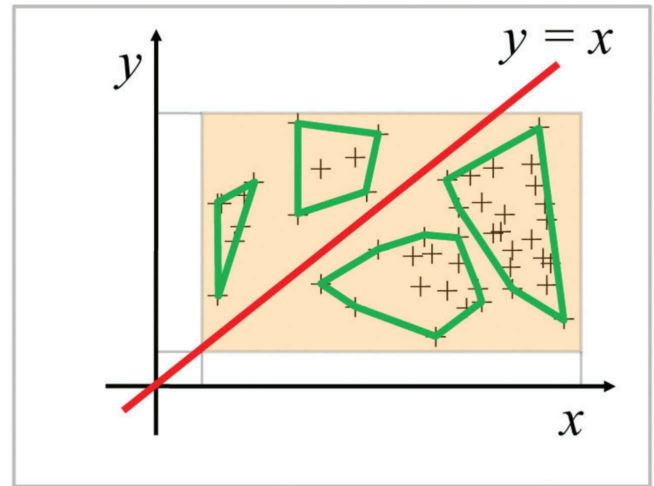


Рисунок 3. Абстрактная интерпретация устанавливает правила для построения более точных фигур, представляющих взаимосвязи между X и Y.

В отличие от компиляторов и других статических анализаторов, абстрактная интерпретация не полагается только на идею вычисления взаимосвязей между типами данных и постоянными. Вместо этого эти взаимосвязи выводятся из семантики каждой операции и операнда в программе, и используются в качестве руководства для инспекции кода.

С использованием абстрактной интерпретации следующие элементы программы интерпретируются новыми способами:

- Индекс в for-цикле больше не является целым числом, а представляет дискретную функцию, монотонно возрастающую от нижнего до верхнего предела.
- Параметр, передаваемый в функцию, больше не является переменной или константой, а представляет набор значений, которые могут использоваться для задания ограничений локальных данных, используемых в этой функции.
- Любые глобальные разделяемые данные могут меняться в любой момент времени в многозадачной программе, за исключением тех, что защищены механизмами, такими, как блокировка памяти или критических секций.
- Указатель является типом данных, который может создавать связи между явными данными и производить побочные эффекты и скрытые конкурирующие попытки доступа к разделяемым данным в многозадачных приложениях.
- Переменная обладает не только типом и диапазоном значений, но также набором уравнений, включая взаимосвязи, основанные на потоках управления, которые создали ее.
- В конечном итоге ошибки при выполнении — это уравнения, также называемые условиями корректности, которые могут решаться абстрактной интерпретацией с использованием уравнений, которые связывают все переменные вместе.

Приложение C: Примеры абстрактной интерпретации

Следующие примеры являются конструкциями кода, содержащими ошибки при выполнении. Polyspace Code Prover способен выявлять эти ошибки при выполнении при помощи абстрактной интерпретации.

Анализ управляющей структуры: Разыменование указателя за пределы массива после for-цикла

```
10: int ar[100];
11: int *p = ar;
12: int i;
13: for (i = 0; i < 100; i++; p++)
14:     { *p = 0;}
15: *p = 5;
```

В этом примере p является указателем, который может быть абстрагирован как дискретная функция, возрастающая с шагом 1 с начала массива ar. После выхода из for-цикла, когда i становится равным 100, указатель p также увеличивается до 100. Это приводит к тому, что на строке 15 указатель p указывает за пределы массива, поскольку индекс массива находится в диапазоне от 0 до 99. Абстрактная интерпретация докажет, что этот участок кода является надежным, и определит строку 15 как источник ошибки при выполнении.

Анализ управляющей структуры: Разыменование указателя за пределы массива в двух вложенных for-циклах

```
20: int ar[10];
21: int i,j;
22: for (i = 0; i < 10; i++)
23: {
24:    for (j = 0; j < 10; j++)
25:    {
26:           ar[i - j] = i + j;
27:    }
28: }
```

Как i, так и j являются переменными, монотонно возрастающими от 0 до 9 с шагом 1.

Операция i–j используется как индекс для массива ar и однажды вернет отрицательное значение. Абстрактная интерпретация этого кода докажет, что код является надежным и определит доступ за пределы массива на строке 26.

Ошибки при выполнении в этих примерах часто приводят к повреждению данных, находящихся непосредственно рядом с массивом ar. В зависимости от того, как и когда эти поврежденные данные используются в другом месте программы, отладка такого рода ошибки без абстрактной интерпретации может потребовать существенных усилий.

Межпроцедурный анализ: Деление на ноль

```
30: void foo (int* depth)
31: {
32:   float advance;
33:   *depth = *depth + 1;
34:   advance = 1.0/(float)(*depth);
35:   if (*depth < 50)
36:          foo (depth);
37: }
38:
39: void bug_in _recursive ()
40: {
41:   int x;
42:   if (random_int())
43: {
44:   x = -4;
45:   foo ( &x );
46: }
47:    else
48:   {
49:          x = 10;
50:          foo ( &x );
51:   }
52: }
```

В данной функции depth это целое число, которое сначала увеличивается на 1. Затем оно используется как знаменатель для определения значения advance и поэтому рекурсивно передается в функцию foo. Проверка того, приведет ли операция деления на строке 34 к делению на ноль, требует межпроцедурного анализа для выявления значений, которые будут передаваться в функцию foo (см. bug_in_recursive, строки 45 и 50), поскольку этим определяются значения depth.

В этом примере функция foo может вызываться в двух различных случаях из функции bug_in_recursive. Когда утверждение if на строке 42 является ложным, foo вызывается со значением 10 (строка 50).

Таким образом, *depth становится дискретной монотонно возрастающей функцией, изменяющейся от 11 до 49 с шагом 1. Уравнение на строке 34 не приведет к делению на ноль.

Однако если выражение if на строке 42 является истинным, тогда foo вызывается со значением -4. Таким образом, *depth становится дискретной монотонно возрастающей функцией, меняющейся от -3 до 49 с шагом 1. Рано или поздно *depth станет равным 0, тем самым выражение на строке 34 приведет к делению на ноль.

Простая проверка синтаксиса не обнаружит такую ошибку, также, как и не все тестовые вектора. Абстрактная интерпретация докажет, что весь код является надежным, за исключением строки 45. Это иллюстрирует уникальную способность абстрактной интерпретации осуществлять межпроцедурный анализ и различать проблемные вызовы функций от допустимых вызовов. Если проблему не исправить, она приведет к останову процессора. Ошибки такого рода могут также потребовать значительного времени для отладки в связи с используемыми рекурсивными конструкциями.

Анализ многозадачного кода: конкурирующий доступ к разделяемым данным

Абстрактная интерпретация анализирует потоки данных и управления и позволяет проверять многозадачные приложения. Ключевая задача с такими приложениями заключается в обеспечении того, что разделяемые данные и критические ресурсы не подвергаются неконтролируемому конкурирующему доступу. Неявный доступ к данным (data aliasing) и перемежение задач усложняют поиск такого типа конкурирующего доступа.

При неявном доступе к данным указатели используются для доступа к общему региону памяти. Такой подход может привести к созданию скрытых или неявных зависимостей между переменными, так что одна переменная может быть неожиданно изменена через указатель во время выполнения программы, вызывая случайные аномалии. Анализ такой проблемы требует очень эффективных алгоритмов анализа указателей. С использованием абстрактной интерпретации это реализуется путем создания списка разделяемых данных и списка доступов для чтения и записи функциями и задачами, а также соответствующего граф конкурирующего доступа.

Перемежение задач усложняет отладку многозадачных приложений, поскольку дефекты очень сложно воспроизвести, когда они зависят от последовательности задач, выполняемых в определенном порядке в реальном времени. Абстрактная интерпретация принимает во внимание все возможные перемежения задач для полного анализа потоков управления. Создание таких графов конкурирующего доступа вручную было исключительно сложной задачей.

Дополнительная информация и контакты

Информация о продуктах
matlab.ru/products

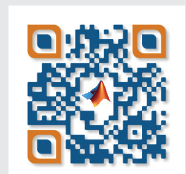
Пробная версия
matlab.ru/trial

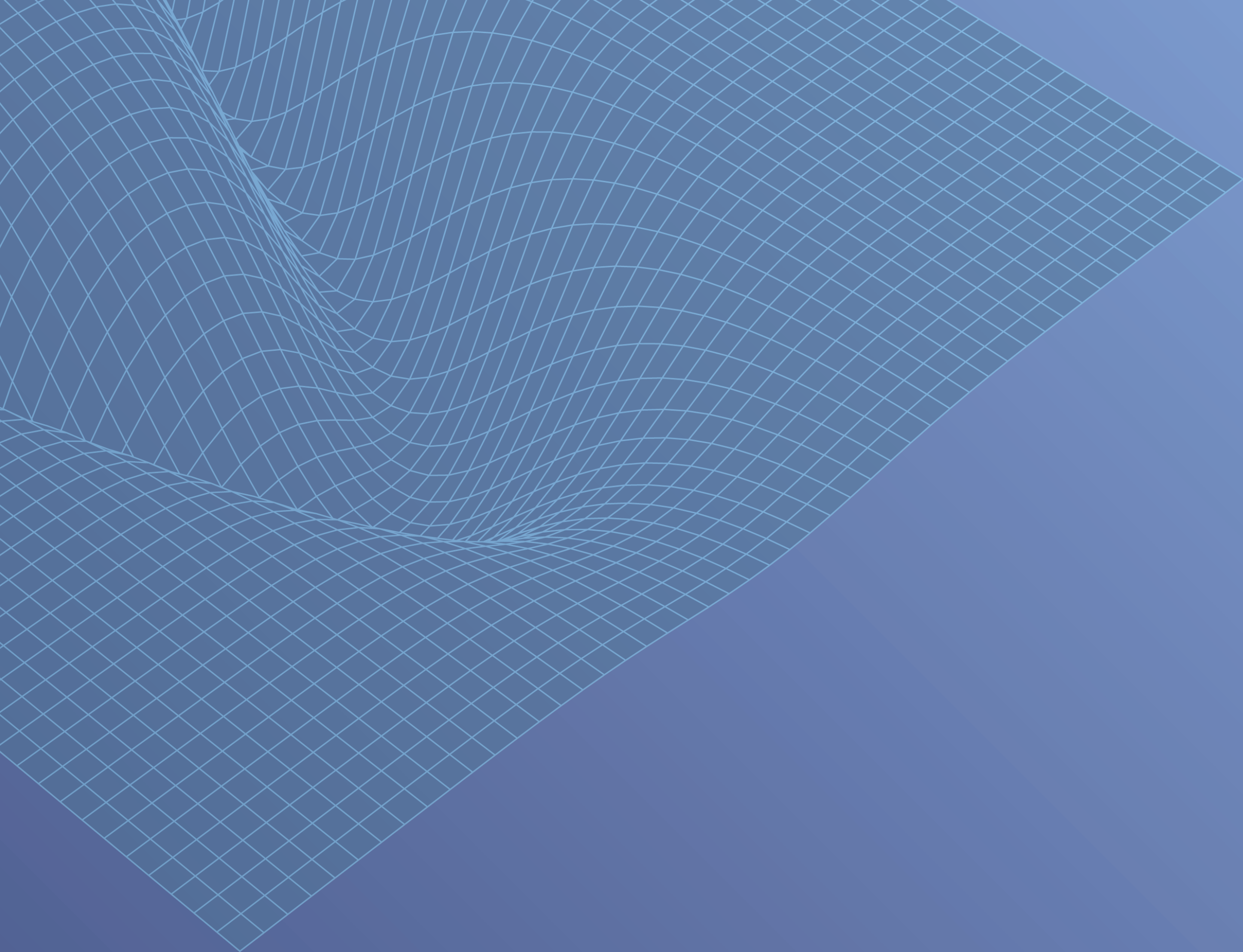
Запрос цены
matlab.ru/price

Техническая поддержка
matlab.ru/support

Тренинги
matlab.ru/training

Контакты
matlab.ru
E-mail: **matlab@sl-matlab.ru**
Тел.: +7 (495) 232-00-23, доб. 0609
Адрес: 115114 Москва,
Дербеневская наб., д. 7, стр. 8





Департамент MathWorks компании Softline

115114 г. Москва, Дербеневская наб., д. 7, стр. 8
Деловой квартал «Новоспасский Двор»
Email: matlab@softline.ru
Тел.: +7 (495) 232-00-23 доб. 0609
matlab.ru

softline®