Статья "Верификация кода и обнаружение ошибок исполнения путем абстрактной интерпретации"

Решение современных проблем тестирования встроенного программного обеспечения

Тестирование встроенных систем является трудной задачей, сложность которой еще более возрастает из-за давления сроков и растущей сложности встроенного программного обеспечения. В настоящее время существует три основных варианта обнаружения ошибок при выполнении встроенных приложений: анализ кода, применение статических анализаторов и динамическое тестирование методом проб и ошибок. Анализ кода является трудоемким, а для больших и сложных приложений нередко оказывается непрактичным. Статические анализаторы выявляют относительно небольшое число проблем и, что наиболее важно, оставляют непроверенной большую часть исходного кода. Динамическое тестирование или тестирование по «методу белого ящика» требует от инженеров написания и выполнения множества тестов. Если результаты тестирования оказываются отрицательными, потребуется дополнительное время на поиск причины проблемы путем выполнения неопределенного процесса отладки.

Абстрактная интерпретация предполагает другой подход. Вместо простого обнаружения ошибок проводится автоматическая верификация важных динамических свойств программ, включая проверку на наличие или отсутствие ошибок при выполнении. При этом сочетается детально точный анализ кода и автоматизация, которая обеспечивает раннее обнаружение ошибок и подтверждение надежности кода. При верификации правильности динамических свойств встроенных приложений абстрактная интерпретация отслеживает все возможные поведения программного обеспечения и все возможные изменения входных данных, включая данные о возможных отказах программного обеспечения. Кроме того, проверяется правильность кода и подтверждается его надежность. Используя инструментальные средства тестирования, которые реализуют абстрактную интерпретацию, компании смогут уменьшить расходы и ускорить поставку надежных встроенных систем.

В настоящей статье описывается работа абстрактной интерпретации и возможность ее использования для преодоления ограничений обычных методов тестирования встроенного программного обеспечения.

Статический анализ	Верификация кода путем абстрактной интерпретации
■ Проверяется выполнение синтаксических и статических семантических правил	■ Проверяются динамические свойства программ
■ Обнаруживаются некоторые ошибки и формируются многочисленные предупреждения	■ Проверяется, какие разделы кода являются правильными, какие неправильными и какие недостижимыми.
■ Важные и неизвестные части кода остаются непроверенными	■ Тщательно анализируются все разделы кода программы, проверяется их надежность при всех возможных значениях данных

Проблемы, касающиеся тестирования встроенного программного обеспечения

В связи с уменьшением затрат на средства обработки и использование памяти, встроенные приложения проникли почти в каждый сектор промышленности от потребительской электроники и до приложений в области аэрокосмической отрасли, обороны, автомобилестроения, авиации, связи и медицинских приборов. В течение нескольких последних лет сильный спрос на полные многоцелевые приложения привел к появлению гораздо больших и более сложных встроенных систем. В некоторых отраслях количество встроенного программного обеспечения удваивается через каждые 18 месяцев. По мере роста количества и сложности этих приложений безопасность, предназначение, жизнеспособность или важность для бизнеса многих встроенных приложений по-прежнему требуют высокой надежности и живучести.

Давление рынка оказывает влияние на любое развитие программного обеспечения. Разрабатываемое приложение подвергается непрерывным изменениям, включая обновления спецификации и изменения проекта. Организации-разработчики нередко должны следовать конфликтующим целям бизнеса:

предоставлять высококачественное встроенное программное обеспечение, уменьшая при этом время выхода на рынок. Эти проблемы возрастают в связи со сложностью разрабатываемых приложений и частой нехваткой технических специалистов.

Одним из решений является повышение эффективности путем использования программных средств для создания кода, проектирования и включения инструментального кода. Эти средства дали возможность командам разработчиков встроенного программного обеспечения делать большее за меньшее время. Однако средства тестирования и отладки не справляются с ростом объема и сложности встроенного программного обеспечения. В результате этого в настоящее время стоимость тестирования встроенной системы может достигать 50% от общей стоимости разработки.

Верификация правильности кода на ранней стадии является эффективным способом сокращения затрат времени и денежных средств. Разумеется, ошибки, обнаруживаемые на поздних стадиях цикла разработки, устранить более трудно, потому что нужно проследить их источник, проанализировав десятки или сотни тысяч строк кода, нередко это выполняют тестировщики, которые не писали код сами. Стоимость устранения проблем, обнаруженных на поздней стадии тестирования, в 10-20 раз превышает стоимость устранения тех же ошибок при написании кода[1].

Ограничения общих методов обнаружения ошибок исполнения

Методы и инструментальные средства, обычно используемые для обнаружения и устранения ошибок при выполнении, основаны на технологии 20-30-летней давности. Эти подходы делятся на три категории: ручной анализ кода, использование статических анализаторов и динамическое тестирование (или тестирование по методу «белого ящика»). Целью этих методов является поиск ошибок. Они не могут подтвердить их отсутствие.

Ручной анализ кода может быть эффективным средством нахождения ошибок при выполнении для относительно небольших приложений (имеющих 10-30 тысяч строк кода). Однако для систем большего размера ручной анализ является трудоемким. Он требует опытных инженеров, которые должны проанализировать исходный код и сообщить об опасных или ошибочных конструкциях; эта работа сложна, бесконечна, неповторяема и дорогостояща. Подтверждение отсутствия ошибок выполнения является гораздо более сложной задачей, которую нельзя решить путем анализа кода.

Статические анализаторы дают очень ограниченную помощь в обнаружении ошибок выполнения. Компиляторы, линкеры и средства оценки качества (например, средства, которые сообщают о количестве комментариев, глубине графа вызовов или сложности структуры кода) могут обнаруживать только те ошибки, которые не зависят от выполнения программы, например, статические выражения, состоящие из констант и буквенных значений (такие, как статическое переполнение), статическое деление на постоянную величину, равную нулю, и явно неинициализированные данные. Прочие ошибки при выполнении, такие как обращение за пределы массива, условно инициализированные данные и неверное разыменование указателей не могут обнаруживаться подобными средствами.

Динамическое тестирование или тестирование по методу «белого ящика» требует от инженеров написания и выполнения тестов. Для типичного встроенного приложения могут потребоваться тысячи тестов. После выполнения теста инженер-тестировщик должен проанализировать результаты и найти источники ошибок. За последние два десятилетия этот процесс тестирования не улучшился. Хотя инженеры могут предпринять другие шаги, чтобы повысить шансы обнаружить ненормальности, такие как включение инструментального кода и выполнять покрытие кода, динамическое тестирование основано на методе проб и ошибок, который обеспечивает только частичный, если не крохотный охват всех возможных комбинаций величин, что могут встретиться в ходе выполнения. Как и ручной анализ кода, этот процесс требует больших затрат ресурсов и времени. Время, затрачиваемое на написание тестов и ожидание конца разработки выполняемой системы, нередко вынуждает отложить динамическое тестирование до конца цикла разработки, когда стоимость исправления ошибок уже наиболее высока.

Инженеры, которые применяют динамическое тестирование, чтобы найти во встроенных приложениях ошибки выполнения, сталкиваются и с другими проблемами. Во-первых, затраты на тестирование растут экспоненциально с увеличением размера приложения. Поскольку число ошибок в фиксированном количестве строк кода является почти постоянным (по обычной оценке, в тысяче строк их от 2 до 10), с ростом общего количества строк в приложении, вероятность нахождения этих ошибок сильно уменьшается. Когда размер исходного кода удваивается, усилия на тестирование должны быть увеличены вчетверо или более, чтобы достичь прежнего уровня уверенности.

Во-вторых, динамическое тестирование выявляет только *признаки*, а не причину ошибки. В результате нужно потратить дополнительное время на отладку, чтобы выявить причину каждой ошибки после ее обнаружения в ходе выполнения программы.

Поиск причины аномалии в коде может потребовать больших затрат времени, если ошибка обнаруживается на поздней стадии разработки. Ошибка, обнаруженная при валидации, может потребовать сотню часов на отладку, тогда как в случае ее обнаружения в ходе юнит тестирования причина может быть найдена за час или менее. Тестирование можно автоматизировать, а отладку – нет. Если в каждой тысяче строк нового кода имеется от 2 до 10 ошибок, в приложении, имеющем 50 тысяч строк, будет, как минимум, 100 ошибок. Разработчику, который тратит на поиск и исправление каждой ошибки в среднем 10 часов, на отладку приложения потребуется 1000 часов.

В-третьих, динамическое тестирование нередко заставляет разработчиков добавлять в свой код инструментальные средства, чтобы при выполнении программы было легче обнаружить аномалии. Однако использование инструментального кода требует времени, увеличивает затраты на выполнение и даже может скрыть такие ошибки, как нарушение памяти и конфликты доступа к совместно используемым данным. Более того, методы, основанные на применении инструментального кода, обнаруживают ошибки только в том случае, если выполненные тесты выявляют аномалию.

В-четвертых, эффективность выявления ошибок выполнения зависит также от способности тестов анализировать различные комбинации значений и условий, которые могут появиться при выполнении. Обычно тесты охватывают только часть всех возможных комбинаций. При этом остается большое число не протестированных комбинаций, в том числе и тех, которые могут вызвать ошибки выполнения.

Методы предотвращения ошибок и обеспечения отказоустойчивости

Некоторые языки разработки обеспечивают возможности обработки исключительных ситуаций и позволяют обнаружить аномалии, возникающие при выполнении, а также принять решение, стоит ли продолжить выполнение программы, восстановить ее или перезапустить. Эти возможности обычно увеличивают затраты на выполнение, которые связаны с предварительными и конечными условиями, а также с обработчиками исключительных ситуаций, а это может противоречить эксплуатационным требованиям к встроенным приложениям.

В результате инженеры-тестировщики иногда используют их только при тестировании и исключают из продукта при его поставке. Ошибки при выполнении рабочего кода могут привести к серьезным и даже катастрофическим результатам.

Другим способом предотвращения ошибок является использование формализованных методов, например доказательства теорем. Хотя доказательство теорем, является мощным средством, оно очень трудоемко и требует больших математических знаний, чтобы быть эффективным, а это ограничивает возможность применения для промышленных приложений.

Верификация кода путем абстрактной интерпретации

Абстрактная интерпретация объединяет обычные статические методы анализа и динамическое тестирование путем проверки динамических свойств программ во время компиляции. Не выполняя саму программу,

абстрактная интерпретация исследует все ее возможные поведения (то есть все возможные комбинации значений) за единственный проход, чтобы определить, как и при каких условиях может возникнуть сбой программы. Абстрактная интерпретация представляет собой развитый и мощный математический метод. Ее можно рассматривать как расширение методов компиляции, используемых программистами для того, чтобы до выполнения реальных тестов предсказать будущее поведение программы.

Аналогия из реального мира

У инженера, которому нужно предсказать траекторию ракеты в воздухе есть три следующих варианта действий.

■ Провести полный учет всех различных частиц, которые могут встретиться на пути ракеты, изучить их свойства и определить, как столкновение с каждой частицей повлияет на траекторию ракеты. Очевидно, что этот подход непрактичен из-за большого числа и разнообразия частиц, встречающихся на траектории полета. Даже если бы оказалось возможным заранее знать все условия, которые могут возникнуть в любой момент времени (например, скорость ветра и дождь), они будут меняться с каждым новым полетом. Это означает проведение нового полного анализа перед каждым запуском.

■ Запустить много ракет, чтобы определить эмпирические законы движения и соответствующие пределы отклонений. Эти результаты могут быть использованы для оценки траекторий в пределах некоторого доверительного интервала. Данный подход требует больших затрат как времени, так и денежных средств, а каждая попытка приведет к новым выводам. Более того, полная проверка полета ракеты при каждой возможной комбинации условий является практически невозможной.

■ Использовать законы физики и известные значения (силы тяжести, коэффициента аэродинамического торможения, начальной скорости и т. п.), чтобы превратить задачу в набор уравнений, которые могут быть решены путем применения формализованных или численных математических методов. Данный подход дает решения для широкого диапазона условий, которые становятся параметрами в математической модели, и позволяет инженеру предсказывать поведение ракеты при различных условиях.

Абстрактная интерпретация сходна с математическим моделированием. Она определяет динамические свойства данных в исходном коде программы (то есть уравнения, связывающие переменные) и применяет их для проверки определенных динамических свойств.

Как работает абстрактная интерпретация

Абстрактная интерпретация опирается на широкую базу математических теорем, которые определяют правила для анализа сложных динамических систем, таких как программные приложения. Вместо проведения численного анализа каждого состояния программы, абстрактная интерпретация представляет эти состояния в более общем виде и задает правила работы с ними. Абстрактная интерпретация не только создает математическую абстракцию, но и интерпретирует эту абстракцию.

Для того чтобы получить математическую абстракцию состояний программы, абстрактная интерпретация тщательно анализирует все переменные кода. Значительные вычислительные мощности, необходимые для этого анализа, в прошлом отсутствовали. Абстрактная интерпретация, в сочетании с алгоритмами неэкспоненциальной сложности и возросшей современной мощностью компьютеров, представляет собой практическое решение сложных проблем тестирования.

В случае применения для обнаружения ошибок выполнения абстрактная интерпретация выполняет полную верификацию всех операций, вызывающих риск, и автоматически выполняет диагностику состояния каждой операции («проверена», «не выполнена», «недоступна» или «не проверена»). Инженеры могут использовать абстрактную интерпретацию, чтобы получить результаты в момент компиляции, то есть на самой ранней стадии тестирования.

Применение абстрактной интерпретации

Для того чтобы лучше понять, как работает абстрактная интерпретация, рассмотрим программу P, которая использует две переменные: X и Y. Она выполняет следующую операцию:

X=X/(X-Y)

Для проверки этой программы на наличие ошибок выполнения исследуем все возможные причины, вызывающие ошибку операции:

■ X и Y могут не иметь начальных значений

■ X-Y может иметь положительное или отрицательное переполнение

■ X и Y могут быть равны и вызывать деление на нуль

■ X/(X–Y) может иметь положительное или отрицательное переполнение

Поскольку любое из этих состояний может вызвать ошибку выполнения, на следующих шагах исследуется возможность деления на нуль.

Все возможные значения X и Y в программе P можно представить на диаграмме. Красная линия на рис. 1 представляет набор значений (X, Y), которые могут привести к делению на нуль.

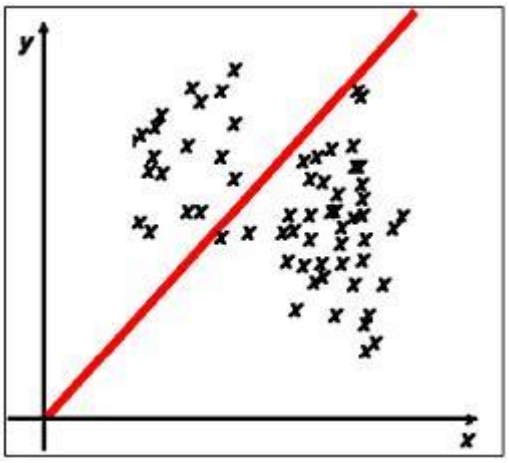

Рис. 1. Все возможные значения X и Y в программе P.

Очевидный способ проверки деления на нуль заключается в переборе всех состояний и определения того, находятся ли они на красной линии. Этот подход применяется при обычном тестировании по методу «белого ящика», но он имеет существенные ограничения. Во-первых, количество возможных состояний в реальном приложении обычно очень велико из-за большого числа используемых переменных. На перебор всех возможных состояний уйдут годы, что делает полную проверку совершенно невозможной.

В отличие от «метода грубой силы», который заключается в переборе всех возможных значений, абстрактная интерпретация устанавливает обобщенные правила для анализа всего множества состояний. Она создает абстрактное представление программы, которое можно использовать для проверки свойств.

Один из примеров этой абстракции, называемой «анализом типов», приведен на рис. 2. Этот тип абстракции используется компиляторами, редакторами связей и основными статическими анализаторами путем применения правил вывода типов. При анализе типов набор точек проецируется на оси X и Y, определяются минимальные и максимальные значения X и Y и вычерчивается соответствующий прямоугольник. Поскольку прямоугольник охватывает все возможные значения X и Y, если свойство подтверждено для прямоугольника,

оно будет справедливым и для программы. В этом случае нас интересует пересечение красной линии и прямоугольника, поскольку, если они не пересекаются, деления на нуль никогда не будет.

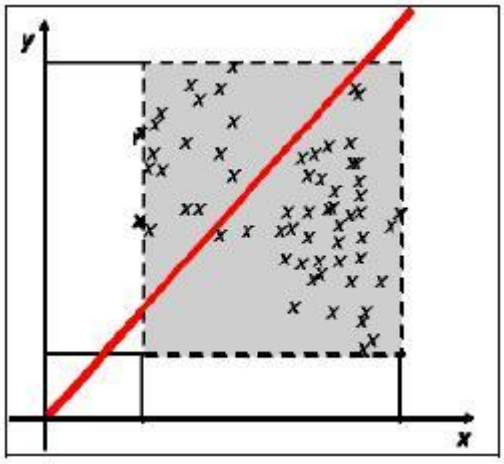

Рис. 2. При анализе типов все значения X и Y проецируются на оси координат.

Основным недостатком анализа типов является то, что прямоугольник содержит слишком много нереальных значений X и Y. Это дает плохие и неточные результаты, а также приводит к появлению большого числа предупредительных сообщений, которые нередко остаются непрочитанными, если только инженер не отключит их вообще.

Вместо больших прямоугольников абстрактная интерпретация устанавливает правила для построения более точных фигур (см. рис. 3). Она использует методы, основанные на целочисленных решётках, совокупности многоугольников и базах Грёбнера, чтобы представить соотношения данных (X и Y), которые учитывают операторы управления (такие как условный оператор, циклы for и while и переключатель), межпроцедурные операции (вызовы функций) и многозадачный анализ.

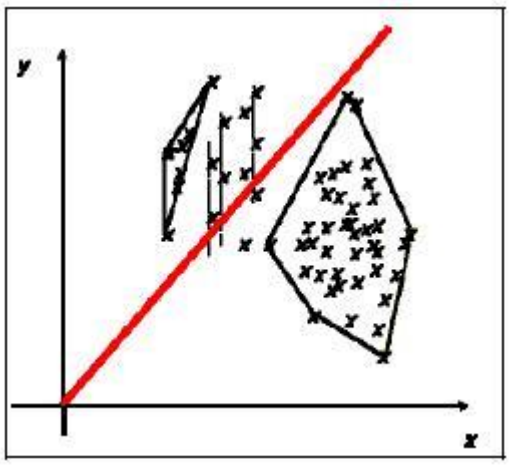

Рис. 3. Абстрактная интерпретация устанавливает правила для построения более точных фигур, представляющих соотношения между X и Y. В отличие от компиляторов и статических анализаторов абстрактная интерпретация не полагается только на идею вычислять соотношения между типами данных и постоянными величинами. Вместо этого она определяет эти соотношения по семантике каждой операции и операнда программы и использует их для анализа исходного кода.

При выполнении абстрактной интерпретации указанные ниже элементы программы толкуются по-новому.

■ Индекс в цикле for не является более целочисленной величиной, а представляет собой монотонную возрастающую дискретную функцию от нижнего до верхнего предела.

■ Параметр, передаваемый в функцию, более не является переменной или константой, а представляет собой набор значений, которые можно использовать с целью ограничения локальных данных, используемых в функции.

■ Любые глобальные совместно используемые данные могут быть в любой момент изменены в многозадачной программе, за исключением того случая, когда включены механизмы защиты, такие как блокировки памяти или критические секции.

■ Указатель является типом данных, который может создавать связи между явными данными и приводить к побочным эффектам и скрытым одновременным обращениям к совместно используемым данным в многозадачных приложениях.

■ Переменная имеет не только тип и диапазон значений, но и набор уравнений (включая соотношения, зависящие от потока управления), которые порождают ее.

■ В конечном итоге, ошибки при выполнении являются уравнениями, которые также называются условиями правильности. Абстрактная интерпретация может их решить, используя уравнения, связывающие переменные друг с другом.

Примеры абстрактной интерпретации

Приведенные ниже примеры являются общими конструкциями кода, которые вызывают ошибки при выполнении, автоматически обнаруживаемые абстрактной интерпретацией.

Анализ структуры управления: разыменование указателя выхода за пределы после цикла for

```
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, его можно абстрагировать как дискретную возрастающую функцию, которая изменяется на 1 с начала массива ar. При выходе из цикла for, когда i становится равным 100, указатель p также увеличивается до 100. Это приводит к тому, что указатель p выходит за пределы в строке 15, поскольку индекс массива изменяется от 0 до 99. Абстрактная интерпретация подтвердит надежность этой части кода и укажет на строку 15 как на источник ошибки при выполнении.

Анализ структуры управления: доступ за пределы массива в двух вложенных циклах for

```
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, которые монотонно увеличиваются на 1 от 0 до 9. Операция i–j, используемая как индекс массива ar, возвратит в итоге отрицательную величину. Абстрактная интерпретация этого кода подтвердит его надежность и укажет об обращении за пределы массива в строке 26.

Следует заметить, что ошибки при выполнении в этих примерах нередко приводят к порче данных, хранящихся рядом с массивом ar. В зависимости от того, как и когда эти испорченные данные используются в каком-то месте программы, отладка этого типа ошибки без использования абстрактной интерпретации может стоить значительных усилий.

Межпроцедурный анализ: деление на нуль

```
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 этой функции depth является целой величиной, которая сначала увеличивается на 1. Затем она используется в качестве знаменателя для определения степени продвижения, после чего рекурсивно передается в функцию foo. Проверка того, приведет ли операция, указанная в строке 34, к делению на нуль, требует проведения межпроцедурного анализа с целью определения того, какие значения будут переданы в функцию foo (см. bug_in_recursive, строки 45 и 50), поскольку она определит величину *depth.

В предшествующем коде функция foo может быть вызвана в двух разных ситуациях в функции bug_in_recursive. Если результат выполнения оператора if в строке 42 является «ложью», foo вызывается со значением 10 (строка 50).

Поэтому *depth становится монотонной дискретной возрастающей функцией, изменяющейся на 1 от 11 до 49. Уравнение в строке 34 не даст деления на нуль.

Однако, если результат оператора if в строке 42 является «истиной», то foo вызывается со значением -4. Поэтому *depth становится монотонной дискретной возрастающей функцией, изменяющейся на 1 от -3 до 49. В итоге *depth станет равной нулю, что вызовет деление на нуль в уравнении, указанном в строке 34.

Простая проверка синтаксиса не выявит эту ошибку, так же, как и все тесты. Абстрактная интерпретация подтвердит надежность всего кода, кроме строки 45. Это демонстрирует уникальную способность абстрактной интерпретации выполнять межпроцедурный анализ и отличать проблематичные вызовы функций от приемлемых. Если не ликвидировать ошибку, связанную с делением на нуль, она приведет к остановке процессора. Этот тип ошибки может также потребовать значительно длительного времени отладки вследствие использования рекурсивных конструкций.

Многозадачный анализ: одновременное обращение к совместно используемым данным

Абстрактная интерпретация проводит анализ потока управления и данных и может также проверять многозадачные приложения. Основная проблема, связанная с такими приложениями – это обеспечение того, чтобы не было одновременного обращения к совместно используемым данным и критическим ресурсам.

Псевдонимизация данных и чередование задач делают трудным выявление проблем такого типа, связанных с одновременным обращением.

При псевдонимизации данных для доступа к совместно используемой памяти применяются указатели. Такой подход может вызвать наличие скрытых или неявных связей между переменными, при этом одна переменная может быть неожиданно изменена указателями при выполнении программы, что вызовет эпизодические аномалии. Анализ данной проблемы требует использования высокоэффективных алгоритмов анализа указателей. При абстрактной интерпретации эти алгоритмы могут быть реализованы, чтобы они дали список совместно используемых данных и список доступов для чтения и записи с указанием функций и задач, а также соответствующий граф одновременного доступа.

Чередование задач делает проблематичной отладку многозадачных приложений, потому что серьезные ошибки в программе трудно воспроизвести, если они зависят от последовательности задач, которые выполняются в определенном порядке в режиме реального времени. Абстрактная интерпретация учитывает каждое возможное чередование задач, чтобы выполнить полный анализ потока управления. Получить эти результаты или граф одновременного доступа вручную было бы исключительно трудным.

Выгоды абстрактной интерпретации для обнаружения ошибок выполнения

Абстрактная интерпретация представляет собой эффективный, экономичный и быстрый способ разработки надежных встроенных систем. Она дает четыре преимущества: подтверждение надежности кода, повышение эффективности, уменьшение накладных расходов и упрощение отладки.

Подтверждение надежности кода

Благодаря полному анализу исходного кода абстрактная интерпретация не только обеспечивает обнаружение ошибок при выполнении, но и проверяет правильность кода. Это особенно важно для приложений с особыми требованиями к обеспечению безопасности, в которых системные отказы могут быть катастрофическими. Обычные средства отладки ориентированы на обнаружение ошибок, но не проверяют надежность остального кода. Абстрактная интерпретация делает возможным выявление кода, который никогда не вызовет сбой программного обеспечения, и, тем самым, она устраняет любую неуверенность в надежности программного обеспечения.

Повышение эффективности

Проверяя динамику приложений, абстрактная интерпретация дает возможность разработчикам и тестировщикам выявить те участки кода в их программах, которые не имеют ошибок выполнения, в отличие от тех, которые приведут к нарушению надежности. Поскольку ошибки могут быть обнаружены до компиляции кода, абстрактная интерпретация помогает рабочим группам существенно сократить затраты времени и денежных средств благодаря обнаружению и ликвидации ошибок выполнения в то время, когда их легче всего исправить.

Уменьшение накладных расходов

Абстрактная интерпретация не требует выполнения программ и поэтому дает полные результаты без написания и выполнения тестов. Аналогичным образом отсутствует необходимость добавлять инструментальный код, а потом удалять его перед поставкой программного обеспечения. Кроме того, абстрактная интерпретация может быть реализована в текущих проектах без изменения существующих процессов разработки.

Упрощение отладки

Абстрактная интерпретация упрощает отладку, поскольку она непосредственно указывает источник каждой ошибки, а не только ее признак. Она не приводит к потере времени на отслеживание источников аварийных отказов и ошибок, приводящих к порче данных, а также время, ранее тратившееся на попытку воспроизвести эпизодические серьезные ошибки. Абстрактная интерпретация повторяема и исчерпывающа. Каждая операция в коде автоматически выявляется, анализируется и проверяется для всех возможных комбинаций входных величин.

Ошибки выполнения, обнаруживаемые абстрактной интерпретацией

■ Одновременное обращение к совместно используемым данным

■ Проблемы с разыменованием указателей (нулевой, обращения за пределы)

■ Обращения за пределы массива

■ Считывание данных, которые не имеют начальных значений

■ Неверные арифметические операции (такие как деление на нуль и извлечение квадратного корня из отрицательных чисел)

■ Вещественное и целочисленное положительное или отрицательное переполнение

■ Неверное преобразование типа (например, float J int, long J short).

Прочие динамические свойства, обнаруживаемые абстрактной интерпретацией

■ Недостижимый код

■ Не заканчивающиеся циклы

■ Инициализированные возвращаемые значения

Инструментальные средства абстрактной интерпретации

Перечисленные ниже продукты компании MathWorks являются единственными имеющимися на рынке инструментальными средствами для верификации правильности кода, которые реализуют абстрактную интерпретацию.

■ PolySpace™ Client for C++

■ PolySpace™ Server for C/C++

■ PolySpace™ Client for Ada

---

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

Контакты
exponenta.ru
E-mail: **info@exponenta.ru**
Тел.: +7 (495) 009 65 85
Адрес: **115088 г. Москва,**
2-й Южнопортовый проезд, д. 31, стр. 4

mathworks.com