Дэниэл Джексон

программы проверяют программы

Не прочнее
самого слабого звена —
так можно говорить не только
о мостах, но и о компьютерных программах.
Несмотря на то, что программное обеспечение
стало важнейшим компонентом современной
инфраструктуры, эффективная процедура проверки
его надежности была разработана совсем недавно

Сегодня компьютеры обслуживают авиационные, банковские, телекоммуникационные, торговые и промышленные системы. Однако надежность программного обеспечения по-прежнему оставляет желать лучшего

Высокотехнологичной жемчужиной нового международного аэропорта в Денвере, открывшегося 11 лет назад, должна была стать автоматизированная багажная линия. Предполагалось, что почти 42 км конвейеров будут быстро и без задержек доставлять чемоданы и дорожные сумки к багажным отсекам самолетов и в залы прилета. Однако проблемы с программным обеспечением задержали открытие аэропорта на 16 месяцев и повлекли за собой дополнительные расходы в сотни миллионов долларов. Настройка автоматизированной линии длилась еще несколько лет, но требуемого уровня надежности так и не удалось достичь. Прошлым летом проблема была наконец решена: теперь багаж доставляют вручную на обычных грузовых тележках. Фирма *BAE Automated Systems*, разработавшая багажный конвейер, была ликвидирована, а компания *United Airlines*, ее главный заказчик, оказалась на грани банкротства.

Ежедневно миллионы раздосадованных пользователей страдают из-за низкого качества программ. Например, в 1997 г. Налоговое управление США без толку потратило $4 млрд. на новый компьютерный проект, а потом еще $8 млрд. на его модернизацию. Неудачу потерпело и ФБР, потерявшее в 2005 г. $170 млн. на разработке виртуальной системы управления делами, которая оказалась неработоспособной. Несмотря на огромные затраты, Федеральное управление гражданской авиации США до сих пор не может справиться с обновлением морально устаревшей системы управления воздушным движением.

Практически все серьезные неудачи обусловлены тем, что принципиальные ошибки, допущенные на этапе проектирования, выявляются только в процессе программирования. Иногда это происходит по недосмотру или из-за несогласованности в работе, но чаще всего причина кроется в общей непродуманности проекта. Его детальная структура проясняется, лишь когда программисты начинают вносить исправление за исправлением. Однако в результате программный код становится похожим на лоскутное одеяло, расползающееся по швам.

Недавно появились новые инструментальные средства разработки, в которых заложены принципы, используемые при проверке аппаратного обеспечения. Сначала общая логика программы описывается на языке программирования высокого уровня, а затем запускается аналитический модуль, просчитывающий миллионы возможных ситуаций и выявляющий условия, которые могут привести к нарушению работы. Это позволяет отловить неявные ошибки еще до того, как будут написаны первые строчки кода. В результате получается надежная и тщательно отлаженная программа. Один из таких инструментов был создан нашей рабочей группой и получил название *Alloy*. Он свободно распространяется в Интернете и может применяться при тестировании специального программного обеспечения для авиации, телефонии, медицинского приборостроения и других отраслей (*рис. на стр. 55*).

Уже четверть века ученые занимаются поиском строгих математических способов проверки программ. В *Alloy* и аналогичных инструментах используется метод автоматических рассуждений, при котором проект рассматривается как большая головоломка. Поскольку аналитический модуль обрабатывает не исходный код, а только логическую схему программы, он не может гарантировать, что в конце концов она будет работать без сбоев. Однако разработчик получает возможность проверить проект на наличие концептуальных ошибок и заложить прочный фундамент для создания надежного программного обеспечения.

Оценка проектов

Разговоры о кризисе программного обеспечения звучат с 1960-х гг. Сейчас, когда компьютеры используются почти во всех сферах деятельности, эта тема стала особенно актуальной. Сегодня каждая программа проходит отладку и тестирование, при которых проверяется ее работоспособность в широком диапазоне начальных условий, что позволяет выявить множество мелких недостатков, но глобальные ошибки, заложенные в проект, часто остаются незамеченными. Складывается ситуация, когда разработчик за деревьями не видит леса.

Кроме того, исправления, вносимые в процессе отладки, неизбежно усложняют программу и нередко приводят к снижению эффективности и появлению новых ошибок. Здесь можно провести аналогию с геоцентрической системой Птолемея, созданной древними греками. Средневековые астрономы подгоняли ее под результаты своих наблюдений, надстраивая эпициклы над эпициклами, однако довести ее до ума им так и не удалось, потому что она изначально была основана на неверных принципах.

Точно так же и плохое программное обеспечение со временем становится все более сложным и ненадежным, несмотря на вкладываемые в отладку время и деньги. Известно, что наиболее серьезные проблемы связаны не с ошибками программирования, а с концептуальными ▸

Неисправная багажная линия в Денверском международном аэропорту

недочетами, допущенными на этапе проектирования, задолго до того, как за дело берутся программисты. Более того, усилия, потраченные на моделирование и анализ требований, спецификаций и блок-схем, стоят гораздо дешевле, чем утомительное тестирование всего программного кода. Внимательное отношение к проектированию на ранней стадии окупается сполна и в дальнейшем избавляет от денежных затрат и головной боли.

Программное обеспечение не повинуется физическим законам, поэтому инструменты для его проектирования только начинают появляться. Например, если балка выдерживает большой вес, то можно быть уверенным, что она не сломается при меньших нагрузках. В повседневной практике малые изменения одного параметра сопровождаются малыми изменениями другого. К сожалению, это совершенно неприменимо к компьютерным программам, которые представляют собой дискретные математические объекты, состоящие из битов. Результаты тестирования не поддаются ни экстраполяции, ни интерполяции: даже если одна часть программы работает правильно, то о работоспособности других ее частей на этом основании нельзя сказать ничего.

На заре информатики специалисты надеялись, что правильность программ можно доказывать так же, как математические теоремы. Но поскольку автоматизировать многие этапы анализа не удавалось, львиную долю работы приходилось выполнять вручную. Строгие формальные методы применялись только для проверки простых, но чрезвычайно важных алгоритмов, таких как, например, алгоритм управления железнодорожными стрелками.

В последнее время исследователи пытаются использовать современные мощные процессоры для рассмотрения всех возможных сценариев. Так называемая модельная проверка уже широко применяется при тестировании проектов интегральных схем. Идея заключается в том, чтобы смоделировать все состояния системы, которые могут возникнуть в процессе работы, и убедиться, что ни одно из них не ведет к сбою. У современного микрочипа может быть порядка 10^{100} различных состояний, а у сложной программы и того больше. Однако интеллектуальные методы представления данных, когда большие наборы состояний программы даются в очень сжатом виде, позволяют довольно быстро просматривать каждое из них.

Как ни печально, тотальный перебор не годится для проверки сложных состояний модели, столь характерных для большинства программ. Мы разработали подход, в котором реализован несколько иной механизм: при анализе возможных состояний учитываются определенные ограничения. Кроме того, проверяются не все сценарии, но ведется поиск лишь тех, что ведут к сбою.

Решение головоломки

Чтобы понять, как *Alloy* ищет дефекты в программах, рассмотрим старую загадку про волка, козу и кочан капусты. Некоему крестьянину необходимо переправить всех троих через реку на лодке, в которой кроме него самого может поместиться только один предмет. Трудность заключается в том, что оставленный без присмотра волк непременно съест козу, которая, в свою очередь, не прочь полакомиться капустой. Решая задачи подобного рода, мы анализируем различные сценарии, удовлетворяющие определенным условиям. Сначала крестьянин перевозит козу. Потом возвращается и переправляет волка. Затем сажает козу в лодку и плывет за капустой. Оставив козу, везет капусту, а потом снова возвращается за козой. Проверяя каждый шаг на выполнение ограничительных условий, мы убеждаемся, что все без проблем перебираются через реку.

В успешном проекте должен быть заложен набор аналогичных, хотя и более сложных ограничений. Инструмент для проверки проектов должен уметь находить контрпримеры, т.е. такие ситуации, которые удовлетворяют всем заданным условиям, но при этом приводят к неправильной работе программы. Каждый обнаруженный контрпример указывает на ошибку проектирования или на неточность описания нежелательного результата и свидетельствует о том, что нужно либо скорректировать проект, либо пересмотреть ожидания разработчика.

Поиск осложняется тем, что даже для простых проектов существует огромное количество сценариев, а контрпримеров среди них очень мало. Допустим, требуется рассадить

ОБЗОР: СЕМЬ РАЗ ПРОВЕРЬ

- Несмотря на все возрастающую важность программного обеспечения, разработчики редко анализируют его надежность на этапе проектирования. Лишь недавно появились инструменты для проверки проектов программ.
- Утилита *Alloy* объединяет в себе язык программирования высокого уровня для моделирования сложных программных комплексов и аналитический модуль для автоматического поиска концептуальных и структурных ошибок в проектах программ.
- Такие инструменты, как *Alloy*, помогут существенно повысить надежность программного обеспечения за счет применения прогрессивных методов проектирования.

ALLOY В ДЕЙСТВИИ

Alloy помогает находить и исправлять недостатки проектов программного обеспечения, предоставляя язык программирования высокого уровня для описания структуры программы и аналитический модуль для поиска возможных ситуаций, приводящих к сбою. В приведенном примере показано, как можно использовать *Alloy* для проверки проекта файловой системы, т.е. программного обеспечения, ответственного за хранение файлов и папок (директорий) на диске. Главная задача *Alloy* заключается в том, чтобы оценить результаты всевозможных операций над файлами и папками. Здесь рассматриваются моделирование и проверка операции перемещения одной директории в другую

Программный код модели

Результат

Корневой каталог

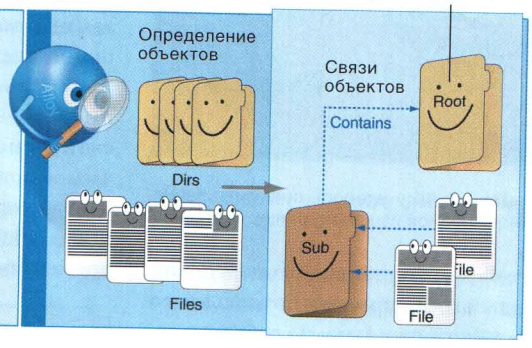
ШАГ 1: ОПРЕДЕЛЕНИЕ ОБЪЕКТОВ

Сначала разработчик определяет объекты системы (файлы, директории и файловую систему в целом) и их взаимоотношения. Согласно модели, файловая система (*FS*) состоит из трех компонентов: множество файлов (*files*), множество директорий (*dirs*) и множество связей, определяющих, какие файлы и папки содержатся в каждой папке (*contains*)

```
module filesystem

abstract sig Object {}
sig File, Dir extends Object {}

sig FS {
  dirs: set Dir,
  files: set File,
  contains: dirs -> (dirs + files)
  }
```



ШАГ 2: МОДЕЛИРОВАНИЕ ОПЕРАЦИИ

Затем моделируется переход системы из состояния *fs* в состояние *fs'* при переносе директории *d* в директорию *to*. В трех строках указываются условия, описывающие предполагаемый результат операции. В первой строке записано, что и перемещаемый объект, и его новое расположение являются директориями файловой системы. Второй строкой представлена сама операция: старая связь с директорией *d* удаляется и добавляется новая, которая связывает ее с папкой *to*. В третьей строчке говорится, что никаких иных изменений в файловой системе не производится

```
pred move_dir (fs, fs': FS, d, to: Dir) {
d + to in fs.dirs
fs'.contains = fs.contains - Dir->d + to->d
fs'.files = fs.files and fs'.dirs = fs.dirs
  }
```

ШАГ 3: ОПРЕДЕЛЕНИЕ ТРЕБОВАНИЙ

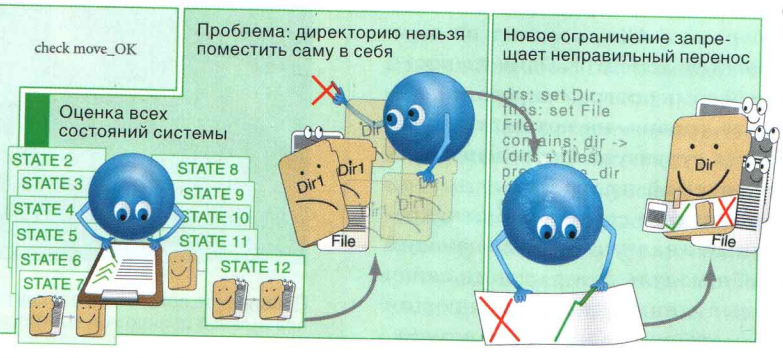
Следующим этапом разработчик формулирует обязательное требование, согласно которому все файлы и папки должны быть доступны по некоторому пути из корневой директории. В модели появляется утверждение *move_OK*, которое гласит, что в результате операции перемещения все директории и файлы остаются доступными из корневого каталога

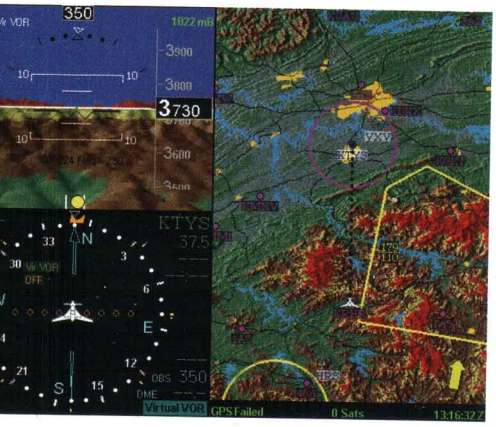
```
pred reachable (fs: FS) {
  some root: fs.dirs | fs.(dirs+files) in root.*(fs.contains)
  }

assert move_OK {
  all fs, fs': FS, d, to: Dir |
    reachable (fs) and move_dir (fs, fs', d, to) implies reachable (fs')
  }
```

ШАГ 4: ИСПРАВЛЕНИЕ ОШИБОК

Alloy запускает процедуру *check move_OK* и проверяет справедливость утверждения для всевозможных сценариев переноса файлов и директорий. Вскоре тестирующая программа находит контрпример: в качестве модели допускается перемещение директории саму в себя, в результате которого она становится недоступной. Чтобы исправить ошибку, разработчик добавляет ограничение, запрещающее перемещать директорию саму в себя или в своих потомков





С помощью *Alloy* удалось повысить защищенность одной из авиационных систем

гостей за свадебным столом. Если каждый гость приходит отдельно, то решение тривиально. Но добавьте несколько пар разведенных супругов, которые не захотят сидеть рядом, и проблема сразу усложнится. А теперь представим свадебный банкет Ромео и Джульетты. Если дано двадцать стульев и десять гостей, то их можно рассадить 10^{20} способами. Даже если компьютер будет проверять миллиард сценариев в секунду, ему потребуется три тысячи лет.

В 1980-х гг. математики выделили отдельный класс задач, которые в худшем случае решаются перебором всех возможных вариантов. Однако десять лет назад появились новые стратегии и алгоритмы поиска, позволяющие современным компьютерам сравнительно легко справляться с ними. Сейчас существует множество программ, которые способны на удивление быстро решать так называемые задачи на выполнимость с миллионами условий.

Важность абстракций

Программа *Alloy* состоит из двух основных элементов. Один из них — новый язык программирования высокого уровня, на котором описываются структура и поведение проектируемой программы, другой представляет собой автоматизированный анализатор, включающий в себя модуль для решения задач на выполнимость и позволяющий рассматривать множество возможных сценариев.

Применение *Alloy* начинается с создания точной модели проекта, которая детально описывает желательное и нежелательное поведение системы и всех ее компонентов. Вначале программист составляет описание различных типов объектов, а затем группирует сходные по структуре и поведению и связанные математическими соотношениями объекты во множества. Затем анализируются ограничения, связанные со структурой проектируемой системы и предполагаемым поведением пользователей. Например, никто из присутствующих на банкете не может быть одновременно и Монтекки, и Капулетти, а рядом с каждым гостем должны сидеть ровно два других гостя. Некоторые положения определяются самим проектом: каждый стол на банкете (за исключением стола для новобрачных) предназначается либо для Монтекки, либо для Капулетти. Из неизбежных ограничений выводится утверждение о том, что система не может принимать некоторые состояния, равно как и не могут происходить определенные последовательности событий. В нашем примере ни один Монтекки не окажется за столом рядом с Капулетти (исключение составляют только Ромео и Джульетта). С помощью модуля для решения задач на выполнимость

ОТЛАДКА МЕДИЦИНСКОГО ОБОРУДОВАНИЯ

Сегодня медицинское оборудование всецело полагается на специализированное программное обеспечение. В сложных приборах даже кнопка аварийной остановки — это не просто электрический выключатель: она запускает компьютерную программу из 15 тыс. строчек кода, которая останавливает систему.

С помощью *Alloy* мы исследовали программное обеспечение установки для протонной терапии. Сначала был рассмотрен проект новой системы планирования, которая определяет, в какой из процедурных кабинетов следует направить протонный луч. *Alloy* помог нам найти сценарии взаимодействия терапевта с оператором в диспетчерской, приводящие к непредусмотренным результатам.

Мы также проанализировали протокол позиционирования пациента под протонным лучом и выявили неожиданный дефект: угол наклона кушетки постоянно «уплывал». Результаты моделирования в программе *Alloy* показали, что использование правильной абстракции позволяет упростить проблему и свести ее к давно решенной задаче фиксации водительского сиденья в автомобиле. Разумеется, установка была оснащена многими защитными системами, и непредусмотренное покачивание кушетки не представляло никакой опасности для пациентов. Однако если бы на этапе проектирования была выбрана правильная абстракция, то программное обеспечение было бы гораздо надежнее и проще в управлении.

Правильное положение пациента, критически важное для обеспечения правильной дозировки, контролируется специальной компьютерной программой

проводится поиск контрпримеров: *Alloy* пытается вывести утверждения, которые должны быть истинными, если проект программы написан правильно. Иными словами, анализатор пытается построить сценарии, удовлетворяющие ограничениям, но противоречащие выведенным из них утверждениям. В нашем примере возможна ситуация, когда Капулетти (но не Джульетта) оказывается рядом с Монтекки (но не Ромео). Чтобы исправить ошибку, нужно добавить новое ограничение: за столом новобрачных сидят только Ромео и Джульетта.

Определения множеств, отношения объектов, накладываемые ограничения и следующие из них утверждения формируют абстракцию, отражающую все существенные стороны проекта программного обеспечения. Главная задача разработчика заключается в том, чтобы на этапе проектирования выбрать абстракцию, которая будет работать наиболее успешно. В противном случае система получится излишне сложной и ненадежной.

Путь к надежности

Сегодня инструменты для тестирования программного обеспечения применяются в основном в научно-исследовательских целях. Их использовали для анализа архитектуры новейших телефонных коммутаторов, при проектировании защищенных от взлома авиационных процессоров, а также для описания системы управления доступом к телекоммуникационным сетям. С помощью *Alloy* мы убедились в надежности распространенного протокола поиска сетевых принтеров и службы синхронизации файлов в локальной сети.

К сожалению, *Alloy* выявил серьезные изъяны во многих широко используемых программах. Программисты искренне удивлялись, когда скрытые недостатки обнаруживались даже в самых простых проектах. Не за горами тот день, когда общество не сможет больше мириться с ненадежным программным обеспечением, и будет учрежден контроль за его качеством. Тогда важнейшим инструментом для разработчиков станут программы, проверяющие программы. ■

Alloy проверяет протокол поиска принтеров в беспроводной сети

ИНСТРУМЕНТЫ ДЛЯ ПРОВЕРКИ ПРОГРАММ

Специалисты разработали новое поколение инструментов для проверки проектов программного обеспечения и выявления в них структурных и концептуальных недочетов. В основе каждого из них, как правило, лежит тот или иной язык программирования высокого уровня, облегчающий описание и моделирование различных программных продуктов.
Любой подобный инструмент включает в себя аналитический модуль, исследующий бесчисленное множество различных состояний системы и обнаруживающий ее неявные дефекты. Часто в состав таких продуктов входят средства визуализации, наглядно отображающие структуру проверяемого проекта.

ЯЗЫК	ИНСТРУМЕНТ	РАЗРАБОТЧИК	ВЕБ-САЙТ
B	*B-Toolkit*	*B-Core*	*www.b-core.com*
	Atelier-B	*Steria*	*www.atelierb.societe.com*
	Pro-B	Саутгемптонский университет	*www.ecs.soton.ac.uk/~mal/systems/prob.html*
CSP	*FDR*	*Formal Systems Europe*	*www.fsel.com*
FSP	*LTSA*	Лондонский имперский колледж	*www.doc.ic.ac.uk/~jnm/book/ltsa/LTSA.html*
Lotos	*CADP*	НИИ *INRIA* (*INRIA Research Institute*)	*www.inrialpes.fr/vasy/cadp/*
OCL	*USE*	Бременский университет	*www.db.informatik.uni-bremen.de/projects/USE/*
PROMELA	*Spin*	*Bell Laboratories*	*spinroot.com*
Statecharts	*Statemate*	*I-Logix*	*www.ilogix.com*
VDM	*VDMTools*	*CSK Corp.*	*www.csk.com/support-e/vdm/ www.vdmbook.com/tools.php*
Z	*Jaza*	Университет Вайкато	*www.cs.waikato.ac.nz/~marku/jaza/*
Zing	*Zing*	*Microsoft Research*	*research.microsoft.com/zing/*

ОБ АВТОРЕ

Дэниел Джексон (Daniel Jackson) возглавляет группу проектирования программ в Лаборатории информатики и искусственного интеллекта Массачусетского технологического института. Он занимается разработкой и анализом программного обеспечения для критических систем. Заядлый фотограф, Джексон недавно выставлял свои работы в Публичной библиотеке Ньютона в пригороде Бостона.