Верификация программ и темпоральные логики

Ю. Лифшиц*

7 декабря 2005 г.

Содержание

1 О верификаци программ

1.1 Мотивации

В данной статье будет рассмотрен вопрос поиска ошибок в программе при использовании метода верификации модели самой программы. Основная цель исследований в этой области состоит в том, чтобы сформулировать ясную логическую основу для создания автоматических систем верификации программного обеспечения.

План лекции:

1. О верификации программ

2. Темпоральные логики и моделирование программ

*Законспектировал М. Смачных.

3. Алгоритм верификации CTL

При разработке программного обеспечения цена ошибки бывает столь высока, что порой приводит не только к разорению компании, выпустившей данный продукт, но даже к гибели людей. Особенно велика цена ошибки в следующих областях:

- Медицинское обслуживание
- Военная промышленность (самолетостроение, конструирование подводных лодок)
- Атомная промышленность (конструирование и эксплуатация АЭС)
- Управление транспортом
- Медицинские системы
- Электронный бизнес
- Телефонные сети

Классическими примерами критических ошибок в разработке программного обеспечения являются:

- Падение ракеты Ariane-5. *Космическая ракета потерпела крушение сразу после взлета, ошибка сбоя системы была установлена через месяц и заключалась в неправильном переводе числа из 16-ричной системы в 64-ричную.*
- Медицинский ускоритель Therac-25. *Доза лечебного облучения, предназначенного для пациентов, превысила допустимое значение в 100 раз. Неисправное действие прибора было обнаружено лишь по прошествии 6 месяцев, вследствие чего только количество смертельных исходов достигло 6 человек.*

1.2 Методы поиска ошибок

Для повышения качества выпускаемого программного обеспечения, проводят несколько стадий тестирования. Перечислим основные из них:

1. Тестирование прототипа
2. Тестирование полной программы
   - Регрессивное тестирование
   - Нагрузочное тестирование
   - Функциональное тестирование
3. Дедуктивный анализ
4. Верификация модели программы

2 Темпоральные логики

2.1 Стадии верификации программ

В данной статье нас будет интересовать вопрос верификации модели программы. Основными этапами стадии верификации программ являются:

1. Моделирование *Модель предназначена для изучения объекта путем его упрощения, выбора тех параметров, которые существенны.*

2. Спецификация *Спецификация предназначена для формулирования основных требований, предъявляемых к модели.*

3. Верификация модели *На стадии верификации модели происходит анализ работы алгоритма и его корректировка. Так, если алгоритм не справляется с поставленной задачей, модель уменьшается. В случае возникновения "ложных опровержений", т.е при обнаружении ошибки в модели, а не в программе - изменяется модель.*

2.2 Неформально о моделировании

Рассмотрим типы систем, проверяющих корректность работы программы.

1. Одноразовый запуск (проверка input-output). *Программе на вход подается входное слово, результат работы сравнивается с заранее известным корректным выходным словом. В качестве примера можно привести функциональное тестирование.*

2. Реагирующая система с бесконечным временем работы. *Данный тип тестирующих систем работает в постоянном режиме, отслеживая возможные сбои в работе программы. Модель данной реагирующей системы представима в виде множества состояний и множества возможных переходов между состояниями.*

2.3 Модель Крипке

Определение 1
Пусть AP — множество атомарных высказываний.
Модель Крипке над AP — четверка $M = (S, S_0, R, L)$, в которой:

1. S - конечное множество состояний

2. $S_0 \subseteq S$ — множество начальных состояний

3. $R \subseteq S \times S$ отношение переходов

4. $L : S \to 2^{AP}$ — функция истинности

Определение 2
Путем в модели Крипке называется последовательность $\pi = s_0 s_1 \dots$ из состояния s, если $s_0 = s$ и для всех i выполнено $R(s_i, s_{i+1})$.
Многие классы программ могут быть сведены к модели Крипке. Основными из них являются:

1. Булевы (логические) схемы
2. Последовательные программы
3. Параллельные программы

2.4 Темпоральные логики

В предыдущем разделе мы рассмотривали модель Крипке, которая служит для описания программ. Для описания требований к программе используются темпоральные логики. Темпоральные логики являются языком, на котором удобно формулировать утверждения, использующие понятия времени. Допустим, у нас имеется набор переменных, которые могут изменяться в процессе работы программы. Темпоральные логики позволяют формулировать утверждения типа:

- Значение a всегда будет равно значению b
- Наступит момент, когда c станет равно 0
- Значение d принимает значение 1 бесконечно много раз

В данной статье будут рассмотрены следующие два типа темпоральных логик:

1. CTL
2. CTL*

Аббревиатура CTL расшифровывается как **Computation Tree Logic**
Для того чтобы охарактеризовать темпоральную логику, определяют ее синтаксис и семантику. Введем понятия синтаксиса и семантики темпоральной логики.
Определение 3
Синтаксисом темпоральной логики являются правила составления формальных утверждений.
Определение 4
Семантикой темпоральной логики являются интерпретации данных формальных утверждений.

2.5 Темпоральная логика CTL*

Опишем синтаксис и семантику темпоральной логики CTL*
Синтаксис CTL* включает в себя следующие наборы кванторов пути и темпоральных операторов:

- Кванторы пути
  1. **A**. Квантор всеобщности, указывающий на то, что данное свойство выполнено для всех путей.
  2. **E**. Квантор существования, указывающий на то, что данное свойство выполнено для некоторого пути.
- Темпоральные операторы.
  Темпоральная логика CTL* включает в себя пять темпоральных операторов:
  1. **X** : (Ne**x**t time operator).
     Унарный оператор, указывающий на то, что данное свойство выполняется на следующем состоянии нашего пути.
  2. **G** : (**G**lobal time operator)
     Унарный оператор, указывающий на то, что данное свойство выполняется для каждого состояния нашего пути.
  3. **F** : (**F**uture time operator)
     Унарный оператор, указывающий на то, что данное свойство выполняется на некотором состоянии нашего пути.
  4. **U** : (**U**ntil time operator)
     Бинарный оператор, указывающий на то, что первое свойство выполняется для всех состояний пути, предшествующих состоянию, где выполняется второе свойство.
  5. **R** : (**R**elease time operator)
     Бинарный оператор, указывающий на то, что второе свойство выполняется для всех состояний, следующих до состояния s включительно, в котором выполняется первое свойство.

В логике CTL* выделяют следующие два типа формул:

1. Формулы состояний
2. Формулы пути

Формулы состояния описывают свойства соответствующего состояния, аналогично, формулы пути описывают свойства соответствующего пути.
Приведем синтаксис формул состояний и синтаксис формул пути.

1. Синтаксис формул состояний
   - Если $p \in AP$, то p — формула состояния

- Если f и g — ф.с., то $\neg f$, $f \vee g$, $f \wedge g$ — ф.с.
- Если f — формула пути, то $\mathbf{A}f$ и $\mathbf{E}f$ — формулы состояния

2. Синтаксис формул пути

   - Если f — формула состояния, то f — формула пути
   - Если f и g — формулы пути, то $\neg f$, $f \vee g$, $f \wedge g$, $\mathbf{X}f$, $\mathbf{G}f$, $\mathbf{F}f$, $f\mathbf{U}g$, $f\mathbf{R}g$ — формулы пути

Приведем несколько примеров формул пути.

- **X** : (Ne**x**t time operator).
  $powerOn \quad X \quad lightOng$. Электроэнергия включена, и на следующем состоянии нашего пути включится свет.

- **G** : (**G**lobal time operator)
  $G \quad possibleToChangeBlub$. На каждом состоянии нашего пути есть возможность поменять лампочку.

- **F** : (**F**uture time operator)
  $F \quad lightOn$. В пути существует состояние, на котором свет будет включен.

- **U** : (**U**ntil time operator)
  $lightOff \quad U \quad lightOn$. Свет выключен во всех состояниях нашего пути, предшествующих состоянию, в котором свет включен.

- **R** : (**R**elease time operator)
  $changeBlub \quad R \quad brokenBlub$. Лампочка будет сгоревшей до тех пор, пока мы ее не поменяем.

Модель Крипке может быть рассмотрена как диаграмма переходов с конечным набором начальных состояний. Модель Крипке часто представляют в виде леса, состоящего из деревьев вычислений (forest of computation trees). Для каждого начального состояния такой лес деревьев может быть построен путем разворачивания графа в дерево конечной или бесконечной высоты.

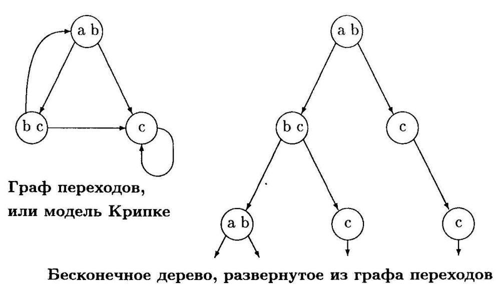

рис. 1

В этом примере наша модель Крипке имела только одно начальное состояние, поэтому получившийся лес состоит только из одного дерева вычислений. В общем случае количество деревьев вычислений равняется количеству начальных состояний в модели Крипке.
Введем несколько новых обозначений для описания семантики логики CTL*

- $\boxed{M, s \models f}$ — формула состояния f выполненана на модели M со стартовой вершиной s.
- Отношение $\models$ определяется естественным образом индукцией по строению формулы.

3 Алгоритм верификации CTL

Определение 5
Логикой CTL называется сужение логики CTL*, допускающая только конструкции вида:

- $\neg f$
- $f \vee g$
- $\mathbf{EX}f$
- $\mathbf{EG}f$
- $\mathbf{E}[f\mathbf{U}g]$

3.1 Постановка задачи

Рассмотрим постановку задачи верификации модели темпоральной логики CTL. Пусть имеется модель Крипке $M = (S, R, L)$ и формула темпоральной логики f. Требуется найти множество $\{s \in S \quad | \quad M, s \models f\}$

3.2 Алгоритм

Идея алгоритма верификации модели заключается в следующем:

- Найти множество всех подформул состояния f
- Для каждого состояния $s \in S$ создать список выполненных подформул
- Вести построение "индукцией по построению f"

Рассмотрим несколько примеров.

1. **Простой случай**
    - Нам уже даны выполняющие множества для атомарных формул

- Знаем выполняющее множество для $f \Rightarrow$ построим и для $\neg f$
- Знаем выполняющие множества для f и $g \Rightarrow$ построим и для $f \vee g$
- Сделаем один шаг назад от выполняющего множества f — получим выполняющее множество для $\mathbf{EX}f$
- $\mathbf{E}[f\mathbf{U}g]$ — отмечаем все g и строим деревья обратных путей вдоль f-вершин

2. **Построение выполняющего множества для оператора** EGf

   - Выкидываем вершины, не выполняющие f
   - Находим компоненты сильной связности [Алгоритм Тарьяна]
   - Строим обратные деревья от этих компонент
     Трудоемкость итогового алгоритма составляет $O(|f|(|S| + |R|))$