Формальные модели программ

Введение в Model Checking. Темпоральные логики LTL и CTL

Software Model Checking via Counterexample Guided Abstraction Refinement

maxim.krivchikov@gmail.com

Материалы курса: https://maxxk.github.io/formal-models-2015/

Источники

Классическая книга:

Clarke E.M., Grumberg O., Peled D.A. Model Checking. Cambridge, Mass: The MIT Press, 1999. 314 p.

Э.М. Кларк и др. Верификация моделей программ. Model checking. М.:МЦНМО, 2002.

Презентации:

Bor-Yuh Evan Chang

https://www.cs.colorado.edu/~bec/courses/csci5535/meetings/meeting03.pdf

https://www.cs.colorado.edu/~bec/courses/csci5535/meetings/meeting04.pdf

Model checking

Верификация моделей программ

— подход, обеспечивающий выполнение требуемых свойств путём исчерпывающей проверки всего возможного множества состояний.

На настоящее время наиболее успешно используемый на практике подход к формальной верификации программного обеспечения.

Общие принципы Model Checking

Задача — верификация свойств программ или поиск ошибок в программах.

Автоматизированный подход, который

верифицирует модели состояний и переходов,

обеспечивает выполнение *темпоральных* свойств.

Выполняет фальсификацию гипотезы путём генерации контрпримеров.

Фальсифицируемость

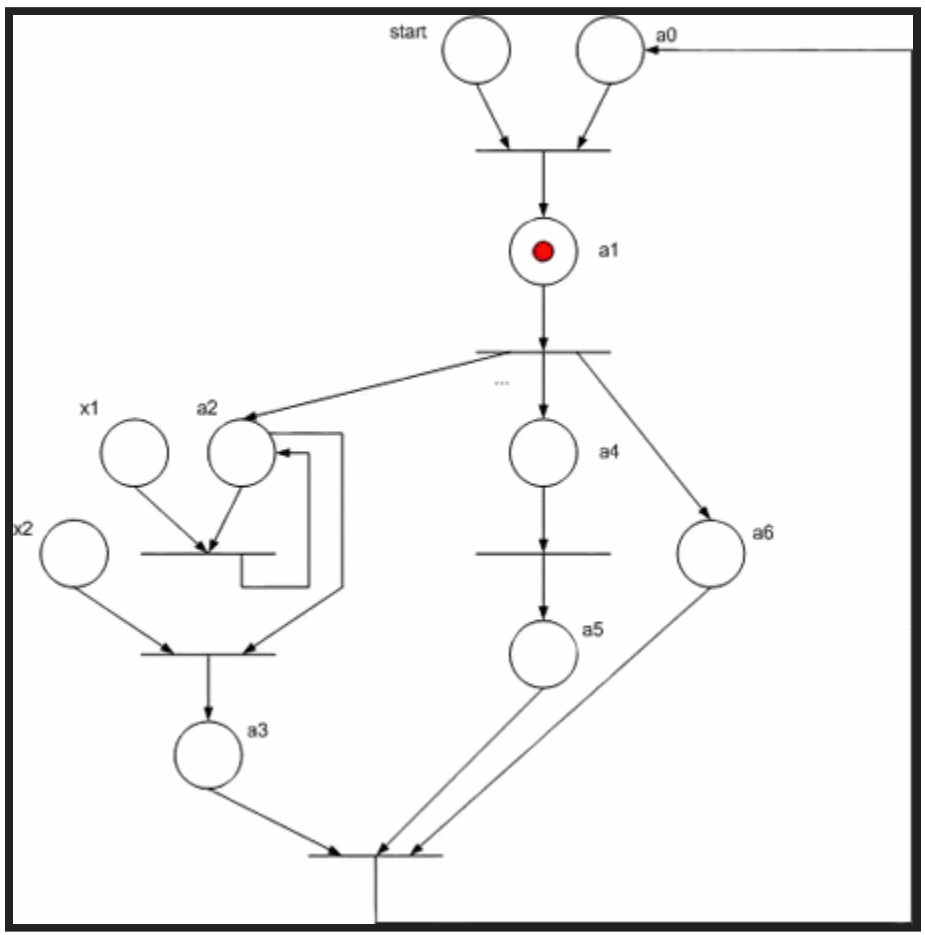
К. Поппер, 1935 г. — современный критерий научности теории (гипотезы).

Теория научна, если она удовлетворяет критерию фальсифицируемости — принципиально может быть поставлен тот или иной эксперимент, один из возможных исходов которого опровергает теорию.

Модели состояний и переходов

Простейший пример — конечные автоматы.

Более сложный — сети Петри:



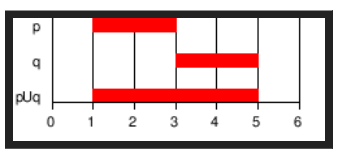
Темпоральная логика

Обычные логические операторы

Оператор	**Описание**	**Значения**
$\neg A$	**Не** A	$\neg 0 = 1, \neg 1 = 0$
$A \wedge B$	A **и** B	$(0 \wedge 0) = (0 \wedge 1) = (1 \wedge 0) = 0; (1 \wedge 1) = 1$
$A \vee B$	A **или** B	$(0 \vee 0) = 0; (0 \vee 1) = (1 \vee 0) = (1 \vee 1) = 1$
$A \rightarrow B$	Из A **следует** B	$(0 \rightarrow 0) = (0 \rightarrow 1) = (1 \rightarrow 1) = 1; (0 \rightarrow 1) = 0$

Темпоральная логика

В темпоральной логике переменные — последовательности логических переменных, т.е. A на самом деле — это множество состояний A_i в i-е моменты времени.

Оператор	Описание	Диаграмма
X α , ◯α	α будет верно в **следующем** состоянии	
G α , □ α	α верно **во всех следующих** состояниях	
F α , ◇ α	α будет верно **в одном из следующих** состояний	
ε U δ	ε верно **до некоторого** состояния, **после которого** становится верно δ	

https://en.wikipedia.org/wiki/Temporal_logic

Темпоральная логика

Позволяет выразить свойства, связанные со временем, такие как «инвариантность» и «гарантированная достижимость».

α — **инвариант** (invariant) для данного состояния i, если начиная с этого состояния α выполняется во всех последующих на любом возможном пути исполнения.

α **гарантированно достижимо** (eventual, «когда-нибудь произойдёт») из данного состояния i, если на любом пути исполнения начиная с этого состояния найдётся хотя бы одно состояние, в котором выполняется α.

Пример — параллельная программа

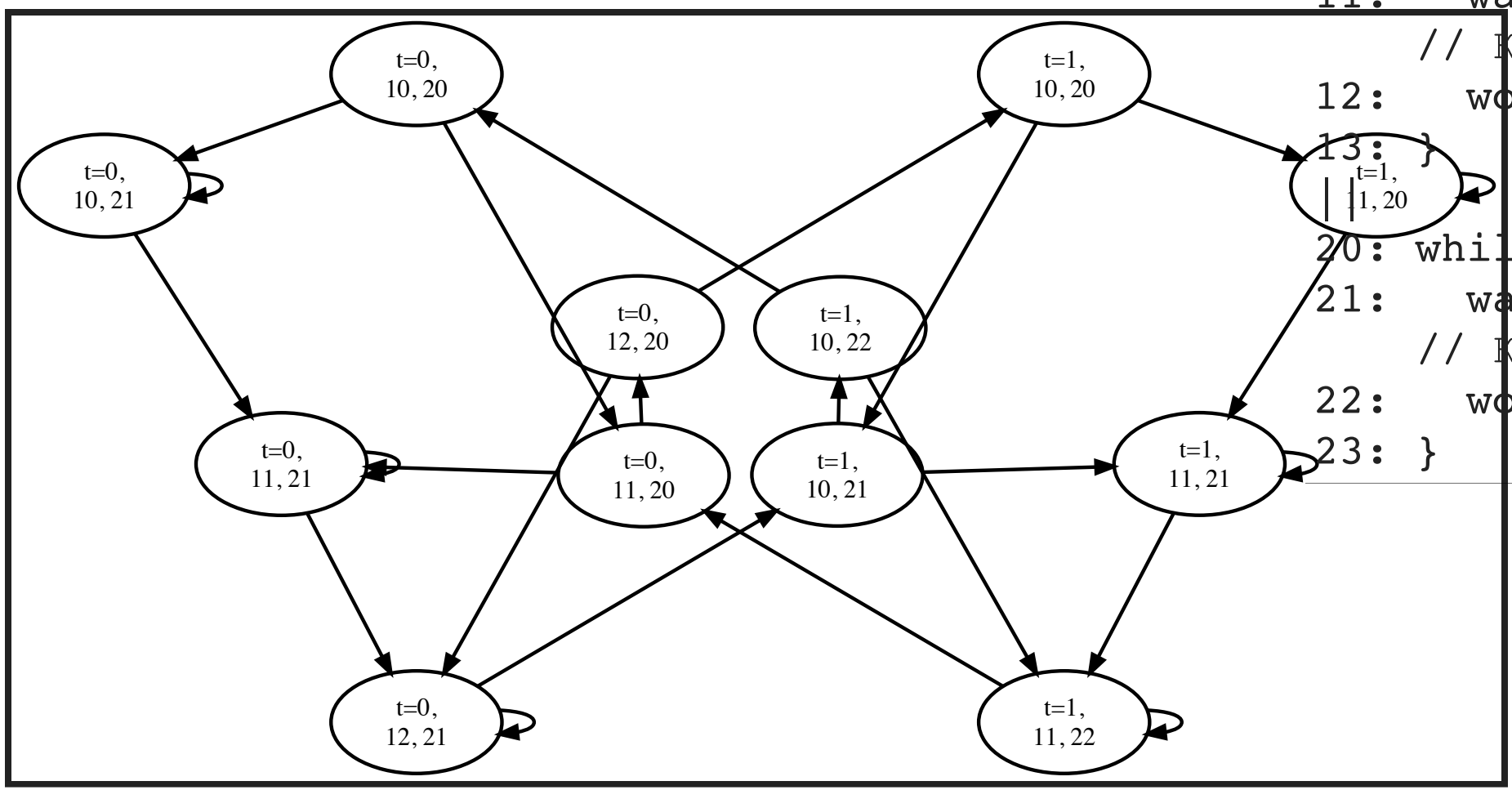
- Два процесса выполняются параллельно.
- В общей памяти задана переменная `turn`.
- Общая переменная используется для определения критической секции, в которой в каждый момент времени может находиться только один поток.

```
10: while(true) {
11:   wait(turn == 0)
    // Критическая секция
12:   work(); turn = 1
13: }
```

```
20: while(true) {
21:   wait(turn == 1)
    // Критическая секция
22:   work(); turn = 0
23: }
```

Граф достижимых состояний примера

Состояние — все возможные сочетания значения переменной `turn` и счётчиков инструкций двух процессов.




```
10: while(tr
11:   wait(t
      // Крити
12:   work()
13: }
20: while(tr
21:   wait(t
      // Крити
22:   work()
23: }
```

Модель состояний и переходов

Напоминает недетерминированный автомат:

$$T = (S, I \subseteq S, R \subseteq S \times S, L : S \to 2^{AP})$$

- S — множество состояний (конфигураций)
- I — начальные состояния
- R — переходы
- L — функция аннотаций (labeling function)
- AP — множество атомарных утверждений о программе (например, $x = 5$)
  - описывают основные утверждения
  - для программ обычно — в терминах значений переменных
  - функция аннотаций помечает каждое состояние множеством истинных в этом состоянии атомарных утверждений

Примеры свойств

1. во всех достижимых конфигурациях системы два процесса *никогда не находятся одновременно в критической секции*
   (`pc1=12`, `pc2=22` — атомарные утверждения «процесс находится в критической секции»)
   `Invariant(¬(pc1 = 12 ∧ pc2=22))`
2. первый процесс *обязательно попадёт* в критическую секцию
   `Eventually(pc1 = 12)`

Пути исполнения

Путь в модели состояний и переходов — это бесконечная последовательность состояний, последовательно связанных переходами:

$$(s_0, s_1, s_2, \ldots) \qquad (s_i, s_{i+1}) \in R$$

Путь исполнения — путь, который начинается из начального состояния ($s_0 \in I$)

Дополнительно к уже рассмотренным операторам темпоральной логики можно ввести квантификацию по путям (∀ x, ∃ x)

Отношения выводимости

В «обычной» логике, как на прошлом занятии:
$A, B, C \vdash D$ — D выводимо при наличии выводов A, B, C.

В темпоральной логике вводится следующее отношение:
$h \models p$

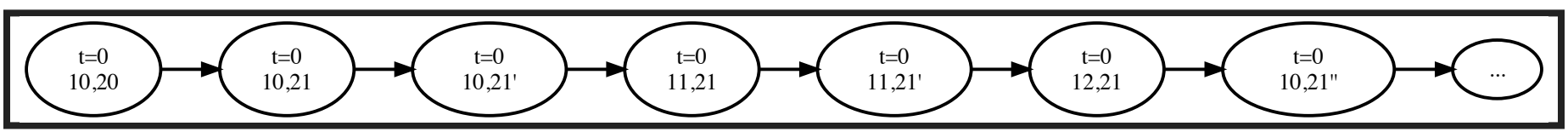
Для пути h выполняется предикат p.

Пример (1):

```
∀ h . h ⊨ G (¬ (pc1 = 12 ∧ pc2 = 22))
```

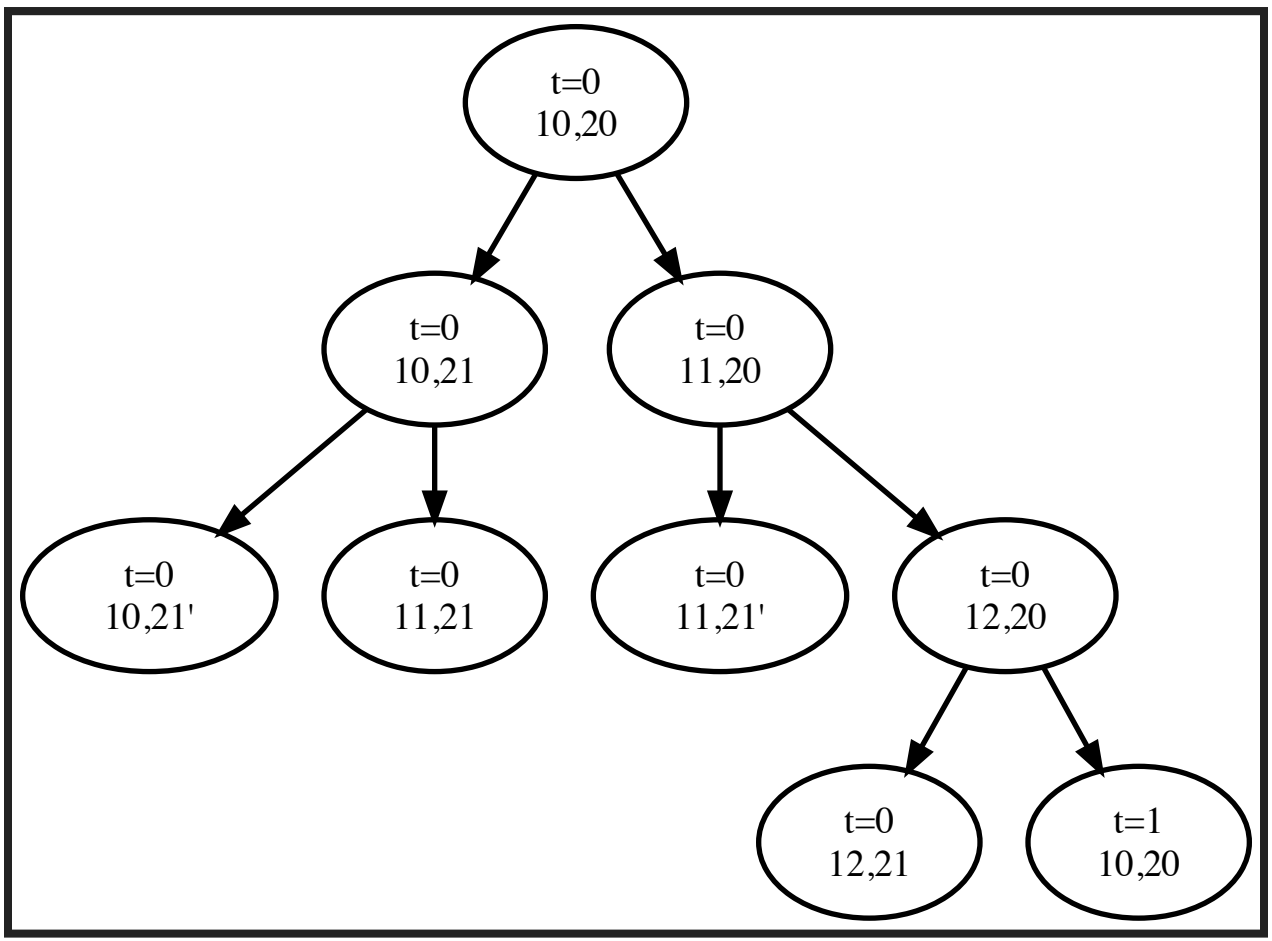
Linear Time Logic

LTL (Linear Time Logic) — путь рассматривается как линейная последовательность переходов, значение предикатов определено на путях.



Computational Tree Logic

CTL (Computational Tree Logic) — рассматривается дерево всех возможных путей; именно в этой логике используется квантификация по путям.



Вычислительная сложность

Для множества состояний S и переходов R проверить, удовлетворяет ли модель формуле f можно за время:

$$O(|f| \cdot (|S| + |R|))$$

Сложность растёт линейно по отношению к размеру модели состояний и переходов.

Однако размер модели состояний и переходов растёт экспоненциально по отношению к количеству переменных и числу параллельных процессов.

Основная проблема model checking — проблема «комбинаторного взрыва» количества состояний.

Проверка утверждений с кванторами

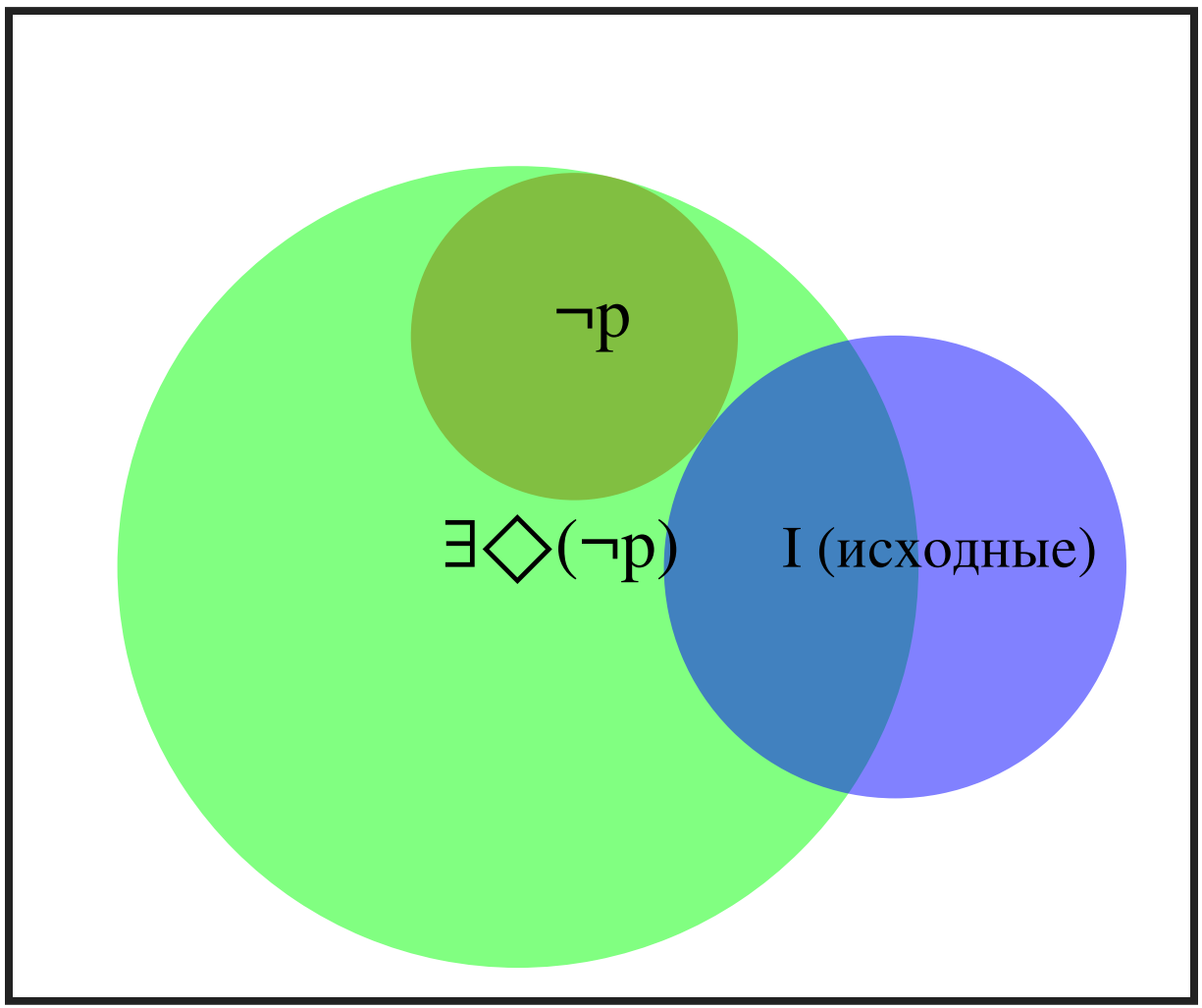
Квантифицированные свойства = неподвижные точки

$\forall\Box(p) \equiv \exists\Diamond(\neg p)$ (p — глобальный инвариант = не существует состояния, из которого достижимо другое состояние, в котором p — не выполняется)

Алгоритм:

1. Положим Func : $2^S \to 2^S$, Func(Z) ≡ ¬p ∪ состояния, из которых Z достижимо за один шаг.
2. Вычислим ∃F(¬p) как наименьшую неподвижную точку Func:
   - начинаем с Z=⊘ , применяем Func, пока не дойдём до неподвижной точки (Func(Z') = Z')

Неподвижные точки



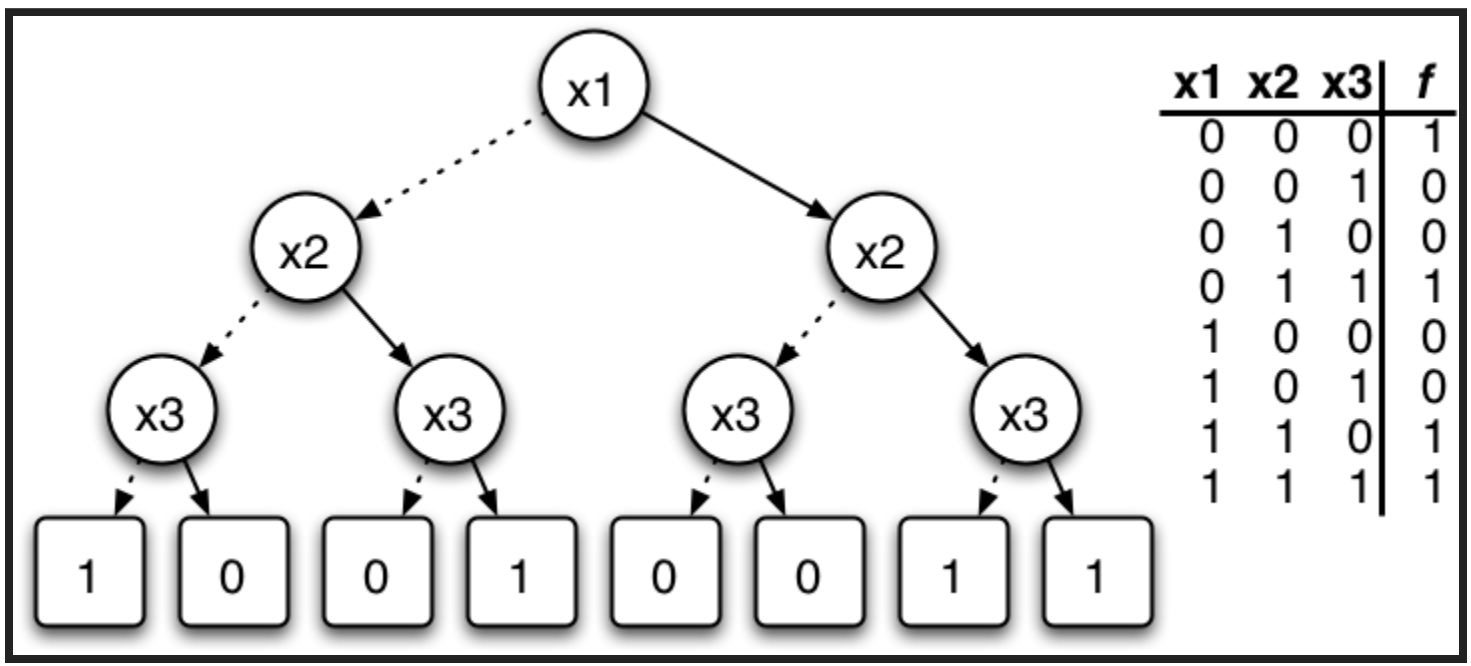
Symbolic Model Checking

Символьная верификация моделей

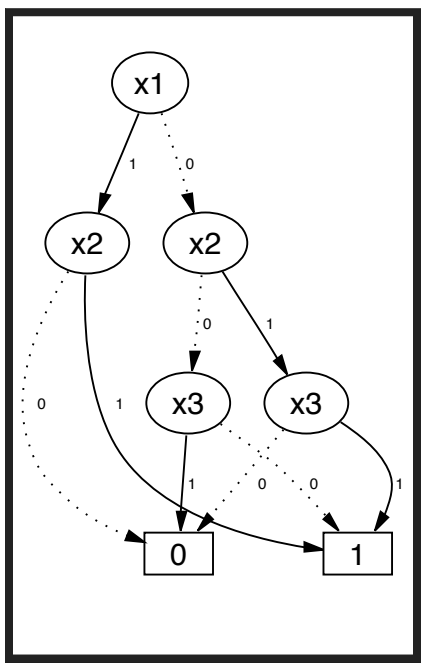
- множества состояний и отношение перехода представляются в виде булевых формул
- неподвижные точки можно вычислить итеративной подстановкой в такие формулы
- пример средства — SMV (Symbolic Model Verifier), выполняет проверку свойств в логике CTL с использованием бинарных диаграмм решений (Binary Decision Diagrams, BDD)
- BDD используются для представления множества в виде функции принадлежности

Binary Decision Diagrams

Бинарные диаграммы решений



x1	x2	x3	*f*
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



Binary Decision Diagrams

- дизъюнкция и конъюнкция формул вычисляется не более чем за квадратичное время
- отрицание — за константное время (очевидно — поменяем местами 0 и 1)
- проверка эквивалентности формул — константа или линейное время
- образ (выполнимость; множество всех значений переменных, при которых формула выполняется) — может быть экспоненциальным

Software Model Checking via Counterexample Guided Abstraction Refinement (SLAM)

приблизительно — «Верификация моделей программ с использованием абстракции и уточнения по контрпримерам»

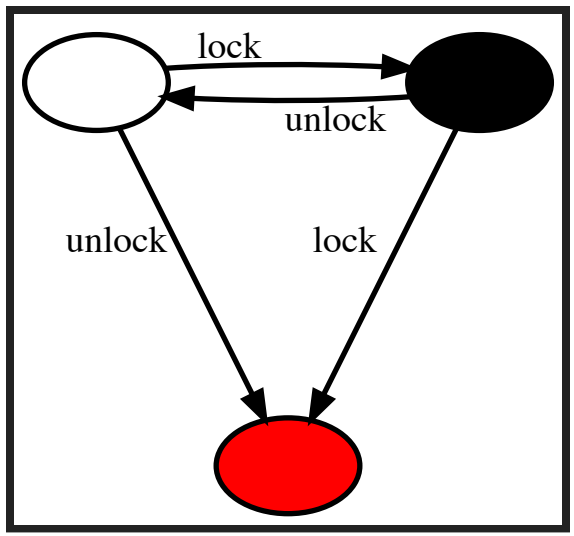
Реализация — анализатор BLAST (Berkeley Lazy Abstraction Software verification Tool), разработан в Беркли; на настоящее время поддерживается в России, в ИСП РАН.

Статья с примером работы BLAST

Общие принципы SLAM

1. **Входные данные**
   - программа (на языке C!)
   - частичная спецификация (задаётся в виде модели состояний и переходов, более-менее в терминах атомарных утверждений — «программа использует блокировки корректно», а не «программа реализует Web-сервер»)
2. **Выходные данные**
   - программа удовлетворяет спецификации (в некоторых случаях возможно получить доказательство)
   - есть контрпример — конкретный путь исполнения программы, нарушающий спецификацию
3. **Схема работы**
   - преобразуем программу в набор булевскую программу
   - проверяем выполнение спецификации
   - нет ошибок в булевской программе — значит нет ошибок в оригинальной
   - возможны ложные положительные

Свойство 1 — двойная блокировка



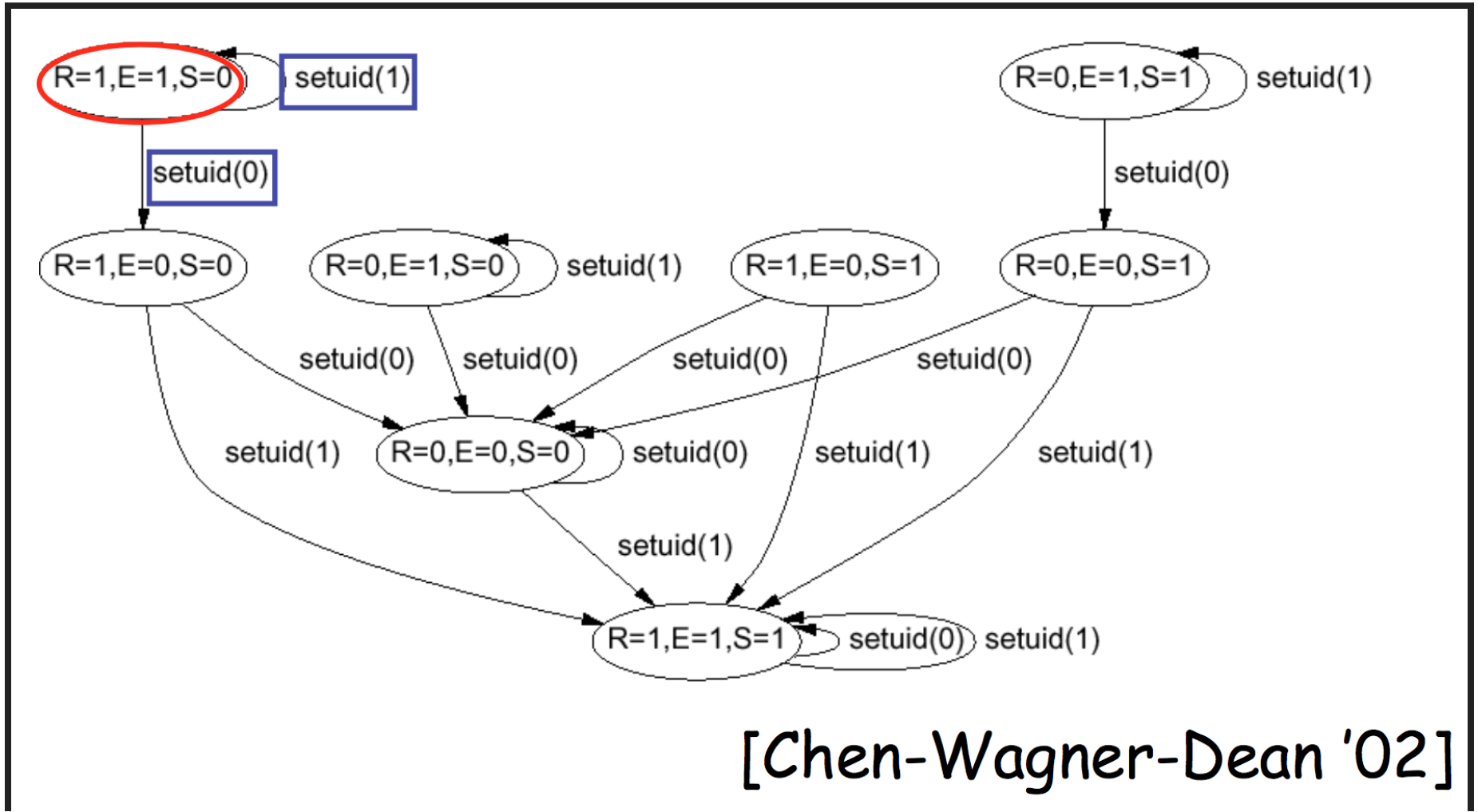
Повторный вызов `lock` или `unlock` приводит к deadlock.

Свойство: вызовы `lock` и `unlock` должны чередоваться.

Свойство 2

понижение привилегий суперпользователя

Произвольные пользовательские приложения не должны быть запущены с привилегиями суперпользователя. Свойство: при вызове `execv` всегда `suid ≠ 0`.



Псевдокод SLAM

```
SLAM(Program p, Spec s) =
    Program q = incorporate_spec(p,s);
    PredicateSet abs = { };
    while true do
        BooleanProgram b = abstract(q,abs);
        match model_check(b) with
            | No_Error ! print("no bug"); exit(0)
            | Counterexample(c) !
              if is_valid_path(c, p) then
                  print("real bug"); exit(1)
              else
                  abs ← abs U new_preds(c)
    done
```

Выводы

- Model checking — хороший, работающий на практике подход
- Проблема экспоненциального взрыва состояний
- Проблема абстракции и уточнения
- Главная (на мой взгляд) проблема — очень низкоуровневое описание свойств

Обычно хочется всё-так доказывать свойства вида «эта программа — web-сервер, понимающий стандарт HTTP 1.1», а лучше даже — «этот комплекс программ реализует сайт, на котором обычный пользователь не может получить доступ к закрытым от него данным». Это не всегда возможно, но нужно стремиться к этому.

Задача со звёздочкой

Задача 4.1** Представить пример доказательства не совсем тривиального свойства для не совсем тривиальной программы на используемых на практике языках программирования :) (> 40 строк кода без учёта комментариев и пустых строк) с использованием BLAST или других средств model checking.

Например, интересно было бы проверить корректность реализации сортировки.