НАДЁЖНОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Лекция 1:
Введение в разработку надёжного ПО

ВМиК МГУ им. М.В. Ломоносова,
Кафедра АСВК, Лаборатория Вычислительных Комплексов
Ассистент Волканов Д.Ю.

Краткий план лекций

- Как избежать следующих ситуаций:

План лекции

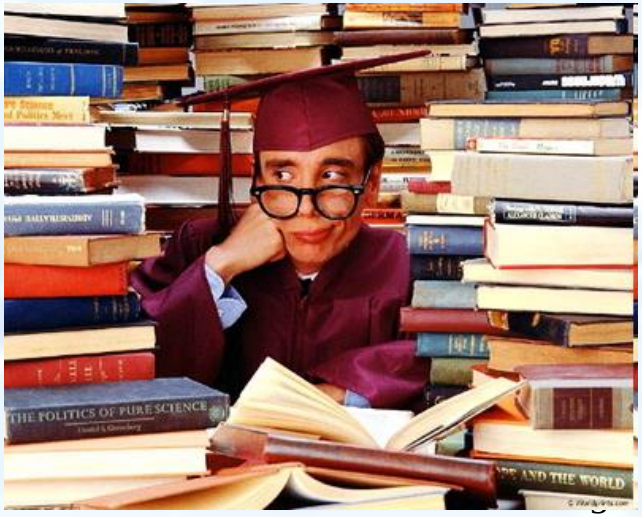
- Введение в курс
- Надёжность ПО
- Разработка надёжного ПО
- Введение в процесс разработки надёжного ПО

О чём этот курс

- Концепции и связи
- Аналитические модели и средства поддержки
- Методы для увеличения надёжности ПО:
  - Верификация
  - Статический анализ
  - Тестирование
  - Расчёт надёжности
  - *Анализ статистики ошибок*
  - *Безопасность*

Как это сдавать

- В январе экзамен
- В процессе семестра с/р
  - 80% - 100% -> автоматом 5
  - 60% - 79,9% -> 4
  - 40% - 59,9% -> 3
  - 20% - 39,9% -> 2
  - 0% - 19,9% -> 1
  - экзамен письменно-устный

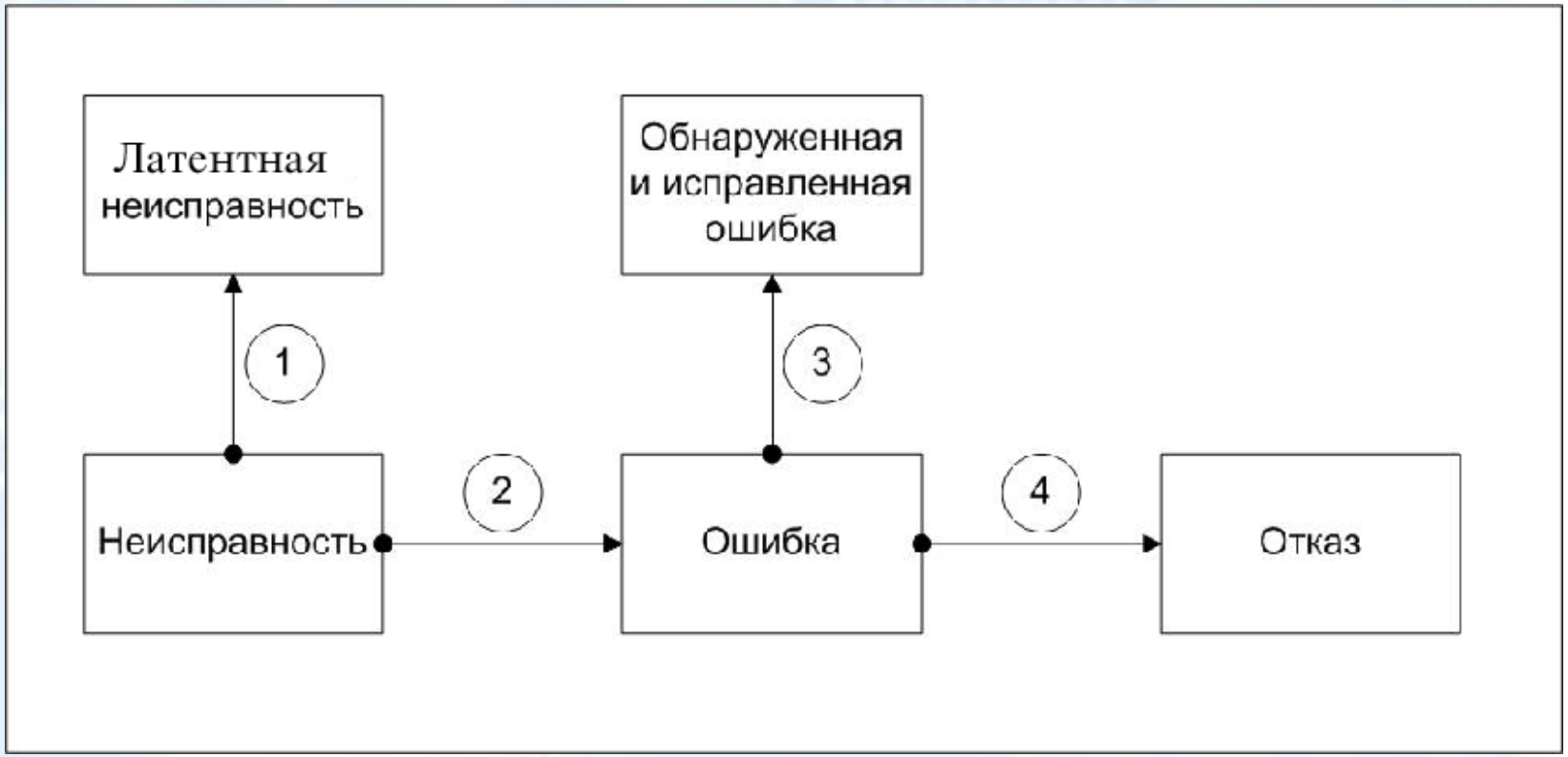


Практикум

- Несколько заданий, по мере прохождения курса
- Первое задание – 16.09
- Умение работать в программных средствах
- Каждое задание - 10 баллов
- Каждая неделя сверх

Дедлайна –N задания

Неисправность, ошибка, отказ

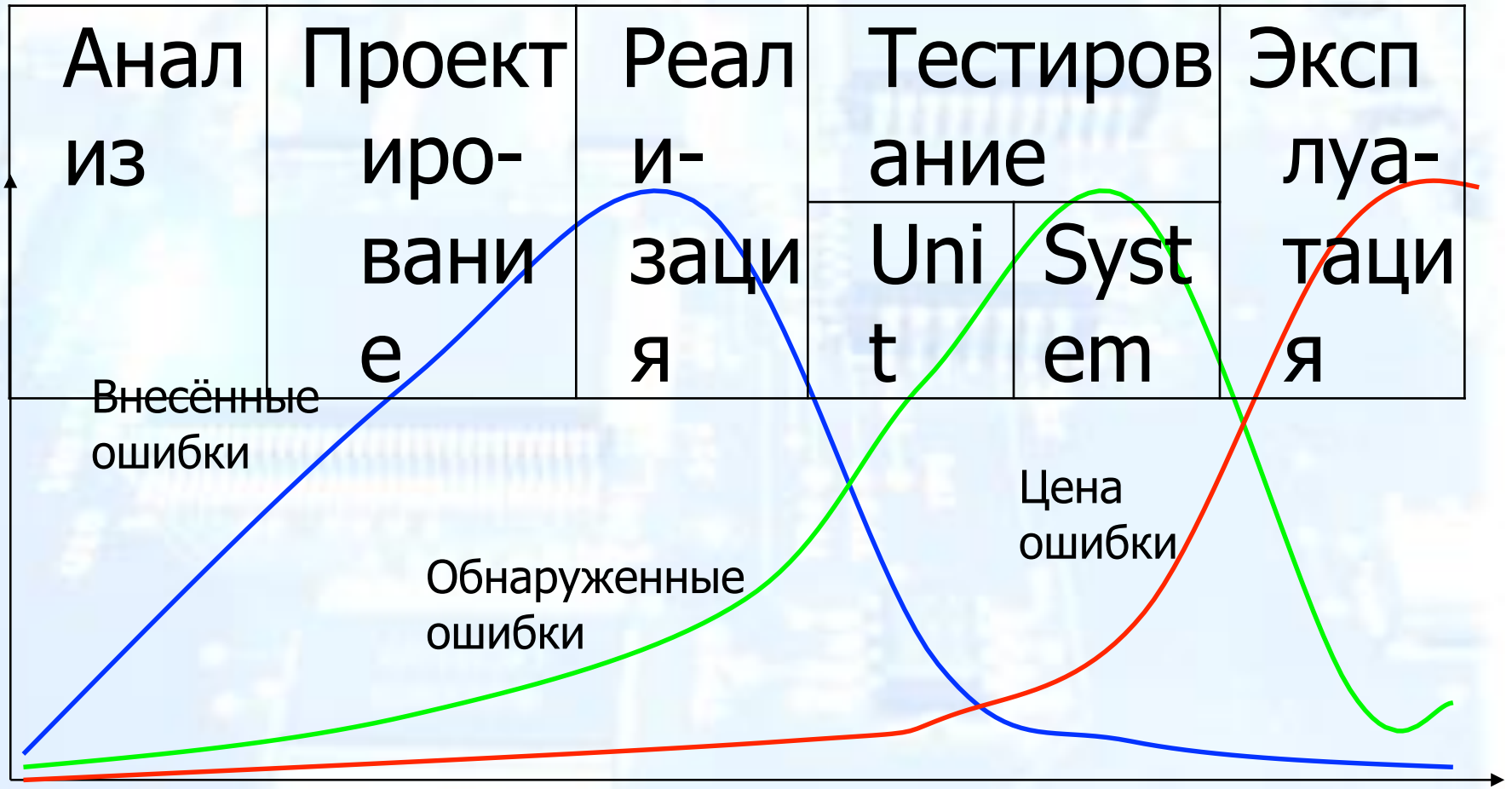


Пример, поясняющий понятие отказа

```
Program definitions;
var s:integer;
begin
   read(s);
   s:=s*2; {неисправность}
L: {ошибка}
   s:=s mod 5;
   write(s); {отказ}
end.
```

Ожидаемые данные	Текущие данные
6	6
36	12
36	12
1	2
1	2

Разработка программы



Правильность программ

- Нет требований – нет правильности
- Ошибка – несоответствие требованиям
- Ошибки:
  - в формулировке требований,
    building the wrong system,
  - в соблюдении требований,
    building the system wrong.

Правильность программ

- **Валидация** – исследование и обоснование того, что спецификация ПО и само ПО через реализованную в нём функциональность удовлетворяет требованиям пользователей,
- **Верификация** – исследование и обоснование того, что программа соответствует своей спецификации.

найти ошибки или доказать, что их нет

NB: ошибки формализации требований

Цена ошибки

- В ряде приложений ошибки не критичны:
  - сводятся к лёгким моральным травмам,
  - возможность обнаружения сбоя и восстановления,
  - возможность быстрого исправления ошибки.

Цена ошибки

- Системы с повышенными требованиями к надёжности (Safety-critical)
- Ошибки приводят к:
  - Гибели или травмам людей,
  - Крупным финансовым потерям
  - Ущербу окружающей среде
  - Итд.

Цена ошибки: Ariane-5

- Июнь 1996 года, взрыв ракеты спустя 40 сек. после старта,
- Ущерб – $500млн (разработка – $7 млрд.),
- Причина – 64bit float -> 16bit int.

Цена ошибки: Patriot

- Февраль 1991 года, Patriot промахнулся мимо ракеты Scud,
- Ущерб – 28 убитых, >100 раненых,
- Причина – ошибка округления из-за 24bit fixed, Scud успел пролететь 500м.

Цена ошибки: Sleipner A

- август 1991 года, Северное море, платформа Sleipner A затонула после разрушения основания,
- Ущерб – $700 млн, землетрясение силой 3 балла,
- Причина – ошибка округления при моделировании платформы.

Кое-что посвежее (2010)

- Ошибки в ПО автомобилей: Toyota Prius (нарушен директивный интервал включения ABS), Chrysler (можно вытащить ключ, не переключившись в режим парковки)
- Ошибки в программе обработки заявлений доноров органов в UK (у 25 человек взяли не те органы)
- Ошибка в электронной системе налоговой службы США (не обслуживались 64-летние люди)
- Ошибка в антивирусе McAfee (системный файл Windows распознан как вредоносный и удалён, бесконечный ребут)
- Отключение Skype (вышла из строя одна из версий клиента, что привело к сбою всей P2P сети)
- Ошибка в апдейте NYSE Euronext (S&P упал на 10%)
- 88 критических ошибок в Android Froyo (доступ к личным данным)





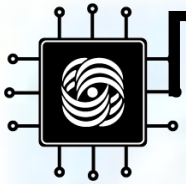
facebook

at&t

GPS

CareFusion

Baxter



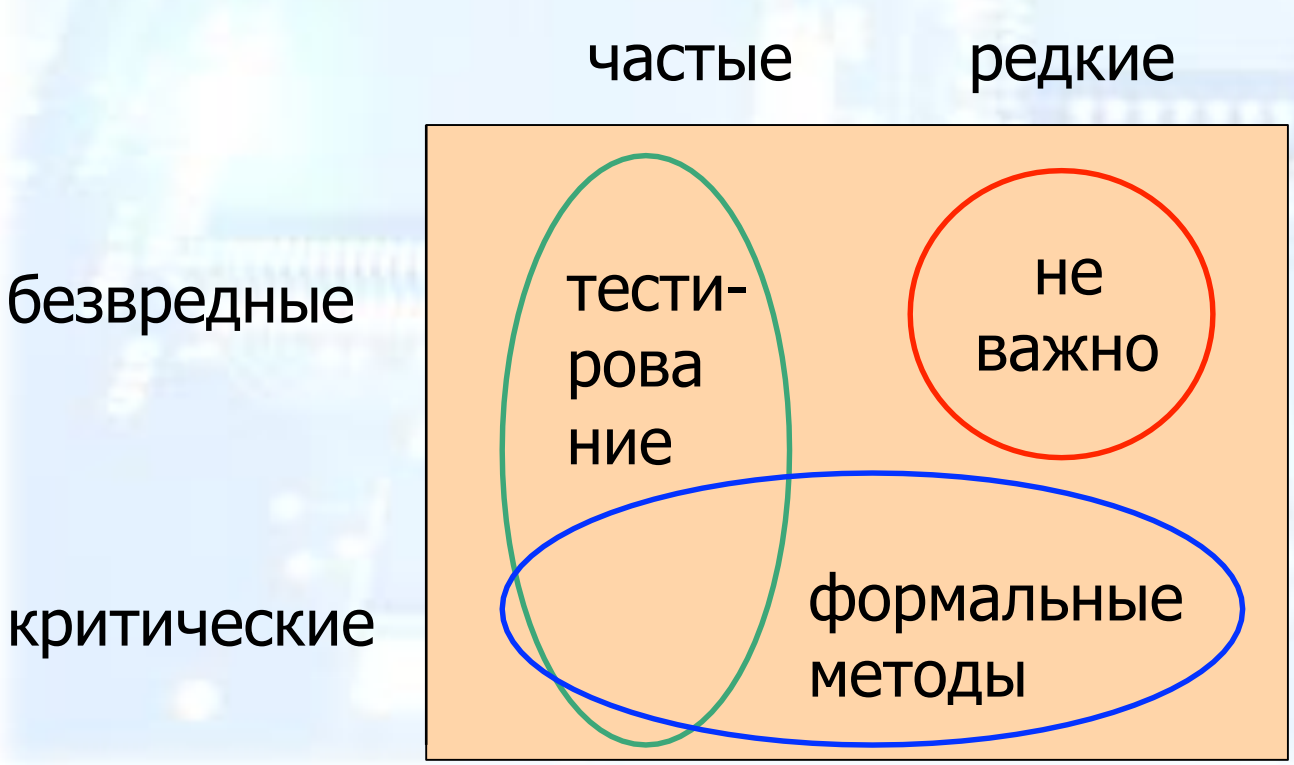
Повышенные требования к надежности

- Не только спутники, самолёты и АЭС!
- Системы, в которых
  - затруднено исправление ошибок (потребительская электроника)
  - велик масштаб использования (та же электроника, важные веб-сервисы)
  - высокая степень доверия человека (augmented reality)
- Критически-важных систем становится всё больше, они становятся более сложными и интероперабельными.

Выборочное тестирование

«Тестирование может показать присутствие ошибок, но не может показать их отсутствия» (с) Дейкстра.

Выявление ошибок



Reminder

ВЕРИФИКАЦИЯ ПРОГРАММЫ В ОБЩЕМ СЛУЧАЕ АЛГОРИТМИЧЕСКИ НЕРАЗРЕШИМА

См., например, задачу останова программы, теорему Райса, etc

Формальные методы

«Использование математического аппарата, реализованного в языках, методах и средствах спецификации и верификации программ»

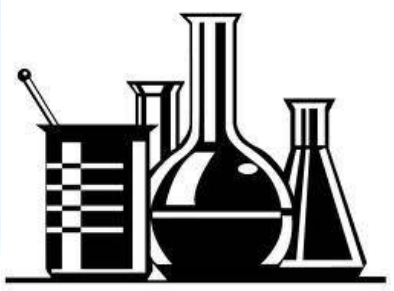
- Методы формальной спецификации
- Методы формальной верификации:
  - Доказательство теорем
  - Верификация на моделях
  - Кое-что ещё

Методы верификации

- «Полное» тестирование
- Имитационное моделирование
- Доказательство теорем
- Статический анализ
- Верификация на моделях
- Динамическая верификация

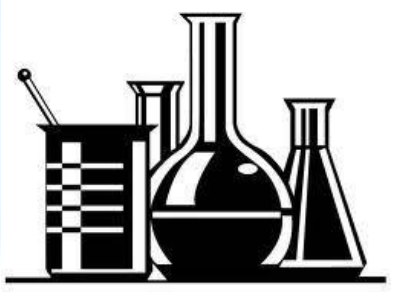
Тестирование

- Обоснование полноты тестового покрытия,
- Метод «чёрного ящика» (ЧЯ) -- полное покрытие входных данных,
- Метод «прозрачного ящика» (ПЯ) -- полное покрытие кода программы.



Тестирование: плюсы

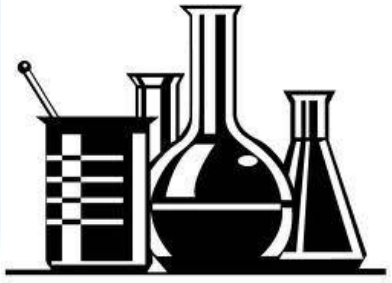
- Проверяется та программа, которая будет использоваться,
- Не требуется (знания) дополнительных инструментальных средств,
- Удобная локализация ошибки.



Тестирование: минусы

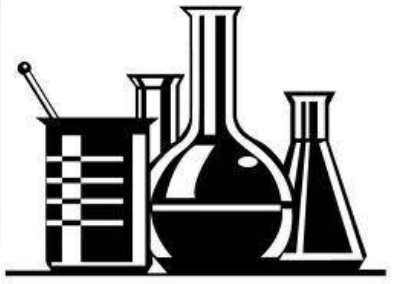
- Не всегда есть условия для тестирования системы,
- Проблема с воспроизводимостью тестов.

частичное решение – имитационное моделирование



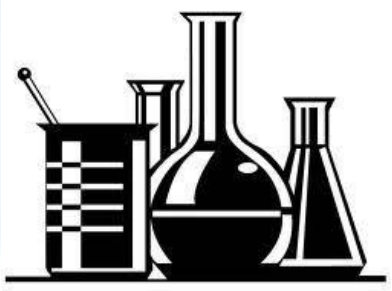
Тестирование: подводные камни

- **Полнота тестового покрытия:**
  - **ЧЯ**: для последовательных программ сложно перебрать все входные данные,
  - **ЧЯ**: для параллельных – очень сложно,
  - **ЧЯ**: для динамических структур данных, взаимодействия с окружением – **невозможно**.
  - **ПЯ**: большой размер покрытия,
  - **ПЯ**: часто **невозможно** построить 100% покрытие,
  - **ПЯ**: полное покрытие **не гарантирует** отсутствия ошибок.



Полное покрытие для черного ящика

- Поиск выигрышной стратегии в шашках:
  - 10^{14} тестов,
  - 18 лет,
  - постоянно работало от 50 до 200 десктопов.



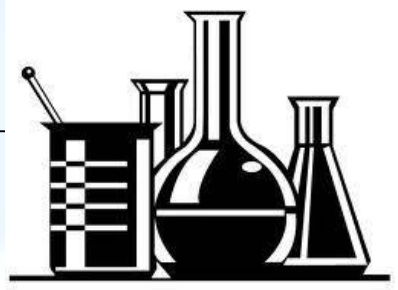
Полное покрытие для прозрачного ящика

```
if (B1) {
   S1;
}
```

- Два теста

```
if (B1) {
   S1;
}
if (B2) {
   S2;
}
```

- Четыре теста

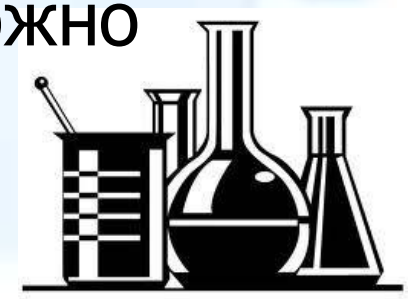


...exp...

Полное покрытие для прозрачного ящика

```
int x = 1;
if  (x==  1)
  std::cout     “Okay” << std::endl;
} else {     <<
  std::cout << “Error” << std::endl;
}
```

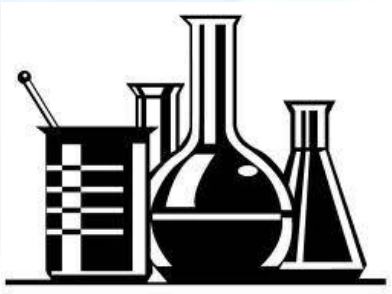
-- полное покрытие кода невозможно



Полное тестовое покрытие – не панацея

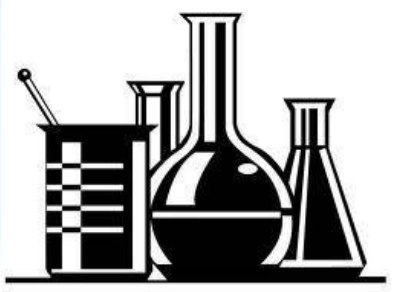
```
int strlen(const char* p) { int
        len = 0;
do  {
++len;
}   while   (*p++); return len;
}
```

«a», «bbb» -- полное покрытие кода, ошибка не найдена



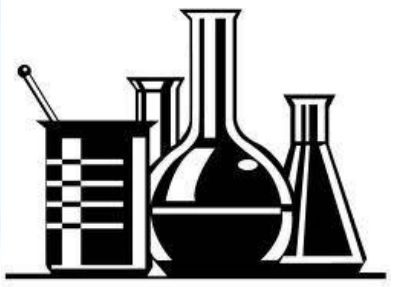
Полнота покрытия: итоги

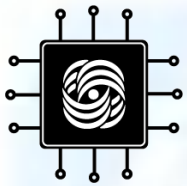
- Полный перебор входных данных – невозможен, плохой критерий,
- Полнота покрытия кода – не гарантирует правильности, плохой критерий,
- Ошибка – ошибочное вычисление системы,
- Полнота в терминах возможных вычислений – хороший критерий.



Кое-что о тестировании

- Как правило, разработка критически важных систем регулируется каким-либо стандартом; Например, RTCA/DO-178B (авионика);
- Стандарты разрабатывались давно, поэтому основной способ верификации там – тестирование;
- Пример обоснования полноты тестового покрытия для критических систем – **MC/DC** (http://techreports.larc.nasa.gov/ltrs/PDF/2001/tm/NASA-2001-tm210876.pdf);
- В стандарт RTCA/DO-178C (2011г) включены model-driven development и верификация





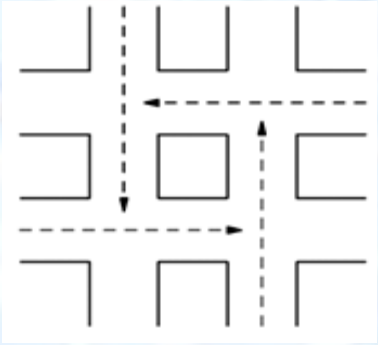
Дополнительные трудности:
Реактивные программы

- Традиционные программы:
  - Завершаются,
  - Описание «вход/выход»,
  - **Число состояний зависит от входных данных и переменных**;
- Реактивные программы:
  - Работают в бесконечном цикле,
  - Взаимодействуют с окружением,
  - Описание «стимул/реакция»,
  - (Не обязательно параллельные),
  - **Дополнительный источник сложности.**

Дополнительные трудности:

Параллельные программы

- Большое количество возможных вычислений,
- Неочевидные ошибки,
- Пример – системы с разделением ресурсов.
- Исключительная ситуация:

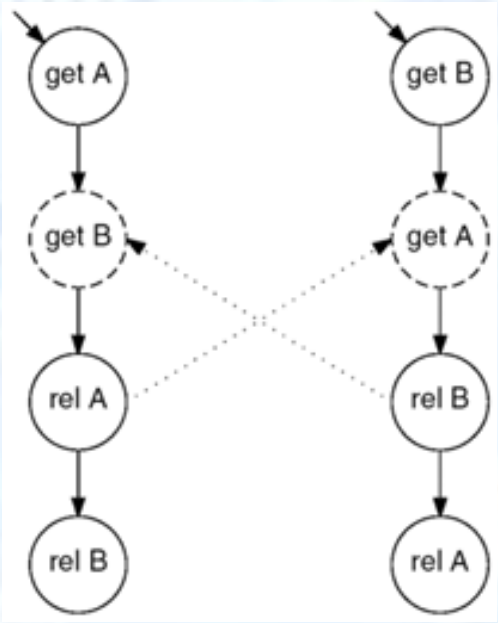


- Правила, реализованные в программах, должны быть универсальны

Системы с разделением ресурсов

Примеры:

- Дорожный траффик,
- Телефонные сети,
- Операционные системы,
- …

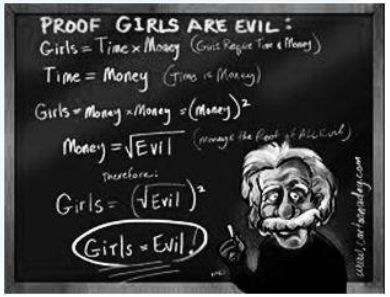


Дополнительные трудности: Параллельные системы

- Новый источник ошибок – совместная работа проверенных компонентов,
- Невоспроизводимость тестов,
- Ограниченные возможности по наблюдению.

Доказательство теорем

- Система и свойства – формулы
- Набор аксиом и правил вывода
- Строится доказательство свойства-теоремы
- Качественный анализ системы



Доказательство теорем

- Система:

$$A = c \bullet a \bullet$$

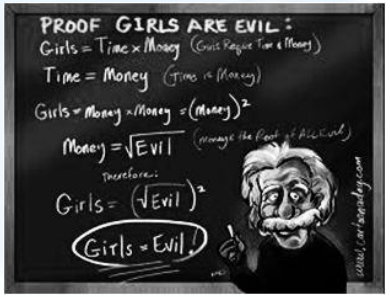
$$B\,B = b \bullet A$$

- Свойство: $a \bullet c$?
- Правила вывода:

$$\frac{S_1 = a_1 \bullet S_2 \vee S_2 = a_2 \bullet S_3}{S_1 = a_1 \bullet a_2 \bullet S_3}, \quad \frac{s_1 \bullet s_2 \vee s_2 \bullet s_3}{s_1 \bullet s_3}$$

Доказательство теорем

- Достоинства:
  - работа с бесконечными пр-вами состояний,
  - даёт более глубокое понимание системы.
- Недостатки:
  - медленная скорость работы,
  - может потребоваться помощь человека (построение инвариантов циклов),
  - В общем случае нельзя построить полную систему аксиом и правил вывода (теорема неполноты Гёделя).

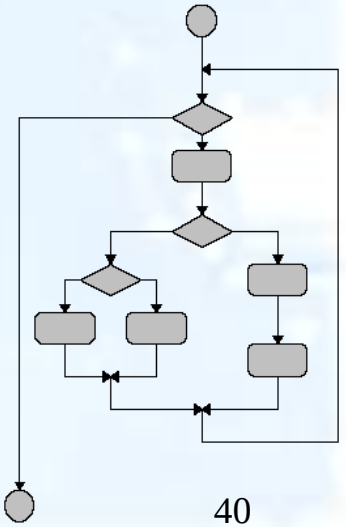


Статический анализ

- Более грубый и прагматичный подход,
- Анализ исходного текста программы без её выполнения,
- В общем случае задача неразрешима

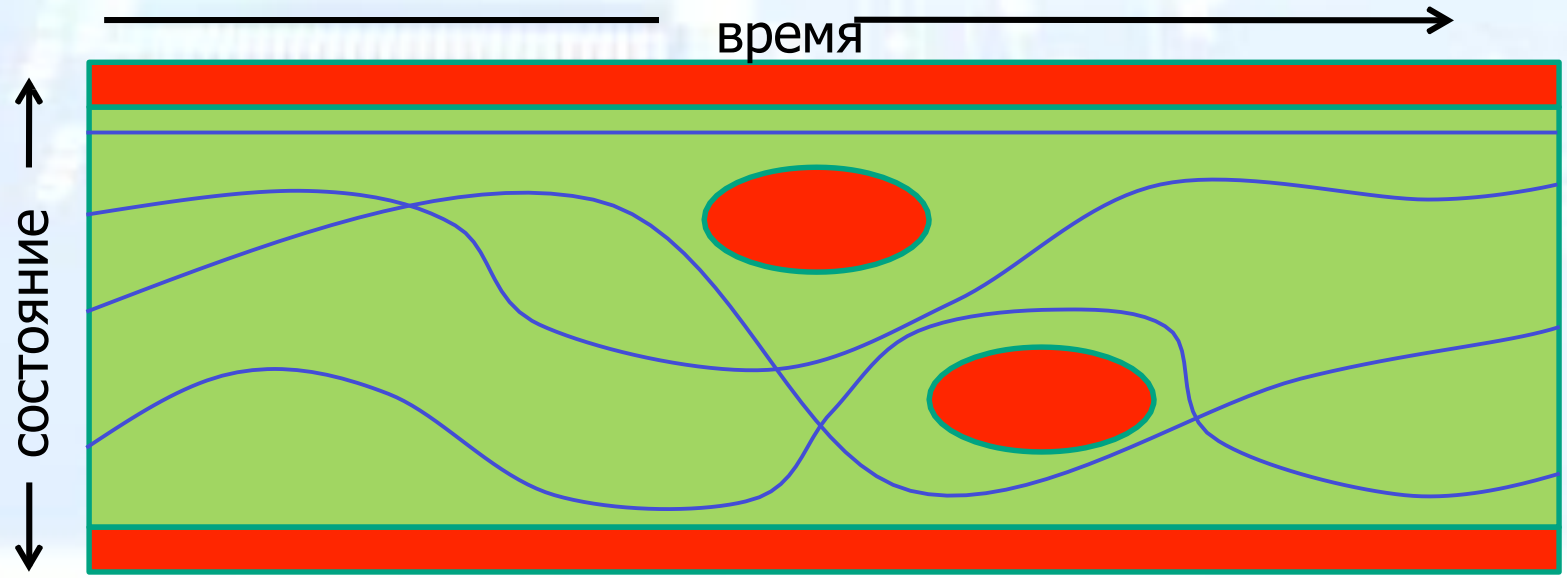
(сводится к анализу достижимости оператора программы),

- Поиск компромисса между потребностями и возможностями.



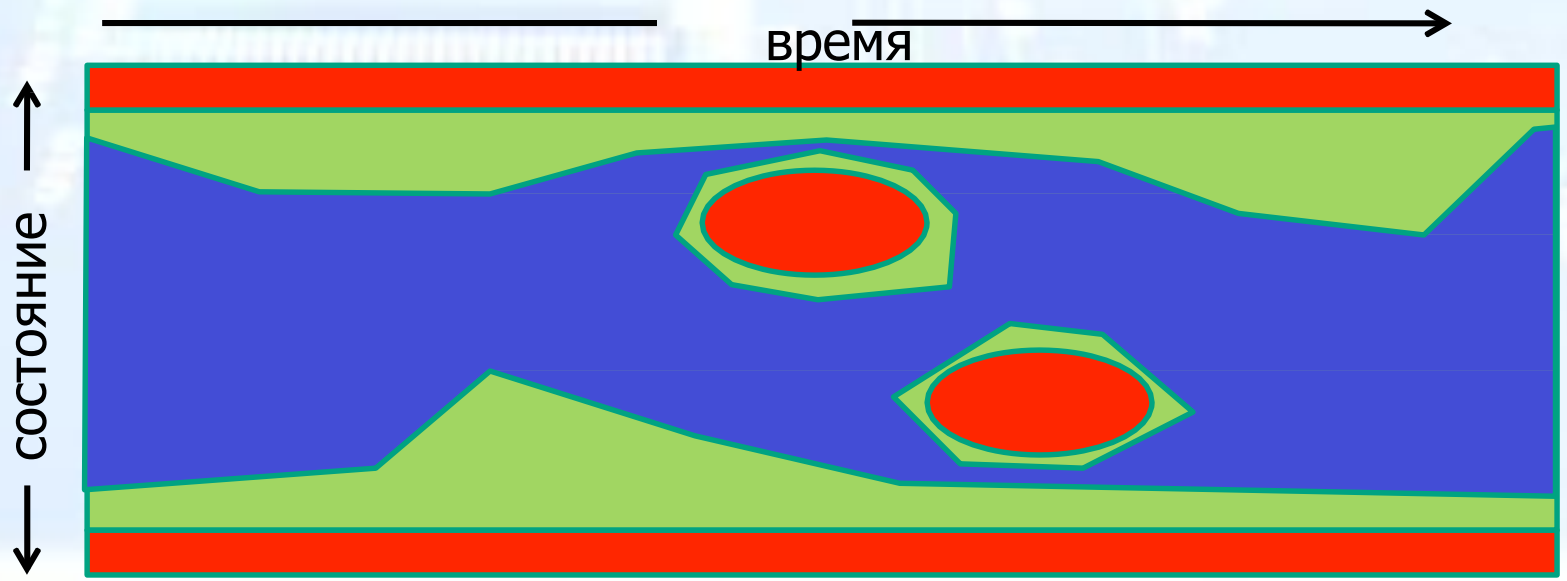
Статический анализ

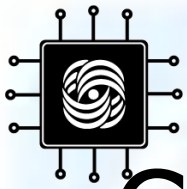
- Абстрактная интерпретация: построение абстрактной семантики языка программирования и интерпретация текста программы в соответствии с этой семантикой,
- В случае тестирования: проверяем, что конкретные вычисления программы не приводят её в ошибочные состояния



Статический анализ

- **Абстрактная интерпретация:** построение абстрактной семантики языка программирования и интерпретация текста программы в соответствии с этой семантикой,
- Статический анализ: аппроксимируем «сверху» множество вычислений программы,
- Возможны ложные сообщения о нарушении свойств!





Статический анализ: пример

Проверка инициализированности переменной:

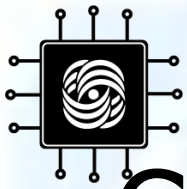
```
int min(int* arr, int n) {
  int m;
  if (n > 0) {
    m = arr[0];
  }
  int i = 0;
  while (i < n) {
    if (m > arr[i]) {
      m = arr[i];
    }
    i++;
  }
  return m;
}
```

dom(m) = Int + { ω }

NI = { ω }

I = Int

v: Expr -> {NI, I}



Статический анализ: пример

Проверка инициализированности переменной:

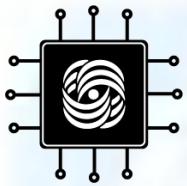
```
int min(int* arr, int n) {
  int m;          <v = { NI }>
  if (n > 0)      <v = { NI }> {
    m = arr[0]; <v = { I }>
  }  <v = { NI, I }> (!)
  int i = 0; <v = { NI, I }>
  while (i < n) <v = { NI, I }> {
    if (m > arr[i]) <v = { NI, I }> {
      m = arr[i]; <v = { I }>
    }
    i++; <v = { NI, I }>
  }
  return m; <v = { NI, I }>
}
```

dom(m) = Int + { ω }

NI = { ω }

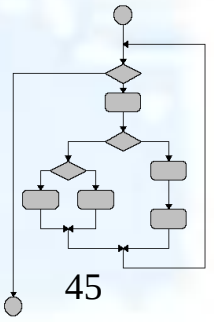
I = Int

v: Expr -> {NI, I}



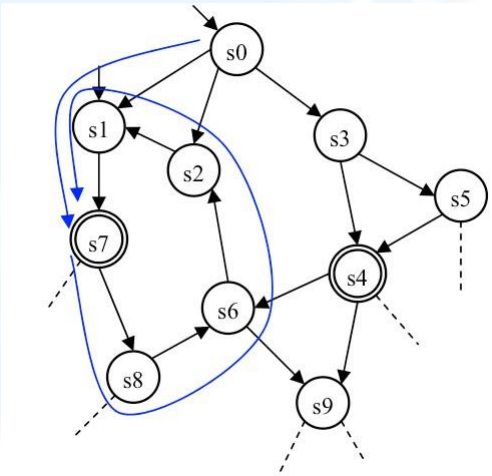
Статический анализ

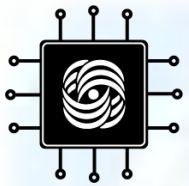
- Достоинства
  - Высокая скорость работы,
  - Если ответ дан, ему можно верить.
- Недостатки
  - Узкая область применения (оптимизация в компиляторах, анализ похожести кода, анализ безопасности итп.),
  - Ручная настройка при изменении проверяемых свойств



Верификация программ на моделях (model checking)

- Проверка свойства на конечной модели программы,
- Свойства – в терминах значения предикатов в состоянии программы и последовательности значений.
- Исчерпывающий поиск по пространству состояний.

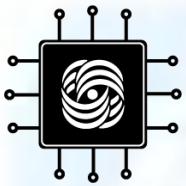




Верификация программ на моделях: пример

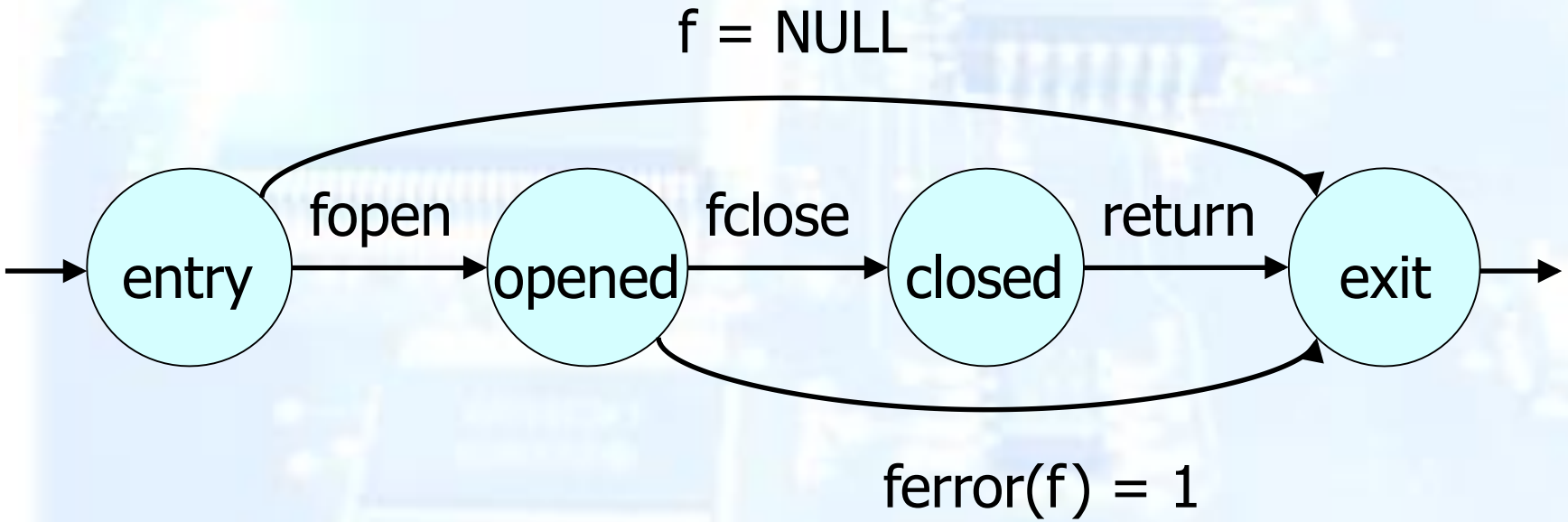
```
int count_lines(const char* filename) {
    int c, count = 0;
    FILE* f = fopen(filename, "r");
    if (f != NULL) {
        c = fgetc(f);
        while (c != EOF) {
            if (c == '\n') {
                ++count;
            }
            c = fgetc(f);
        }
        if (ferror(f)) {
            return -1;
        }
        fclose(f);
        return count;
    } else {
        return -1;
    }
}
```

Всегда ли функция закрывает открытый файл ?



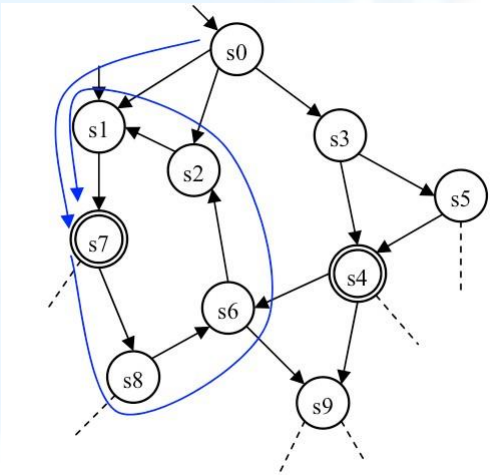
Верификация программ на моделях: пример

- Модель функции count_lines:



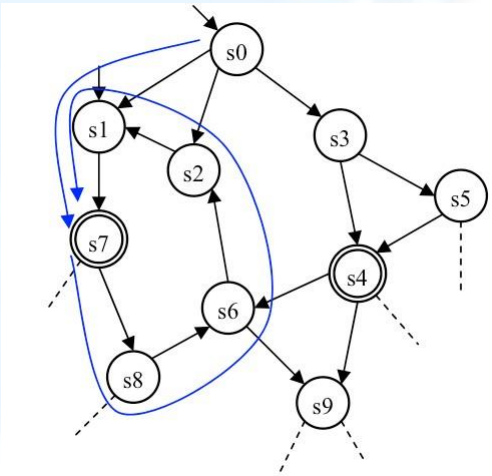
Процесс верификации программ на моделях

- Моделирование
  - Построить адекватную и корректную модель,
  - Избежать «лишних» состояний;
- Спецификация свойств
  - Темпоральная логика,
  - Полнота свойств;
- Верификация
  - Построение контрпримера,
  - Анализ контрпримера.



Верификация программ на моделях

- Достоинства
  - Хорошо автоматизируем,
  - Если модель конечна, корректна и адекватна проверяемому свойству, то даётся точный ответ,
  - Выявляет редкие ошибки.
- Недостатки
  - Работает только для конечных моделей.



Динамическая верификация

- Иногда на этапе разработки системы невозможно гарантировать её правильную работу в ходе эксплуатации:
  - Динамическое изменение конфигурации системы

пример: компьютерная сеть

  - Неполное описание компонентов системы

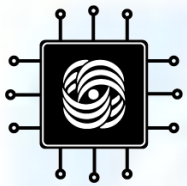
пример: мультивендорная система

  - Работа в сложном окружении

пример: взаимодействие с Интернет

Динамическая верификация

- Т.е. невозможно составить достаточно детальное конечное описание системы
- Решение: в систему добавляется компонент, выполняющий
  - **мониторинг** поведения системы,
  - **анализ** наблюдаемого поведения,
  - **реакцию** на обнаруженные нарушения спецификации



Динамическая верификация

- Проверяется правильность не **описания программы**, а её **наблюдаемого поведения**
- Примеры:
  - Система контроля поведения приложений в ОС
  - Система обнаружения атак
  - Встроенная система контроля

Что можно получить из данных по отказам

Что можно получить из данных по отказам (2)

- Таблица времени между отказами:

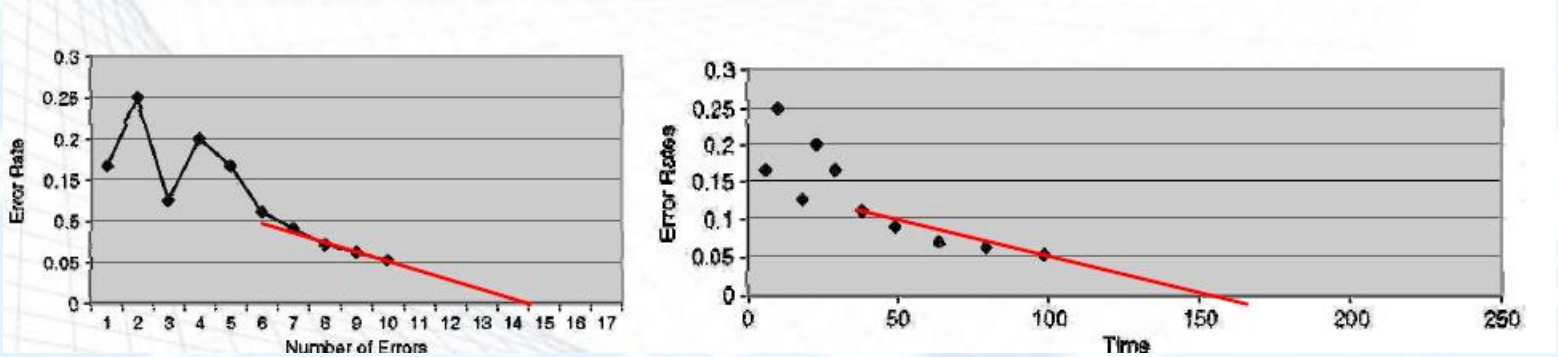
Failure no.	1	2	3	4	5	6	7	8	9	10
Time since last failure (hours)	6	4	8	5	6	9	11	14	16	19

- Что мы можем получить из этих данных:
  - Системная надёжность
  - Апроксимировать количество багов в системе
  - Апроксимировать время для исправления оставшихся ошибок

Что можно получить из данных по отказам (3)

- Интенсивность отказов:

Error no.	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
Time since last failure (hours)	*6*	*4*	*8*	*5*	*6*	*9*	*11*	*14*	*16*	*19*
Failure intensity	*0.166*	*0.25*	*0.125*	*0.20*	*0.166*	*0.111*	*0.09*	*0.071*	*0.062*	*0.053*



Что можно получить из данных по отказам (4)

- Среднее время до отказа

MTTF = (6+4+8+5+6+9+11+14+16+19)/10 = 9.8 часов

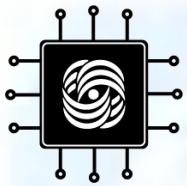
- Системная надёжность
- Если апроксировать график (а), то ошибок будет 15
- Если апроксимировать (б), то все ошибки будут около 160. Соответственно нужно 62 единицы времени

Что можно получить из данных по отказам (4)

- Среднее время до отказа

MTTF = (6+4+8+5+6+9+11+14+16+19)/10 = 9.8 часов

- Системная надёжность
- Если апроксировать график (а), то ошибок будет 15
- Если апроксимировать (б), то все ошибки будут около 160. Соответственно нужно 62 единицы времени



Спасибо за внимание!