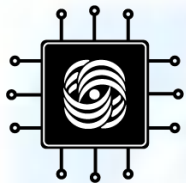


НАДЁЖНОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

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:





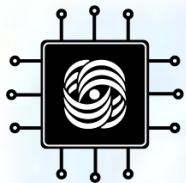
План лекции

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



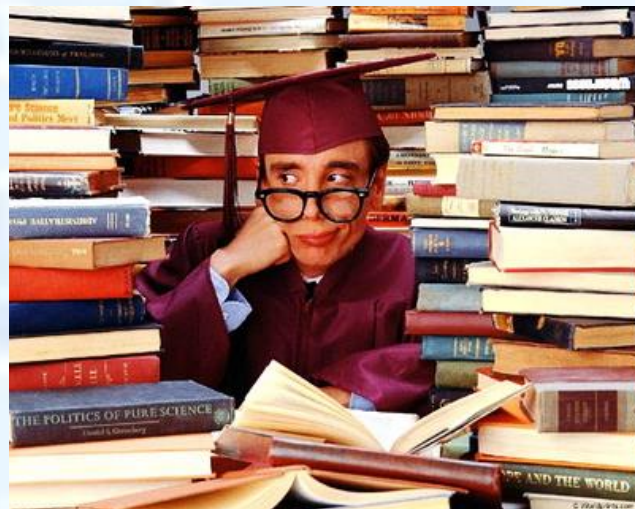
О чём этот курс

- 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

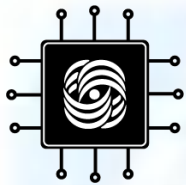




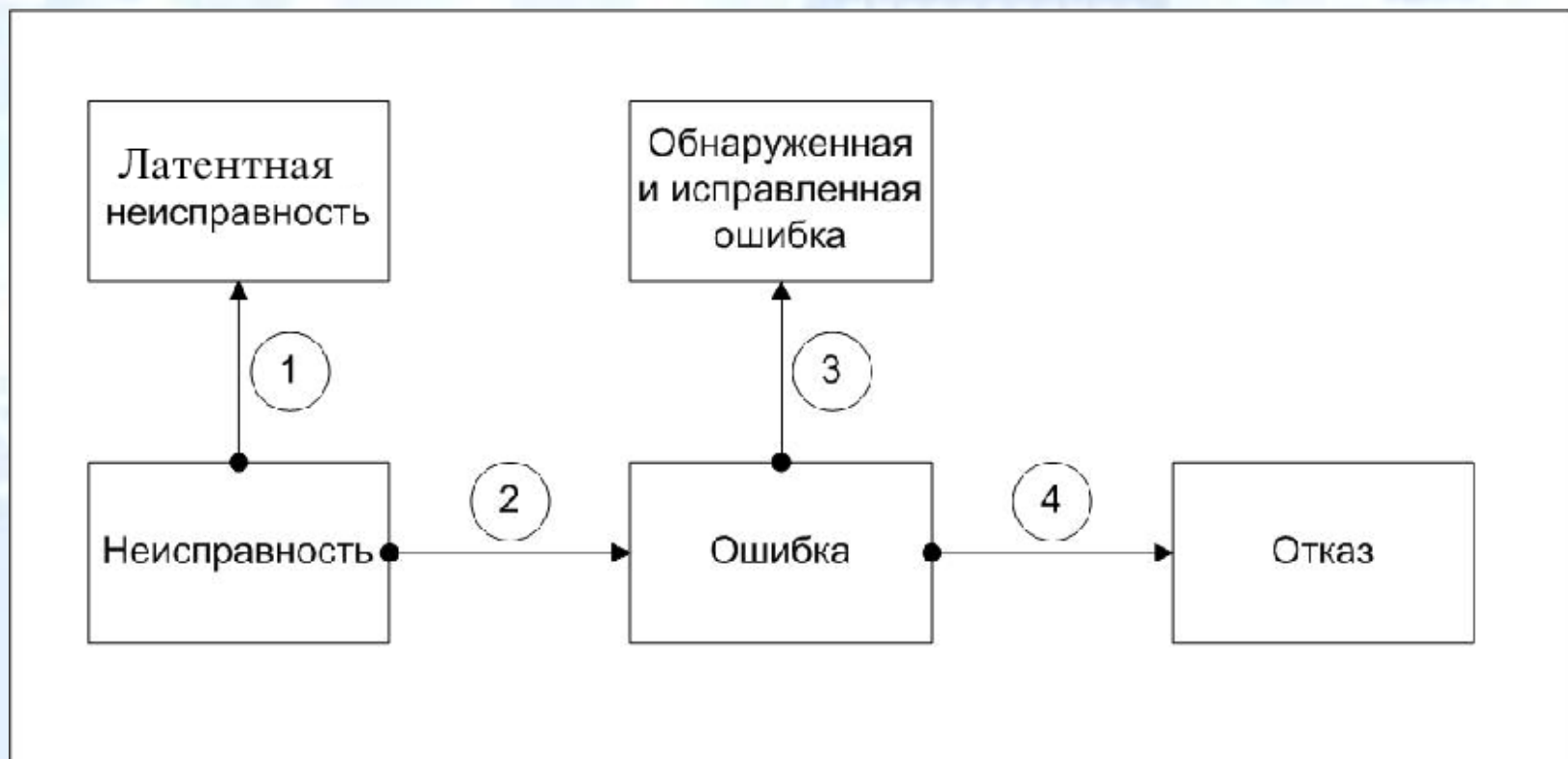
Практикум

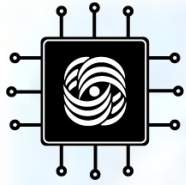
- Несколько заданий, по мере прохождения курса
- Первое задание – 16.09
- Умение работать в программных средствах
- Каждое задание - 10 баллов
- Каждая неделя сверх
Дедлайна –N задания





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;

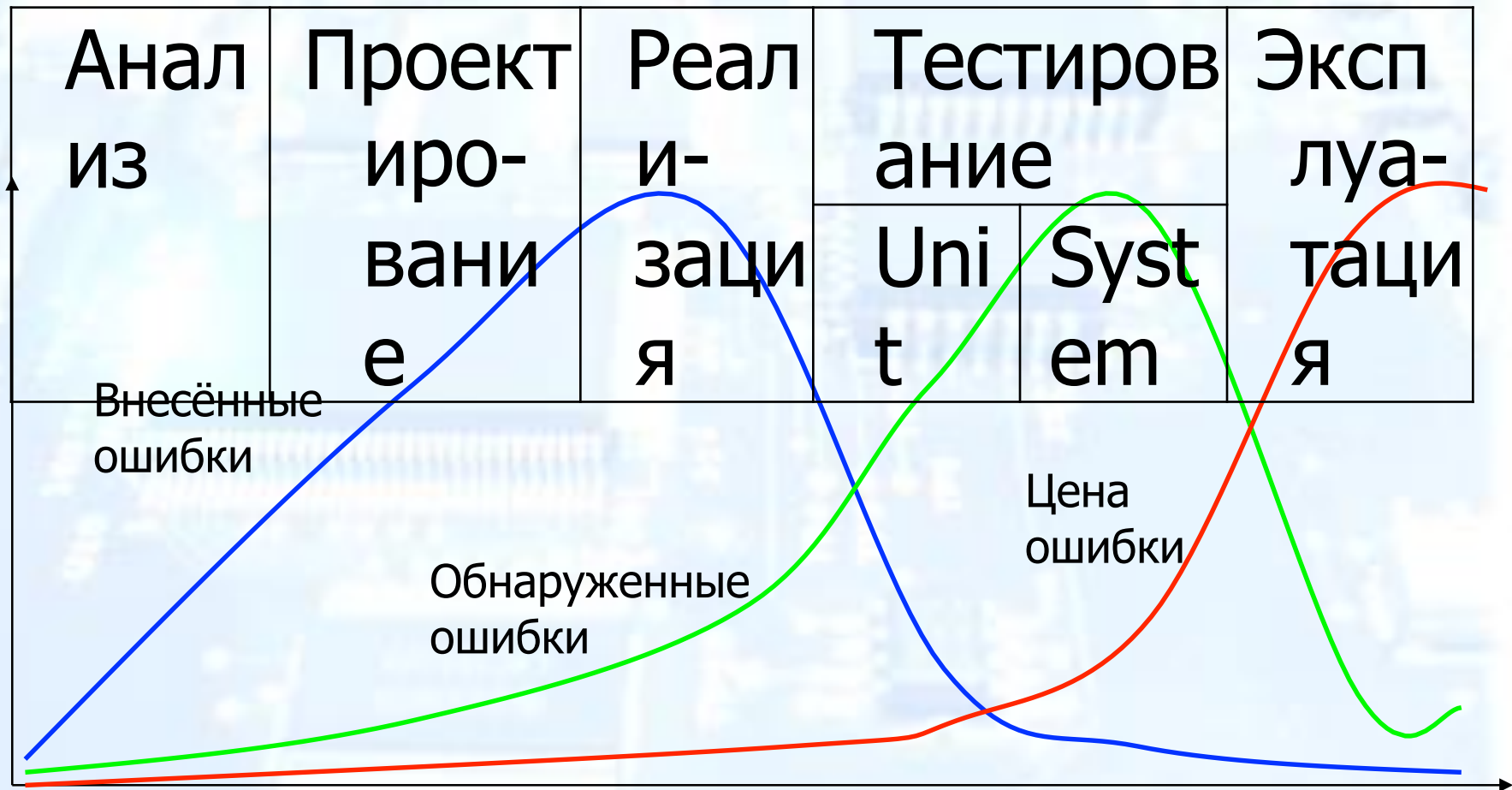
write(s); {отказ}

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

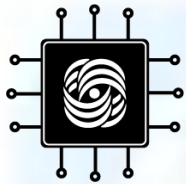
- В ряде приложений ошибки не критичны:
 - сводятся к лёгким моральным травмам,
 - возможность обнаружения сбоя и восстановления,
 - возможность быстрого исправления ошибки.





Цена ошибки

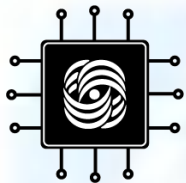
- Системы с повышенными требованиями к надёжности (Safety-critical)
- Ошибки приводят к:
 - Гибели или травмам людей,
 - Крупным финансовым потерям
 - Ущербу окружающей среде
 - Итд.



Цена ошибки: Ariane-5

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

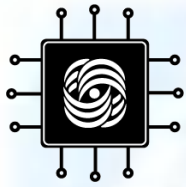




Цена ошибки: Patriot

- 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)



at&t

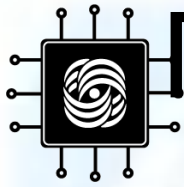


GPS



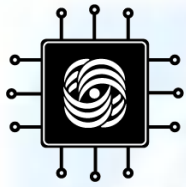
CareFusion

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

частые

редкие

безвредные

тести-
рова
ние

не
важно

критические

формальные
методы



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.



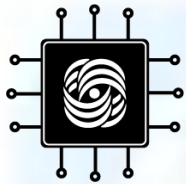


Тестирование: минусы

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

частичное решение –
имитационное моделирование



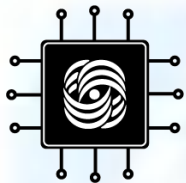


Тестирование: подводные камни

- **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...



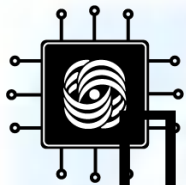


Полное покрытие для прозрачного ящика

```
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

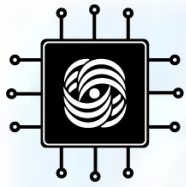




Полнота покрытия: итоги

- 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.

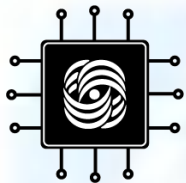




Кое-что о тестировании

- 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.**

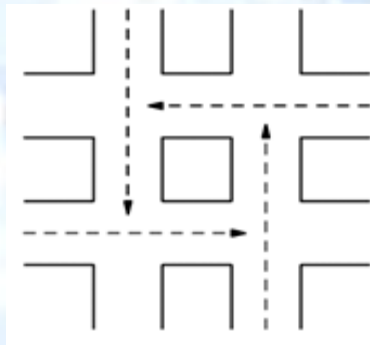




Дополнительные трудности:

Параллельные программы

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



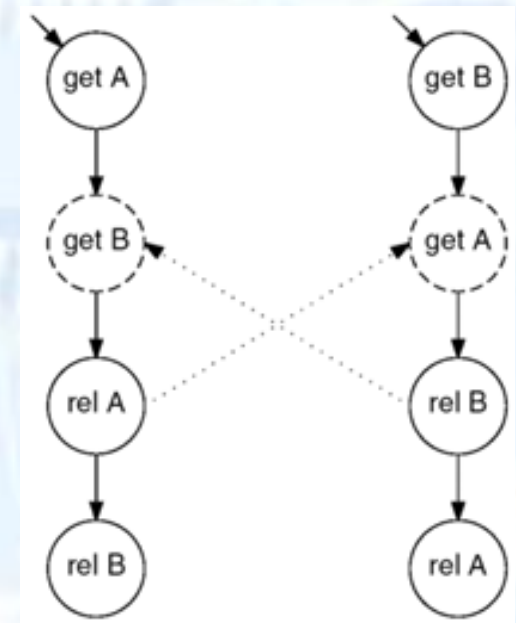
- Правила, реализованные в программах, должны быть универсальны



Системы с разделением ресурсов

Примеры:

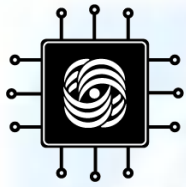
- Дорожный трафик,
- Телефонные сети,
- Операционные системы,
- ...





Дополнительные трудности: **Параллельные системы**

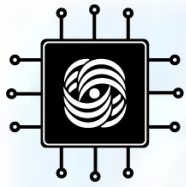
- Новый источник ошибок – совместная работа проверенных компонентов,
- Невоспроизводимость тестов,
- Ограниченные возможности по наблюдению.



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:

$$\frac{S_1 = a_1 \bullet S_2 \vee S_2 = a_2 \bullet S_3}{S_1 = a_1 \bullet a_2 \bullet S_3}, \frac{S_1 \bullet S_2 \vee S_2 \bullet S_3}{S_1 \bullet S_3}$$

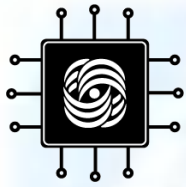




Доказательство теорем

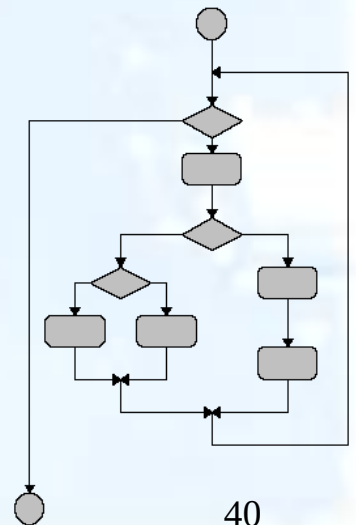
- Достоинства:
 - работа с бесконечными пр-вами состояний,
 - даёт более глубокое понимание системы.
- Недостатки:
 - медленная скорость работы,
 - может потребоваться помощь человека (построение инвариантов циклов),
 - В общем случае нельзя построить полную систему аксиом и правил вывода (теорема неполноты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





Статический анализ

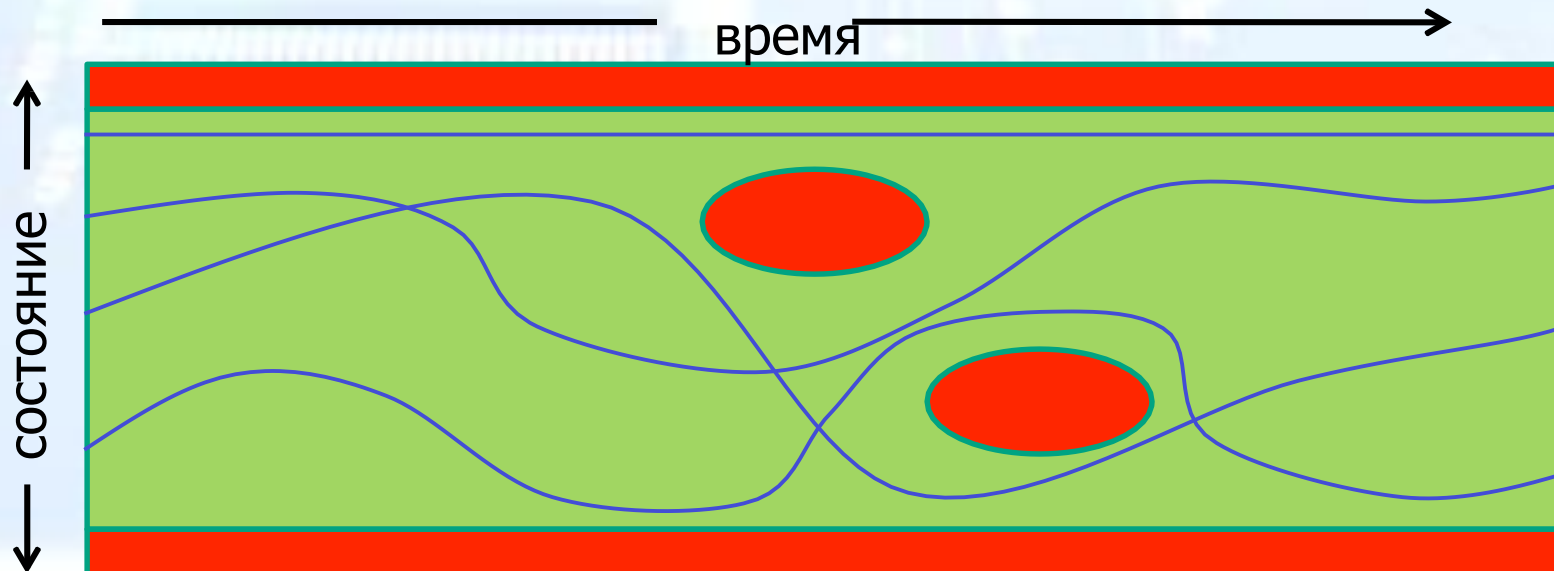
- Более грубый и прагматичный подход,
- Анализ исходного текста программы без её выполнения,
- В общем случае задача неразрешима (сводится к анализу достижимости оператора программы),
- Поиск компромисса между потребностями и возможностями.

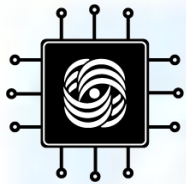




Статический анализ

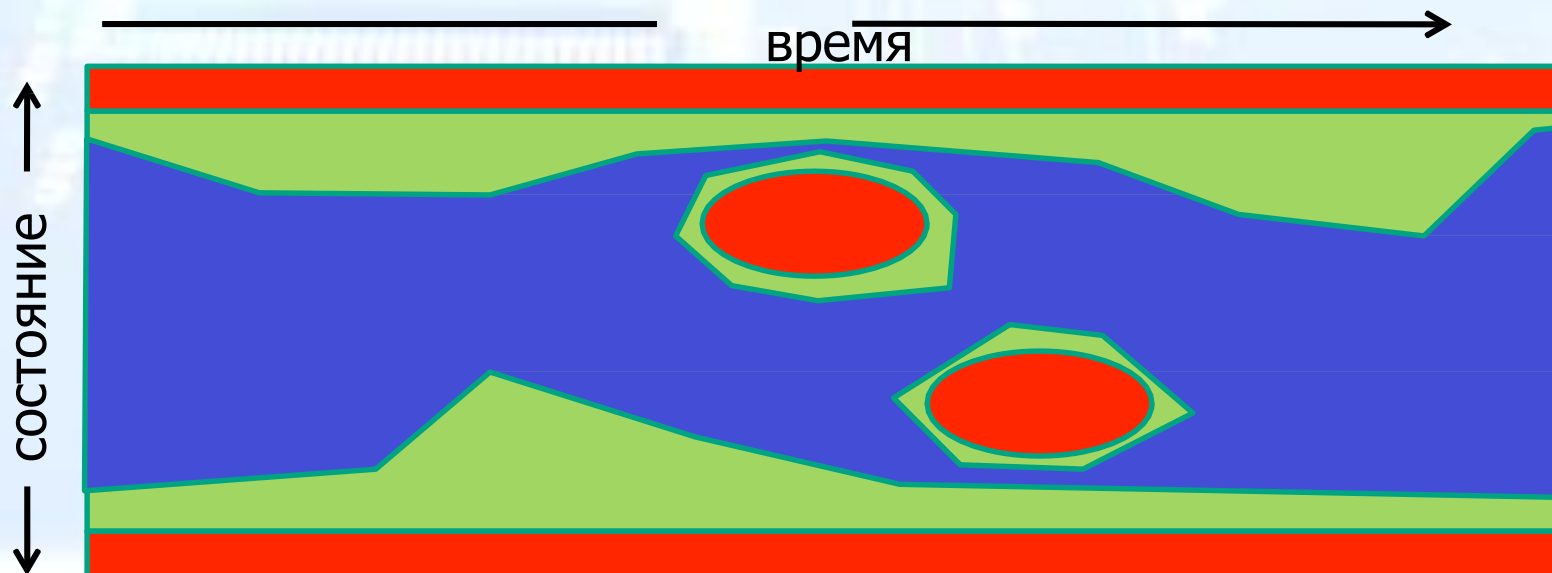
- Абстрактная интерпретация: построение абстрактной семантики языка программирования и интерпретация текста программы в соответствии с этой семантикой,
- В случае тестирования: проверяем, что конкретные вычисления программы не приводят её в ошибочные состояния

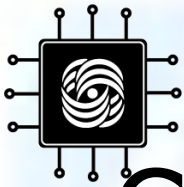




Статический анализ

- **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:

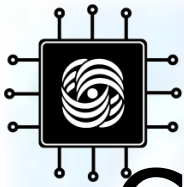
```
int min(int* arr, int n) {  
    int m;  
    if (n > 0) {  
        m = arr[0];  
    }  
    int i = 0;  
    while (i < n) {  
        if (m > arr[i]) {  
            m = arr[i];  
        }  
        i++;  
    }  
    return m;  
}
```

$\text{dom}(m) = \text{Int} + \{ \omega \}$

$\text{NI} = \{ \omega \}$

$I = \text{Int}$

$v: \text{Expr} \rightarrow \{ \text{NI}, I \}$



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 }>  
}
```

$\text{dom}(m) = \text{Int} + \{ \omega \}$

$\text{NI} = \{ \omega \}$

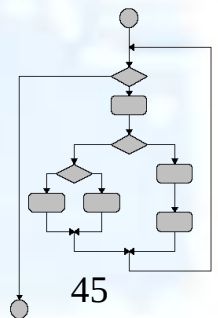
$I = \text{Int}$

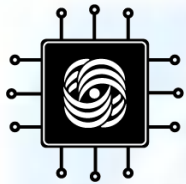
$v: \text{Expr} \rightarrow \{ \text{NI}, I \}$



Статический анализ

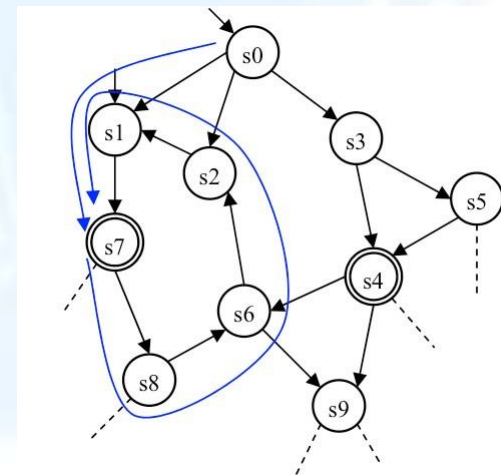
- Достоинства
 - Высокая скорость работы,
 - Если ответ дан, ему можно верить.
- Недостатки
 - Узкая область применения (оптимизация в компиляторах, анализ похожести кода, анализ безопасности итп.),
 - Ручная настройка при изменении проверяемых свойств

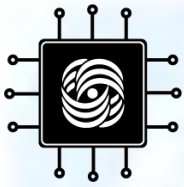




Верификация программ на моделях (model checking)

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

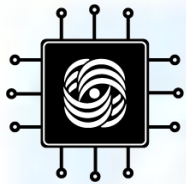




Верификация программ на моделях: пример

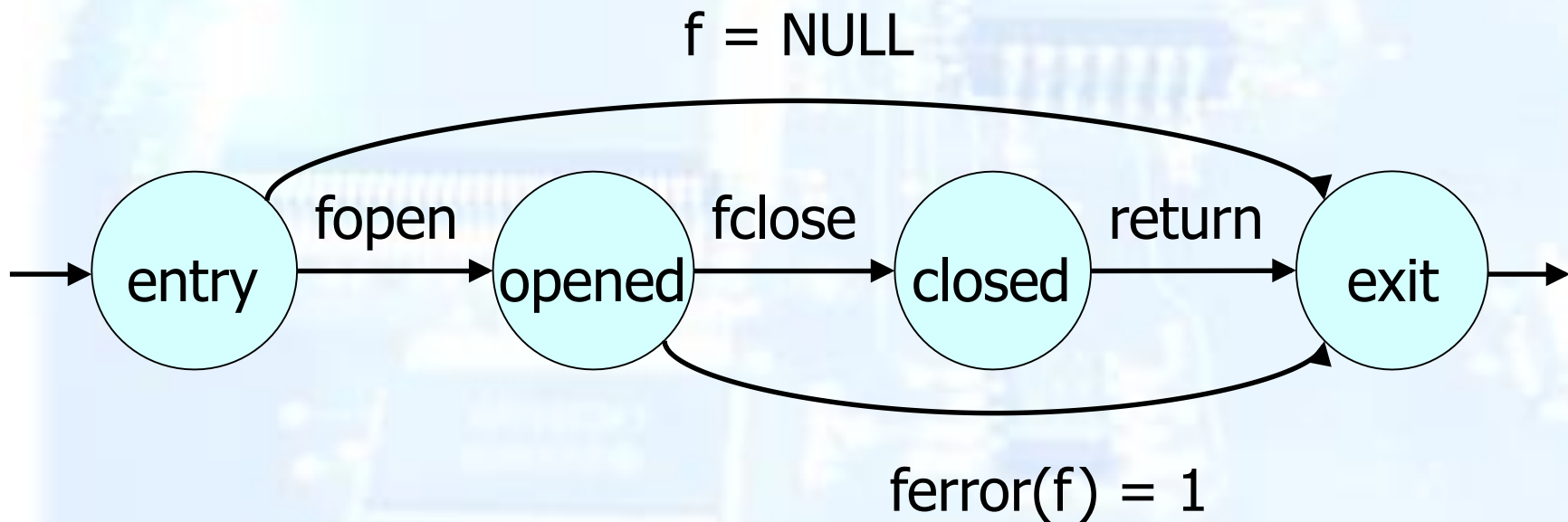
```
int count_lines(const char* filename) {  
    int c, count = 0;  
    FILE* f = fopen(filename, "r");  
    if (f != NULL) {  
        c = fgetc(f);  
        while (c != EOF) {  
            if (c == '\n') {  
                ++count;  
            }  
            c = fgetc(f);  
        }  
  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

Всегда ли
функция
закрывает
открытый
файл
?



Верификация программ на моделях: пример

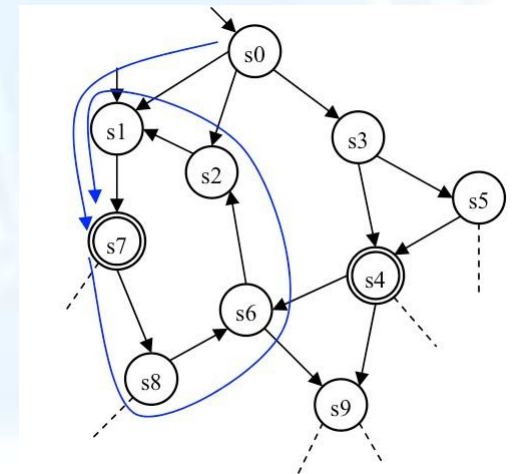
- Модель функции `count_lines`:





Процесс верификации программ на моделях

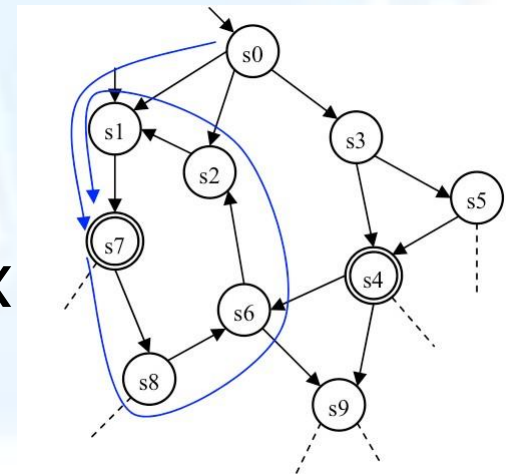
- Моделирование
 - Построить адекватную и корректную модель,
 - Избежать «лишних» состояний;
- Спецификация свойств
 - Темпоральная логика,
 - Полнота свойств;
- Верификация
 - Построение контрпримера,
 - Анализ контрпримера.





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



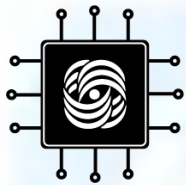
Динамическая верификация

- 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



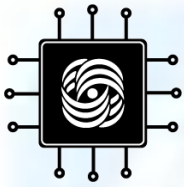
Динамическая верификация

- 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



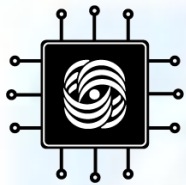


Что можно получить из данных по отказам (2)

- Таблица времени между отказами:

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

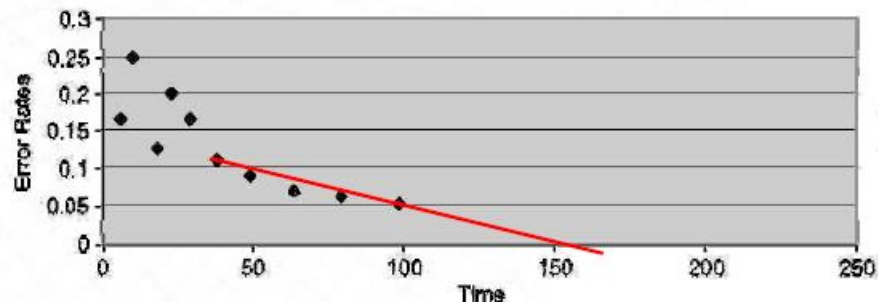
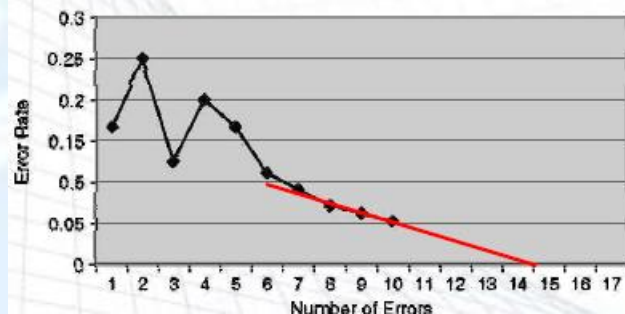
- Что мы можем получить из этих данных:
 - Системная надёжность
 - Апроксимировать количество багов в системе
 - Апроксимировать время для исправления оставшихся ошибок

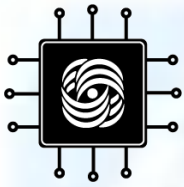


Что можно получить из данных по отказам (3)

- Интенсивность отказов:

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



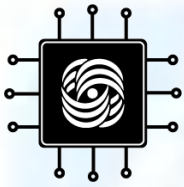


Что можно получить из данных по отказам (4)

- Среднее время до отказа

$$\text{MTTF} = (6+4+8+5+6+9+11+14+16+19)/10 = 9.8 \text{ часов}$$

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

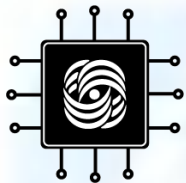


Что можно получить из данных по отказам (4)

- Среднее время до отказа

$$\text{MTTF} = (6+4+8+5+6+9+11+14+16+19)/10 = 9.8 \text{ часов}$$

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



Спасибо за внимание!



НАДЁЖНОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Лекция 2: *Моделирование программ*

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

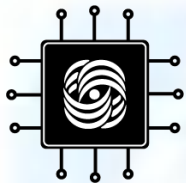


План лекции

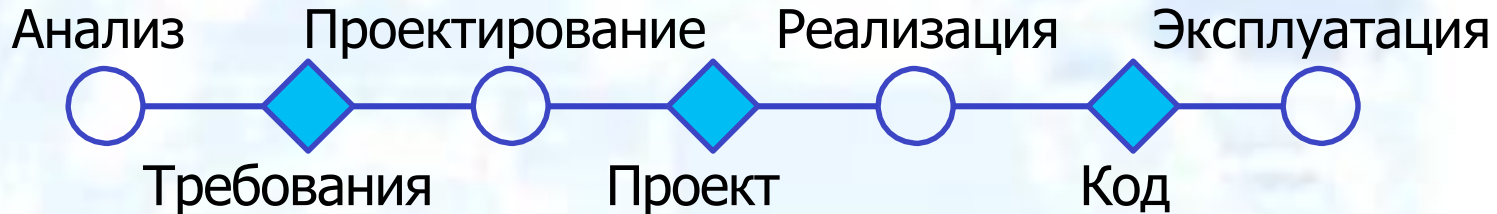
- **Формальные методы проверки правильности программ:** место в жизненном цикле ПО, автоматизируемость
- **Верификация программ на моделях:** области применения, общая схема
- **Моделирование программ:** моделируемые свойства, состояние программы, процесс построения модели



Формальные методы проверки правильности программ (место в жизненном цикле ПО, вопросы автоматизации)



Итоги предыдущей лекции



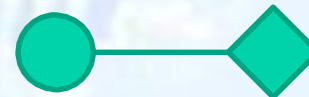
Валидация



Тестирование



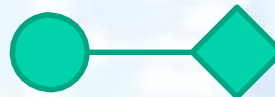
Статический
анализ кода



Доказательств
о теорем



Верификация
на моделях



Динамическая
верификация





Автоматизируемость методов верификации

- **Тестирование:** автоматическое и автоматизированное,
- **Доказательство теорем:** существенное участие человека,
- **Статический анализ:** полностью автоматический для заданной области и свойства,
- **Верификация на моделях:** участие человека при построении модели и при анализе контрпримеров,
- **«Комбинаторный взрыв».**



Верификация программ на моделях

(области применения,
общая схема,
проверяемые свойства)



Области применения верификации на моделях

- Сетевые и криптографические протоколы,
- Протоколы работы кэш-памяти,
- Интегральные схемы,
- Стандарты (напр. FutureBus+),
- Встроенные системы,
- Драйвера.



Примеры использования SPIN

- Верификация системы управления Роттердаме (Нидерланды)

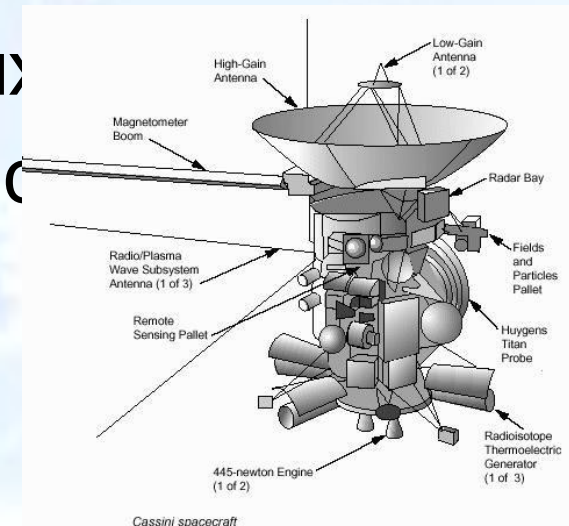


- Верификация АТС

PathStar (Lucent)



- Верификация аэрокосмически (DeerSpace 1, Cassini, Mars Explorer итд.)



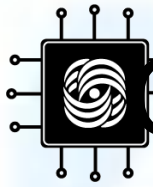
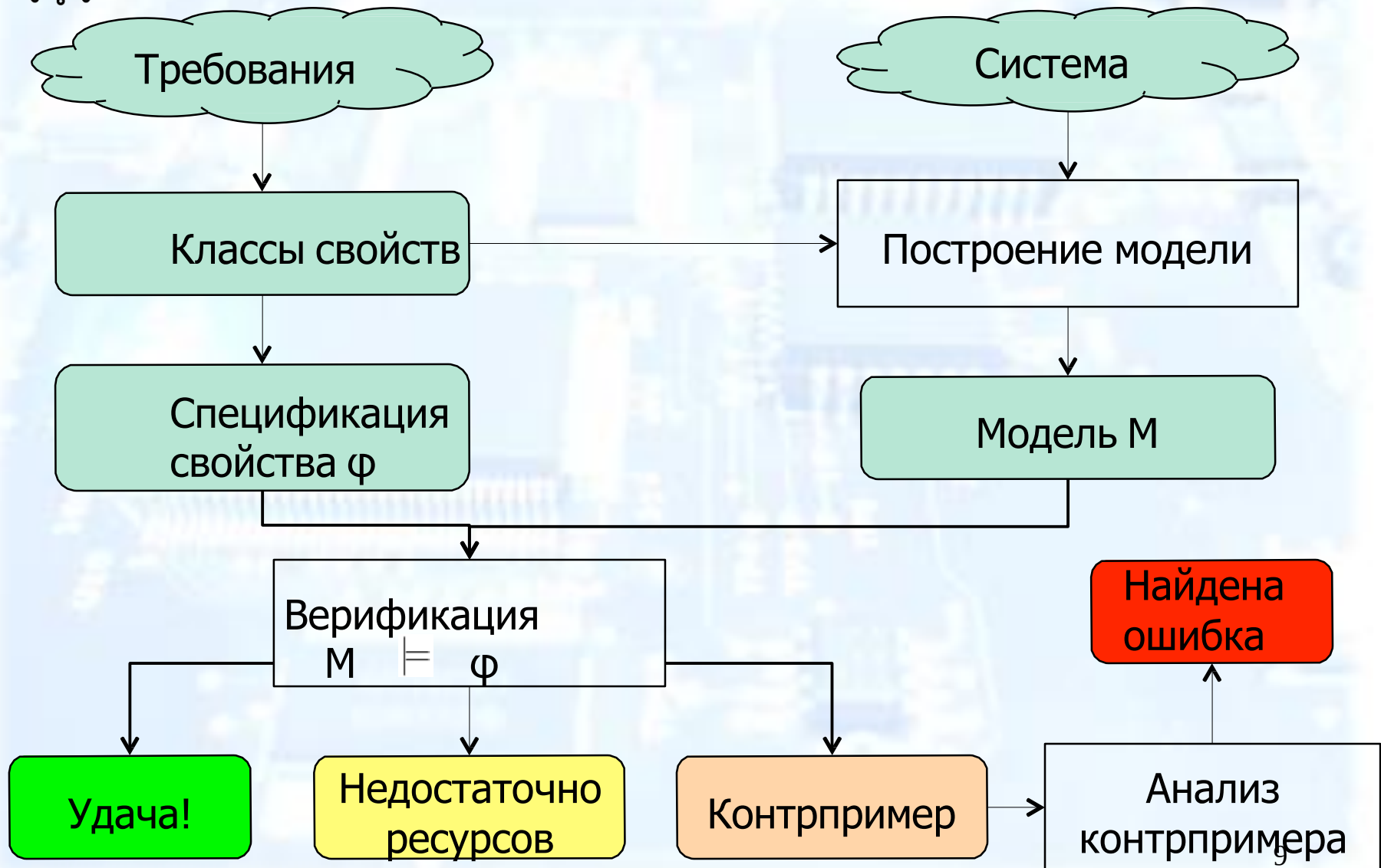


Схема верификации на модели





Примеры классов свойств

- Стандартные:
 - Отсутствие ошибок времени выполнения (RTE),
 - Отсутствие тупика (deadlock),
 - Отсутствие удушения (starvation),
 - Не срабатывают ассерты (assertions).
- Зависящие от приложения:
 - Инварианты системы,
 - Индикаторы прогресса,
 - Корректная завершаемость,
 - Причинно-следственный и темпоральный порядок на состояниях системы,
 - Требования справедливости.



Моделирование программ

(пример построения модели,
общая схема моделирования)



Зачем строить модели программ?

- Свойства задают допустимые состояния и последовательности состояний (вычисления)

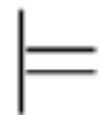
т.н. свойства линейного времени

- Модель генерирует трассы, по которым можно проверить свойства,

Система

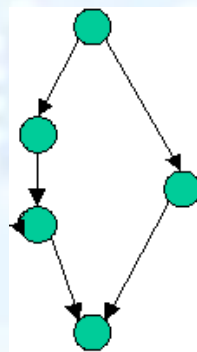
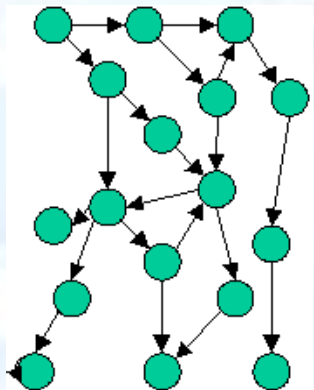


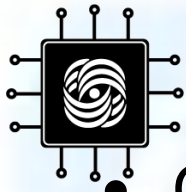
Модель



Свойства

- Абстракция от лишних состояний.





Состояние программы

- Состояние программы – совокупность значений объектов данных и счётчика управления,
- Как правильно оценить количество состояний?

Специфика ция программы	Исходный код на языке высокого уровня	Исполняем ый код (ASM)	Программн о- аппаратная система	Работающи й компьютер
Может вообще не быть состояний (stateless протоколы)	Данные – переменные языка, счетчик – номер выполняемого оператора	Данные – ячейки памяти в адресном пр-ве процессора и регистры, счётчик -- регистр команд процессора	Данные – ячейки памяти различных ЗУ, кэш, буфера выборки, состояние конвейеров, Счётчик – адрес последней команды	Атомы, электроны, спины ...



Состояние программы (на уровне языка C)

```
0:  int compare(int a, int b) {  
1:      int res;  
2:      if (a < b) {  
3:          res = -1;  
4:      } else if (a >= b) {  
5:          res = 1;  
6:      } else {  
7:          res = 0;  
8:      }  
9:      return res;  
10: }
```

Потенциально –
 $8 * 2^{96}$
СОСТОЯНИЙ

3 целочисленных переменных,
 $|\text{int}| = 2^{32}$,
7 операторов языка +
терминальный оператор.



Состояние программы (на уровне языка C)

```
0: void foo(int a) {  
1:     int x= 5;  
2:     if (a < x) {  
3:         x = x-a;  
4:     }  
5: }
```

```
0: void bar(int b) {  
1:     int x;  
2:     x = b;  
3: }
```

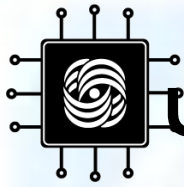
Потенциально –
 $15 \cdot 2^{128}$
СОСТОЯНИЙ

4 целочисленных переменных,
 $|\text{int}| = 2^{32}$,
2 потока управления, в foo – 5
операторов, в bar – 3.



Достижимые СОСТОЯНИЯ

- Состояние называется **достижимым**, если существует вычисление программы, в котором оно присутствует
 - формализация – позже
- Достижимость состояния в программе в общем случае алгоритмически неразрешима.
 - неразрешима даже достижимость точки программы



Число достижимых состояний

0:
1:
2:
3:
4:
5:
6:
7:

```
int compare(int a, int b) {  
    int res;  
    if (a < b) {  
        res = -1;  
    } else if (a >= b) {  
        res = 1;  
    } else {  
        res = 0;  
    }  
  
    return res;  
}
```

$$8 * 4 * 2^{64}$$

переменная `res`
может принимать
только 4 значения

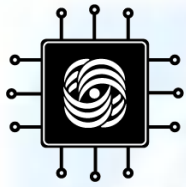
$$7 * 3 * 2^{64}$$

5-й оператор не
достижим



Пример построения модели программы

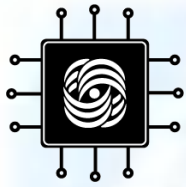
- Программа подсчёта количества слов в файле
- Проверяемое свойство – всегда ли программа закрывает открытый файл?
- См. следующий слайд.



Пример построения модели программы

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc,
          const char* argv[]) {
    FILE* f;
    int c, wordCnt = 0, inWord = 0;
    if (argc < 2) {
        printf("Use: %s filename\n",
               argv[0]);
        return 1;
    }
    f = fopen(argv[1], "r");
    if (f == NULL) {
        printf("Can't open file:
               %s\n", argv
               [1]);
        return 1;
    }
    ↓      ↓      ↓
```

```
while ((c = fgetc(f)) != -1) {
    if (!isspace(c)) {
        if (!inWord) {
            ++wordCnt;
            inWord = 1;
        }
    } else {
        inWord = 0;
    }
    printf("Word count: %d\n",
           wordCnt);
    fclose(f);
    return 0;
}
```



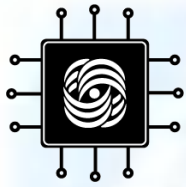
Пример построения модели программы

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc,
        const char* argv[]) {
    FILE* f;
    int c, wordCnt = 0, inWord = 0;
    if (argc < 2) {
        printf("Use: %s filename\n",
               argv[0]);
        return 1;
    }
    f = fopen(argv[1], "r");
    if (f == NULL) {
        printf("Can't open file:
               %s\n", argv
               [1]);
        return 1;
    }
}
```



```
while ((c = fgetc(f)) != -1) {
    if (!isspace(c)) {
        if (!inWord) {
            ++wordCnt;
            inWord = 1;
        }
    } else {
        inWord = 0;
    }
}
printf("Word count: %d\n",
       wordCnt);
fclose(f);
return 0;
}
```





Пример построения модели программы

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
int main() {
```

```
    FILE* f;
```

```
    int c, wordCnt = 0, inWord = 0;
```

```
    enum {FALSE, TRUE} P1 = any();
```

```
    if (P1) {
```

```
        return 1;
```

```
    }
```

```
    f = fopen("sample", "r");
```

```
    if (f == NULL) {
```

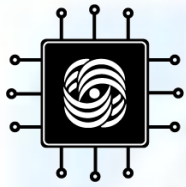
```
        return 1;
```

```
    }
```

```
        ↓           ↓           ↓

while ((c = fgetc(f)) != -1) {
    if (!isspace(c)) {
        if (!inWord) {
            ++wordCnt;
            inWord = 1;
        }
    } else {
        inWord = 0;
    }
}
fclose(f);
return 0;
}
```





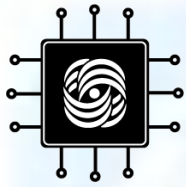
Пример построения модели программы

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    FILE* f;
    int c, wordCnt = 0, inWord = 0;
    enum {FALSE, TRUE} P1 = any();
    if (P1) {
        return 1;
    }
    f = fopen("sample", "r");
    if (f == NULL) {
        return 1;
    }
}
```



```
↓ ↓ ↓

while ((c = fgetc(f)) != -1) {
    if (!isspace(c)) {
        if (!inWord) {
            ++wordCnt;
            inWord = 1;
        }
    } else {
        inWord = 0;
    }
}
fclose(f);
return 0;
}
```



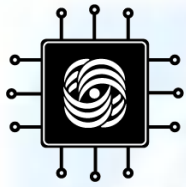
Пример построения модели программы

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    FILE* f;
    int inWord = 0;
    enum {FALSE, TRUE} P1 = any(),
          P2, P3;
    if (P1) {
        return 1;
    }
    f = fopen("sample", "r");
    if (f == NULL) {
        return 1;
    }
}
```

↓ ↓ ↓

```
while (P2 = any()) {
    if (P3 = any()) {
        if(inWord)
            inWord = 1;
        }
    } else {
        inWord = 0;
    }
}
fclose(f);
return 0;
}
```

↓ ↓ ↓



Пример построения модели программы

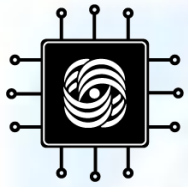
```
#include <stdio.h>
#include <stdlib.h>
```

```
int main() {
    FILE* f;
    int inWord = 0;
    enum {FALSE, TRUE} P1 = any(),
          P2, P3;
    if (P1) {
        return 1;
    }
    f = fopen("sample", "r");
    if (f == NULL) {
        return 1;
    }
}
```

↓ ↓ ↓

```
while (P2 = any()) {
    if (P3 = any()) {
        if(inWord)
            inWord = 1;
    }
    } else {
        inWord = 0;
    }
}
fclose(f);
return 0;
}
```



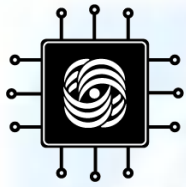


Пример построения модели программы

```
int main() {  
    enum { FCLOSED, FOPEN, FERROR} fileState;  
    enum { V0, V1 } inWord = V0;  
    enum {FALSE, TRUE} P1 = any(),  
           P2, P3;  
    if (P1) {  
        return 1;  
    }  
    if(any()) {  
        fileState = FOPEN;  
    } else {  
        fileState = FERROR;  
    }  
    if (fileState == FERROR) {  
        return 1;  
    }  
}
```

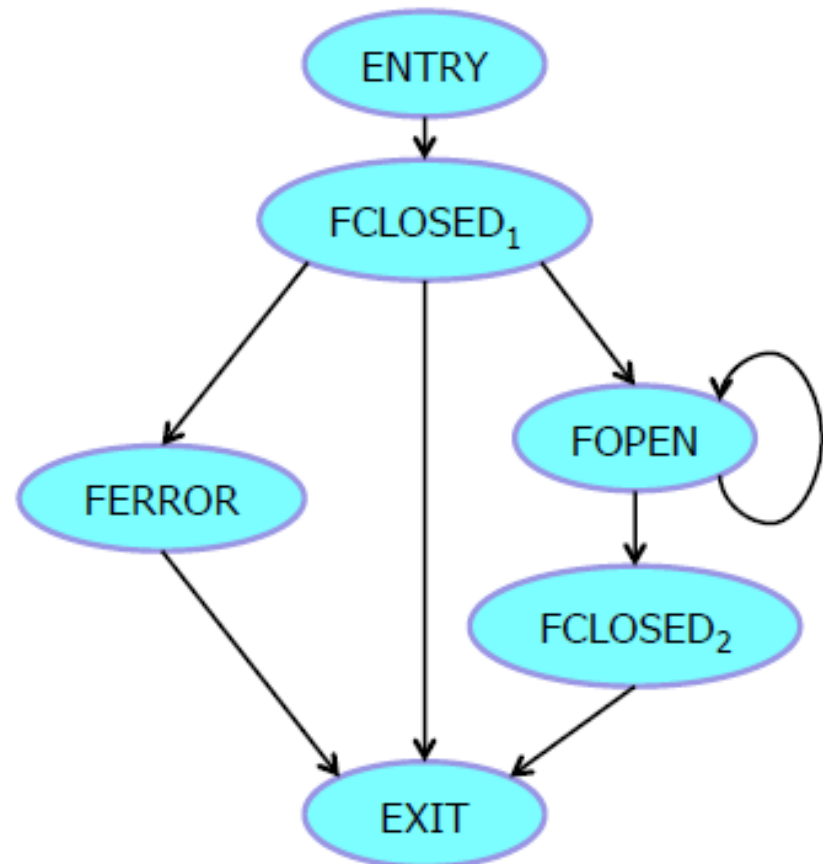
```
                                ↓      ↓      ↓  
while (P2 = any()) {  
    if (P3 = any()) {  
        if (!inWord) {  
            inWord = V1;  
        }  
    } else {  
        inWord = V0;  
    }  
}  
fileState = FCLOSED;  
return 0;  
}
```

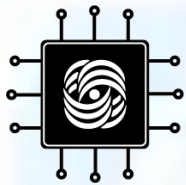




Пример построения модели программы

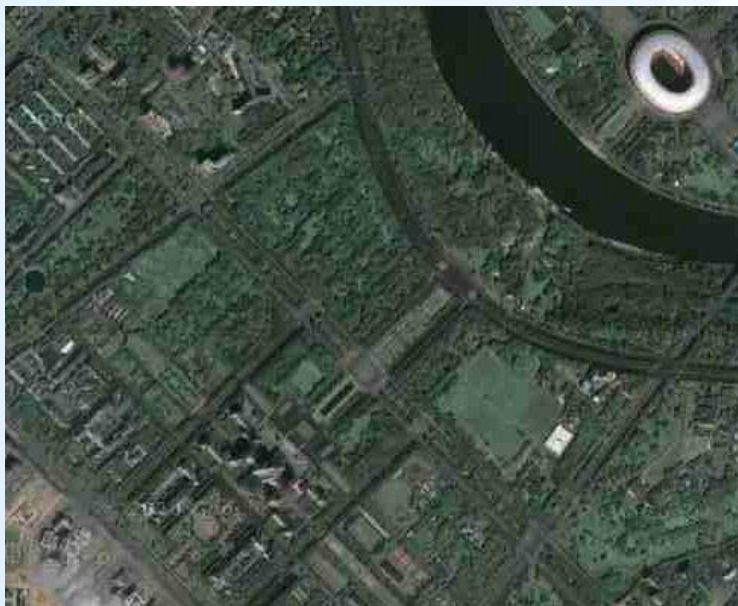
```
int main() {  
    enum { FCLOSED, FOPEN, FERROR} fileState;  
    if(any()) {  
        fileState = FERROR;  
    } else if (any()) {  
        fileState = FOPEN;  
        while (any());  
        fileState = FCLOSED;  
    }  
    return 0;  
}
```





Моделирование

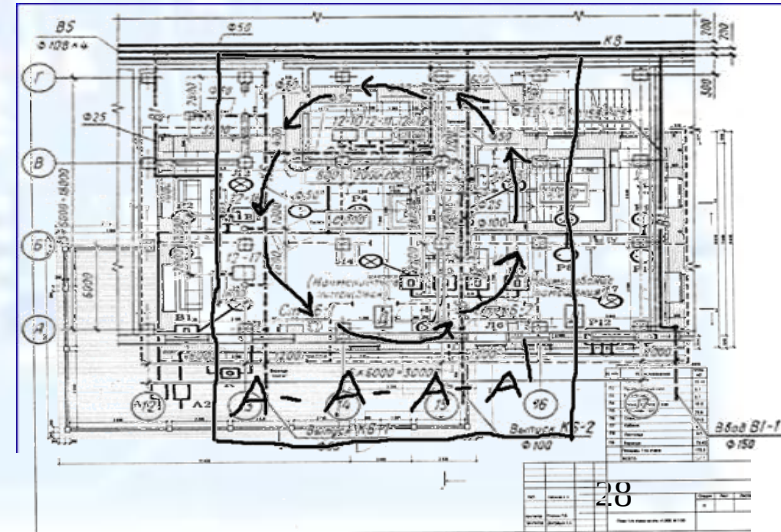
- Модель – упрощённое описание реальности, выполненное с определённой целью.
- Например, карта – модель местности.





Моделирование

- Модель – упрощённое описание реальности, выполненное **с определённой целью.**
- С одним объектом может быть связано несколько моделей.
- Каждая модель отражает свой аспект реальности
- Например, схемы зданий:
 - поэтажный план
 - электропроводка
 - водоснабжение
 - план эвакуации





Моделирование

- Модель должна быть:

- **Простой,**

- (быть проще, чем **реальность**)

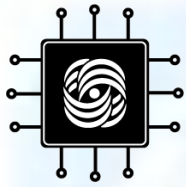
- **Корректной,**

- (не расходиться с **реальностью**)

- **Адекватной.**

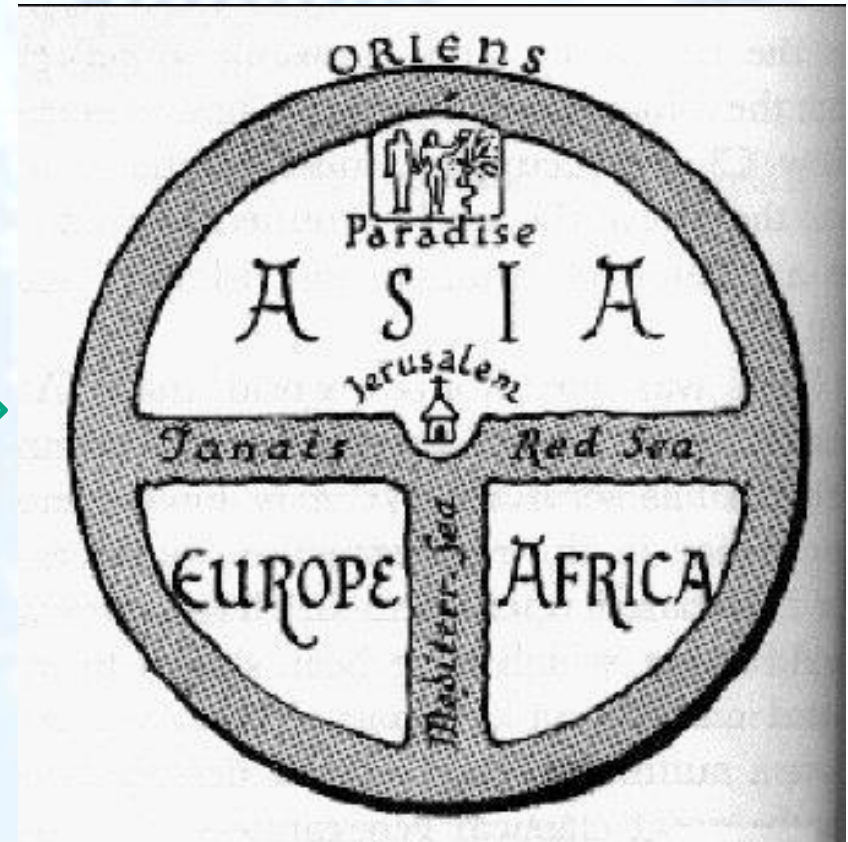
- (соответствовать **решаемой задаче**)

- (Ни одно из этих качеств нельзя проверить на основе только описания модели)



Моделирование

- Пример сильной абстракции:





Построение модели

- 
- Формализуемые требования
(постановка задачи моделирования)
 - Выбор языка моделирования
 - Абстракция системы с учётом требований



Моделирование программ

процесс consumer:

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MSG_LEN 100
```

```
struct msgbuf { mtype;
long mtext[MSG_LEN];
char
};
```

```
int main(int argc,
          const char* argv[]) {
key_t key;
int qid, i;
struct msgbuf msg;
ssize_t size;
key = ftok("./mc_lec2_ex3", 1);
```

consumer

foo

bar

BAR

FOO

```
if ((qid = msgget(key,
IPC_CREAT | 0666)) == -1) {
perror("Error creating
message queue");
}
while (1) {
size = msgrcv(qid, &msg,
MSG_LEN, 1, 0);
for (i = 0; i < size; ++i) {
msg.mtext[i] =
toupper(msg.mtext[i]);
}
/* send it back */
if (size > 0) {
msgsnd(qid, &msg, size, 0);
}
}
}
```



Моделирование программ

процесс producer:



producer



consumer

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MSG_LEN 100
struct msgbuf {
    long    mtype;
    char    mtext[MSG_LEN + 1];
};

int main(int argc,
          const char* argv[]) {
    key_t key; int
    qid;
    struct msgbuf msg;
    ssize_t size;
    msg.mtype = 1;
    key =
        ftok("./mc_lec2_ex3", 1);
```

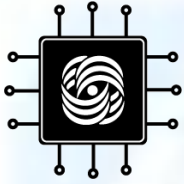
```
    if ((qid = msgget(key, IPC_CREAT
                       | 0666)) == -1) {
        msgget_error:
        perror("Error creating
               message queue");
    }
    while (1) {
        printf("Your message: ");
        fgets(msg.mtext, MSG_LEN, stdin);
        if (msgsnd(qid, &msg,
                   strlen(msg.mtext), 0) == -1) {
            perror("Error sending"); continue;
        }
        size = msgrcv(qid, &msg, MSG_LEN, 1, 0);
        if (size > 0) {
            printf("Reply is: %s\n", msg.mtext);
        } else {
            printf("No reply\n");
        }
    }
}
```




Свойства правильности процесса producer:

- `msgsnd` всегда вызывается только после `msgget`;
- если поток управления попадает в `msgget_error`, то после этого не вызываются функции `msgsnd` и `msgrcv`;
- ни в одном вычислении не встречаются `msgrcv` без `msgsnd` между ними.





СВОЙСТВО №1

процесс producer:

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MSG_LEN 100
struct msgbuf {
long    mtype;
char    mtext[MSG_LEN + 1];
};

int main(int argc,
        const char* argv[]) {
    key_t key;
    int qid;
    struct msgbuf msg;
    ssize_t size;
    msg.mtype = 1;
    key =
        ftok("./mc_lec2_ex3", 1);
```

```
    if ((qid = msgget(key, IPC_CREAT
                        | 0666)) == -1) {
        msgget_error:
        perror("Error creating
                message queue");
    }
    while (1) {
        printf("Your message: ");
        fgets(msg.mtext, MSG_LEN, stdin);
        if (msgsnd(qid, &msg,
                    strlen(msg.mtext), 0) == -1) {
            perror("Error sending"); continue;
        }
        size = msgrcv(qid, &msg, MSG_LEN, 1, 0);
        if (size > 0) {
            printf("Reply is: %s\n", msg.mtext);
        } else {
            printf("No reply\n");
        }
    }
}
```




СВОЙСТВО №1

процесс producer:

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MSG_LEN 100
struct msgbuf {
long    mtype;
char    mtext[MSG_LEN + 1];
};

int main(int argc,
        const char* argv[]) {
    key_t key;
    int qid;
    struct msgbuf msg;
    ssize_t size;
    msg.mtype = 1;
    key =
        ftok("./mc_lec2_ex3", 1);
```

```
    if ((qid = msgget(key, IPC_CREAT
                        | 0666)) == -1) {
        perror("Error creating
                message queue");
    }
    while (1) {
        printf("Your message: ");
        fgets(msg.mtext, MSG_LEN, stdin);
        if (msgsnd(qid, &msg,
                    strlen(msg.mtext), 0) == -1) {
            perror("Error sending");
            continue;
        }
        size = msgrcv(qid, &msg, MSG_LEN, 1, 0);
        if (size > 0) {
            printf("Reply is: %s\n", msg.mtext);
        } else {
            printf("No reply\n");
        }
    }
}
```



СВОЙСТВО №1

процесс producer:

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stubs.h>
```

```
int main(int argc,
        const char* argv[]) {
```

```
    pass();
    pass();
```

```
        if (msgget() == -1) {

msgget_error:
            pass();

        }
        while (1) {
            pass();
            pass();
            if (msgsnd() == -1) {

                pass();
                continue;

            }
            pass();
            if (any()) {
                pass();
            } else {
                pass();
            }
        }
    }
```



Свойства правильности процесса producer:

- `msgsnd` всегда вызывается только после `msgget`;
- если поток управления попадает в `msgget_error`, то после этого не вызываются функции `msgsnd` и `msgrcv`;
- ни в одном вычислении не встречаются `msgrcv` без `msgsnd` между ними.





СВОЙСТВО №2

процесс producer:

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MSG_LEN 100
struct msgbuf {
long    mtype;
char    mtext[MSG_LEN + 1];
};

int main(int argc,
        const char* argv[]) {
    key_t key;
    int qid;
    struct msgbuf msg;
    ssize_t size;
    msg.mtype = 1;
    key =
        ftok("./mc_lec2_ex3", 1);
```

```
    if ((qid = msgget(key, IPC_CREAT
                        | 0666)) == -1) {
        msgget_error:
        perror("Error creating
                message queue");
    }
    while (1) {
        printf("Your message: ");
        fgets(msg.mtext, MSG_LEN, stdin);
        if (msgsnd(qid, &msg,
                    strlen(msg.mtext), 0) == -1) {
            perror("Error sending"); continue;
        }
        size = msgrcv(qid, &msg, MSG_LEN, 1, 0);
        if (size > 0) {
            printf("Reply is: %s\n", msg.mtext);
        } else {
            printf("No reply\n");
        }
    }
}
```



СВОЙСТВО №2

процесс producer:

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MSG_LEN 100
struct msgbuf {
long    mtype;
char    mtext[MSG_LEN + 1];
};

int main(int argc,
        const char* argv[]) {
    key_t key;
    int qid;
    struct msgbuf msg;
    ssize_t size;
    msg.mtype = 1;
    key =
        ftok("./mc_lec2_ex3", 1);
```

```
    if ((qid = msgget(key, IPC_CREAT
                        | 0666)) == -1) {
        perror("Error creating
                message queue");
    }
    while (1) {
        printf("Your message: ");
        fgets(msg.mtext, MSG_LEN, stdin);
        if (msgsnd(qid, &msg,
                    strlen(msg.mtext), 0) == -1) {
            perror("Error sending");
            continue;
        }
        size = msgrcv(qid, &msg, MSG_LEN, 1, 0);
        if (size > 0) {
            printf("Reply is: %s\n", msg.mtext);
        } else {
            printf("No reply\n");
        }
    }
}
```



Свойство №2

процесс producer:

не
вы
пол
ня
ет
ся
!

```
#include <stub.h>

int main(int argc,
          const char* argv[]) {

    pass();
    pass();

    if (msgget() == -1) {
msgget_error:
        pass();
    }
    while (1) {
        pass();
        pass();
        if (msgsnd(-1) == -1) {
            pass();
            continue;
        }
        msgrcv();
        if (any()) {
            pass();
        } else {
            pass();
        }
    }
}
```

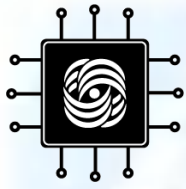
подходит и
для проверки
свойства №3



Свойства правильности процесса producer:

- `msgsnd` всегда вызывается только после `msgget`;
- если поток управления попадает в `msgget_error`, то после этого не вызываются функции `msgsnd` и `msgrcv`;
- ни в одном вычислении не встречаются `msgrcv` без `msgsnd` между ними.





Корректная, но практически бесполезная модель

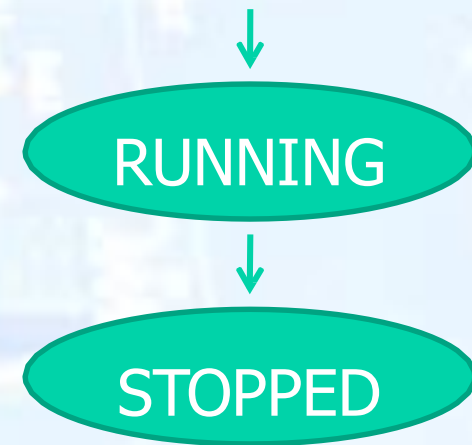
```
int main() {
```

```
enum {RUNNING, STOPPED} state;
```

```
state = RUNNING;
```

```
state = STOPPED;
```

```
}
```





Построение модели

- Задачи:

1. Постановка задачи моделирования

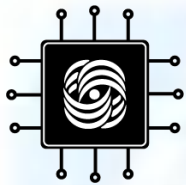
не все требования к программе формализуемы

2. Формализация описания программы

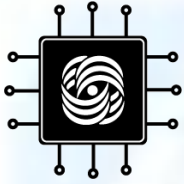
описание программы предназначено для удобства разработки и компиляции, но не для проверки её свойств

3. Абстракция

уменьшение пространства состояний программы



Спасибо за внимание!



НАДЁЖНОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Лекция 3:

Системы переходов (LTS). Корректность и адекватность LTS модели

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



План лекции

- Система понятий, используемых в курсе
- Размеченные системы переходов (LTS)
- Недетерминизм систем переходов
- Пути, вычисления, трассы
- Свойства линейного времени
- Корректность модели
- Адекватность модели



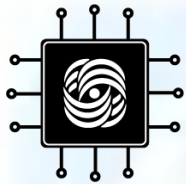
Итоги прошлой лекции

- Свойства проверяются на состояниях и их последовательностях
- Чтобы перебирать меньше состояний, исследуется не исходная система, а её модель
- Модель должна быть простой, корректной и адекватной
- В этом случае из правильности модели следует правильность программы

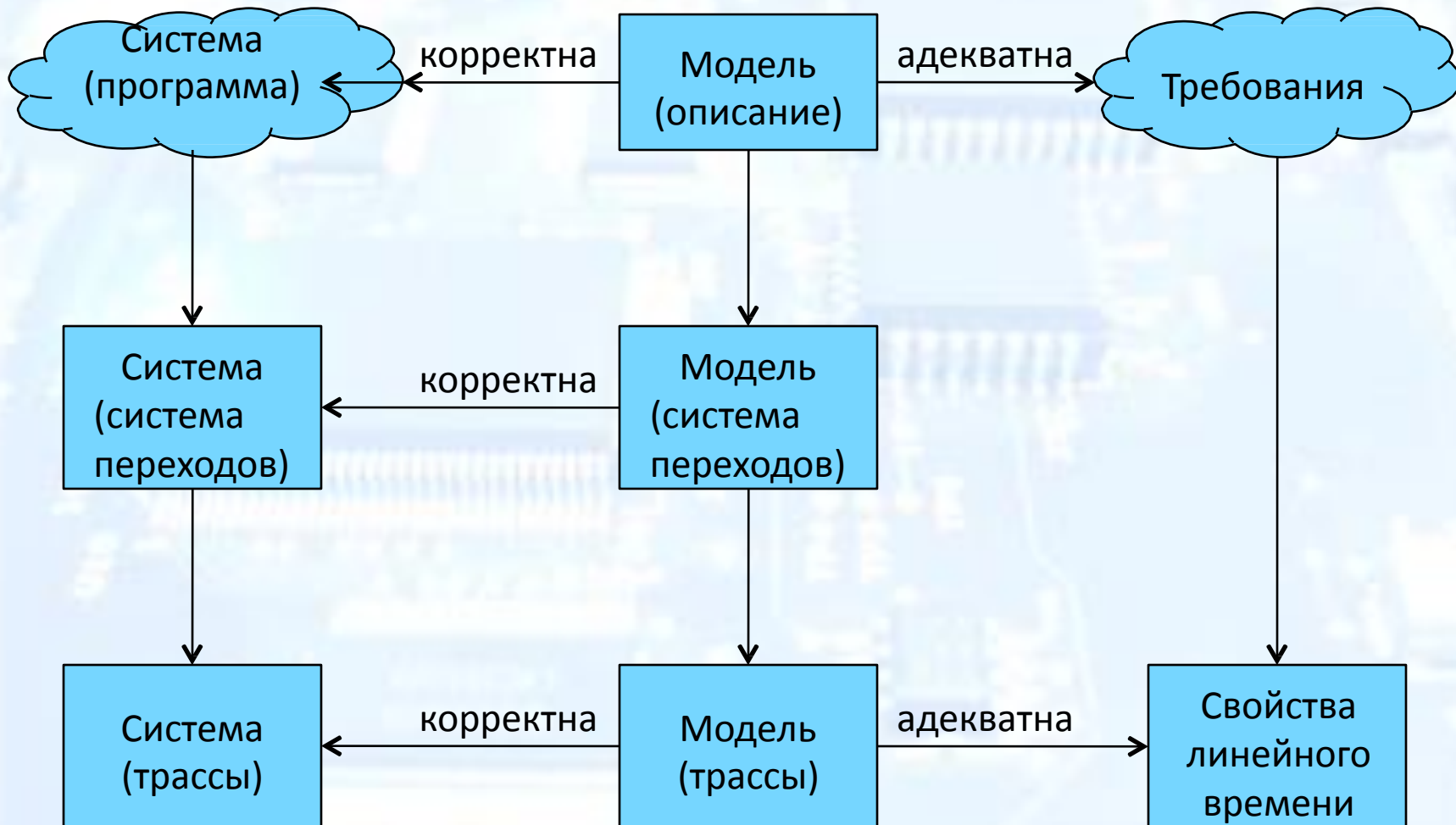


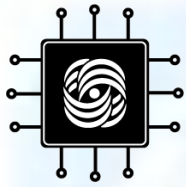
Строго говоря...

Если мы хотим доказать, что после проверки правильности на модели в программе нет ошибок, то все эти понятия стоит сформулировать более строго

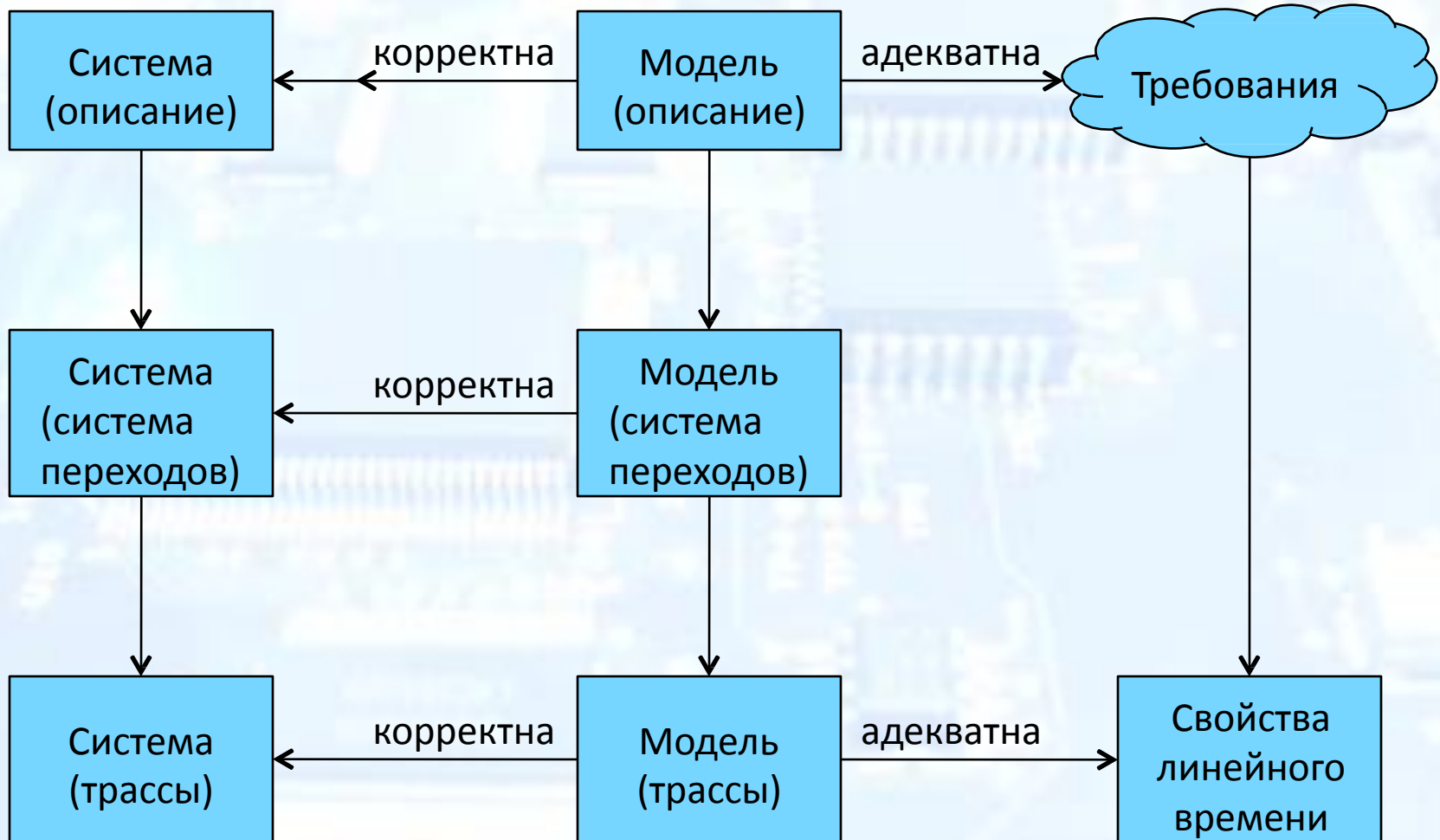


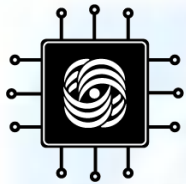
Строго говоря, всё должно быть так:





Строго говоря, всё должно быть так:

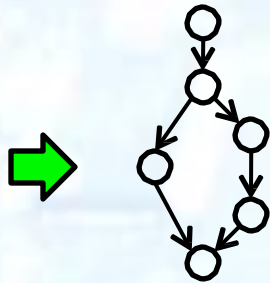




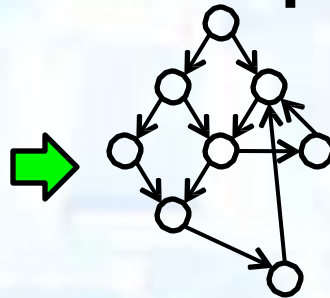
Различные представления программы

```
int main  
( ) {  
  printf(  
  }  
}
```

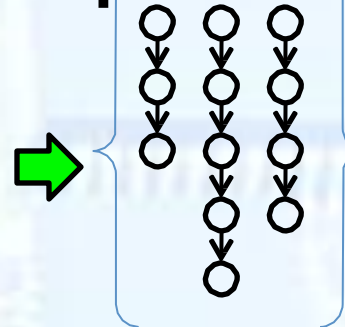
Исходный код программы



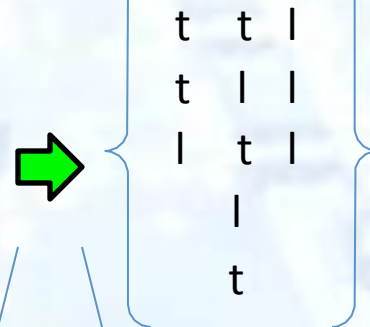
Граф программы (ACFG)



Размеченная система переходов



Множество вычислений



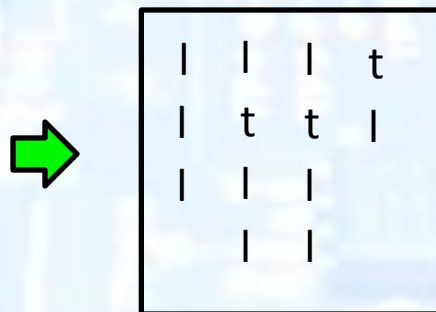
Множество трасс

Взлетает,
не падает,
приземляется

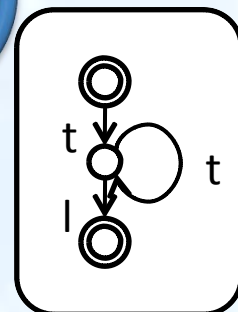
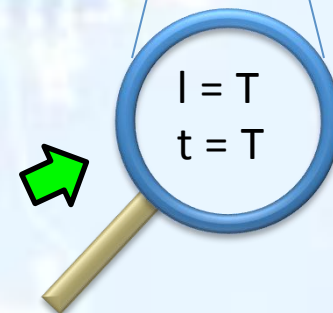
Требования

$\square ($
TAKEOFF $\rightarrow$
(!FALL)
 $\cup$
(LANDED) $)$

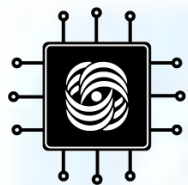
Спецификация
(линейного времени)



Допустимые
последовательности
атомарных высказываний



Язык
допустимых трасс

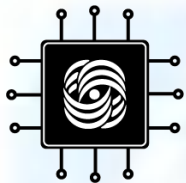


Размеченные системы переходов (LTS)

$$TS = \langle S, Act, \xrightarrow{a}, I, AP, L \rangle$$

- S – множество состояний,
- Act – множество действий, τ – невидимое действие,
- $\xrightarrow{a} \subseteq S \times Act \times S$ – **тотальное** отношение переходов,
- $I \subseteq S$ – **множество** начальных состояний,
- AP – множество атомарных высказываний,
- $L : S \rightarrow 2^{AP}$ – функция разметки.

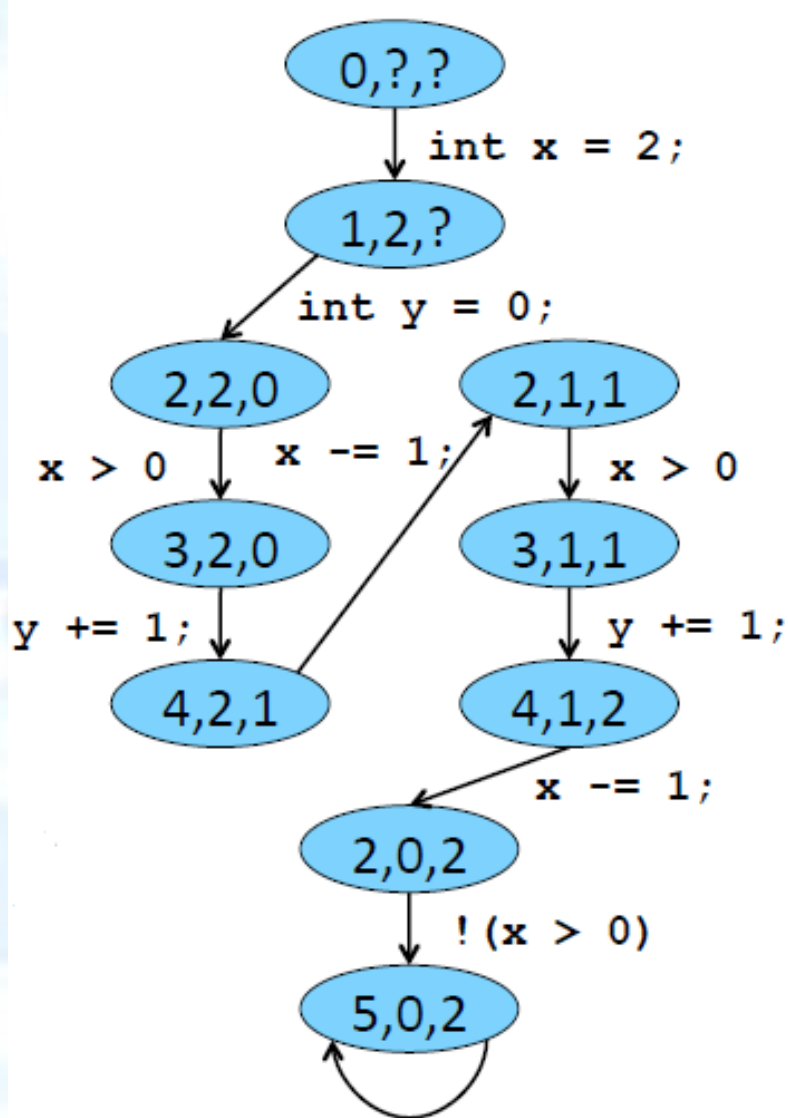
$$\left(\begin{array}{l} S \text{ – конечное или счётное множество, } Act \text{ – конечное} \\ \text{Нотация: } \langle s, a_0, s' \rangle \in \xrightarrow{a} \equiv s \xrightarrow{a_0} s' \end{array} \right)$$

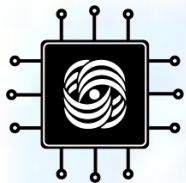


Пример LTS

→ 0: `int x = 2;`
1: `int y = 0;`
2: `while (x > 0)`
 {
3: `y += 1;`
4: `x -= 1;`
 }
5:

[Состояние: (счётчик, x, y)]

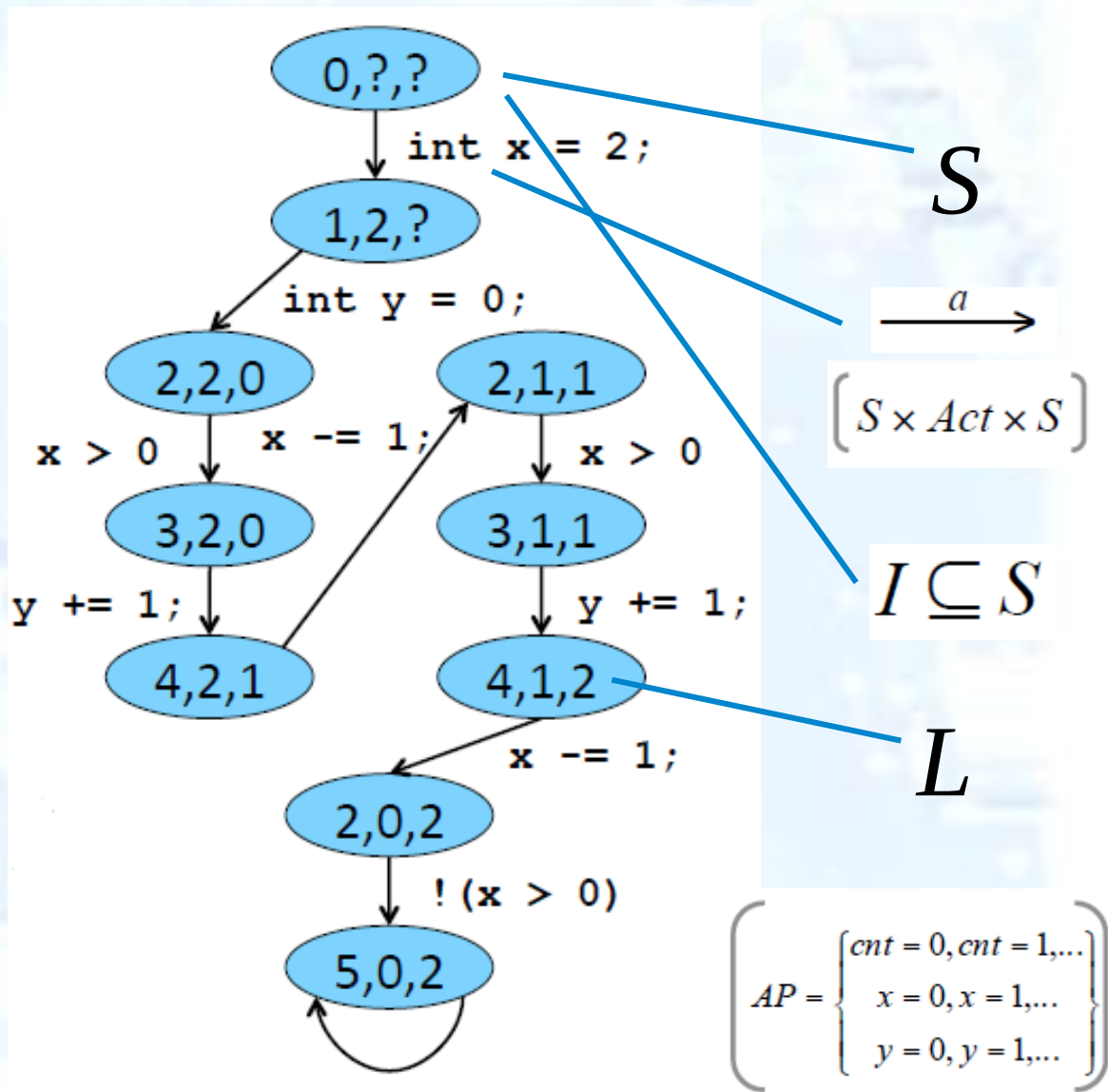


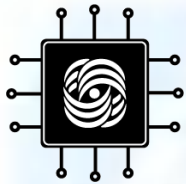


Пример LTS

→ 0: `int x = 2;`
1: `int y = 0;`
2: `while (x > 0)`
 {
3: `y += 1;`
4: `x -= 1;`
 }
5:

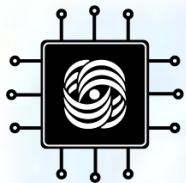
[Состояние: (счётчик, x, y)]





Что включать в состояние

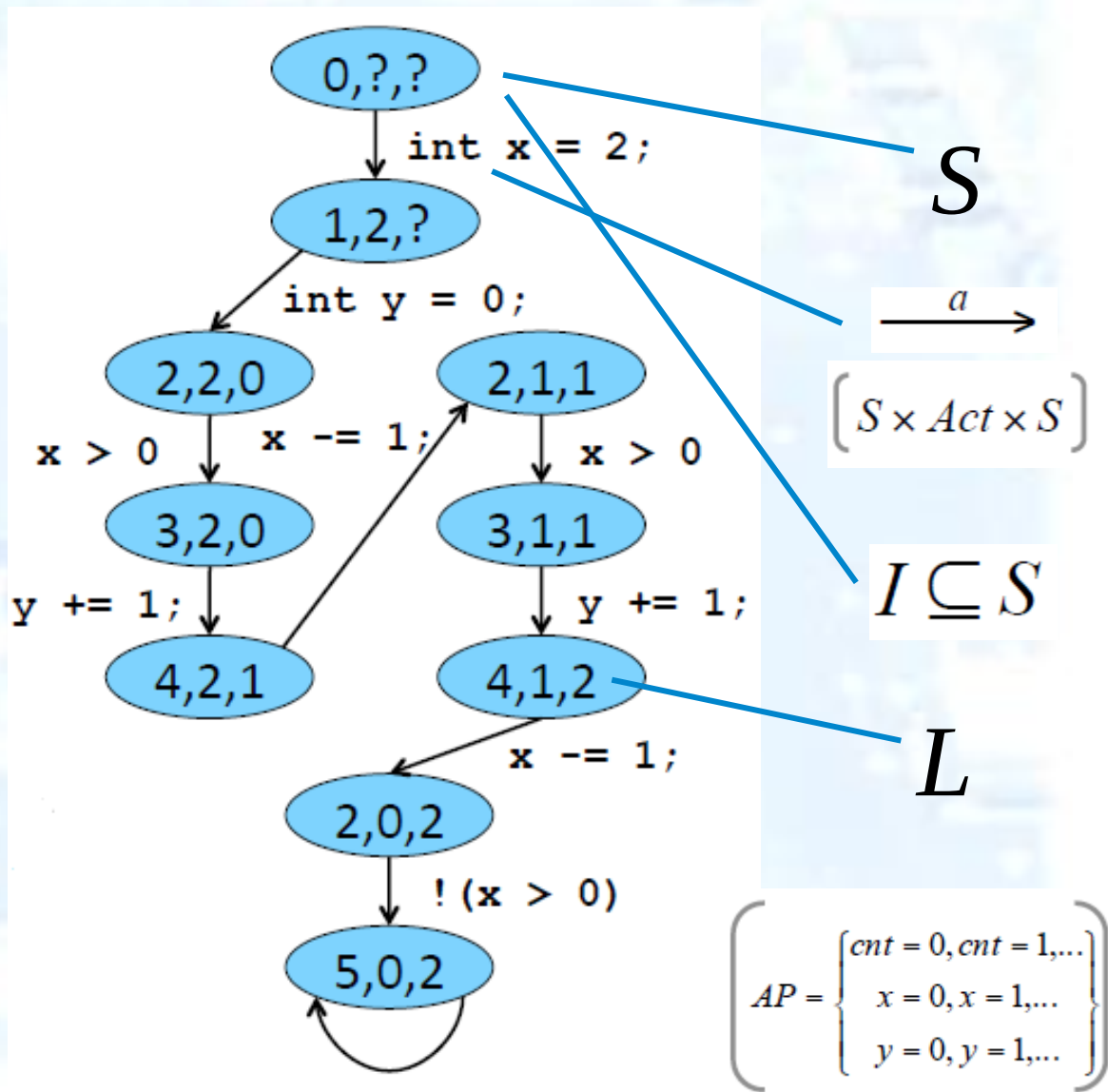
- Набор атомарных высказываний **AP** определяется свойствами, которые нужно проверить
- Изменение состояния связано с изменением выполнимости хотя бы одного атомарного высказывания
- Исходя из этого мы определяем, что включать в состояние программы
- Главное – не «потерять» ни одного изменения атомарных состояний

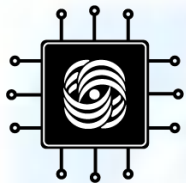


Пример LTS

➔ 0: `int x = 2;`
 1: `int y = 0;`
 2: `while (x > 0)`
 {
 3: `y += 1;`
 4: `x -= 1;`
 }
 5:

[Состояние: (счётчик, x, y)]

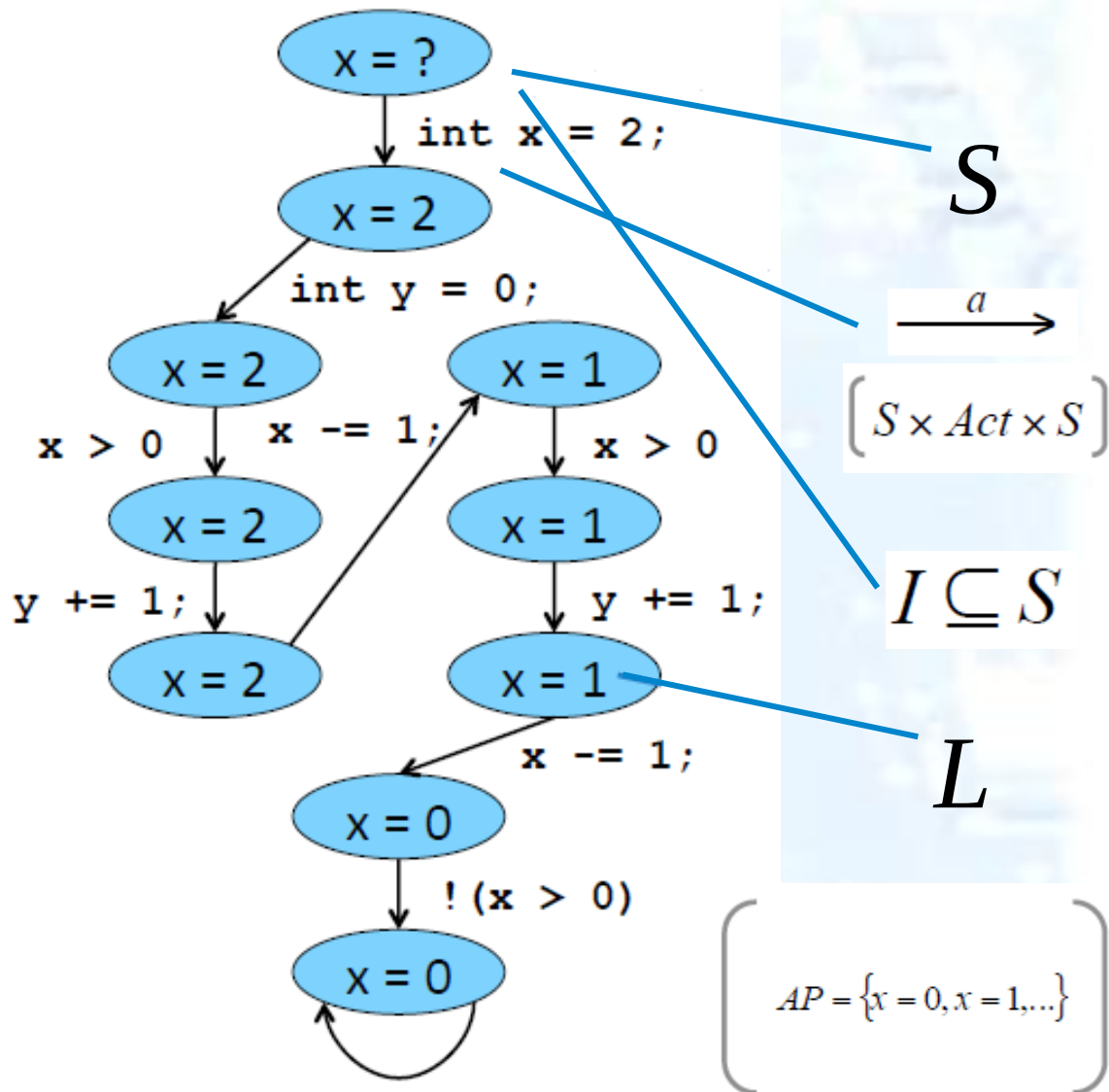


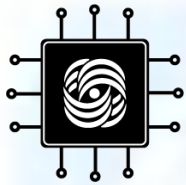


Пример LTS

→ 0: `int x = 2;`
1: `int y = 0;`
2: `while (x > 0)`
 {
3: `y += 1;`
4: `x -= 1;`
 }
5:

[Состояние: (счётчик, x, y)]

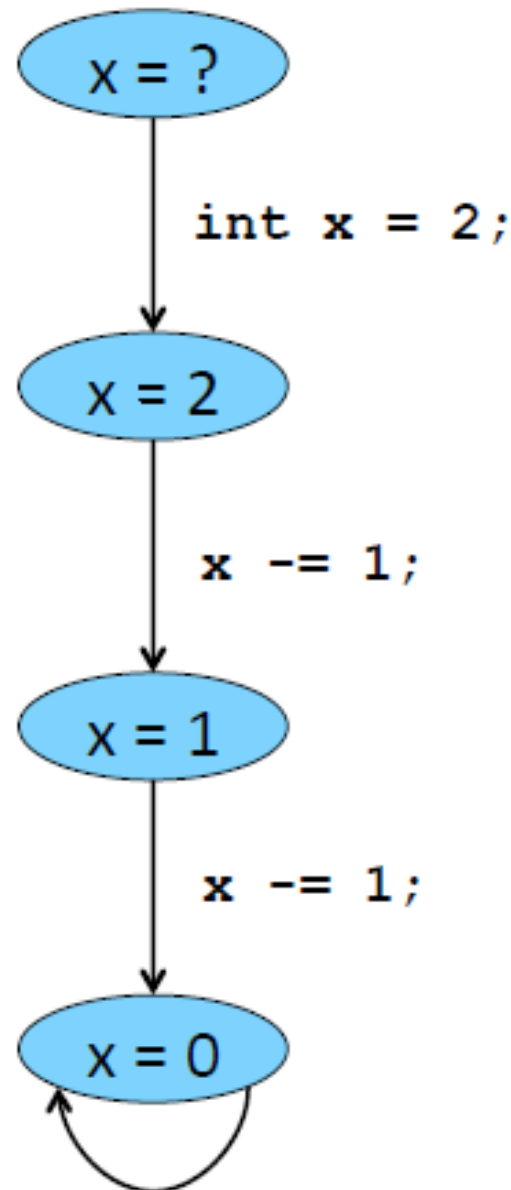




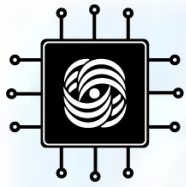
Пример LTS

→ 0: `int x = 2;`
1: `int y = 0;`
2: `while (x > 0)`
 {
3: `y += 1;`
4: `x -= 1;`
 }
5:

[Состояние: (счётчик, x, y)]



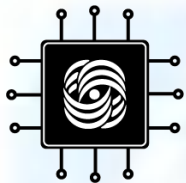
[$AP = \{x = 0, x = 1, \dots\}$]



Что включать в состояние?

В дальнейшем мы будем рассматривать «универсальное» определение состояния, достаточное для проверки свойств линейного времени и локализации их нарушения в описании программы

Совокупность значения счётчиков управления последовательных процессов и переменных программы



Пример LTS

```
int p;
```

```
Process Prod() {
```

```
    while(1)
```

```
1.1:    if(p < 2)
```

```
1.2:    p += 1;
```

```
}
```

```
process Cons() {
```

```
    while(1)
```

```
2.1:    if(p > 0)
```

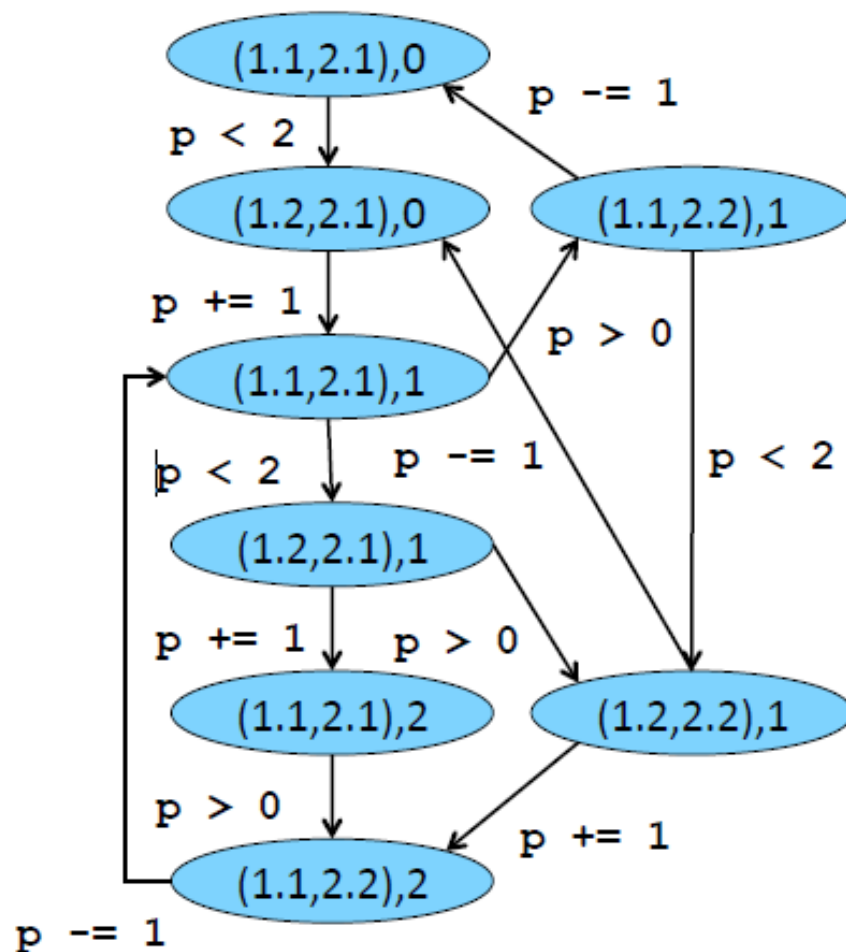
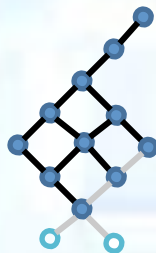
```
2.2:    p -= 1;
```

```
}
```

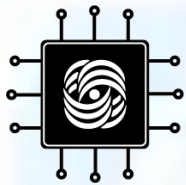
Состояние: (счётчик Prod,
счётчик Cons, p)

Часть состояний не показана
(оператор беск. цикла,
стартовые состояния)

Prod ↔ Cons

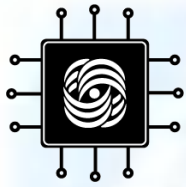


Бесконечное количество вычислений, однако
размеченная система переходов конечна¹⁶



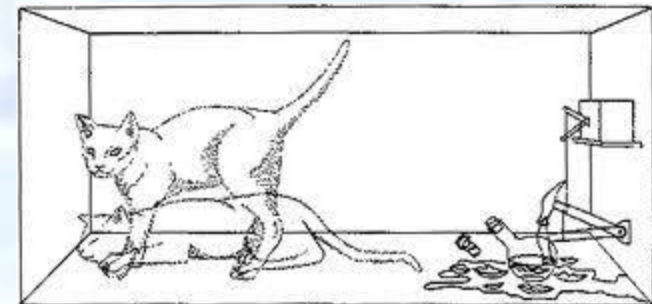
Недетерминизм

- В ряде ситуаций шаг может сделать любой из двух процессов, порядок действий не определён
- **Недетерминизм = неопределённость**
- При построении LTS рассматриваются все возможные варианты последовательности действий



Недетерминизм

- Вообще-то в природе довольно мало недетерминированных процессов
- Да и те считаются недетерминированными, поскольку физические законы, по которым выбирается действие, нам не известны
- Недетерминизм появляется, как только мы не знаем причин выбора действия или считаем их несущественными

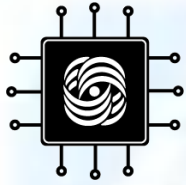




Недетерминизм

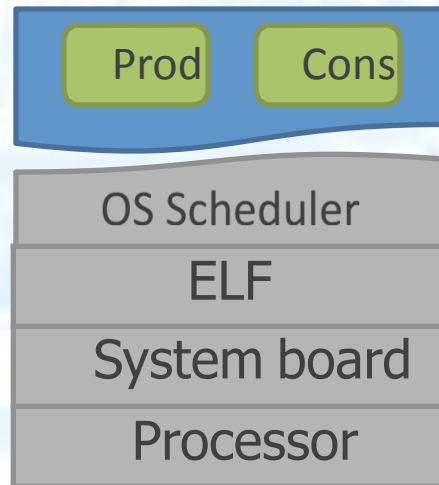
- Порядок выполнения Prod и Cond на конкретном компьютере детерминирован и определяется алгоритмом диспетчера операционной системы и состоянием ресурсов
- Для проверки правильности программы мы решили абстрагироваться от всего, кроме описания двух процессов
- Итог – недетерминизм очерёдности выполнения процессов

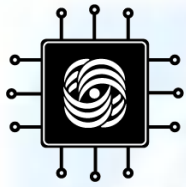
（ Подробнее про разные виды недетерминизма –
ниже ）



Недетерминизм – это фича!

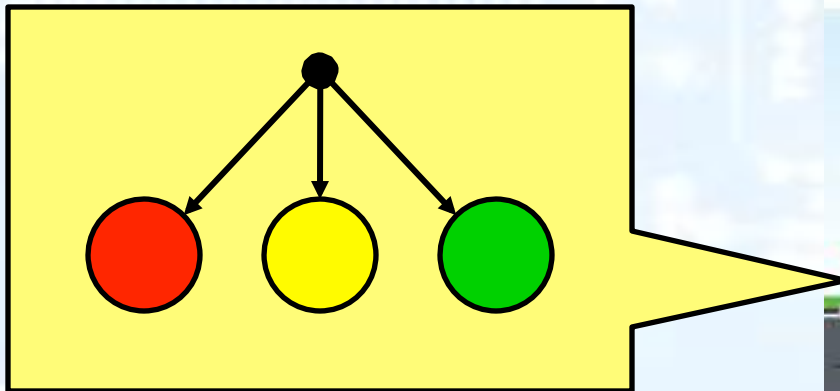
- Используется для:
 - моделирования параллельного выполнения процессов в режиме чередования (интерливинга)
 - позволяет абстрагироваться от алгоритма диспетчеризации и скорости выполнения процессов

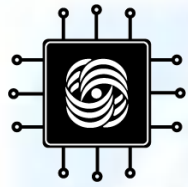




Недетерминизм – это фишка!

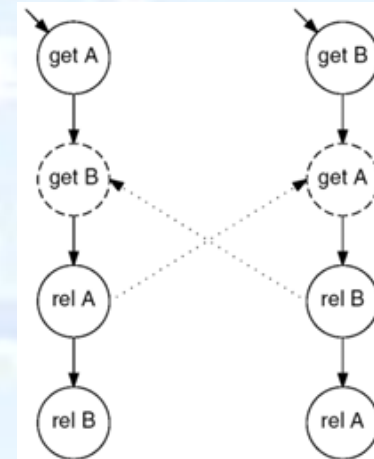
- Используется для:
 - моделирования прототипа системы
 - не ограничивает будущую реализацию заданным порядком выполнения операторов или конкретными входными данными

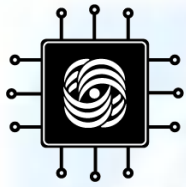




Недетерминизм – это фишка!

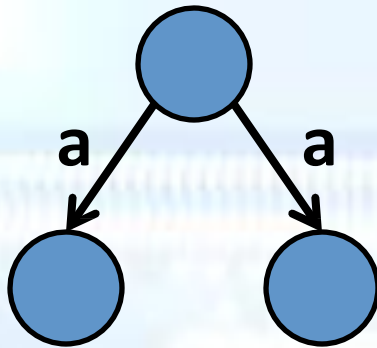
- Используется для:
 - построения абстракции реальной системы
 - для абстрагирования от деталей, несущественных для проверки свойств и для построения модели по неполной информации



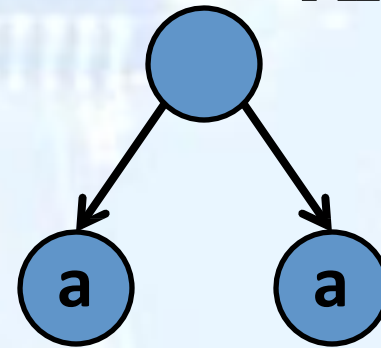


Недетерминизм в LTS

- В LTS недетерминизм проявляется в виде состояний, из которых можно перейти более чем в одно состояние

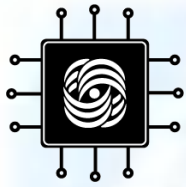


недетерминизм
действий



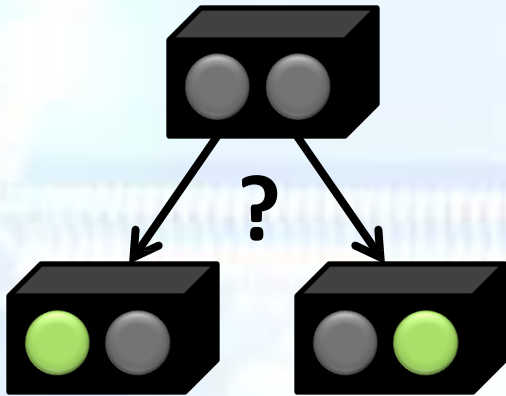
недетерминизм
атомарных высказываний

{ строгие определения – далее }

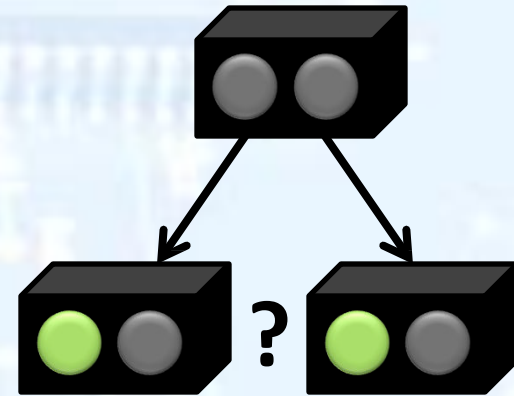


Недетерминизм в LTS

- В LTS недетерминизм проявляется в виде состояний, из которых можно перейти более чем в одно состояние

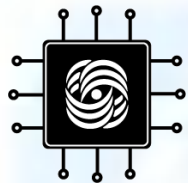


может произойти одно
из нескольких действий



наблюдатель не может
отличить два состояния

{ строгие определения – далее }



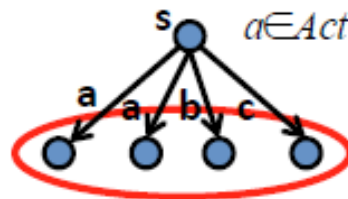
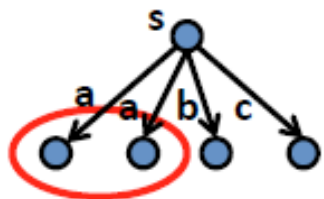
Вспомогательные определения

Прямые потомки вершины s

по действию a

по всем действиям

$$Post(s, a) = \{s' \in S \mid s \xrightarrow{a} s'\}, Post(s) = \bigcup_{a \in Act} Post(s, a)$$

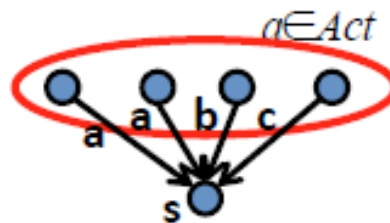
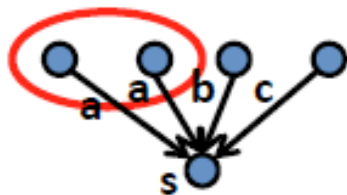


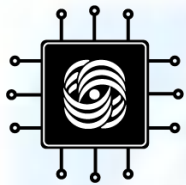
Прямые предки вершины s

по действию a

по всем действиям

$$Pre(s, a) = \{s' \in S \mid s' \xrightarrow{a} s\}, Pre(s) = \bigcup_{a \in Act} Pre(s, a)$$





Детерминизм

- Система $TS = \langle S, Act, \xrightarrow{a}, I, AP, L \rangle$

детерминирована

– по действиям, тогда и только тогда:

$$|I| \leq 1 \quad \text{и} \quad |Post(s, a)| \leq 1, \forall s \in S, a \in Act$$

– по атомарным высказываниям, тогда и только тогда:

$$|I| \leq 1 \quad \text{и} \quad \underbrace{|Post(s) \cap \{s' \in S \mid L(s') = A\}|}_{\text{одинаково размеченные потомки } s} \leq 1, \forall s \in S, A \in 2^{AP}$$

одинаково размеченные потомки s



Путь

- *Конечным путём* σ системы переходов TS называется такая последовательность чередующихся состояний и действий, заканчивающаяся состоянием:

$$\sigma = s_0 a_1 s_1 a_2 s_2 \dots a_n s_n, \text{ что } s_i \xrightarrow{a_{i+1}} s_{i+1} \quad \forall 0 \leq i < n.$$

- *Бесконечным путём* σ системы переходов TS называется такая бесконечная последовательность чередующихся состояний и действий:

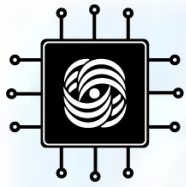
$$\sigma = s_0 a_1 s_1 a_2 s_2 a_3 \dots, \text{ что } s_i \xrightarrow{a_{i+1}} s_{i+1} \quad \forall 0 \leq i.$$

- Путь называется *начальным*, если $s_0 \in I$

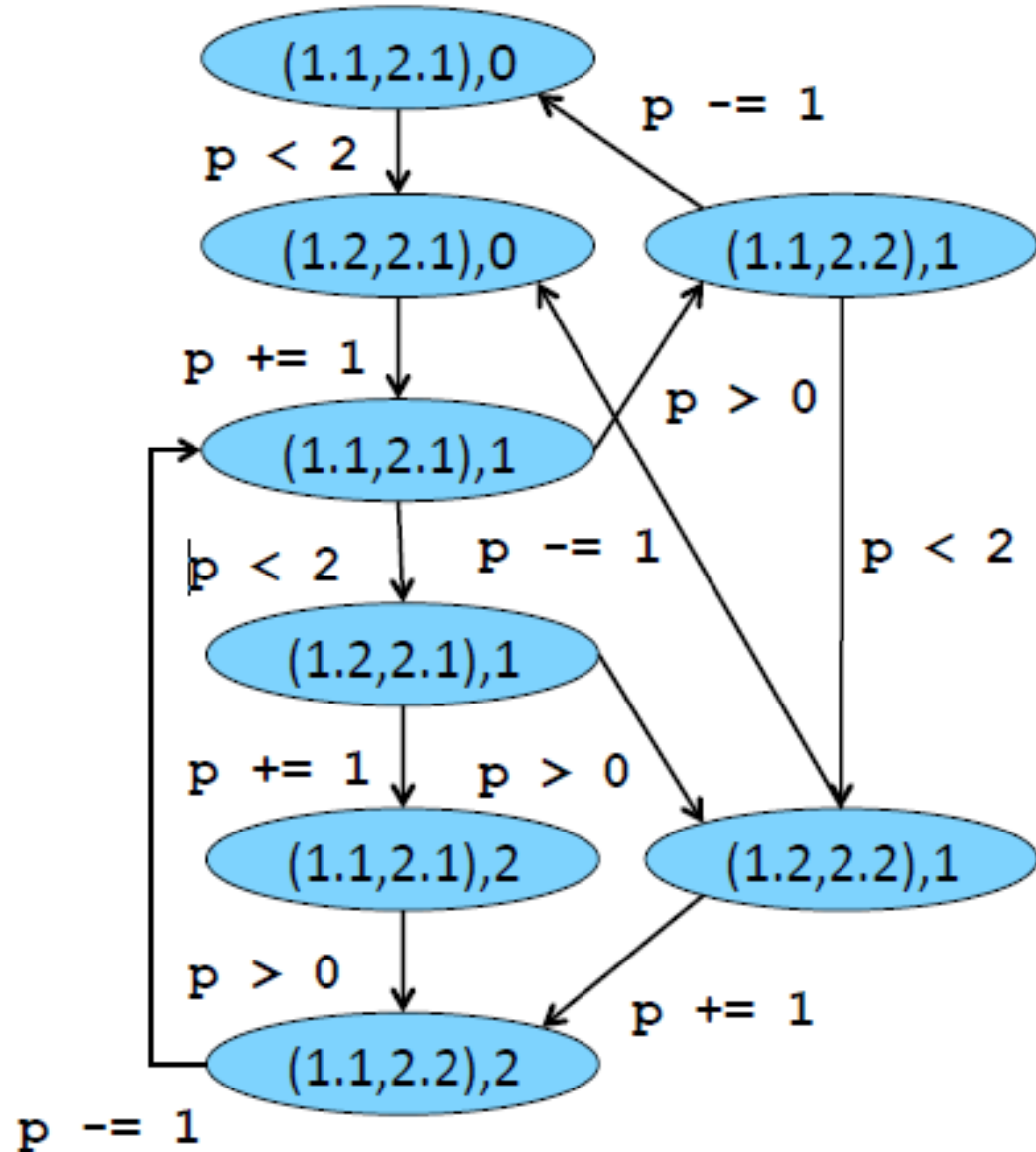
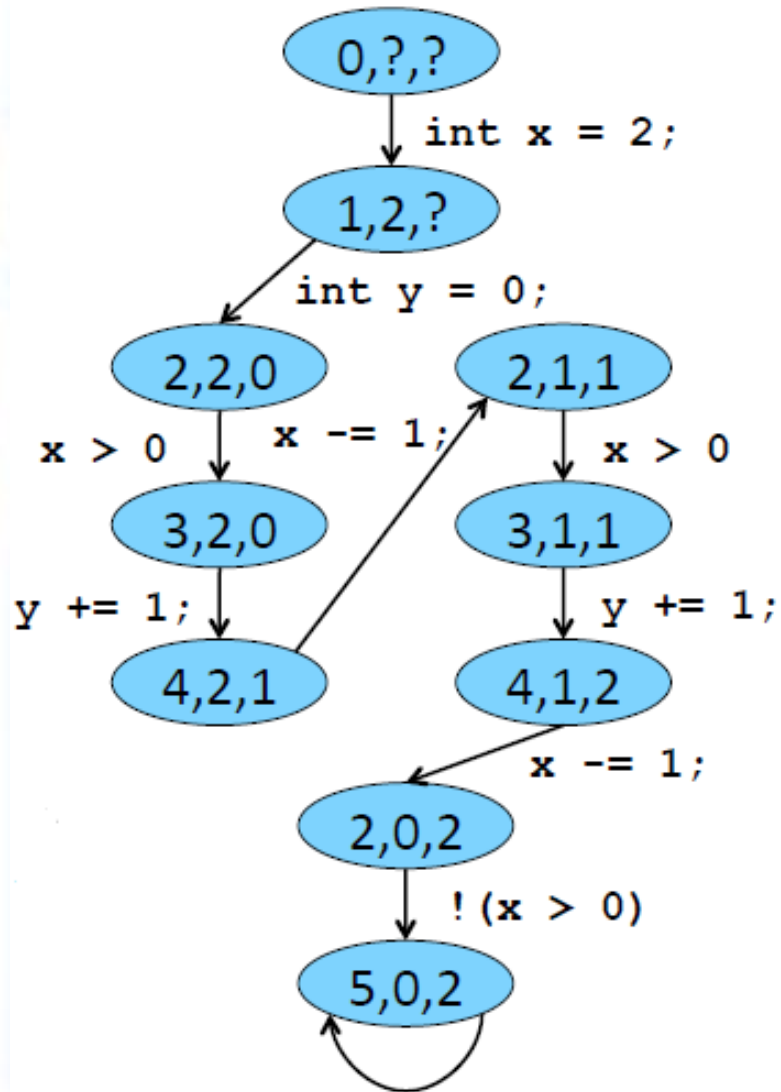


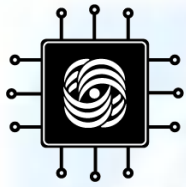
Вычисление

- Путь называется *максимальным*, если он бесконечен.
- *Вычислением* системы переходов TS называется начальный максимальный путь.

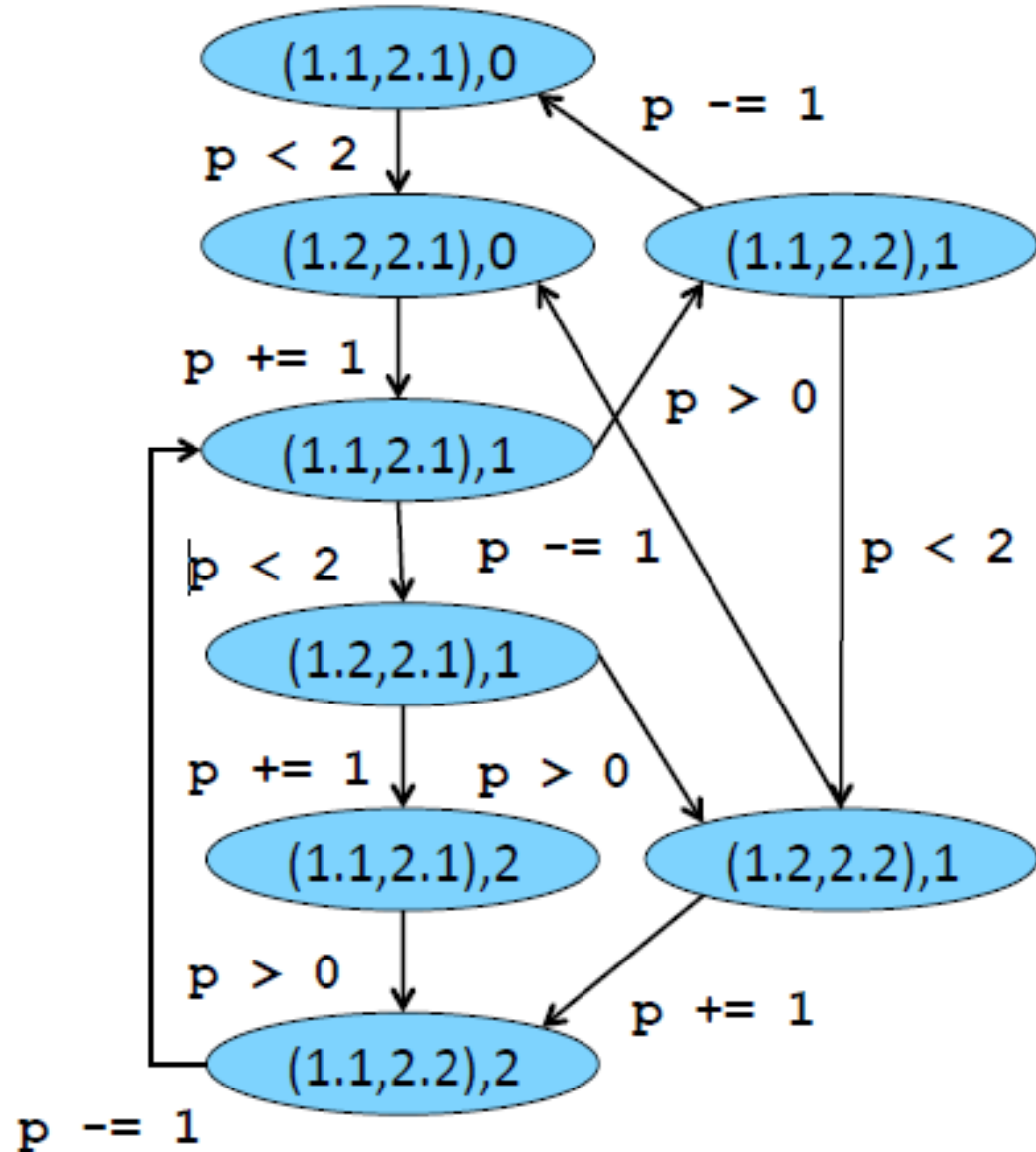
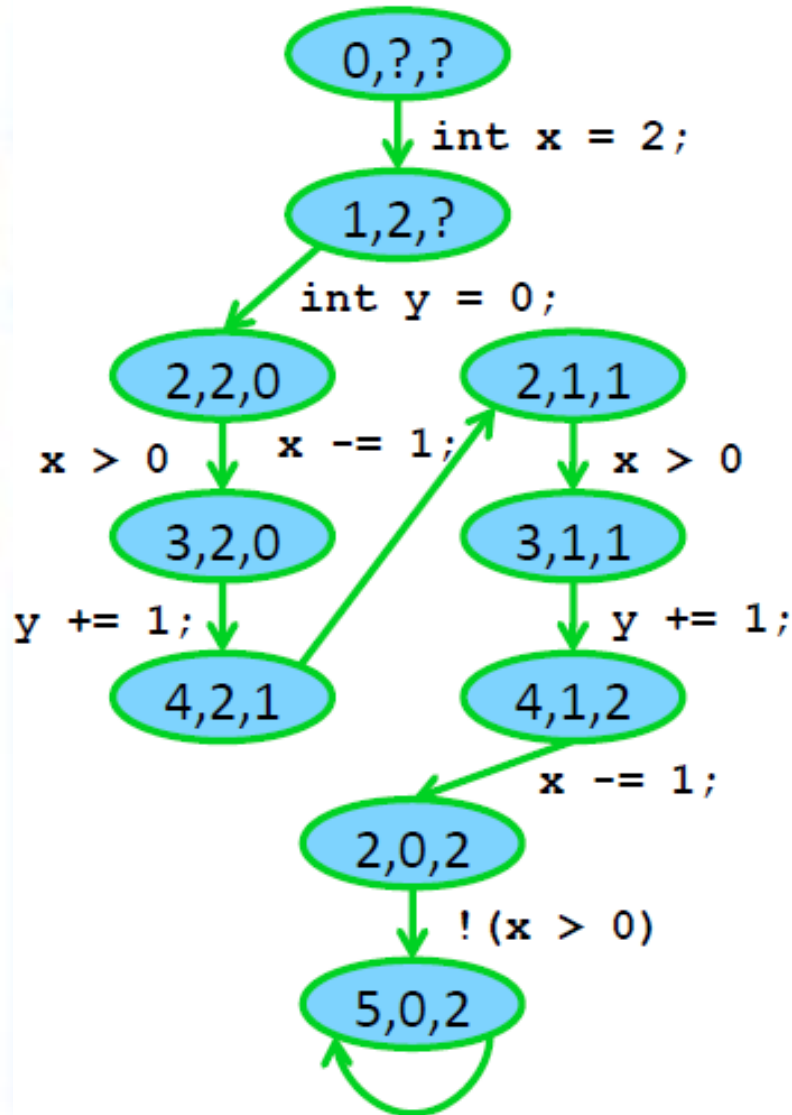


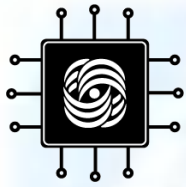
Примеры вычислений



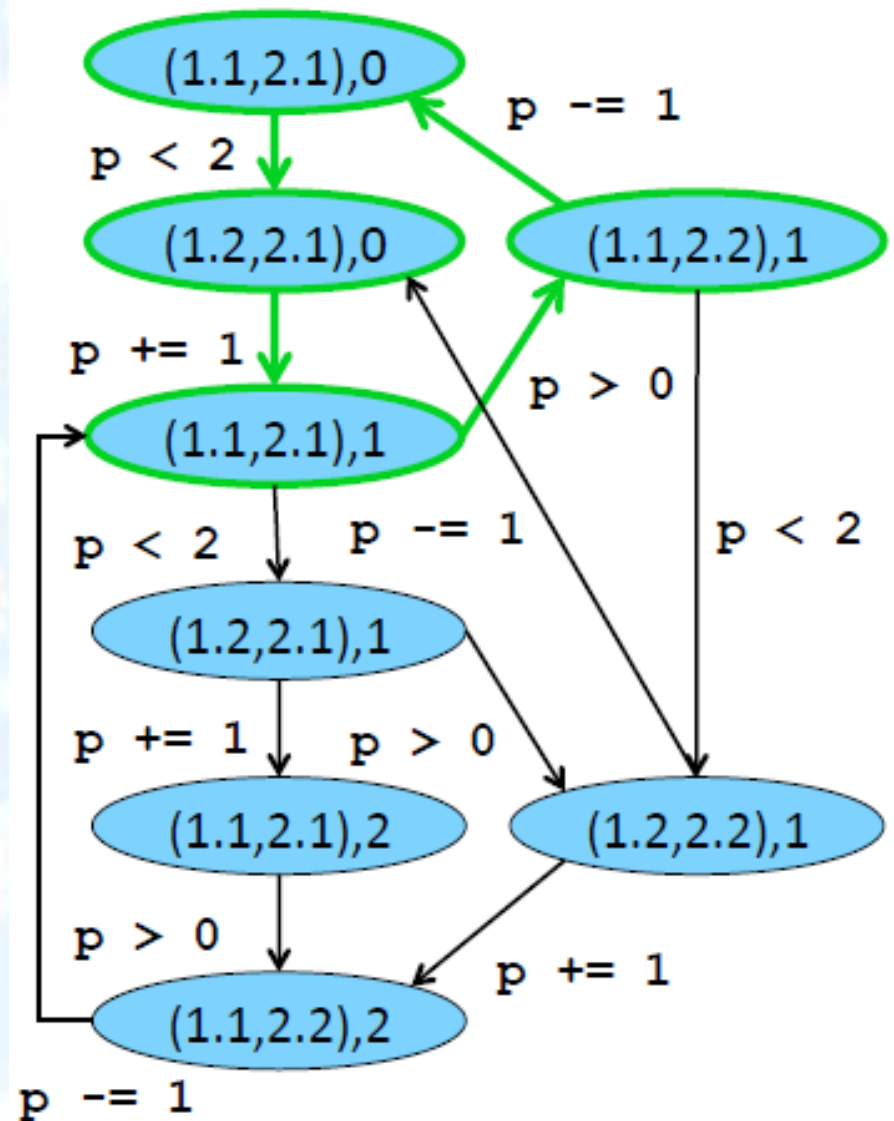
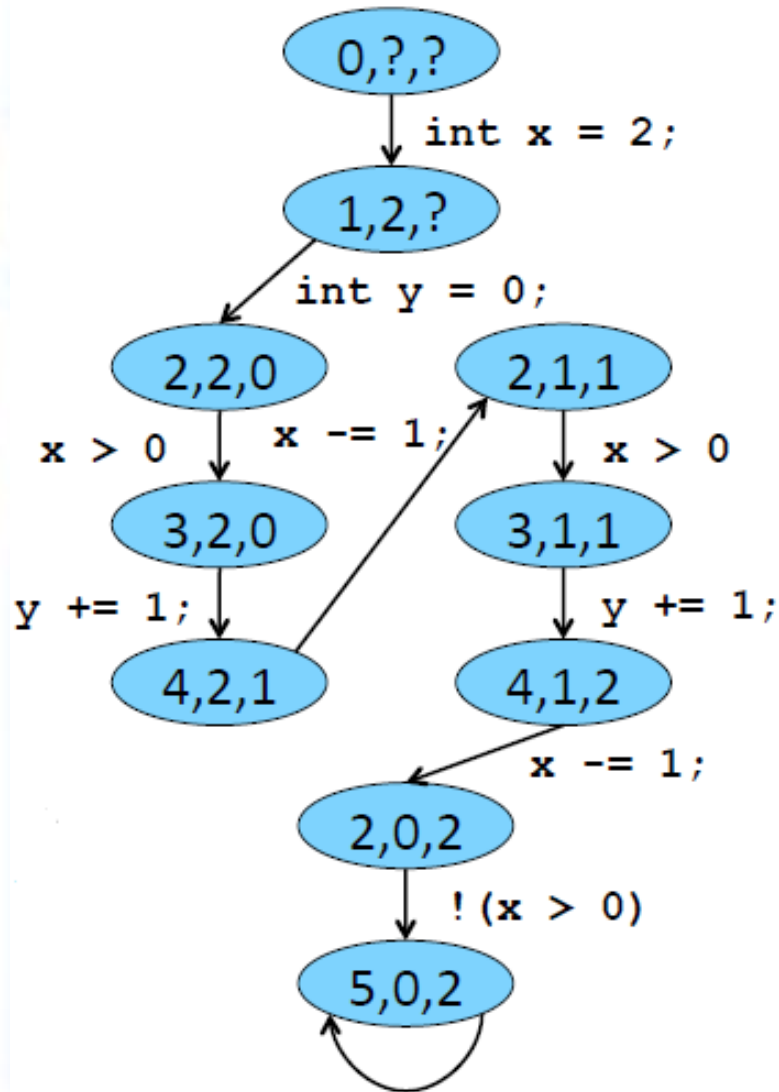


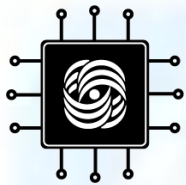
Примеры вычислений



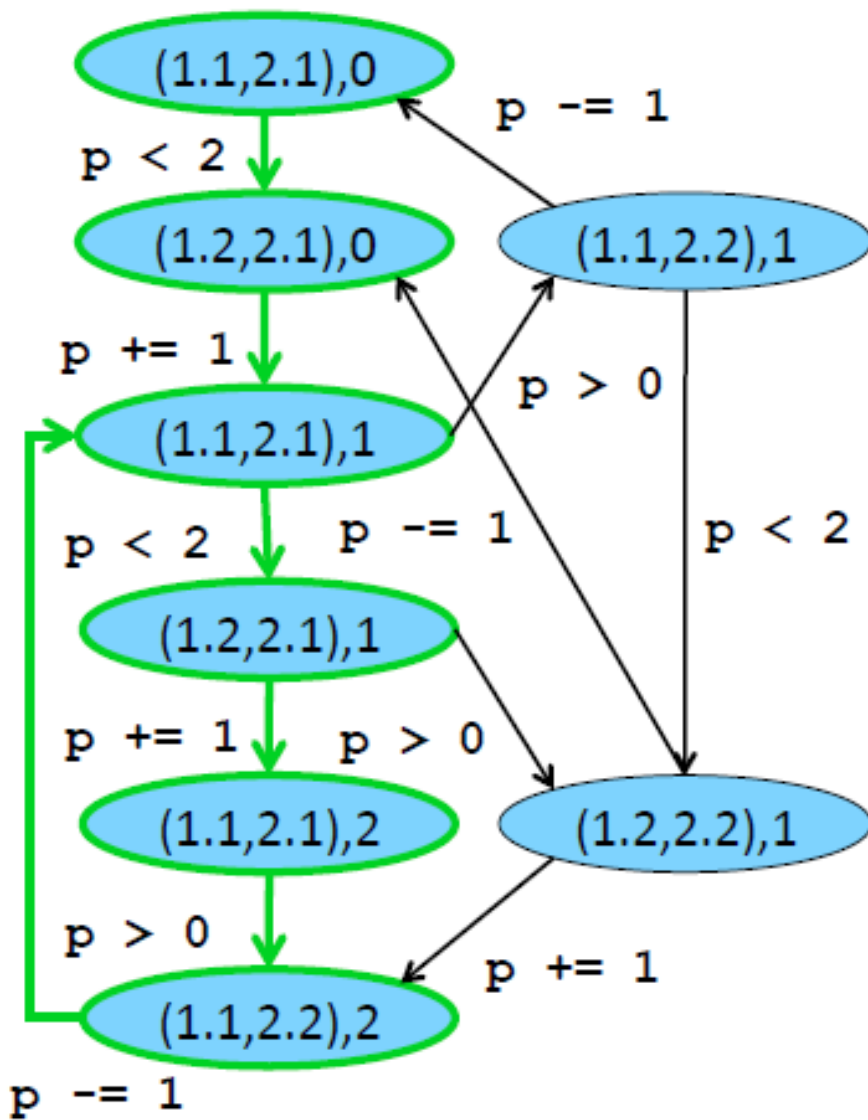
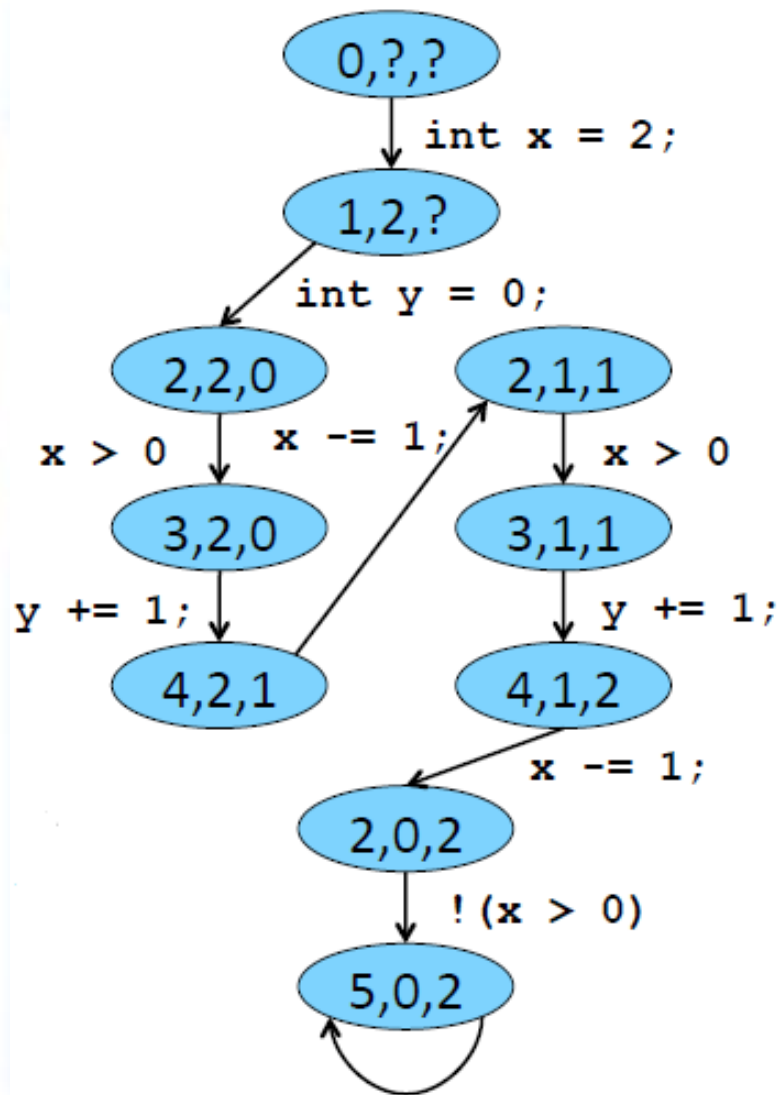


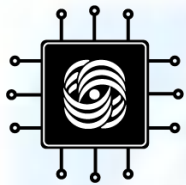
Примеры вычислений



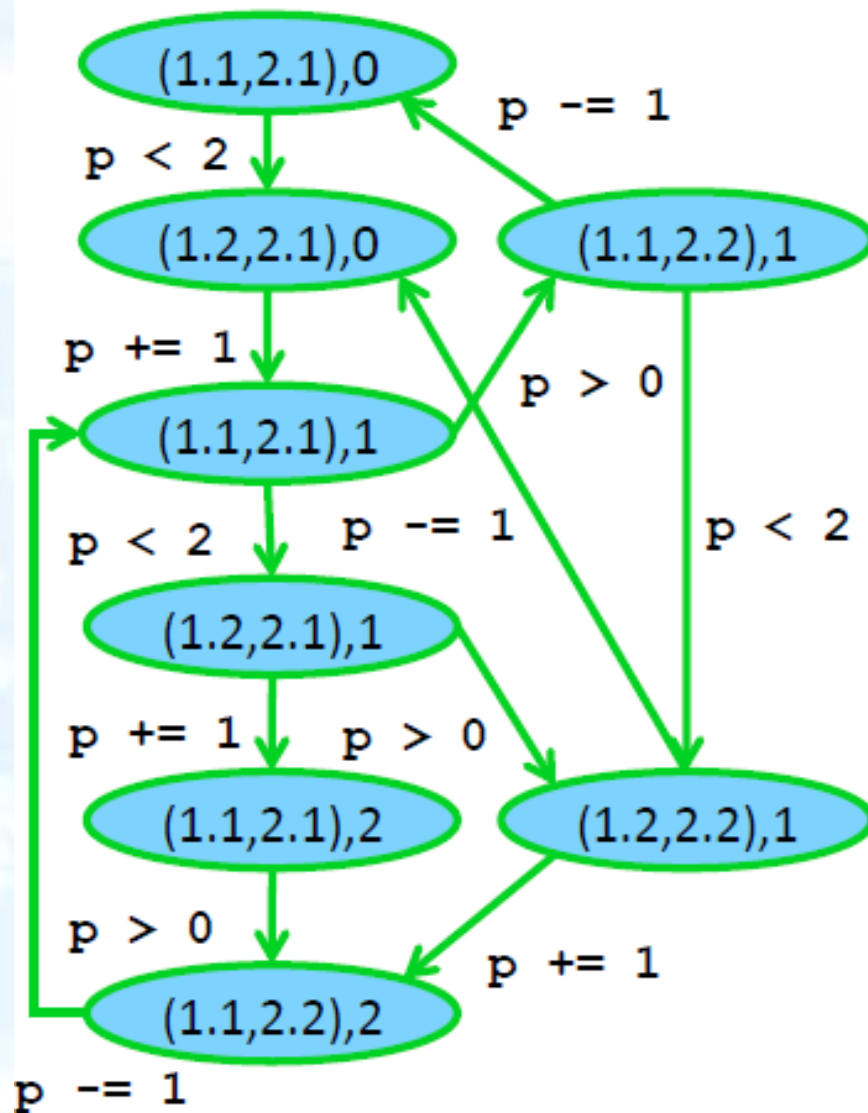
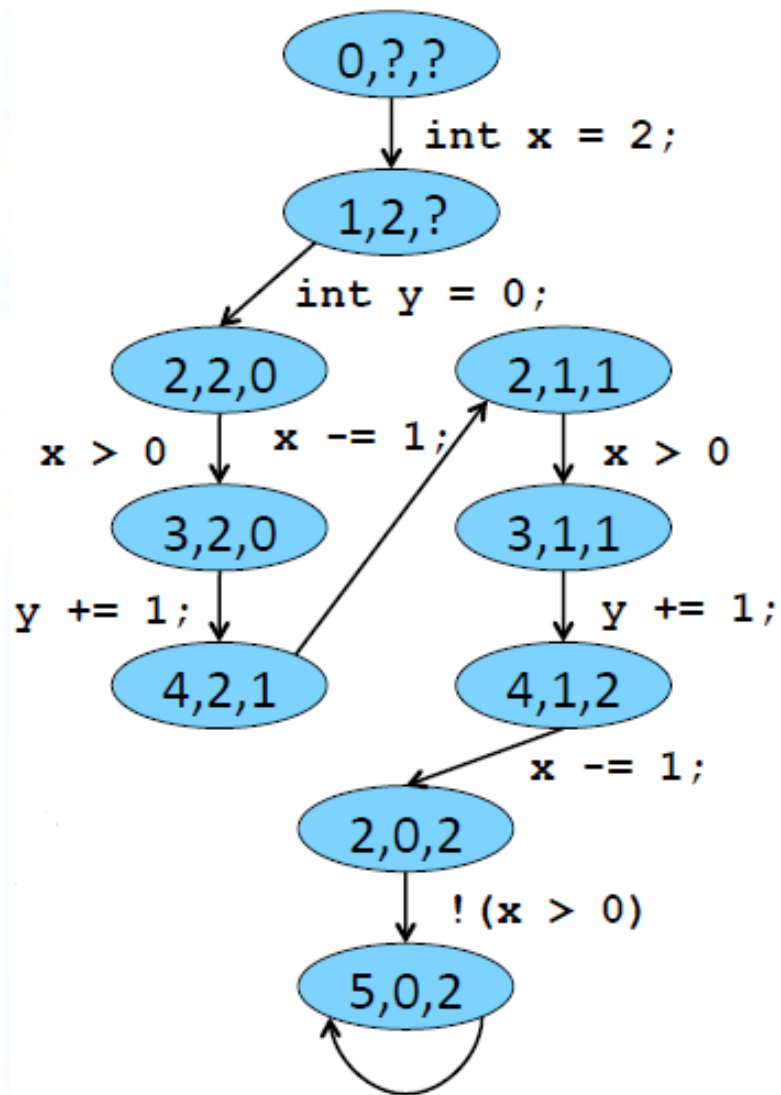


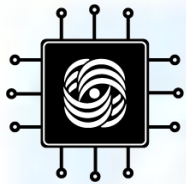
Примеры вычислений





Примеры вычислений





Достижимость состояний

- Состояние $s \in S$ называется достижимым (из начального) в системе переходов TS, если существует начальный, конечный путь

$$S_0 \xrightarrow{a_1} S_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} S_n = S.$$

- $Reach(TS)$ обозначает множество всех состояний, достижимых в TS.



Трассы

- Вычисление описывает последовательность состояний и действий; что происходит в системе. Требуется для описания семантики программы (позже).
- Свойства корректности формулируются в терминах последовательностей значений атомарных высказываний в состояниях модели.



Трассы

- Система переходов:

$$TS = \langle S, Act, \xrightarrow{a}, I, AP, L \rangle$$

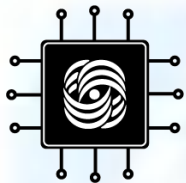
- Путь (фрагмент вычисления):

$$\sigma = s_0 a_1 s_1 a_2 s_2 a_3 \dots$$

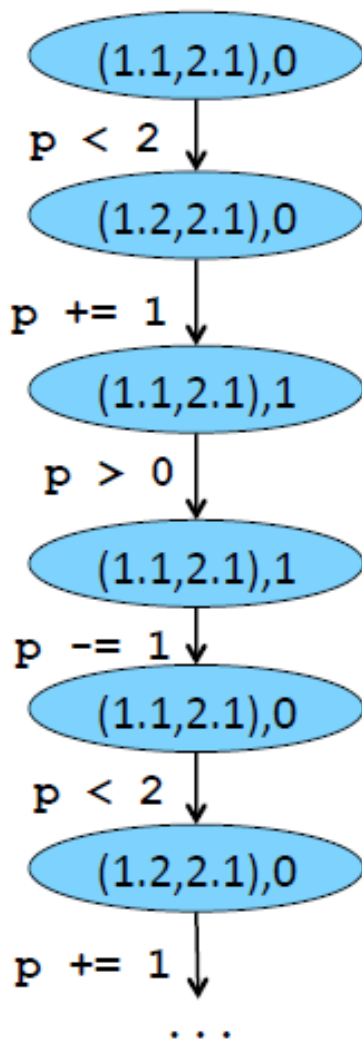
- Трасса:

$$tr = L(s_0)L(s_1)L(s_2)\dots \in (2^{AP})^\omega$$

〔 фокусируемся на «наблюдаемом» поведении 〕



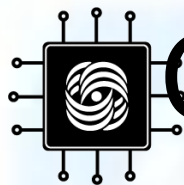
вычисление



Примеры трасс

трасса $\{a \equiv p = 0\}$ трасса 2 $\left\{ \begin{array}{l} a \equiv p \neq 0 \\ b \equiv p = 1 \end{array} \right\}$

a	!a !b
a	!a !b
!a	a
!a	~~a~~ b
a	!a !b
a	!a !b



Свойства линейного времени

- Атомарные высказывания:

$\left\{ \begin{array}{l} \mathbf{a} - \text{«находимся в точке отправки запроса»} \\ \mathbf{b} - \text{«находимся в точке приёма ответа»} \end{array} \right\}$

- СВОЙСТВО:

«после отправки запроса рано или поздно получим ответ»

- Пример допустимых трасс:

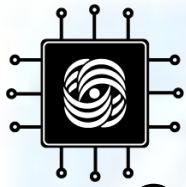
$a \ \emptyset \ \emptyset \ \emptyset \ \emptyset \ \emptyset \ b \ \emptyset \ \dots$

$\emptyset \ \emptyset \ \emptyset \ \emptyset \ \emptyset \ \emptyset \ \emptyset \ \dots$

- Пример недопустимых трасс:

$a \ \emptyset \ \emptyset \ \emptyset \ \emptyset \ \emptyset \ \emptyset \ \emptyset \ \dots$

$a \ \emptyset \ \emptyset \ \emptyset \ \emptyset \ b \ a \ \emptyset \ \dots$



Свойства линейного времени

- Свойство φ определяет набор допустимых трасс

$$\varphi \subseteq (2^{AP})^\omega$$

- Свойство φ выполнимо на трассе σ :

$$\sigma \models \varphi \Leftrightarrow \sigma \in \varphi$$

- Система переходов TS удовлетворяет свойству линейного времени φ :

$$TS \models \varphi \Leftrightarrow \text{Traces}(TS) \subseteq \varphi$$

$\models$

$$TS(P) \models \varphi \equiv P \models \varphi$$

$\models$

$\models$

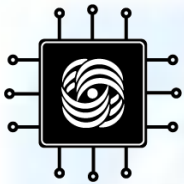
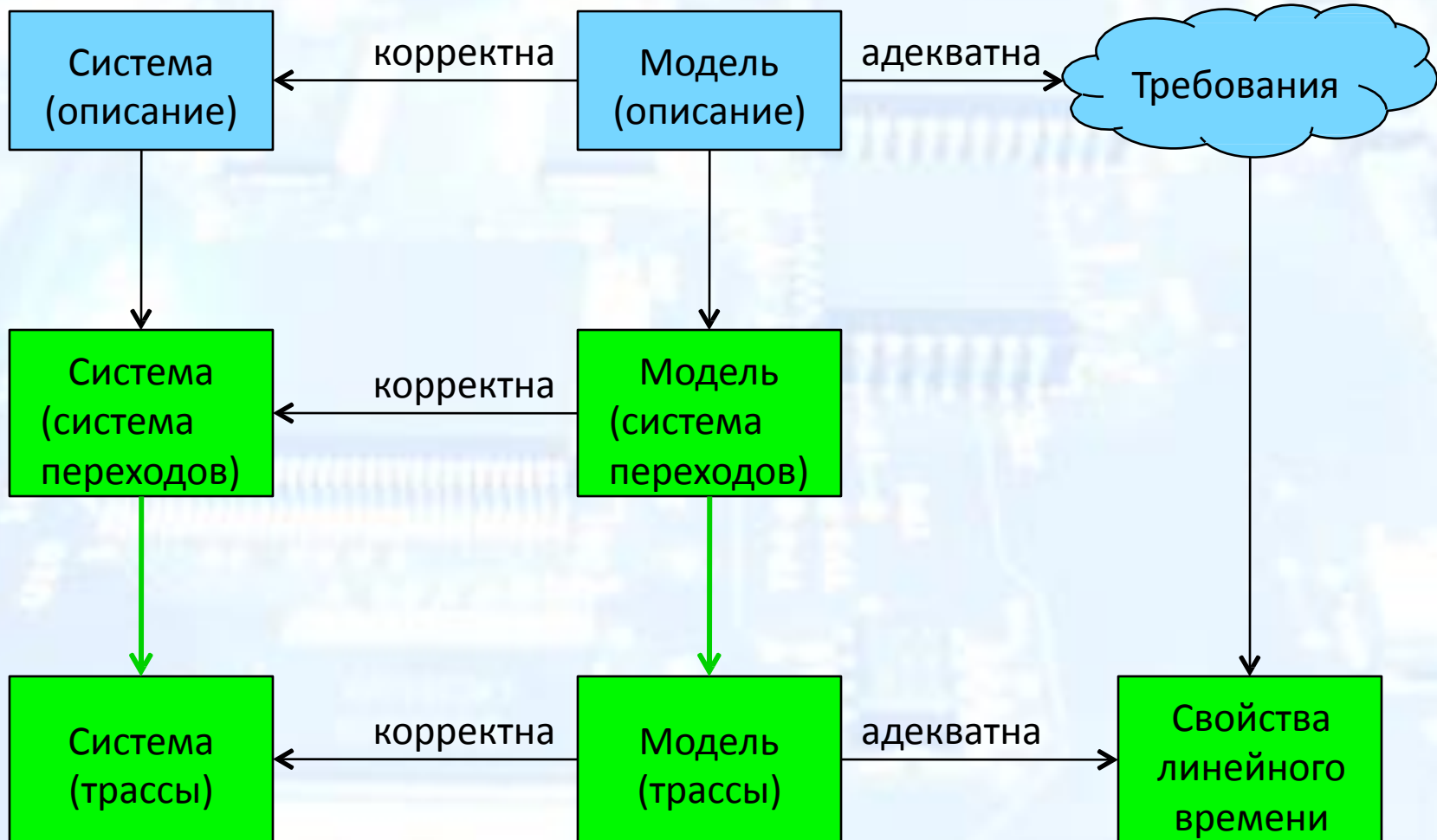


Схема понятий





Абстракция трасс

- Представим трассу в форме интерпретации I :

$$I(tr) = \langle N, \leq, \xi \rangle$$

где N – множество натуральных чисел,

$\leq$ – отношение порядка на N , а

$\xi : N \times AP \rightarrow \{T, \perp\}$, и при этом

$$\forall n > 0, p \in AP$$

$$\xi(n, p) = T \Leftrightarrow p \in L(s)$$



Абстракция трасс

- Рассмотрим трассы tr и tr' такие, что:

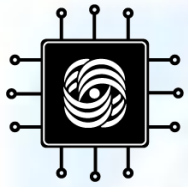
$$I(tr) = \langle N, \leq, \xi \rangle \quad I(tr') = \langle N, \leq, \xi' \rangle$$
$$\xi : N \times AP \rightarrow \{T, \perp\} \quad \xi' : N \times AP' \rightarrow \{T, \perp\}$$

- Будем говорить, что tr' является *(корректной) абстракцией* tr ($tr \prec tr'$), если

$$AP' \subseteq AP$$

$$\exists \alpha : N \rightarrow N : \forall n, k \in N \left(n \leq k \Rightarrow \alpha(n) \leq \alpha(k) \right)$$

$$\forall n \in N, p \in AP' \left(\xi(n, p) = \xi'(\alpha(n), p) \right)$$



Пример абстракции трасс

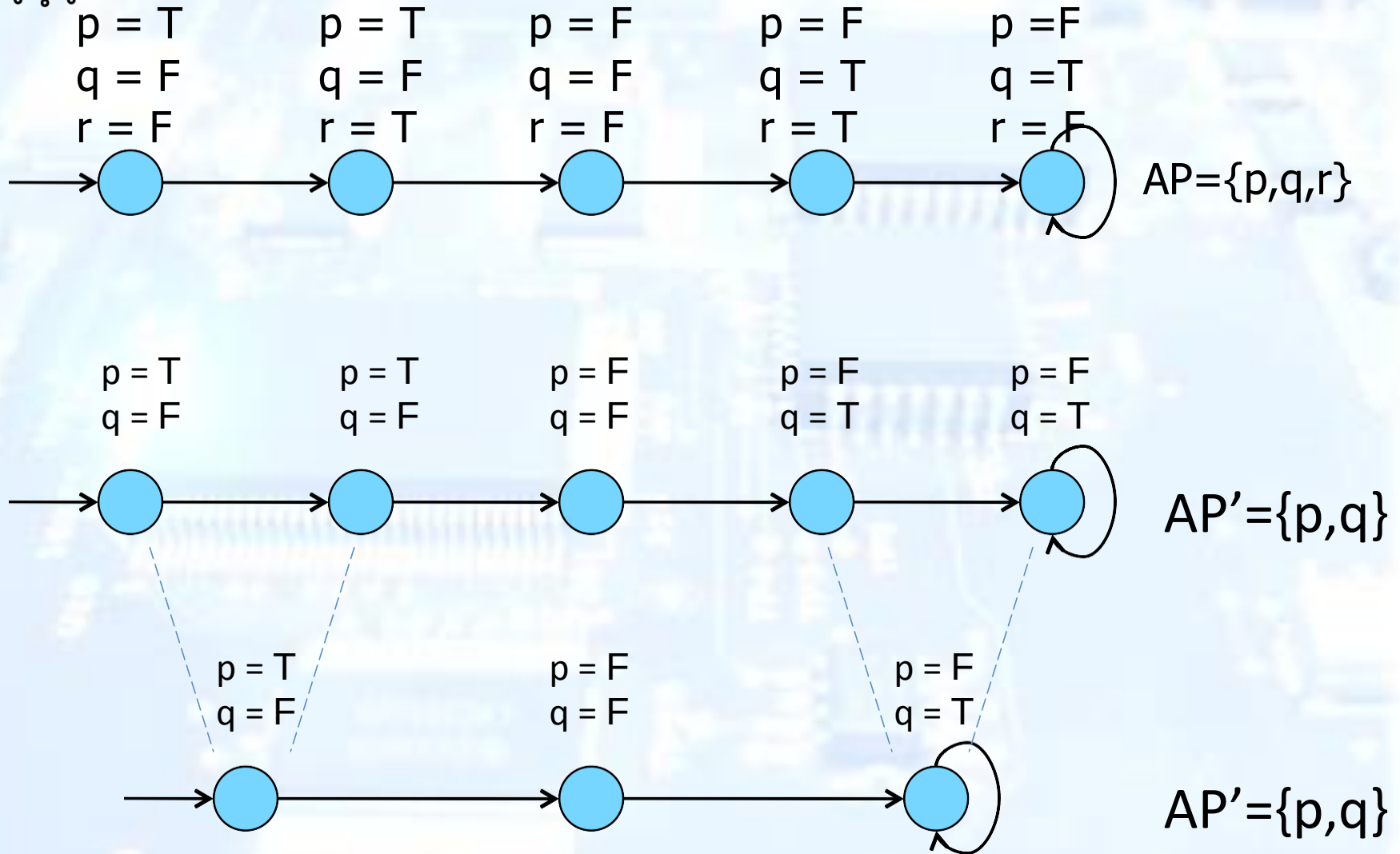
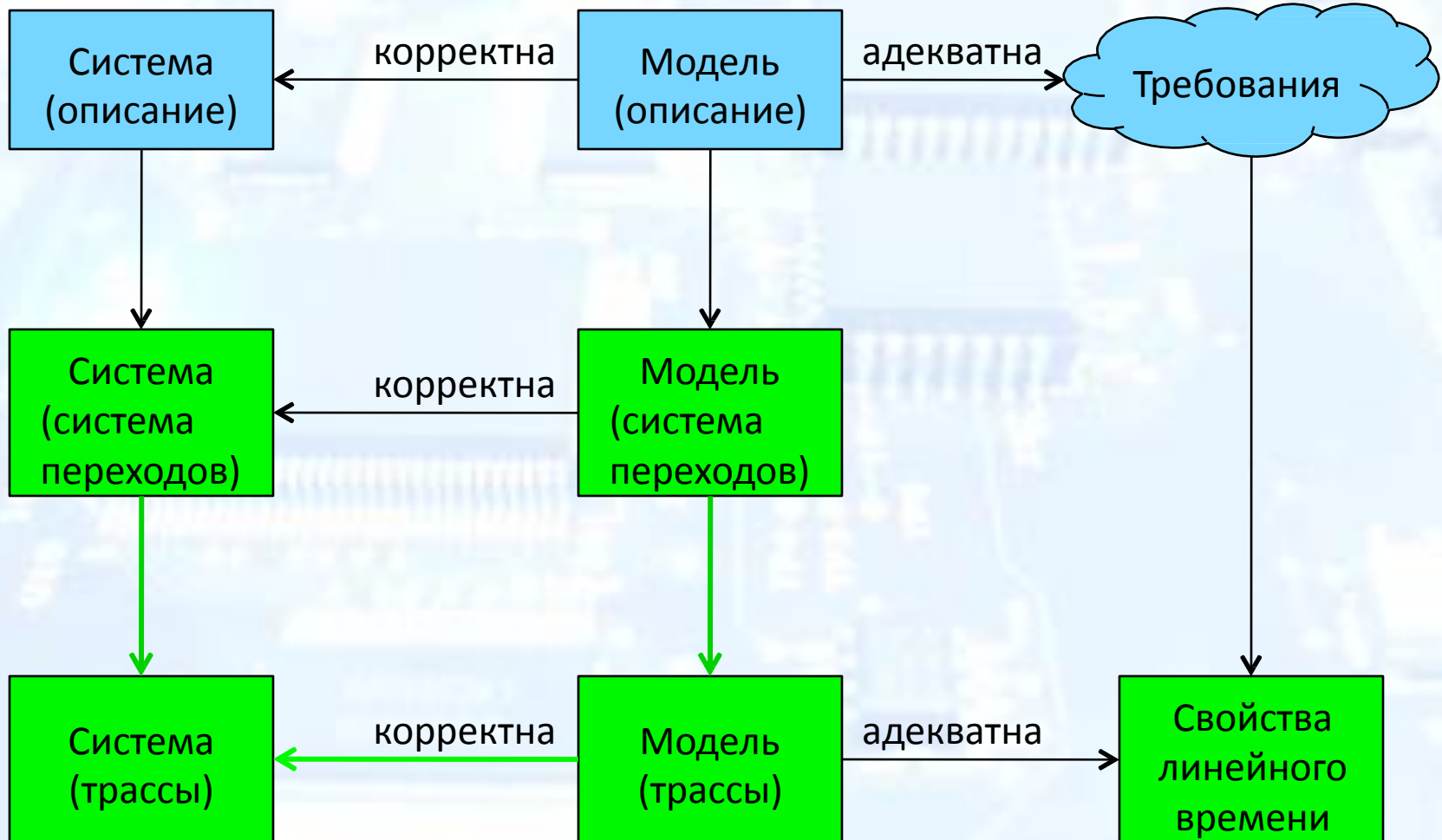
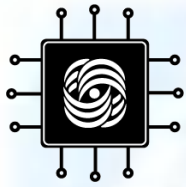




Схема понятий





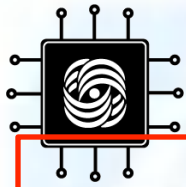
Условие корректности модели

- Пусть P – система, φ – произвольное свойство линейного времени. (Корректной) моделью P называется такое M , что:

если свойство выполняется на модели, то оно выполняется и на системе

- Это выполняется тогда и только тогда, когда:

для любой трассы исходной системы в модели найдётся её корректная абстракция



Условие корректности модели

- Пусть P – система, φ – произвольное свойство линейного времени. (Корректной) моделью P называется такое M , что:

$$M \models \varphi \Rightarrow P \models \varphi$$

⎡ позволяет проверять свойства
программы на её модели ⎤

определение

- Это выполняется тогда и только тогда, когда:

$$\forall tr \in Traces(TS(P)) \exists tr' \in Traces(TS(M)) :$$

$$tr \prec tr'$$

⎡ для проверки такого условия нужно рассмотреть
все трассы исходной системы ⎤

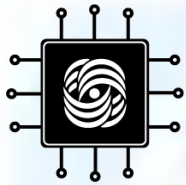
⎡ допускает, что в модели
больше состояний ⎤

необходимое и достаточное условие



Достаточное условие корректности

- Какими же свойствами должна обладать TS модели, чтобы быть корректной?
 - действиям и атомарным высказываниям модели должны быть сопоставлены действия и атомарные высказывания системы,
 - каждому состоянию системы должно быть сопоставлено состояние модели,
 - модель должна корректно сохранять множество начальных состояний,
 - если в системе есть переход между двумя состояниями, в модели должен быть переход по между соотв. состояниями по соотв. действию,
 - соотв. состояния в модели и системе должны быть размечены атомарными высказываниями модели одинаково.



Достаточное условие корректности

- Какими же свойствами должна обладать TS модели, чтобы быть корректной?

$$TS = \langle S, Act, \xrightarrow{a}, I, AP, L \rangle$$
$$TS' = \langle S', Act', \xrightarrow{a}', I', AP', L' \rangle$$

$$Act' \subseteq Act$$

$$AP' \subseteq AP$$

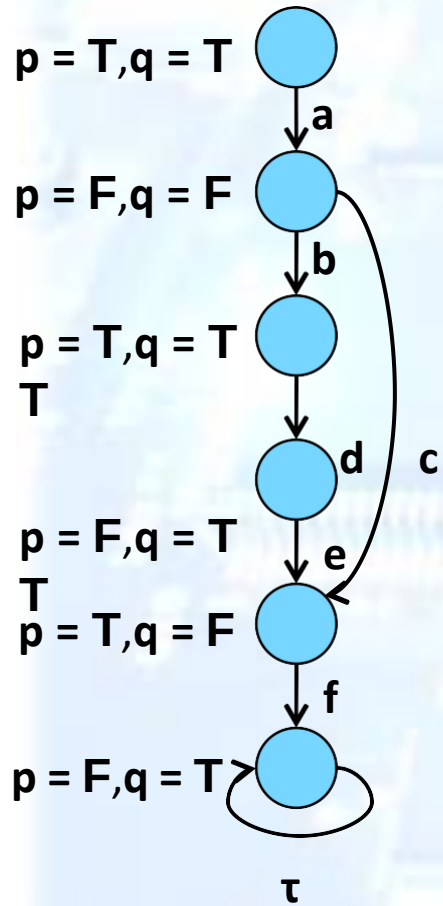
$$\exists \alpha : S \rightarrow S', s_0' = \alpha(s_0)$$

$$s_1 \xrightarrow{a} s_2 \Rightarrow \alpha(s_1) \xrightarrow{a'} \alpha(s_2)$$

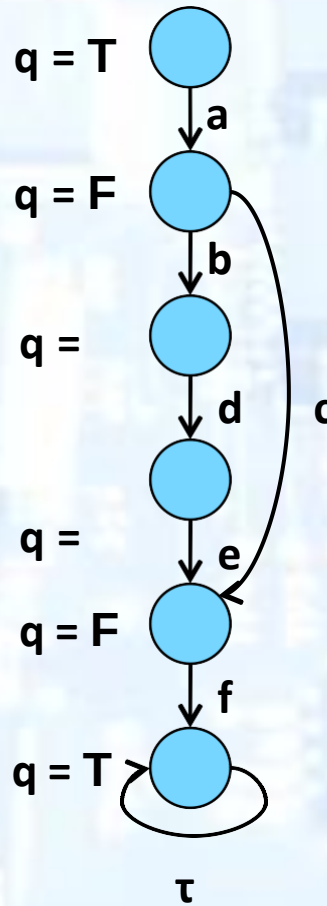
$$\forall s \in S, L'(\alpha(s)) = L(s) \cap AP'$$



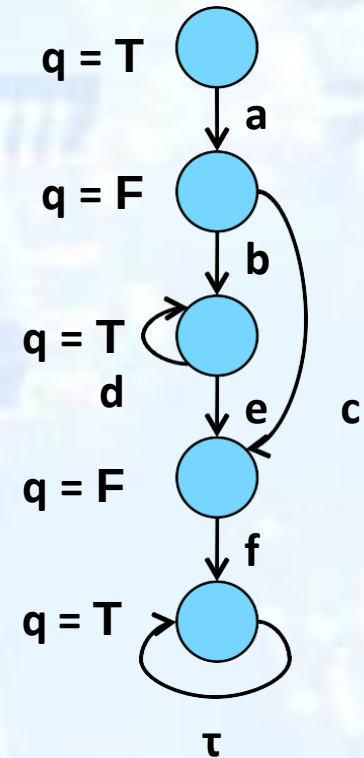
Пример корректной абстракции системы переходов



(P)



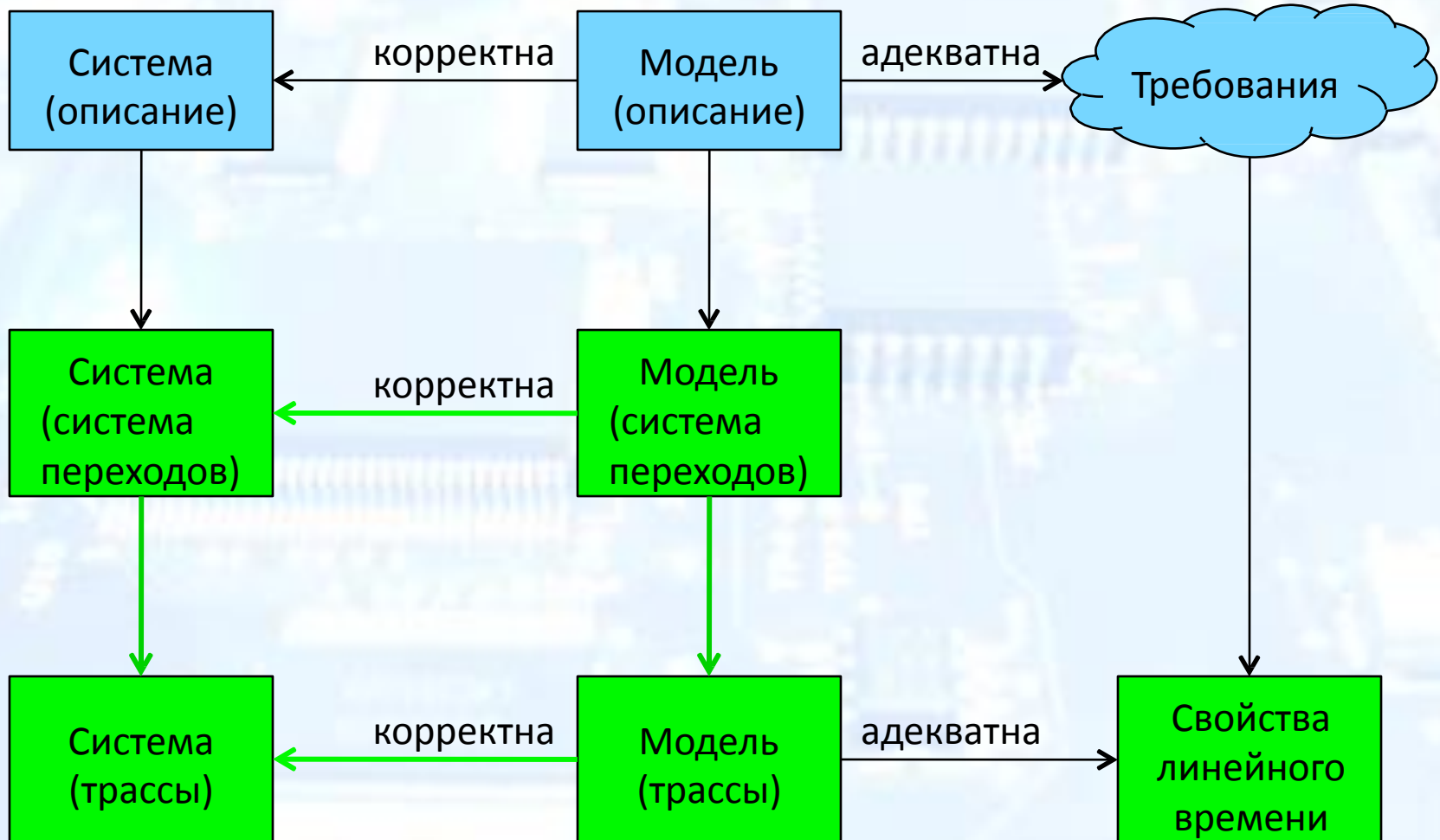
(M_1)



(M_2)



Схема понятий





Адекватность модели

- Модель называется адекватной, если:
 1. Атомарные высказывания, в терминах которых задаются свойства, присутствуют в разметке модели
 2. Из нарушения свойства на модели следует, что оно нарушается и на исходной системе



Адекватность модели

- Модель называется адекватной, если:

$$1. AP_{\varphi} \subseteq AP_M$$

необходимое условие (можно вычислить)

2.

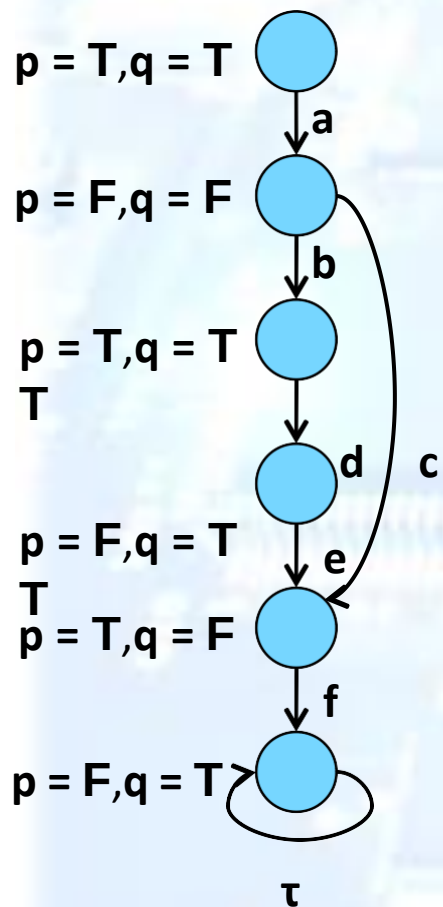
$$M \not\models \varphi \Rightarrow P \not\models \varphi$$

достаточное условие (нельзя вычислить)

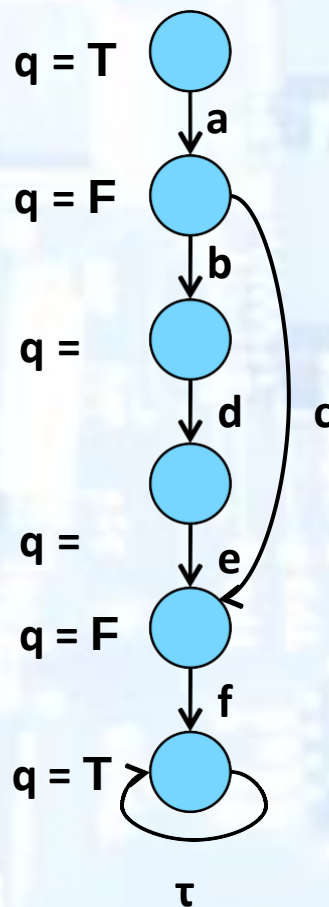
- Определить адекватность при построении нельзя, можно лишь обнаружить несоответствие и исправить модель



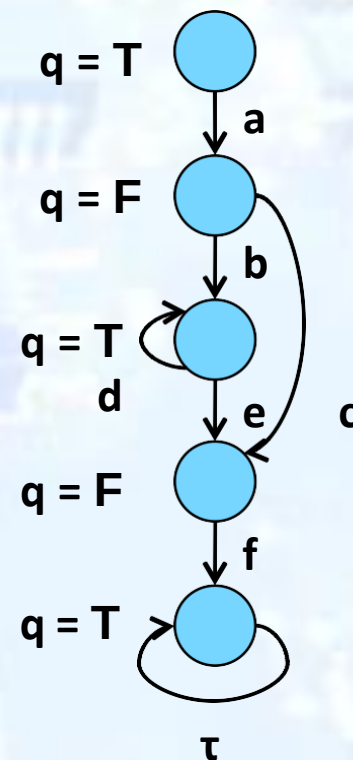
Пример корректной абстракции системы переходов



(P)



(M_1)



(M_2)

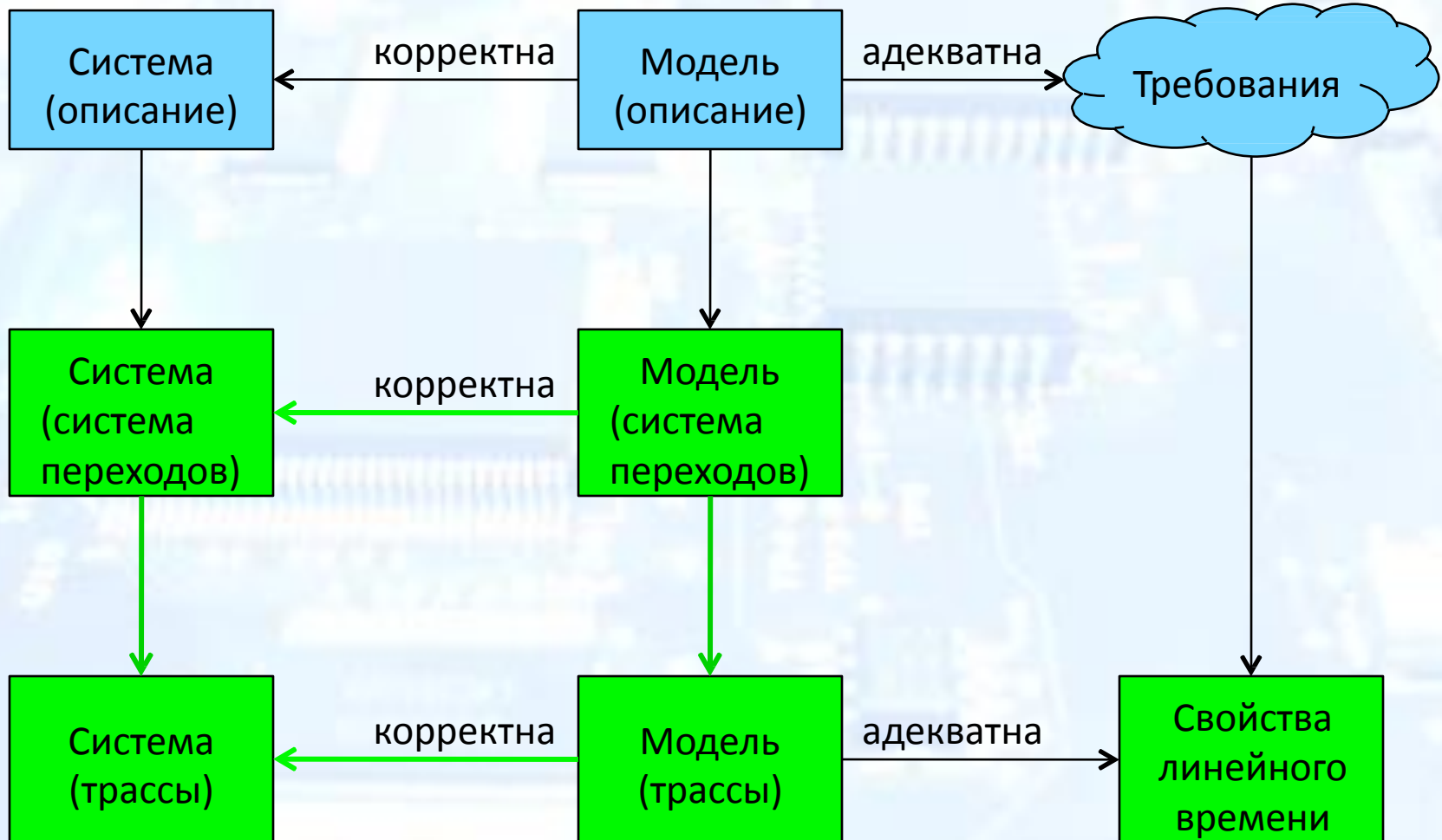


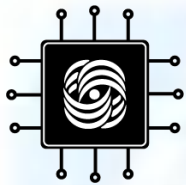
Пример – проверяемые свойства

- В любом вычислении встречается состояние, когда $p=T$ & $q=T$ – *ни одна из моделей не адекватна*,
- Для любого пути верно, что за любым $q=F$ рано или поздно встретится $q=T$ – *все модели адекватны*,
- Между двумя состояниями с $q=F$ встречается не более чем 3 состояния с $q=T$ – *модель M_1 адекватна, M_2 – нет.*



Схема понятий





Спасибо за внимание!



НАДЁЖНОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Лекция 4: *Временные автоматы*

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



План лекции

- Временной автомат:
 - Общее описание
 - Пример
 - Синтаксис
 - Семантика
- Сеть временных автоматов:
 - Описание
 - Пример
- Временные свойства (логика TCTL)



Временной автомат

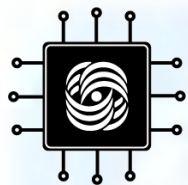
- Описание, пригодное для моделирования и анализа (асинхронных) систем реального времени
- Содержит конечное множество состояний управления
- Дополнительно содержит таймеры, принимающие действительные значения
- В структуре автомата отражено влияние таймеров на поведение системы



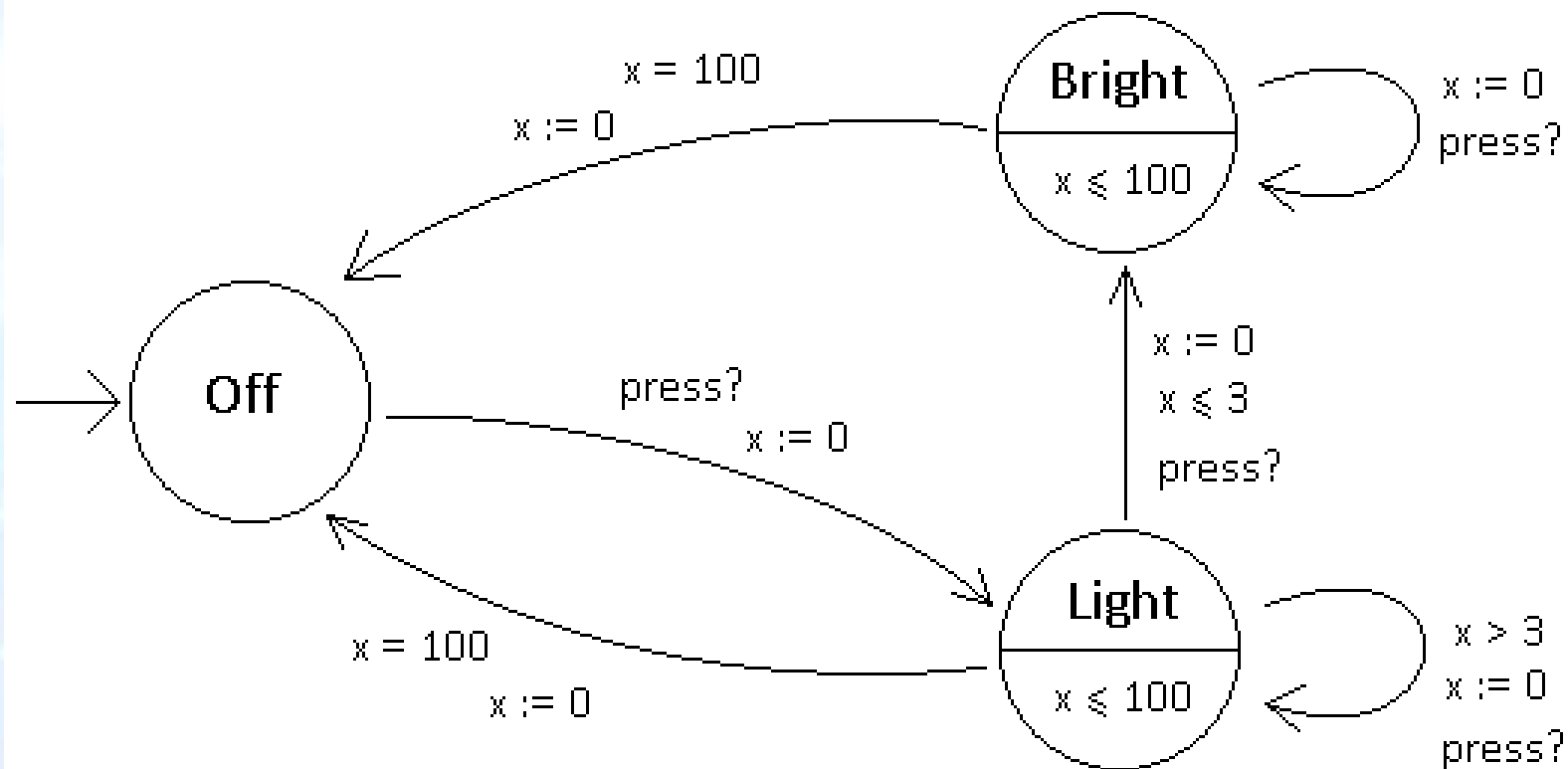
Пример временного автомата

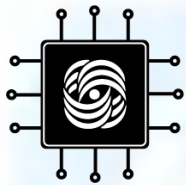
Модель переключателя освещения

- Содержит 3 состояния управления:
 - Off (выключено)
 - Light (обычное освещение)
 - Bright (яркое освещение)
- Свет выключается при 100 секундах неактивности
- Свет включается при нажатии на кнопку (press)
- Яркий свет включается при двойном нажатии (двух нажатиях, отстоящих не более чем на 3 секунды)

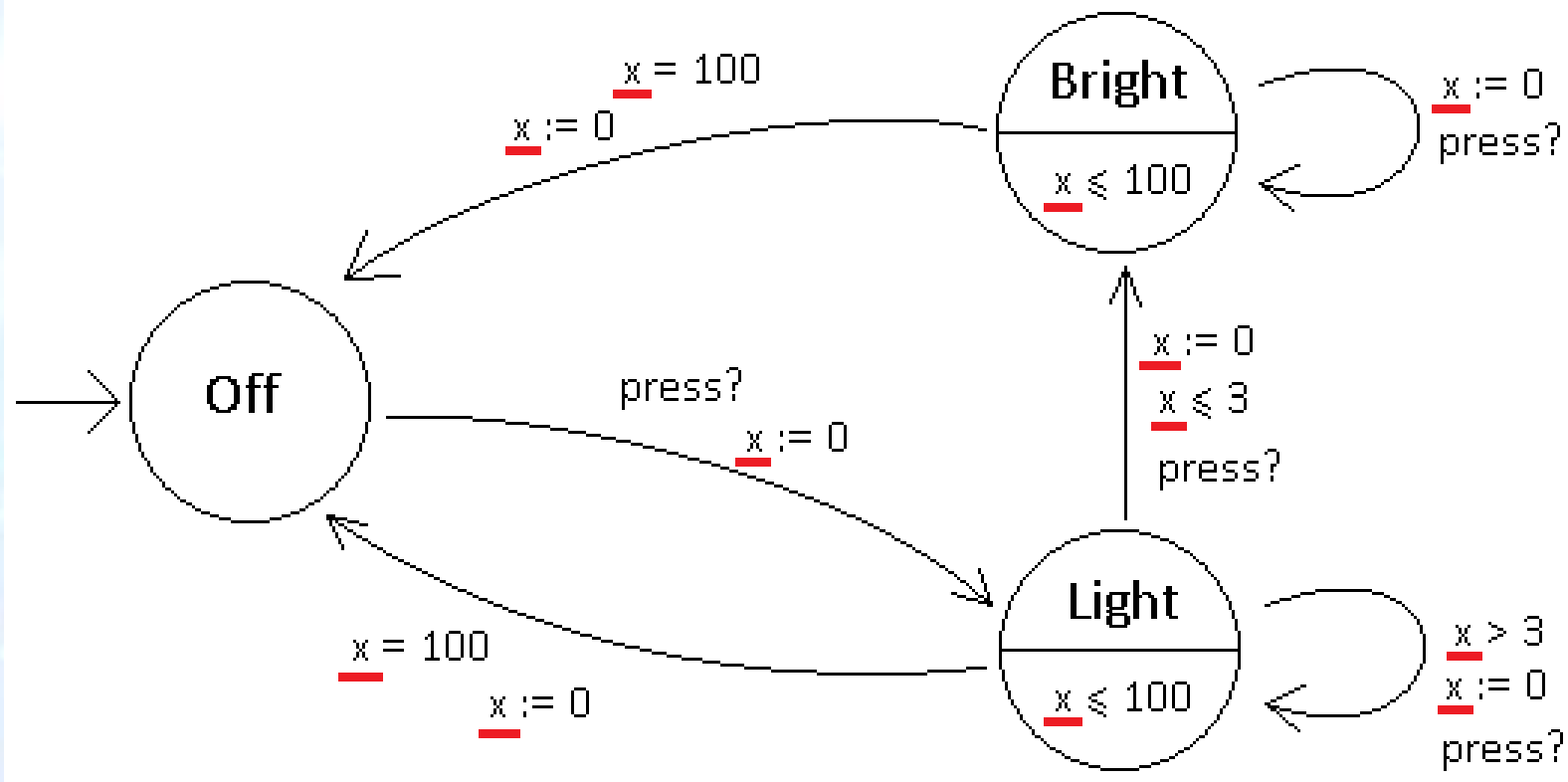


Пример временного автомата

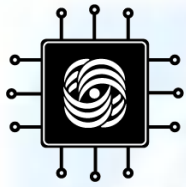




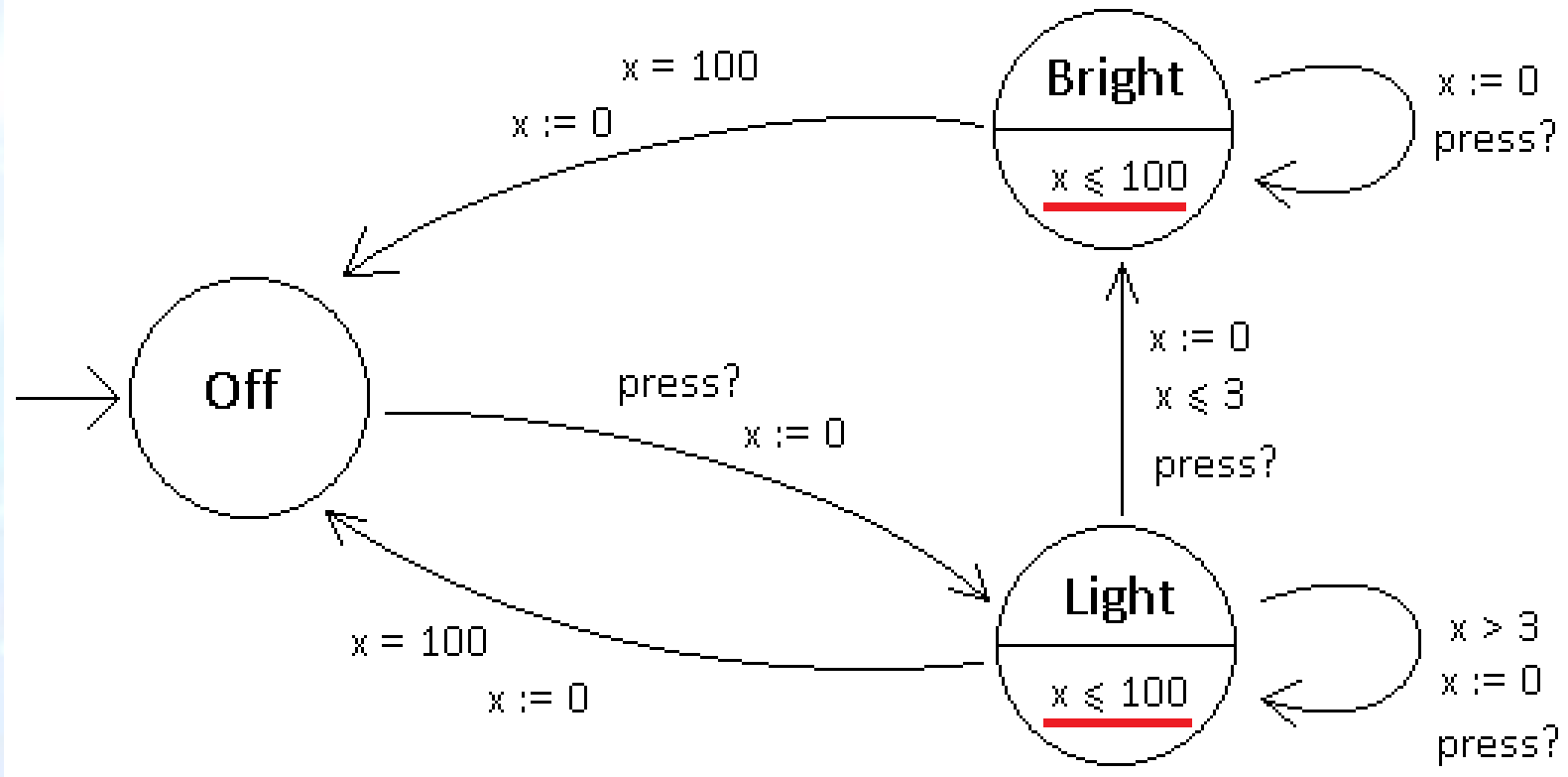
Таймеры



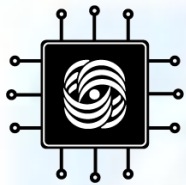
- x – таймер, принимающий действительные неотрицательные значения



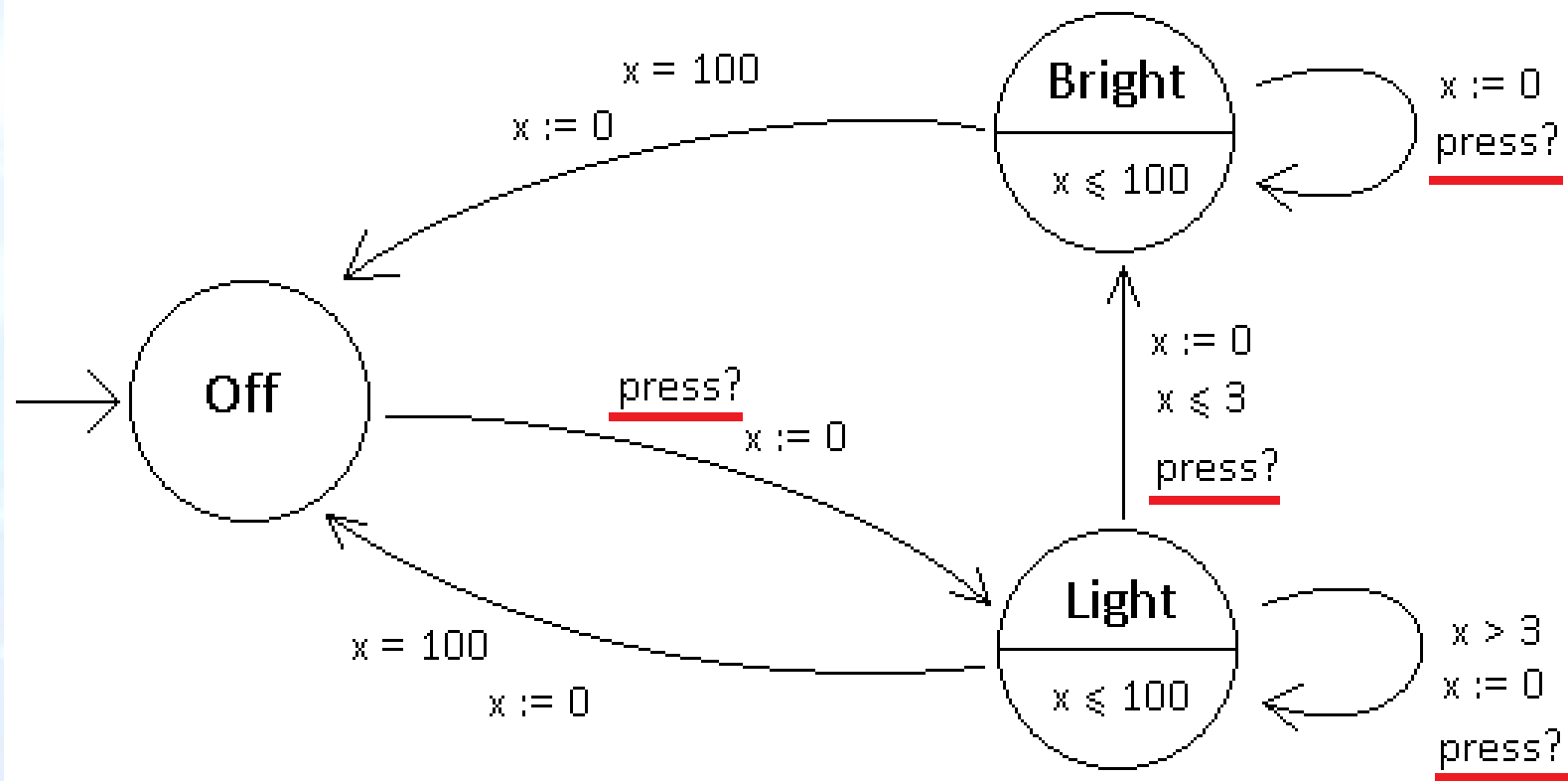
Инварианты



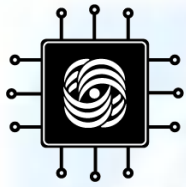
- Инвариант – это необходимое условие активности состояния



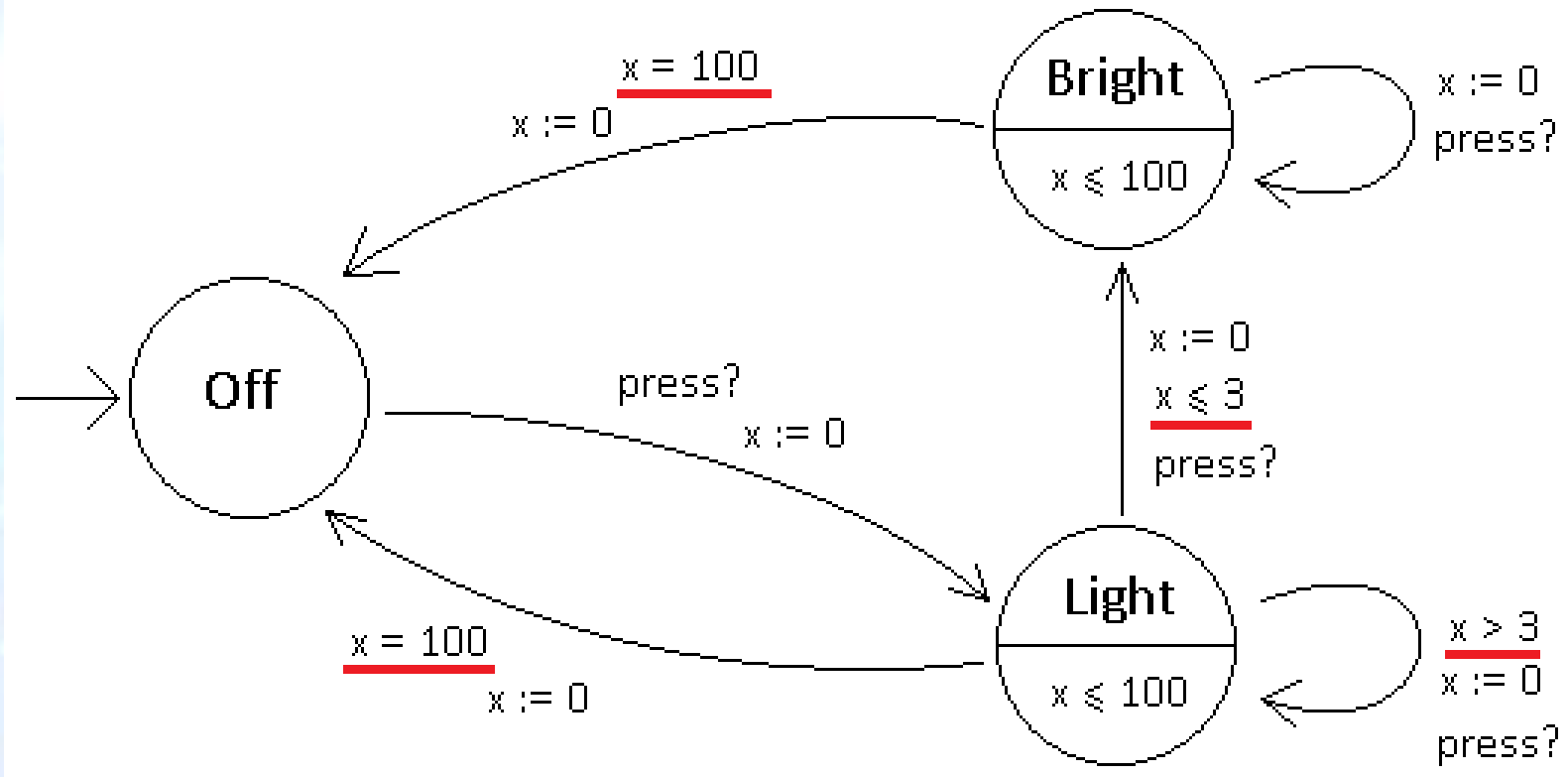
Каналы связи



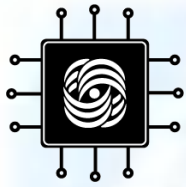
- **press?** – внешнее событие нажатия кнопки включения



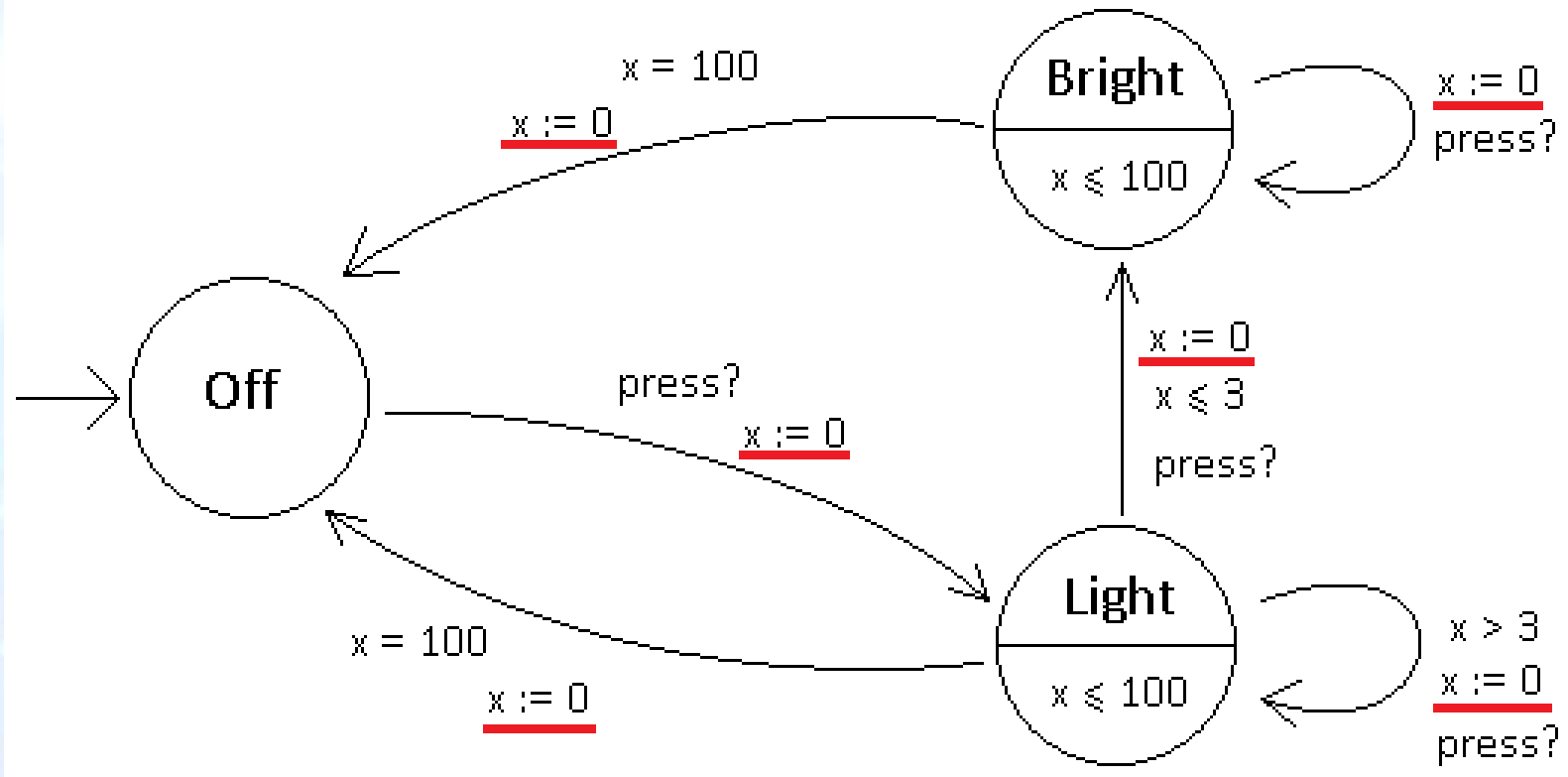
Предусловия



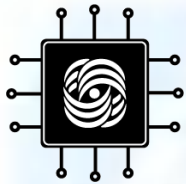
- Предусловие – это необходимое условие выполнения перехода



Сброс таймеров



- При выполнении перехода таймер может быть сброшен



Временные ограничения

- Элементарное временное ограничение:

$\text{ETC} ::= x \text{ op } n \mid x - y \text{ op } n$

$\text{op} ::= < \mid \leq \mid = \mid \geq \mid >$

$(x, y \in \mathbb{C}, n \in \mathbb{Z})$

- Временное ограничение:

$\text{TC} ::= \text{ETC} \mid \text{TC} \wedge \text{TC} \mid \text{TC} \vee \text{TC} \mid \neg \text{TC} \mid (\text{TC})$

- Элементарное инвариантное ограничение:

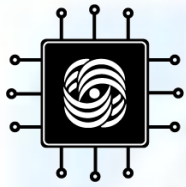
$\text{ETC}\Downarrow ::= x \text{ op}\Downarrow n$

$\text{op}\Downarrow ::= < \mid \leq$

$(x \in \mathbb{C}, n \in \mathbb{Z})$

- Инвариантное ограничение:

$\text{TC}\Downarrow ::= \text{ETC}\Downarrow \mid \text{TC}\Downarrow \wedge \text{TC}\Downarrow$

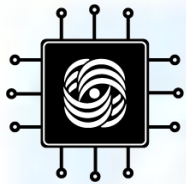


Синтаксис временного автомата

Временной автомат – это система

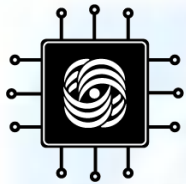
(C, A, L, I_0, P, E) , где

- C – множество таймеров
- A – множество действий
- L – множество состояний управления
- I_0 – начальное состояние
- $P : L \rightarrow TC \Downarrow$ – назначение инвариантов
- $E \subseteq L \times TC \times A \times 2C \times L$ – множество переходов



Конфигурация автомата

- Конфигурация автомата: (l, v)
 - l – состояние управления
 - $v : C \rightarrow R^+$ - оценка таймеров
- Начальная конфигурация: (l_0, v_0)
 - l_0 – начальное состояние автомата
 - $v_0(x) = 0$ для всех таймеров x
- Обозначения:
 - $v+d, d \in R^+$: $v+d(x) = v(x) + d$
 - $v[C'], C' \subseteq C$: $v[C'](x) = 0$ при $x \in C'$; $v[C'](x) = v(x)$ иначе
 - $v \models g$: формула g истинна при значениях таймеров, определяемых оценкой v

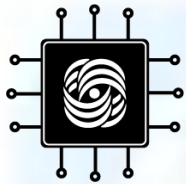


Вычисление автомата

Шаг вычисления автомата состоит в

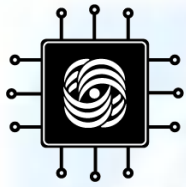
- Продвижении времени:
 - состояние управления не изменяется,
 - все таймеры продвигаются на константу d ;
- Выполнении перехода:
 - выбирается переход из I в I' ,
 - состояние управления меняется на I' ,
 - таймеры, которыми помечен переход, сбрасываются;

После применения шага инварианты всех состояний должны быть выполнены



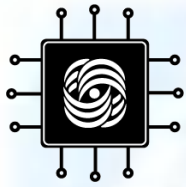
Вычисление автомата

- Продвижение времени: $(l, v) \rightarrow_d (l, v+d)$, если
 - $\forall l \in L (v+d \models P(l))$
- Выполнение перехода: $(l, v) \rightarrow_a (l', v')$, если
 - $(l, g, a, C', l') \in E$
 - $v \models g$
 - $v' = v[C']$
 - $v' \models P(l')$
- $\rightarrow = \rightarrow_d \cup \rightarrow_a$
- Вычисление:
$$c1 \rightarrow c2 \rightarrow \dots \rightarrow c_n \rightarrow \dots$$
- Инициальное вычисление:
 - $c1$ – начальная конфигурация автомата



Синхронизация автоматов

- Выделяются действия синхронизации:
 - Активные: $a!$
 - Пассивные: $a?$
- Они соответствуют посылке и приему сигнала по каналу a
- При выполнении перехода с синхронизацией выбирается второй переход с парной синхронизацией, и эти два перехода в различных автоматах выполняются одновременно



Сеть временных автоматов

- C, A – множество таймеров и множество действий сети
- Сеть – это совокупность автоматов
$$N = (A_1, A_2, \dots, A_n), \text{ где}$$
- $A_i = (C, A, L_i, I_{O_i}, P_i, E_i)$ – временные автоматы
- Сеть – это параллельная композиция автоматов с синхронизацией типа handshake



Семантика сети

- Конфигурация: (I, v)
 - $I = (I_1, I_2, \dots, I_n)$ – вектор управляющих состояний автоматов сети
 - $v : C \rightarrow R^+$ – оценка таймеров
- Начальная конфигурация: (I_0, v_0)
 - I_0 – вектор начальных состояний автоматов
 - $v_0(x) = 0$ для всех таймеров x
- Обозначение:
 - $I[i \leftarrow I']$ – вектор, полученный из I заменой i -й компоненты на I'



Семантика сети

- Правила продвижения времени и выполнения перехода обобщаются естественным образом (отношение $\Rightarrow$)
- Правило выполнения переходов с синхронизацией:

$$(I, v) \Rightarrow (I', v'), \text{ где}$$

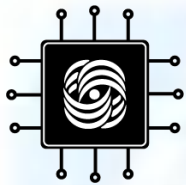
- $I' = I[i \leftarrow I', j \leftarrow I'']$
- $i \neq j$
- $(li, !a, g', C', I') \in Li, (lj, ?a, g'', C'', I'') \in Lj$
- $v' = v[C' \cup C'']$
- $v \models g' \wedge g''$
- $\forall I \in I' (v' \models P(I))$
- Вычисление сети:
$$c1 \Rightarrow c2 \Rightarrow \dots \Rightarrow cn \Rightarrow \dots$$
- Инициальное вычисление сети:
 - $c1$ – начальная конфигурация сети



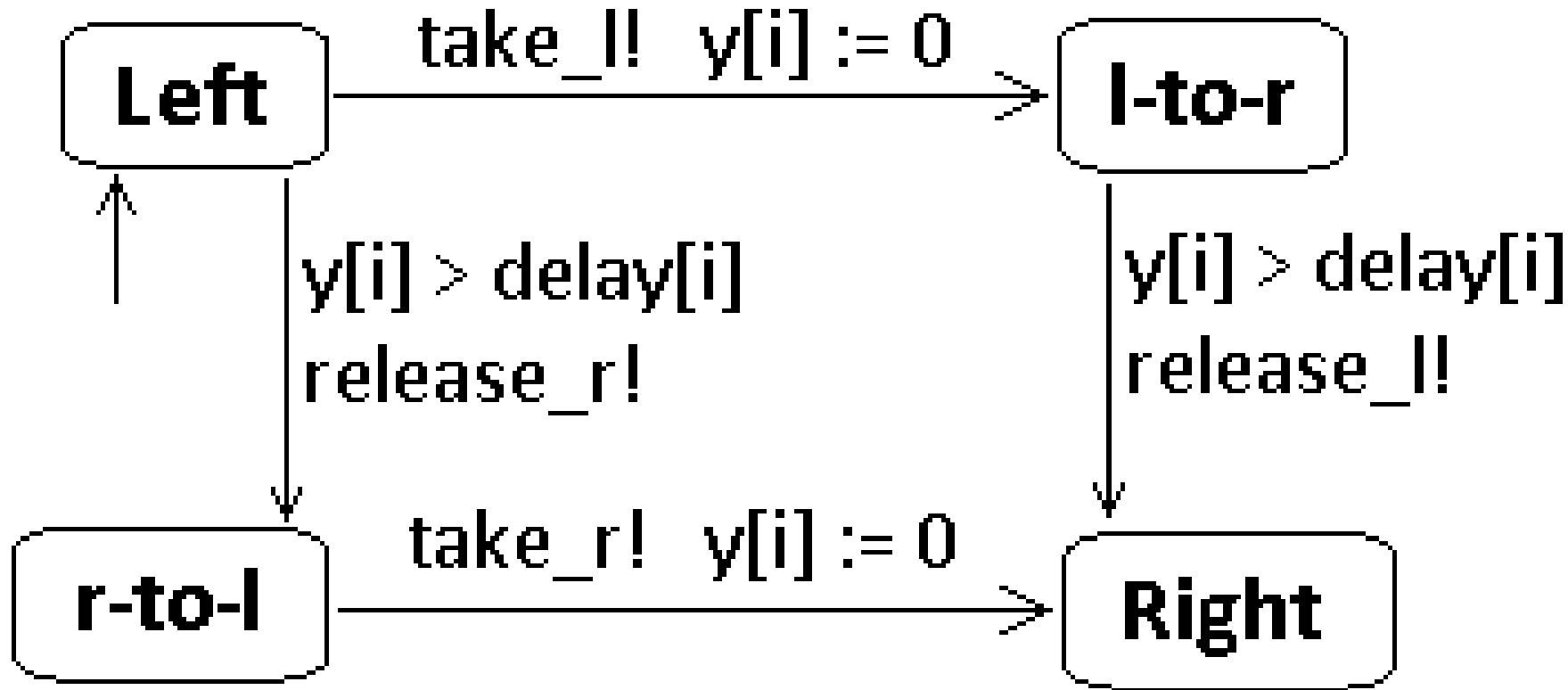
Пример сети

Задача о викингах:

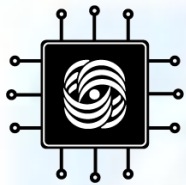
- N викингов хотят перейти через мост
- У каждого викинга своя максимальная скорость
- Имеется один факел
- Идти через мост можно только при факеле одному или вдвоем
- Требуется перевести викингов через мост как можно быстрее



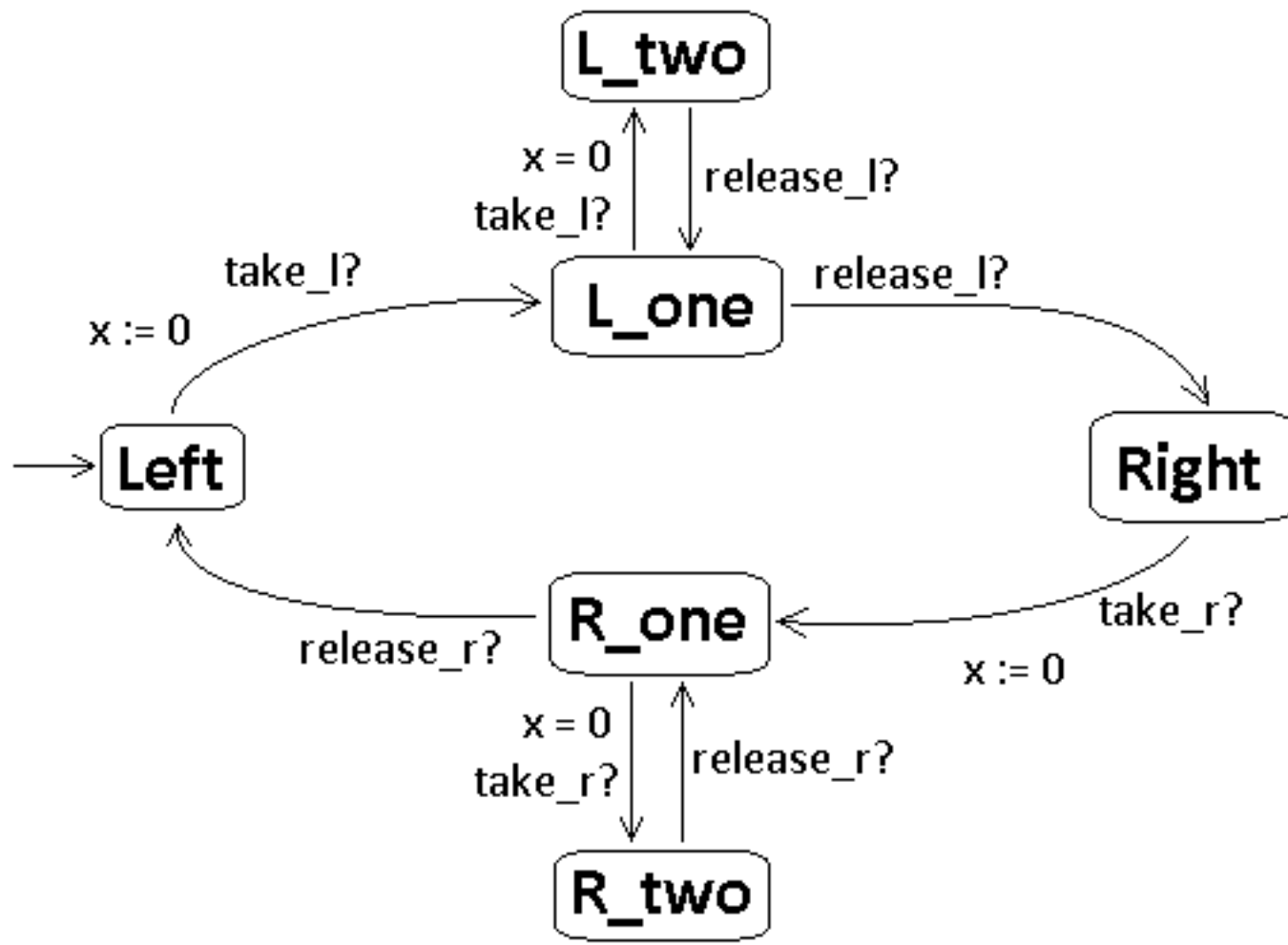
Пример сети



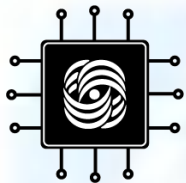
- Автомат для i -го викинга
- $y[i]$ – уникальный для i -го викинга таймер
- $\text{delay}[i]$ заменяется на конкретную константу – минимальное время прохода по мосту



Пример сети



- Временной автомат для факела



Логика TCTL

- Временное ограничение:

$CC ::= x \text{ op } n \mid x - y \text{ op } n$

$\text{op} ::= < \mid \leq \mid = \mid \geq \mid >$

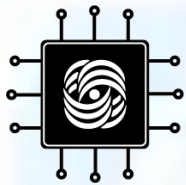
$(x, y \in C; n \in \mathbb{Z})$

- Формула:

$\Phi ::= P \mid CC \mid z.\Phi \mid \Phi \vee \Phi \mid \Phi \wedge \Phi \mid \neg \Phi$

$\mid AG \Phi \mid AF \Phi \mid EG \Phi \mid EF \Phi$

$\mid A(\Phi \cup \Phi) \mid E(\Phi \cup \Phi)$



Логика TCTL (упрощенный вариант)

- Элементарный предикат:

$EP ::= Ai.l \mid x \text{ op } n \mid x - y \text{ op } n$

$op ::= < \mid \leq \mid = \mid \geq \mid >$

$(l \in Li; x, y \in C; n \in Z)$

- Формула:

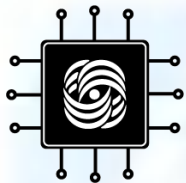
$\Phi ::= EP \mid \Phi \vee \Phi \mid \Phi \wedge \Phi \mid \neg \Phi$

$\mid AG \Phi \mid AF \Phi \mid EG \Phi \mid EF \Phi$

$\mid A(\Phi \cup \Phi) \mid E(\Phi \cup \Phi)$

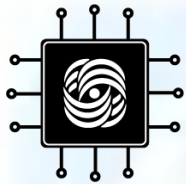
- Обозначение:

- $N, s \models \Phi$: формула Φ выполняется в конфигурации с сети N



Логика TCTL (упрощенный вариант)

- $N, ((l_1, l_2, \dots, l_n), v) \models A_i.l \Leftrightarrow l_i = l$
- $N, (l, v) \models x \text{ op } n \Leftrightarrow v \models x \text{ op } n$
- $N, (l, v) \models x - y \text{ op } n \Leftrightarrow v \models x - y \text{ op } n$
- $N, s \models AG \Phi \Leftrightarrow$ в каждой конфигурации каждого вычисления, начинающегося в s , выполнена формула Φ
- $N, s \models AF \Phi \Leftrightarrow$ хотя бы в одной конфигурации каждого вычисления, начинающегося в s , выполнена формула Φ
- $N, s \models EG \Phi \Leftrightarrow N, s \models \neg AF \neg \Phi$
- $N, s \models EF \Phi \Leftrightarrow N, s \models \neg AG \neg \Phi$
- Соотношения для булевых связок определяются естественным образом



Примеры временных свойств

- Можно перевести всех викингов через мост за время T (здесь `time` – таймер сети)

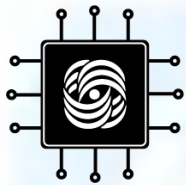
$$EF(Viking[1].Right \wedge \dots \wedge Viking[n].Right \wedge time \leq T)$$

- Нельзя перевести викинга за время, меньшее времени его перехода через мост

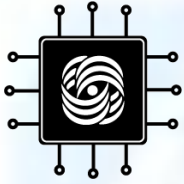
$$AG \neg (Viking[i].Right \text{ and } time < delay[i])$$

- Если факел был захвачен, рано или поздно он будет освобожден (не выполняется)

$$AG((Torch.Left \vee Torch.Right) \vee AF(Torch.Left \vee Torch.Right))$$



Спасибо за внимание



НАДЁЖНОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

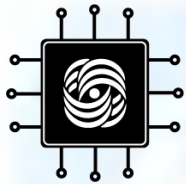
Лекция 5: *Логика линейного времени(LTL)*

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



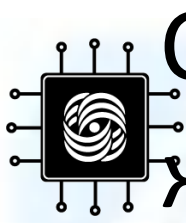
План лекции

- Свойства безопасности и живучести
- Логика линейного времени (LTL)
- Свойства, инвариантные к прореживанию
- Практические приёмы формулирования свойств на LTL



Рассуждения о правильности программы

- Требования правильности задаются как утверждения о возможных или невозможных вариантах выполнения модели,
 - рассуждения о вероятности не допускаются из соображений строгости доказательства;
- Мы утверждаем, что некоторые варианты выполнения модели
 - либо **невозможны**,
 - либо **неизбежны**;
- Двойственность утверждений:
 - если что-то неизбежно, то противоположное – невозможно,
 - если что-то невозможно, то противоположное – неизбежно,
 - используя логику, можно переходить от одного к другому при помощи логического отрицания.



Свойства безопасности и живучести

(автор – Leslie Lamport)

безопасность

• «ничего плохого никогда не произойдёт»;

- пример: инвариант системы
 - x всегда меньше y ;
- задача верификатора – найти вычисления, которые ведут к нарушению свойства безопасности (то «плохое», которое никогда не должно произойти)



живучесть

• «рано или поздно произойдёт что-то хорошее»;

- пример: «отзывчивость»
 - если отправлен запрос, то рано или поздно будет сгенерирован ответ;
- задача верификатора – найти вычисления, в которых это «хорошее» может откладываться до бесконечности





Немного подробнее...

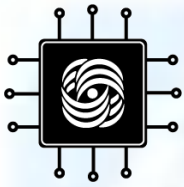
- Свойства безопасности – самые простые свойства («состояние, где истинно φ , недостижимо»).
[Достаточно исследовать всё (конечное)]
- Можно ли в рамках свойства безопасности сформулировать, что φ неизбежно?
- **«состояние, где истинно $!\varphi$, недостижимо»**
– не то, это слишком сильное свойство;
- **«состояние, где истинно φ , достижимо»**
– а это, наоборот, слишком слабое.



Немного подробнее...

- **« φ неизбежно»** – означает, что φ **обязательно достижимо**;
- для спецификации такой модальности и нужны свойства живучести.

Необходимо показать, что свойство достижимо в любом начальном вычислении системы. В частности, нужно исследовать циклы в системе переходов.



Классическая логика не может описать утверждения

- Классическая логика
 - Прimitивная модель истины: “черно-белая” модель, не существует степени уверенности-неуверенности, высказывания статичны, неизменны во времени $\Rightarrow$ неадекватна для высказываний о времени
 - Пример - (некоммутативность конъюнкции, $A \& B \neq B \& A$):
 - “Джону стало страшно и он убил” $\Leftarrow? \Rightarrow$ “Джон убил и ему стало страшно”
 - “Джон умер и его похоронили” $\Leftarrow? \Rightarrow$ “Джона похоронили и он умер”
 - “Джейн вышла замуж и родила ребенка” $\Leftarrow? \Rightarrow$ “Джейн родила ребенка и вышла замуж”
- В обычной логике высказываний не формализуются:
 - *Путин – наш президент* (истинно только в какой-то период)
 - *Мы не друзья, пока ты не извинишься*
 - *Если **m** поступит на вход в канал, то потом **m** появится на выходе*
 - *Каждый запрос к лифту с произвольного этажа, поступивший в любой момент времени, будет когда-нибудь в будущем удовлетворен*

Элементарные (атомарные) утверждения о событиях обычно истинны в один момент времени и ложны в другой!

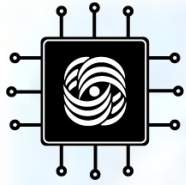
Утверждения о событиях, происходящих в разные моменты времени, нельзя корректно выразить в обычной логике



Темпоральная логика

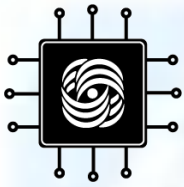
- **Определение** TL - это любая логическая система, которая позволяет, не вводя явно понятие времени, формализовать утверждения, истинность которых изменяется со временем. Используется для описания последовательностей явлений и их взаимосвязи и зависимости во времени
- **Применения TL**
 - **ФИЛОСОФИЯ:** формализм для прояснения философских вопросов о времени;
 - **ЕСТЕСТВЕННЫЙ ЯЗЫК:** формализм для определения семантики утверждений в естественных языках, включающих время;
 - **ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ:** язык для представления знаний, связанных со временем (Д. А. Поспелов (ред) "*Представление знаний о времени и пространстве в интеллектуальных системах*", 1987);
 - **ТЕХНИКА:** для формализации утверждений о свойствах **будущего поведения** технических систем (программ, оборудования, систем управления, ...), для описания свойств их поведения во времени после того, как они будут запущены

Мы будем рассматривать TL с точки зрения верификации ПО и технических систем



Примеры темпоральных свойств

- **р всегда** истинно;
- **р рано или поздно** станет всегда ложным;
- **р всегда рано или поздно** станет ложным хотя бы ещё один раз;
- **р всегда** ведёт к $\neg q$;
- **р всегда** ведёт к тому, что **рано или поздно** станет истинным **q**.



Темпоральные свойства на естественном языке?

- нет строгой семантики => возможно множество трактовок
 - в части области проверки: «индикатор не горит» - в начальном состоянии? или это инвариант?
 - в темпоральной части:
 - попробуйте объяснить разницу: «от события А до события Б» и «между событиями А и Б»
 - «деньги выплачиваются, как только работа будет выполнена» -- требуется ли выполнение работы?
- значение зависит от контекста:
 - если нажата кнопка, рано или поздно будет выпущено шасси
 - от взлёта и до посадки, если нажата кнопка, то рано или поздно будет выпущено шасси (область проверки вложенного свойства изменилась)
 - после взлёта, если была нажата кнопка, то до посадки будет выпущено шасси (более строгая формулировка)
- зависит от знания и понимания естественного языка, который сложнее LTL.



Темпоральная логика LTL

- Ясный, лаконичный и непротиворечивый способ описания требований к программам;
- В явном виде время не присутствует, однако рассуждения ведутся в терминах «никогда», «всегда», «рано или поздно», которые представлены в виде темпоральных операторов.
- Мы рассматриваем темпоральную логику линейного времени – LTL. С её помощью можно описывать свойства, которым должны удовлетворять линейные последовательности наблюдаемых состояний – трассы.
- LTL предложена Амиром Пнуэли (Amir Pnueli) в конце 70-х.



Формулы LTL

- Могут использоваться для описания как свойств живучести, так и безопасности
- LTL = пропозициональная логика
(свойства состояний)
- + темпоральные операторы
(последовательности состояний)
- Формула LTL $f ::=$
 - $p, q, \dots$ - свойства состояний, включая true и false,
 - (f) – группировка при помощи скобок,
 - αf – унарные операторы,
 - $f_1 \beta f_2$ – бинарные операторы.



Операторы LTL

- Унарные:

- $\Box([])$ всегда (в будущем),
- $\Diamond(<>)$ рано или поздно,
- (X) в следующем состоянии,
- $\neg (!)$ логическое отрицание;

*когда-нибудь,
в конце концов*

- Бинарные:

- $U (U)$ пока, *до тех пор*

$$\equiv (\neg p \vee q)$$

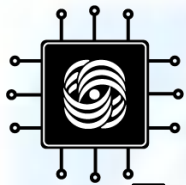
- $\wedge (\&\&)$ логическое И,

- $\vee (||)$ логическое ИЛИ,

- $\rightarrow (->)$ логическая импликация,

$$\equiv (p \rightarrow q) \wedge (q \rightarrow p)$$

- $\leftrightarrow (<->)$ логическая эквивалентность.



Выполнимость формул

- Последовательность состояний прохода σ

$S_0, S_1, S_2, S_3, \dots$

- Набор пропозициональных формул p, q :

$\forall i, i \geq 0$, и $\forall p$, определено $s_i \models p$

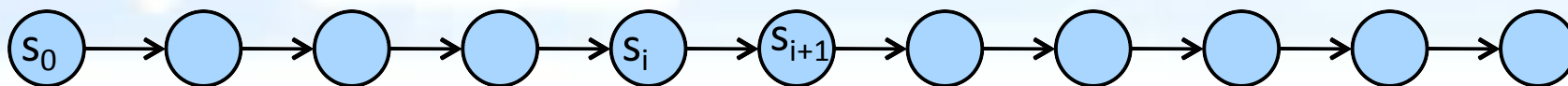
- Семантика LTL:

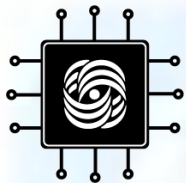
$$\sigma \models f \Leftrightarrow s_0 \models f$$

$$s_i \models []f \Leftrightarrow \forall j, j \geq i : s_j \models f$$

$$s_i \models \Diamond f \Leftrightarrow \exists j, j \geq i : s_j \models f$$

$$s_i \models Xf \Leftrightarrow s_{i+1} \models f$$





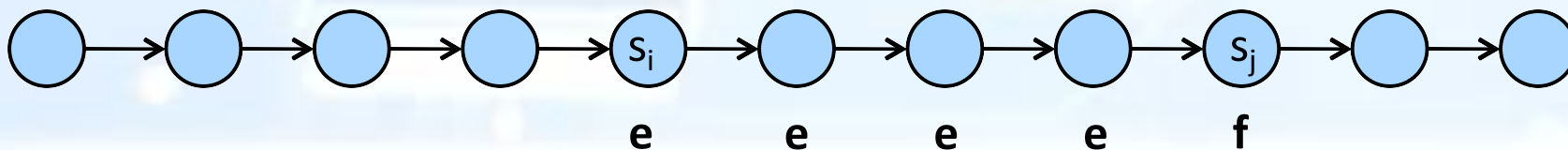
Слабый и сильный until

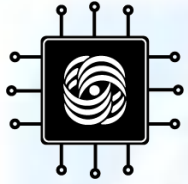
слабый:

$$s_i \models eWf \iff s_i \models f \vee (s_i \models e \wedge s_{i+1} \models (eWf))$$

сильный:
(Spin)

$$s_i \models eUf \iff \begin{array}{l} \exists j, j \geq i : s_j \models f \quad u \\ \forall k, i \leq k < j : s_k \models e \end{array}$$





Практически важные следствия

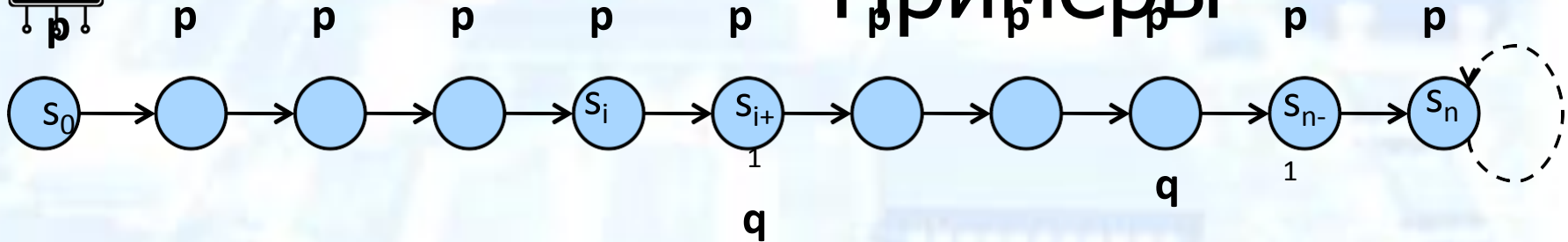
$$\sigma \models e \ W \ false \iff \sigma \models []e$$

$$\sigma \models true \ U \ f \iff \sigma \models \Diamond f$$

$$\sigma \models e \ W \ f \iff \sigma \models []e \vee e \ U \ f$$



Примеры



$$\sigma \models []p$$

$$\sigma \models \Diamond p$$

$$\sigma \models []\Diamond p$$

$$\sigma \models []q$$

$$\sigma \models \Diamond q$$

$$\sigma \models []\Diamond q$$

$$\sigma \models pUq$$

$$\sigma \models [](pUq)$$

$$\sigma \models [](pWq)$$

$$\sigma \models qUp$$

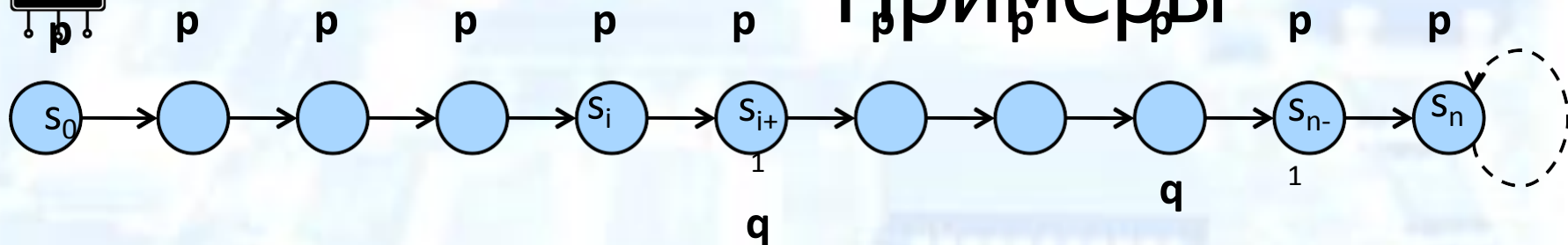
$$\sigma \models [](qUp)$$

$$\sigma \models qWp$$

$$\sigma \models []qWp$$



Примеры



$$\sigma \models []p$$

$$\sigma \models \Diamond p$$

$$\sigma \models []\Diamond p$$

$$\sigma \models []q$$

$$\sigma \models \Diamond q$$

$$\sigma \models []\Diamond q$$

$$\sigma \models pUq$$

$$\sigma \models [](pUq)$$

$$\sigma \models [](pWq)$$

$$\sigma \models qUp$$

$$\sigma \models [](qUp)$$

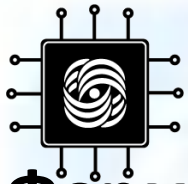
$$\sigma \models qWp$$

$$\sigma \models []qWp$$



Цикличность и стабильность

- Свойством цикличности называется любая темпоральная формула, которая представима в виде $\Box\Diamond p$, где p – формула на состоянии;
- Свойством стабильности называется любая темпоральная формула, которая представима в виде $\Diamond\Box p$, где p – формула на состоянии.



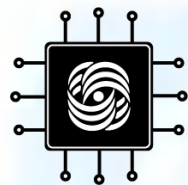
Распространённые LTL-формулы

Формула

Описание

Тип

$\Box p$	всегда p	<i>инвариант</i>
$\langle \rangle p$	рано или поздно p	<i>гарантия</i>
$p \rightarrow \langle \rangle q$	если p , то рано или поздно q	<i>отклик</i>
$p \rightarrow qUr$	если p , то q , затем r	<i>приоритет</i>
$\Box \langle \rangle p$	всегда рано или поздно будет p	<i>цикличность (прогресс)</i>
$\langle \rangle \Box p$	рано или поздно всегда будет p	<i>стабильность (бездействие)</i>
$\langle \rangle p \rightarrow \langle \rangle q$	если рано или поздно p , то рано или поздно q	<i>корреляция</i>



Эквивалентные преобразования

$$![]p \Leftrightarrow \Diamond!p$$

$$!\Diamond p \Leftrightarrow []!p$$

$$!(pUq) \Leftrightarrow !qW(!p \wedge !q)$$

$$!(pWq) \Leftrightarrow !qU(!p \wedge !q)$$

$$[](p \wedge q) \Leftrightarrow []p \wedge []q$$

$$\Diamond(p \wedge q) \Leftrightarrow \Diamond p \wedge \Diamond q$$

$$pW(q \vee r) \Leftrightarrow (pWq) \vee (pWr)$$

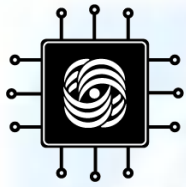
$$(p \wedge q)Wr \Leftrightarrow (pWr) \wedge (qWr)$$

$$pU(q \vee r) \Leftrightarrow (pUq) \vee (pUr)$$

$$(p \wedge q)Ur \Leftrightarrow (pUr) \wedge (qUr)$$

$$[]\Diamond(p \vee q) \Leftrightarrow []\Diamond p \vee []\Diamond q$$

$$\Diamond[](p \wedge q) \Leftrightarrow \Diamond[]p \wedge \Diamond[]q$$



Примеры темпоральных свойств

- **p** всегда истинно;
- **p** рано или поздно станет всегда ложным;
- **p** всегда рано или поздно станет ложным хотя бы ещё один раз;
- **p** всегда ведёт к $\neg q$;
- **p** всегда ведёт к тому, что рано или поздно станет истинным **q**.

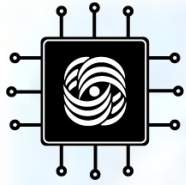
$[]p$

$\langle \rangle [] !p$

$[] (p \rightarrow !q)$

$[] \langle \rangle !p$

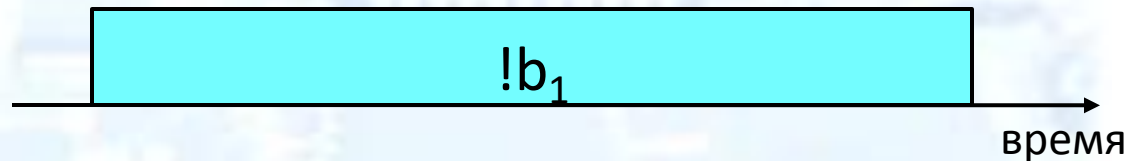
$[] (p \rightarrow \langle \rangle q)$



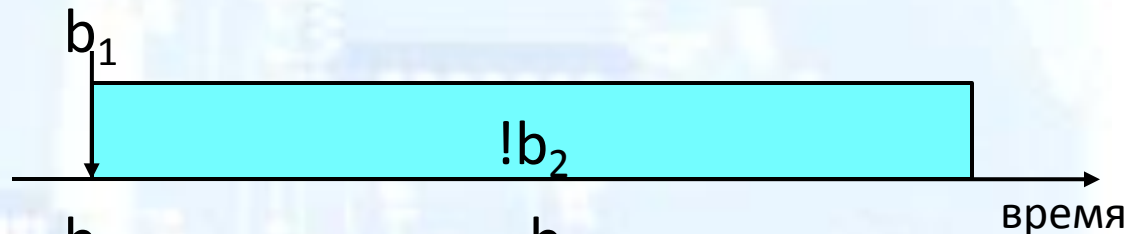
Правильная интерпретация формул LTL

$$LTL: (\langle \rangle (b_1 \wedge (!b_2 U b_2)) \rightarrow [] !a_3)$$

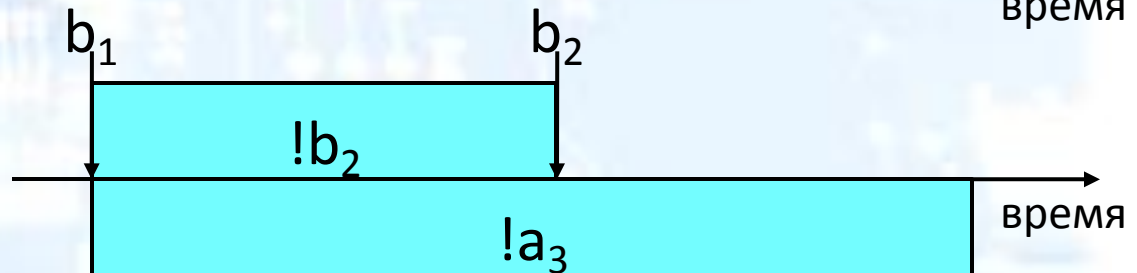
1. Пусть b_1 всегда ложно, $p \rightarrow q$ означает, что $!p \vee q$; **формула выполняется.**



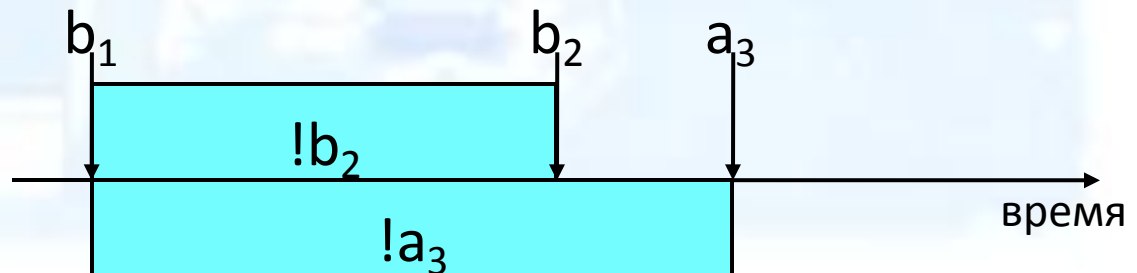
2. Пусть b_1 стало истинно, **формула выполняется.**

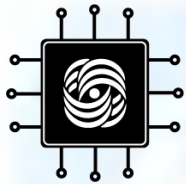


3. Пусть b_1 стало истинно, затем — b_2 , однако a_3 всегда ложно; **формула выполняется.**



4. Пусть b_1 стало истинно, затем — b_2 , затем — a_3 ; **формула не выполняется.**

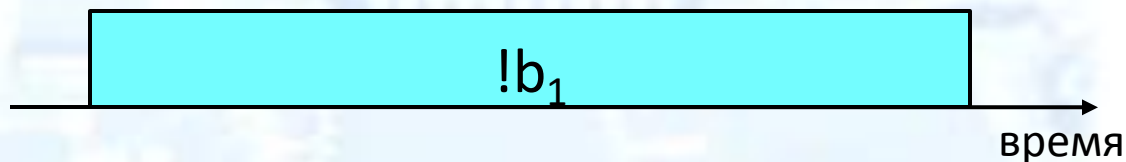




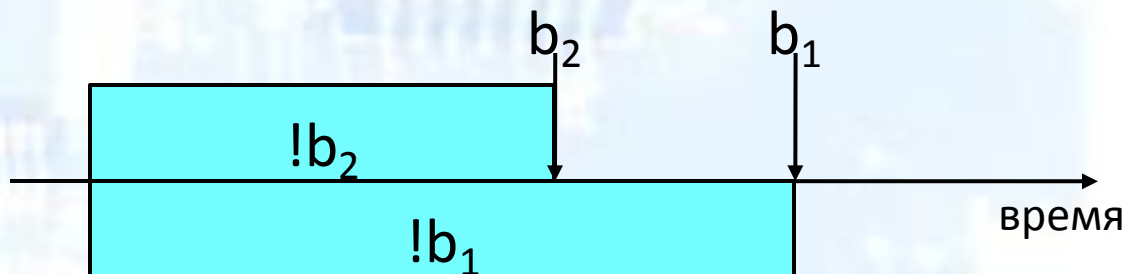
Правильная интерпретация формул LTL

$$LTL: (\langle \rangle b_1) \rightarrow (\langle \rangle b_2)$$

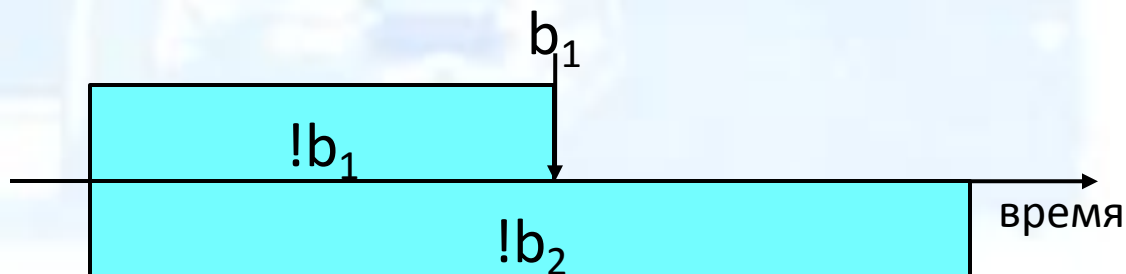
1. Пусть b_1 и b_2 всегда ложно; **формула выполняется.**

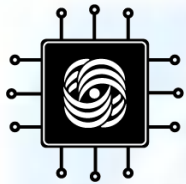


2. Пусть и b_1 , и b_2 становятся истинными; **формула выполняется.**



3. Пусть b_1 становится истинным, но b_2 всегда ложно; **формула не выполняется.**

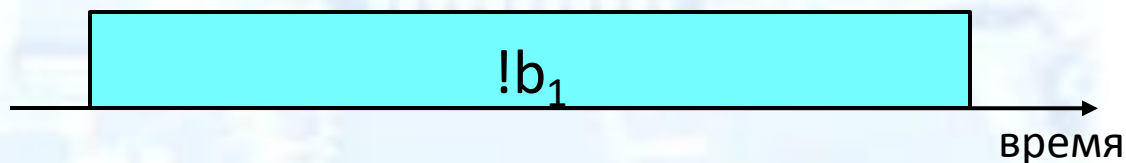




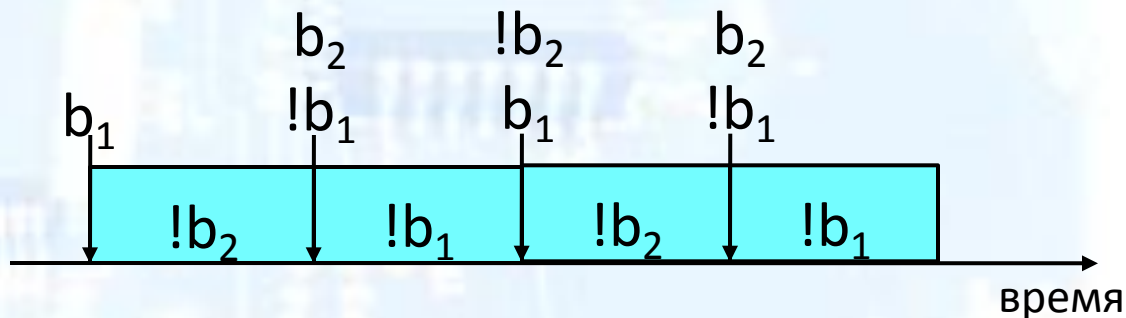
Правильная интерпретация формул LTL

$$LTL: [] (<>b_1) \rightarrow (<>b_2)$$

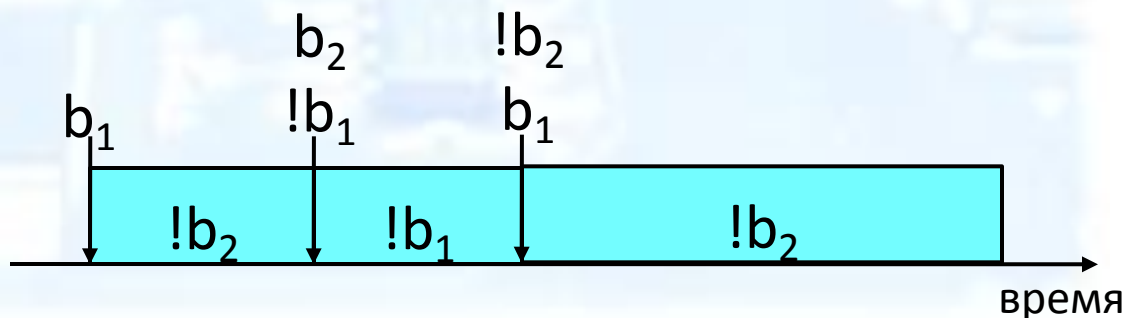
1. Пусть b_1 и b_2 всегда ложно; **формула выполняется.**



2. Пусть и b_1 , и b_2 бесконечно чередуются; **формула выполняется.**



3. Пусть b_2 становится истинным только один раз; **формула не выполняется.**

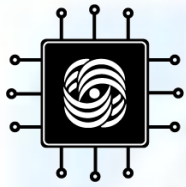




Описание требований при помощи LTL

“ p приведёт к q ”

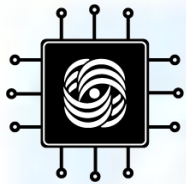
- $p \rightarrow q$
 - нет темпоральных операторов, т.е. применяется только к первому состоянию;
 - выполняется только если $\exists p \vee q$ в первом состоянии, остальная трасса не рассматривается;
 - **не подходит;**
 - **нужно использовать темпоральные операторы.**



Описание требований при помощи LTL

“ p приведёт к q ”

- $[] p \rightarrow q$
 - **правила приоритета!** $[]$ применяется только к p ;
 - означает $([]p) \rightarrow q$;
 - не подходит;
 - **нужно расставить скобки.**

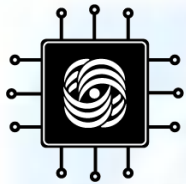


Описание требований при помощи LTL

“ p приведёт к q ”

- **[] ($p \rightarrow q$)**

- проверяем условие во всех состояниях, но причинно-следственная связь между p и q отсутствует;
- выполняется, только если $\neg p \vee q$ во всех состояниях;
- не подходит;
- нужно описать, что p является причиной q .



Описание требований при помощи LTL

“ p приведёт к q ”

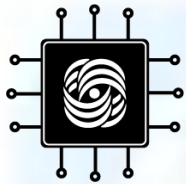
- **[] ($p \rightarrow \langle \rangle q$)**

—уже лучше;

—тем не менее, формула выполнима, если q становится истинным в том же состоянии, что и p — причинно-следственная связь отсутствует;

—не подходит;

—нужно описать, что q не может произойти раньше следующего шага после p .



Описание требований при помощи LTL

“ p приведёт к q ”

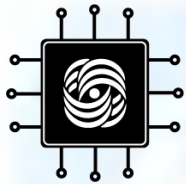
- $[] (p \rightarrow X(<>q))$

- практически то, что нужно;

- формула выполнима, если p всегда ложно;

- возможно, не подходит;

- нужно описать, что p обязательно произойдёт и приведёт к q .



Описание требований при помощи LTL

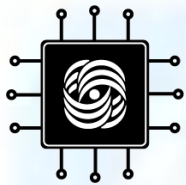
“ p приведёт к q ”

- $[] (p \rightarrow X(<>q)) \ \&\& \ (<>p)$
 - скорее всего, мы имели ввиду именно это;
 - несколько отличается от первоначального $p \rightarrow q$;
 - LTL позволяет выразить множество различных оттенков свойства;
 - подойдёт ли такое свойство для модели параллельной программы?**



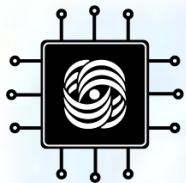
Оператор neXt

- Оператор **X** нужно использовать аккуратно:
 - с его помощью делается утверждение о выполнимости формулы на непосредственных потомках текущего состояния;
 - в распределённых системах значение оператора **X** неочевидно;
 - поскольку алгоритм планирования процессов неизвестен, не стоит формулировать спецификацию в предположении о том, какое состояние будет следующим;
 - стоит ограничиться предположением о справедливости планирования.

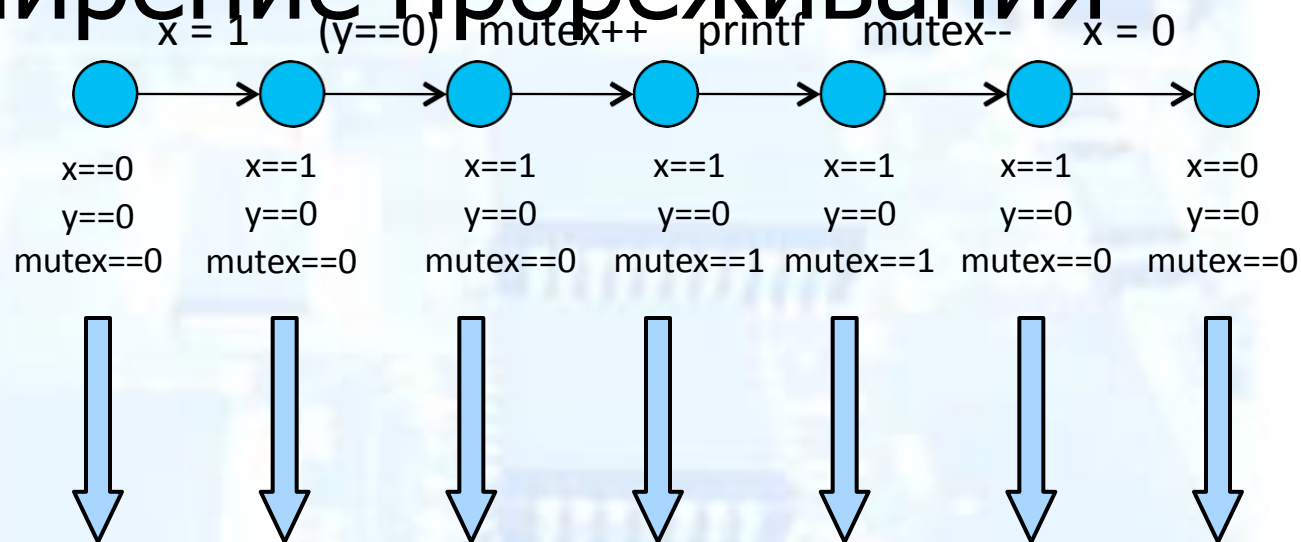


Свойства, инвариантные к прореживанию

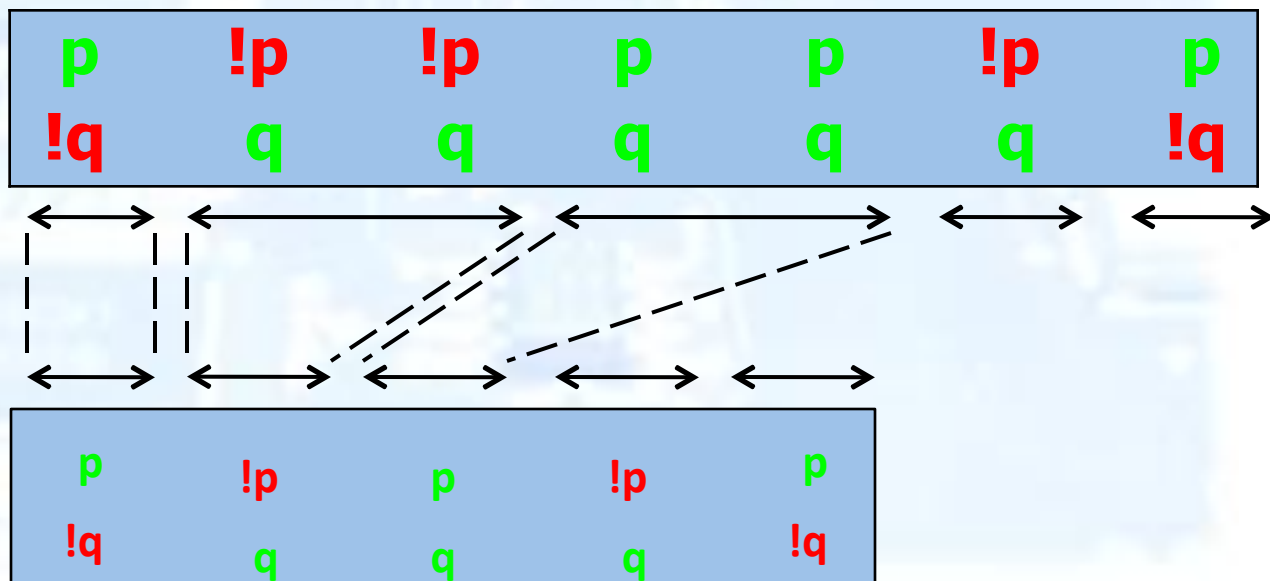
- Пусть φ – трасса некоторого вычисления над пропозициональными формулами P ,
 - по трассе можно определить, выполняется ли на ней темпоральная формула,
 - трассу можно записать в виде $\varphi = \varphi_1^{n1} \varphi_2^{n2} \varphi_3^{n3} \dots$, где значения пропозициональных формул на каждом интервале совпадают.
 - Обозначим $E(\varphi)$ набор всех трасс, отличающихся лишь значениями $n1, n2, n3$ (т.е. длиной интервалов)
 - $E(\varphi)$ называется **расширением прореживания** φ .



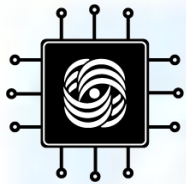
Расширение прореживания



трасса φ



трасса $\varphi_1 \in E(\varphi)$

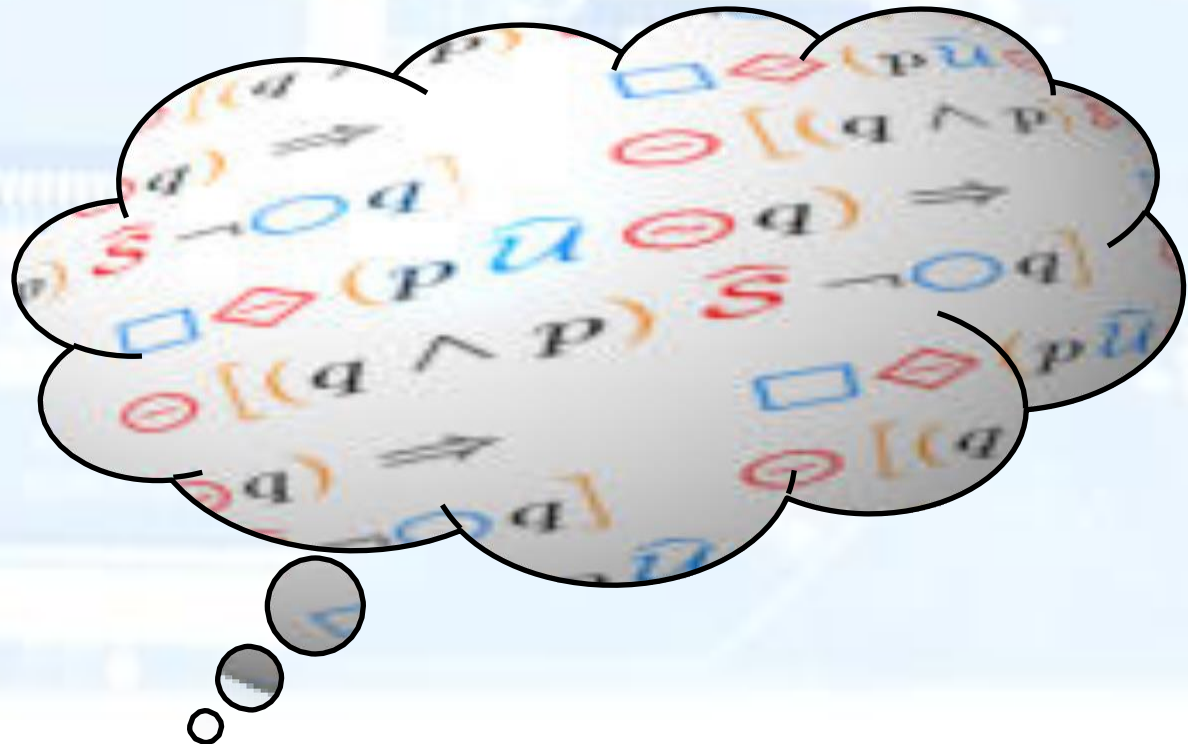
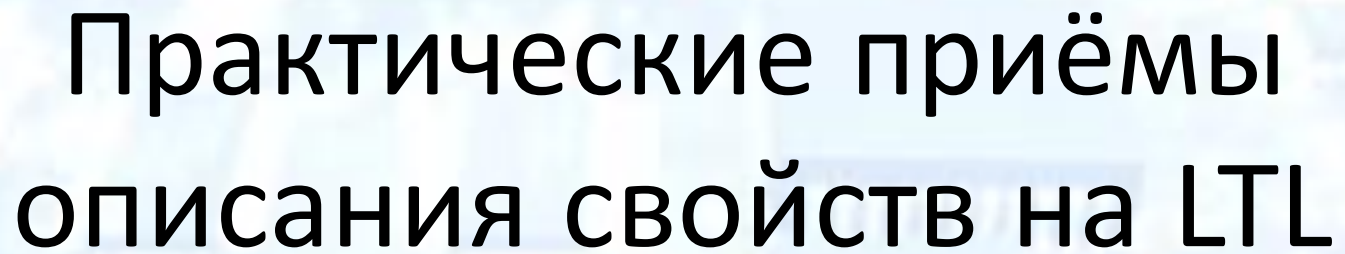


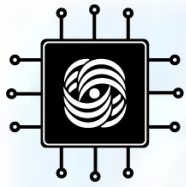
Свойства, инвариантные к прореживанию

- Свойство φ , инвариантное к прореживанию, либо истинно для всех трасс из $E(\varphi)$, либо ни для одной из них:

$$\varphi \models f \rightarrow \forall v \in E(\varphi), v \models f$$

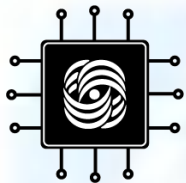
- истинность такого свойства зависит от порядка, в котором пропозициональные формулы меняют свои значения, и не зависит от длины трассы;
- Теорема:** все формулы LTL без оператора X инвариантны к прореживанию.
- Более того, в рамках LTL без X можно описать **все** свойства, инвариантные к прореживанию.





Практические приёмы описания свойств на LTL

- Выполнимость формулы LTL проверяется только для первого состояния в трассе
- Темпоральные операторы управляют проверкой выполнимости своих аргументов
- Сложное свойство можно (и нужно!) строить как суперпозицию простых
- Суперпозиция темпоральных операторов не ограничивает диапазон действия «вложенного» оператора.



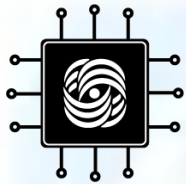
Выполнимость формул LTL

- Выполнимость формулы LTL определена и проверяется для одного (первого) состояния трассы $\left(\begin{array}{l} \text{выполнимость подформул} \\ \text{— уже нет} \end{array} \right)$
- Распространение свойств на другие состояния управляется темпоральными операторами $\left(\begin{array}{l} \text{для этого они и нужны} \end{array} \right)$
- Единственный оператор, который может ограничить сверху проверку выполнимости формулы — **Until** $\left(\begin{array}{l} \text{описать свойство, выполняющееся} \\ \text{для участка программы (одной} \\ \text{итерации, одного вызова функции)} \\ \text{без Until не получится} \end{array} \right)$



Суперпозиция формул LTL

- Составлять сложные формулы LTL нужно методом суперпозиции простых формул
- Внешняя формула задаёт, на каких участках вычислений будет проверяться подформула
- Подформула задаёт свойства, проверяемые для участков вычислений
- Суперпозиция темпоральных операторов не ограничивает диапазон действия «вложенного» оператора ■



База шаблонов темпоральной логики

<http://patterns.projects.cis.ksu.edu>

Логические паттерны (LTL/CTL/GIL)

встречаемость
(occurrence)

отсутствие
(absense)

универсальность
(universality)

существование
(existence)

bounded
existence

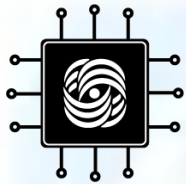
порядок
(order, sequence)

приоритет
(precedence)

chain
precedence

отклик
(response)





База шаблонов темпоральной логики

<http://patterns.projects.cis.ksu.edu>

- Для каждого шаблона – пять вариантов формул:

имя

пример для “absense” и LTL

всегда

$[\] (!p)$

перед r

$\langle \rangle r \rightarrow (!p \cup r)$

после q

$[\] (q \rightarrow [\] (!p))$

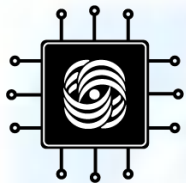
между r и q

$[\] ((r \ \&\& \ !q \ \&\& \ \langle \rangle q) \rightarrow (!p \cup q))$

после r до q

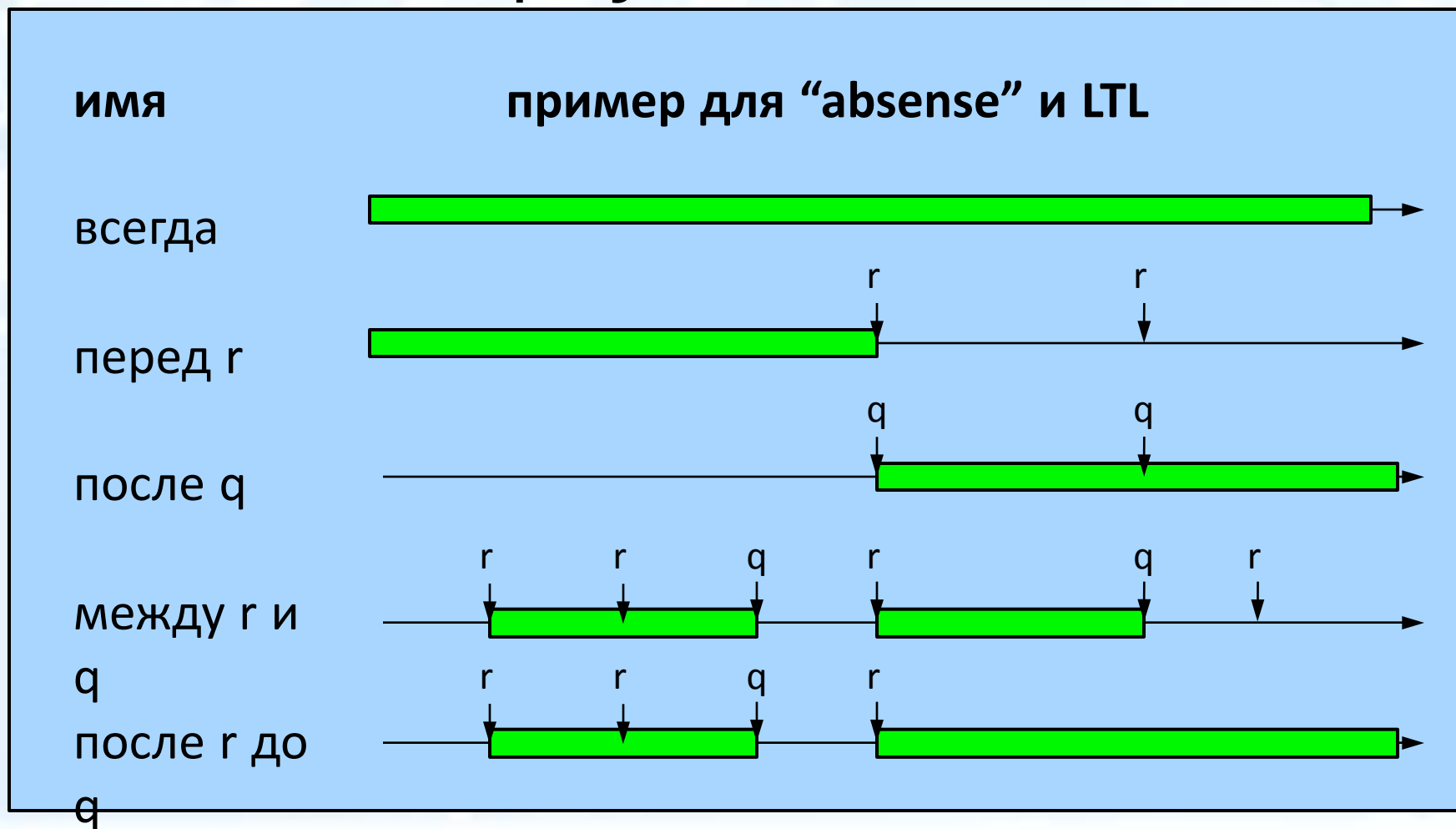
$[\] ((r \ \&\& \ !q) \rightarrow ((!p \cup q) \ || \ [\] !p))$

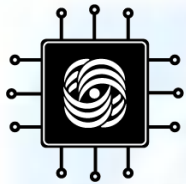
В чём разница?



База шаблонов темпоральной логики <http://patterns.projects.cis.ksu.edu>

- Для каждого шаблона пять вариантов формул:

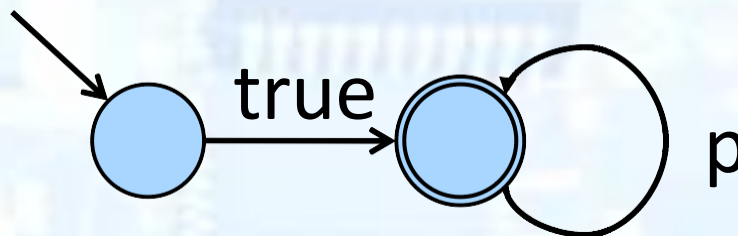




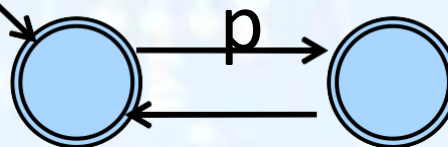
Пример свойства, не выражимого на LTL

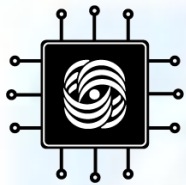
- (p) **может** быть истинным после выполнения системой чётного числа шагов, но **никогда не истинно** после **нечётного**.

- $[]X(p)$ не подходит

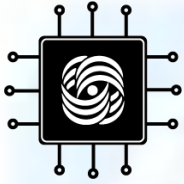


- $p \ \&\& \ [](p \rightarrow X!p) \ \&\& \ [](!p \rightarrow Xp)$ – также не подходит (здесь p **всегда** истинно после чётных шагов)





Спасибо за внимание



НАДЁЖНОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

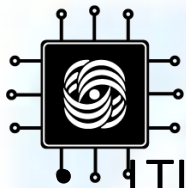
Лекция 6: Логика TCTL *Средство верификации UPPAAL*

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



План лекции

- Логика CTL, CTL* и TCTL
- Общее описание средства UPPAAL
- Модуль описания
 - Пример системы реального времени
 - Описание сети автоматов
 - Синтаксис выражений
- Модуль симуляции
- Модуль верификации
 - Общее описание
 - Язык запросов



Сравнение LTL с другими логиками

- LTL-формула описывает свойство, которое должно выполняться на **всех** вычислениях, начинающихся из исходного состояния системы

Операторы:

!	логическое отрицание
&&	логическое И
	логическое ИЛИ
X	в следующем
U	состоянии сильный until
U	слабый until
<>	рано или поздно
[]	всегда

Серым отмечены операторы, которые можно вывести из других:

$\varphi_1 \varphi_2$	==	$!(\neg \varphi_1 \&\& \neg \varphi_2)$
$\<\> \varphi$	==	$\text{true } U \varphi$
$[] \varphi$	==	$\neg \<\> \neg \varphi$
$\varphi_1 U \varphi_2$	==	$[] \varphi_1 (\varphi_1 U \varphi_2)$

грамматика:

пропозициональные формулы:

p
!f
(f)
f && f
f || f

темпоральные формулы:

f
! φ
(φ)
 $\varphi \&\& \varphi$
 $\varphi || \varphi$
X φ
 $\varphi U \varphi$
 $\<\> \varphi$
 $[] \varphi$

p – некоторый пропозициональный символ

f – некоторая пропозициональная формула

φ – некоторая темпоральная формула



Логика STL*

Логика ветвящегося времени:

- использует кванторы $\forall$ и $\exists$,
- использует F вместо $\langle \rangle$ и G вместо $[]$.

Операторы

!	логическое отрицание
&&	логическое И
	логическое ИЛИ
E	существует путь
A	для всех путей
X	в следующем
U	состоянии until
F	(сильный)
G	рано или поздно всегда

Серым отмечены операторы, которые можно вывести из других:

$$\begin{aligned} \varphi_1 || \varphi_2 &= !(\varphi_1 \&\& ! \varphi_2) \\ &= A \varphi &= !E ! \varphi \\ &= &= true U \varphi \\ &= &= !F ! \varphi \end{aligned}$$

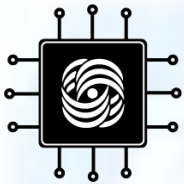
формулы состояния: p

!f
(f)
f && f
f || f
A φ
E φ

формулы пути:

f
! Φ
(φ)
 $\varphi \&\& \varphi$
 $\varphi || \varphi$
X φ
 $\varphi U \varphi$
F φ
G φ

p – некоторый пропозициональный символ
f – некоторая формула состояния
 φ – некоторая формула пути



Логика CTL

- Логика CTL – фрагмент логики CTL*, в котором под управлением квантора пути (E или A) может находиться не более одного оператора X или U.

Корректная CTL формула:

p
 $\neg \varphi$
 $\varphi \ \&\& \ \varphi$
 $\varphi \ || \ \varphi$
 $E \ X \ \varphi$
 $E (\varphi \ U \ \varphi)$
 $A (\varphi \ U \ \varphi)$

p – некоторый пропозициональный символ

f – некоторая формула состояния

φ – некоторая формула пути

Можно вывести:

EF f	==	E(true U f)
AF f	==	A(true U f)
EG f	==	!AF !f
AG f	==	!EF !f
AX f	==	!EX !f



Пример

В LTL $\langle \rangle p$ означает:

$A\langle \rangle p$ **для всех** вычислений, начинающихся
в исходном состоянии s_0 , выполняется $\langle \rangle p$

В CTL можно
выразить:

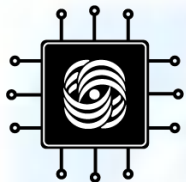
$EF(p)$ **существует** вычисление, для
которого выполняется $\langle \rangle p$

$AF(p)$ **для всех** вычислений выполняется $\langle \rangle p$

$AG(p)$ **для всех** вычислений p – инвариант

$EG(p)$ **существует** вычисление, для которого
выполняется p – инвариант

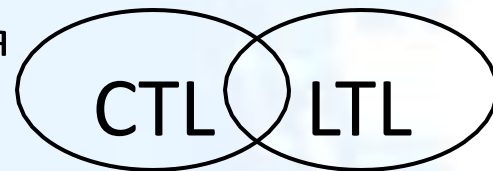
ИТД.



Выразительные возможности CTL* и

- CTL* и CTL описывают подмножества ω -регулярных автоматов над *деревьями*

- автоматы над деревьями более выразительны, чем автоматы над словами (CTL-формула выполнима на дереве трасс, а не на одной трассе);
- CTL и LTL являются подмножествами CTL*;
- CTL и LTL не сравнимы по выразительной мощности (пересекаются, но не включают);
- на LTL можно описать свойства, не выразимые на CTL:
 - CTL не позволяет описать свойства вида $[\]\langle \rangle(p)$,
 - при помощи $[\]\langle \rangle(p)$ в LTL задаются ограничения справедливости;



- на CTL можно описать свойства, не выразимые на LTL:
 - на LTL нельзя описать свойства вида $AGEF(p)$,
 - $AGEF(p)$ используется для описания свойства reset: из любого состояния система может перейти в нормальное.



Выразительная мощность

Модальное μ -исчисление,
Автоматы над ω -деревьями

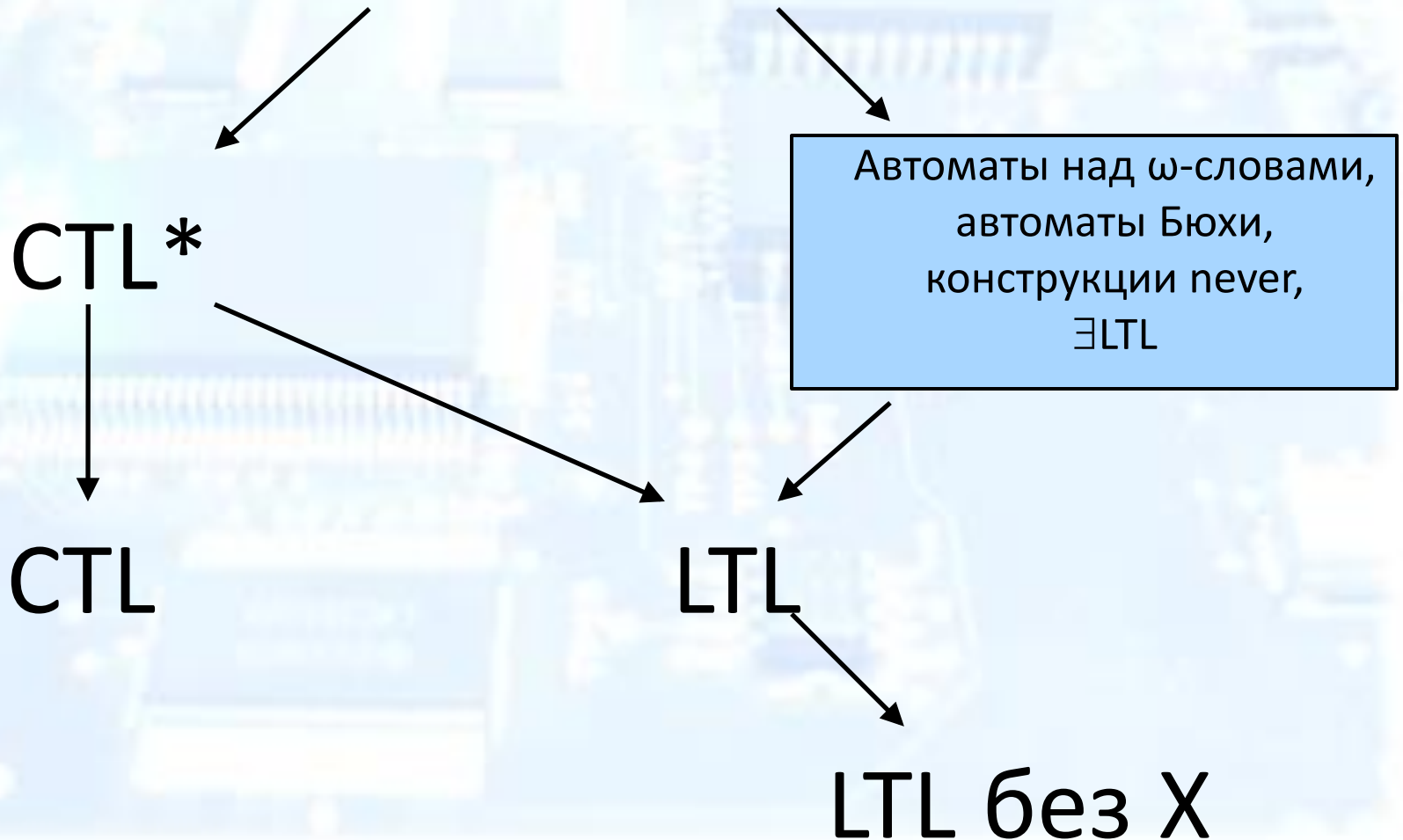
CTL*

CTL

Автоматы над ω -словами,
автоматы Бюхи,
конструкции never,
 $\exists$ LTL

LTL

LTL без X





Примеры формализаций высказываний

- *Джейн вышла замуж и родила ребенка*

$P(\text{Джейн\_выходит\_замуж} \wedge F \text{Джейн\_рожает\_ребенка})$

- *Джейн родила ребенка и вышла замуж*

$P(\text{Джейн\_рожает\_ребенка} \wedge F \text{Джейн\_выходит\_замуж})$

- *Джон умер и его похоронили*

$P(\text{Джон\_умирает} \wedge XF \text{Джона\_хоронят})$

- *Если я видел ее раньше, то я ее узнаю при встрече*

$G(P \text{Увидел} \Rightarrow G(\text{Встретил} \Rightarrow X \text{Узнал}))$

- *Ленин – жил, Ленин – жив, Ленин – будет жить* (В.В.Маяковский)

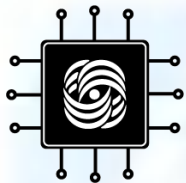
$PG \text{Ленин\_жив}$

- *Любое посланное сообщение будет получено*

$G(\text{Послано}(m) \Rightarrow F \text{Получено}(m))$

- *Вчера он сказал, что придет завтра, значит, он придет сегодня*

$X^{-1}X \text{Приходит} \Rightarrow \text{Приходит} \quad (\text{истинно})$



LTL и CTL – подклассы CTL*

Алгоритмы проверки выполнения формул CTL* сложны $\Rightarrow$ нужны подклассы

В LTL - формулы пути, которые должны выполняться для всех вычислений предваряются квантором пути A

В CTL каждый темпоральный оператор предваряется квантором пути A или E

Формулы LTL:

$AG(p \Rightarrow Fq)$

$A(\neg a \wedge Gb \wedge (aU \neg c))$

$A(aU \neg b)$

Формулы CTL:

$AG(p \wedge \neg EF(q \Rightarrow r))$

$EF(a \wedge E(aU \neg c))$

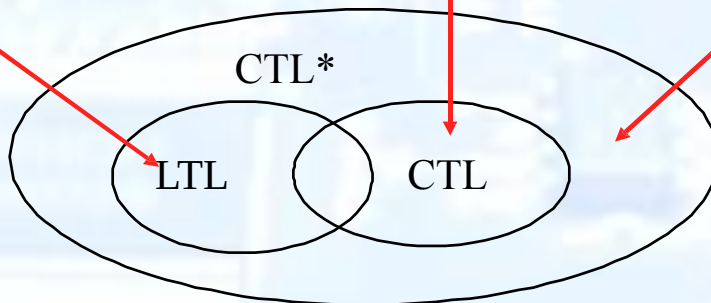
$A(aU \neg b)$

Формулы CTL*:

$E(\neg p \wedge XAFq)$

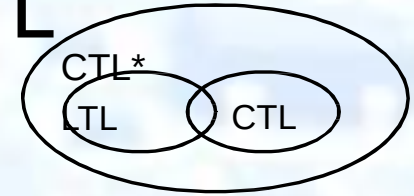
$EX(a \wedge AX(bUc))$

$A(aU \neg (Fb))$

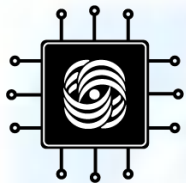




Сравнение логик LTL и CTL



- Формулы этих двух логик характеризуют свойства разных объектов
 - LTL – формулы пути, CTL – формулы состояний
- Выражают свойства вычислений, которые представлены по-разному
 - LTL – множество поведений, CTL – деревья поведений
- Интерпретируются по-разному
 - формулы LTL - на бесконечном множестве поведений
 - формулы CTL – на конечном множестве состояний
- Методы анализа - алгоритмы model checking - совершенно разные
- Выразительная мощь несравнима: есть формулы CTL, невыразимые в LTL, и наоборот
 - например, $FG\varphi$ не может быть выражена в CTL (с некоторого времени в будущем φ будет все время истинным) $AFAGp$ сильнее, $AFEGp$ – слабее
 - например, $AGEF\varphi$ в CTL не может быть выражена в LTL

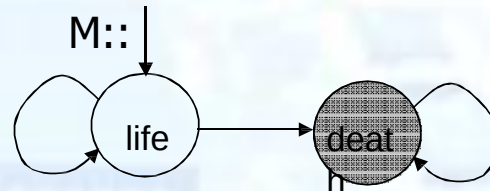


Пример формализации в CTL*

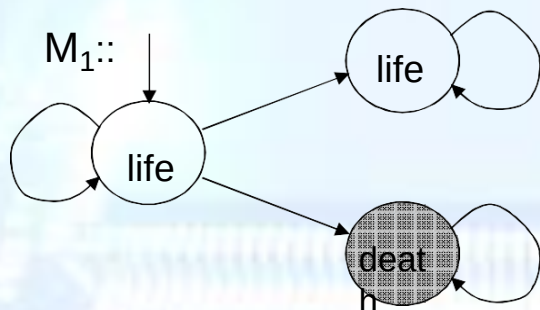
*“Летят за днями дни, и каждый час уносит
Частичку бытия, а мы с тобой вдвоем
Предполагаем жить, и глядь — как раз - умрем”*

А.С.Пушкин

$$\varphi_{br} = \mathbf{A} [(\mathbf{G} \text{ life}) \Rightarrow (\mathbf{GEX} \text{ death})]$$

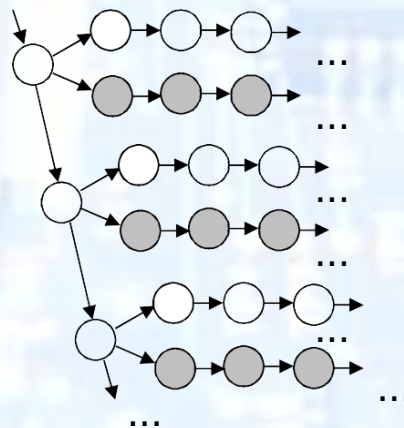


$$M \models \varphi_{br}$$

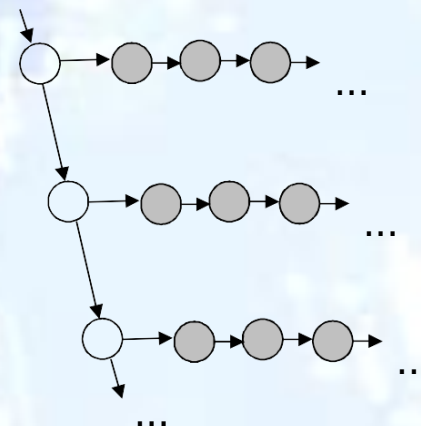


$$M_1 \not\models \varphi_{br}$$

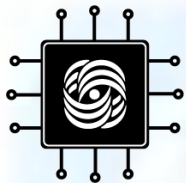
Развертка M1:



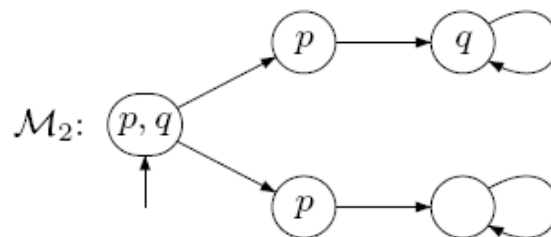
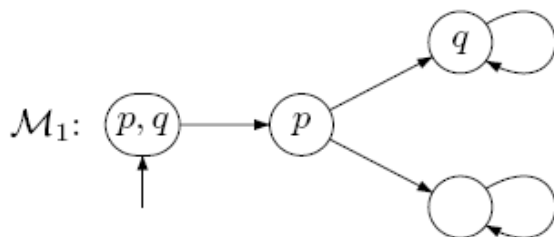
Развертка M:



На развёртке M1 формула φ_{br} не выполняется: ее развертка имеет траектории “вечной жизни” без возможного отклонения на состояния, помеченные *death*



Различия LTL и CTL*



- Эти две модели не различаются LTL, но различаются CTL*:
 - $AG(p \rightarrow EFq)$. В $\mathcal{M}_1$ эта формула выполняется, в $\mathcal{M}_2$ она не выполняется
 - В LTL обе модели представляются одним и тем же множеством из двух вычислений, поэтому LTL их не различает



TCTL

TCTL – Timed CTL – естественное расширение операторов U, F, ... логики CTL количественной информацией.

Грамматика TCTL (= CTL + Time):

$\phi ::= p \mid \alpha \mid \neg\phi \mid \phi \wedge \phi \mid z \text{ in } \phi \mid E[\phi U \phi] \mid A[\phi U \phi]$

p – атомарный предикат

α - ограничение на таймеры и формульные часы z – формульные часы

$z \text{ in } \phi$ - введение новых часов в формулу ϕ

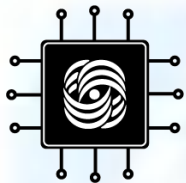
$E[\phi U \phi]$, $A[\phi U \phi]$ – как в CTL

Обозначение: $E[\phi U_{\alpha}\psi] \equiv z \text{ in } E[(\phi \& \alpha)U\psi]$ Выводимые

операторы $EF_{\alpha}\phi \equiv E[\text{True} U_{\alpha}\phi]$ и т.д.

Формулы TCTL включают $E[\phi U_{\sim k}\psi]$, $A[\phi U_{\sim k}\psi]$, $EF_{\sim k}\phi$, $EG_{\sim k}\phi$, $AF_{\sim k}\phi$, $AG_{\sim k}\phi$

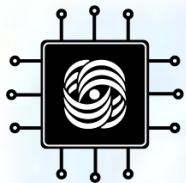
где $\sim$ - любой символ из $\{<, \leq, =, \geq, >\}$ и k – рациональное число



Примеры свойств реального времени

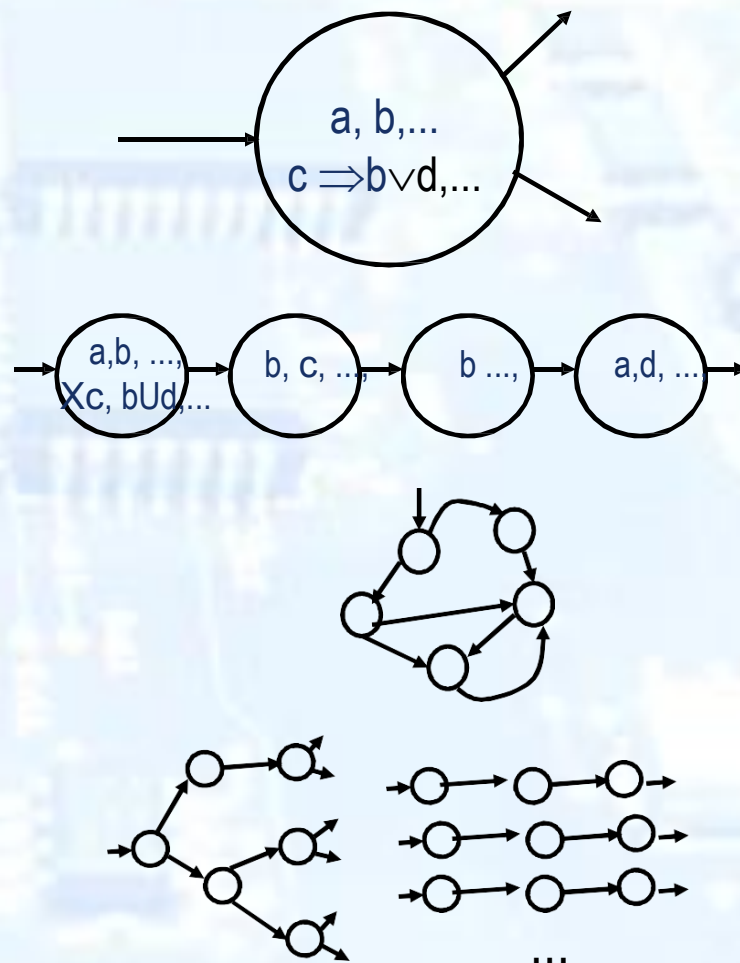
Model checking временных автоматов относительно выполнения заданной формулы TCTL сводится к model checking его регионального графа относительно выполнения формулы CTL

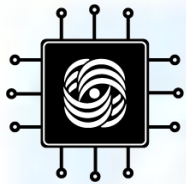
1. $[p \text{ U}_{<2} q]$ – p истинно непрерывно до тех пор, пока не станет истинно q , и истинность q наступит не позднее, чем через 2 единицы времени
2. $\text{AG}(\text{problem} \Rightarrow \text{AG}_{\geq 5} \text{alarm})$ – как только проблема возникла, сигнал alarm зазвучит сразу и будет звучать не менее 5
3. $\text{AG}(\neg \text{far} \Rightarrow \text{AF}_{<7} \text{far})$ – поезд покинет область контроля не позже, чем через 7
4. $\text{AG} [\text{send}(m) \Rightarrow \text{AF}_{<5} \text{receive}(r_m)]$ – подтверждение приходит в пределах 5
5. $\text{EG} [\text{send}(m) \Rightarrow \text{AF}_{>4} \text{receive}(r_m)]$ - подтверждение может быть получено более, чем за 4
6. $\text{AG} [\text{AF}_{=15} \text{tick}]$ – тики следуют периодически точно через 15 е.в. (но, кроме того, могут быть и в промежутках)
7. $\text{AG} (x \leq y)$ - таймер x всегда не больше таймера y
8. $\text{A} [\text{off} \text{ U } x \geq 3]$ - таймер x не будет иметь значение ≥ 3 до тех пор, пока таймер x не будет иметь значение ≥ 3 на любом пути из начального состояния если светофор выключен, то он будет



Общие идеи в TL

- Логика высказываний строится введением
- В логике линейного времени LTL кроме атомарных утверждений и операций логики высказываний вводятся темпоральные операторы $\{U, X\}$ (кроме них удобно использовать еще F и G)
- По конкретной цепочке состояний (миров) в каждом состоянии можем вычислить истинностные значения любой формулы темпоральной логики LTL
- В логике CTL* добавляются кванторы пути, позволяющие различать свойства различных путей
- Формула CTL* определена для конкретной интерпретации (структуры Крипке) и всех возможных ее вычислений
- CTL является подмножеством CTL* - в формулах CTL **каждый** темпоральный оператор предваряется квантором пути





Особенности URRAAL

URRAAL — это весьма популярное средство верификации систем реального времени

- Простое в использовании
- Бесплатное для научных исследований
- Имеет в основе простую, но при этом довольно мощную математическую модель
- Малоприспособно для разработки больших систем
- Не поддерживает описание иерархии вложенности компонентов систем



Структура

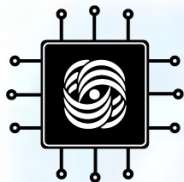
- Модуль описания
 - входная модель – расширение модели **сети временных автоматов**
 - возможность описания параметризованных шаблонов
 - наличие данных различной степени локальности
- Модуль симуляции
 - генерация и визуализации трассы сети
 - различные типы генерации (случайная, из файла)
- Модуль верификации
 - проверка темпоральных свойств
 - предоставление трассы-контрпримера
 - воспроизведение контрпримера в модуле симуляции



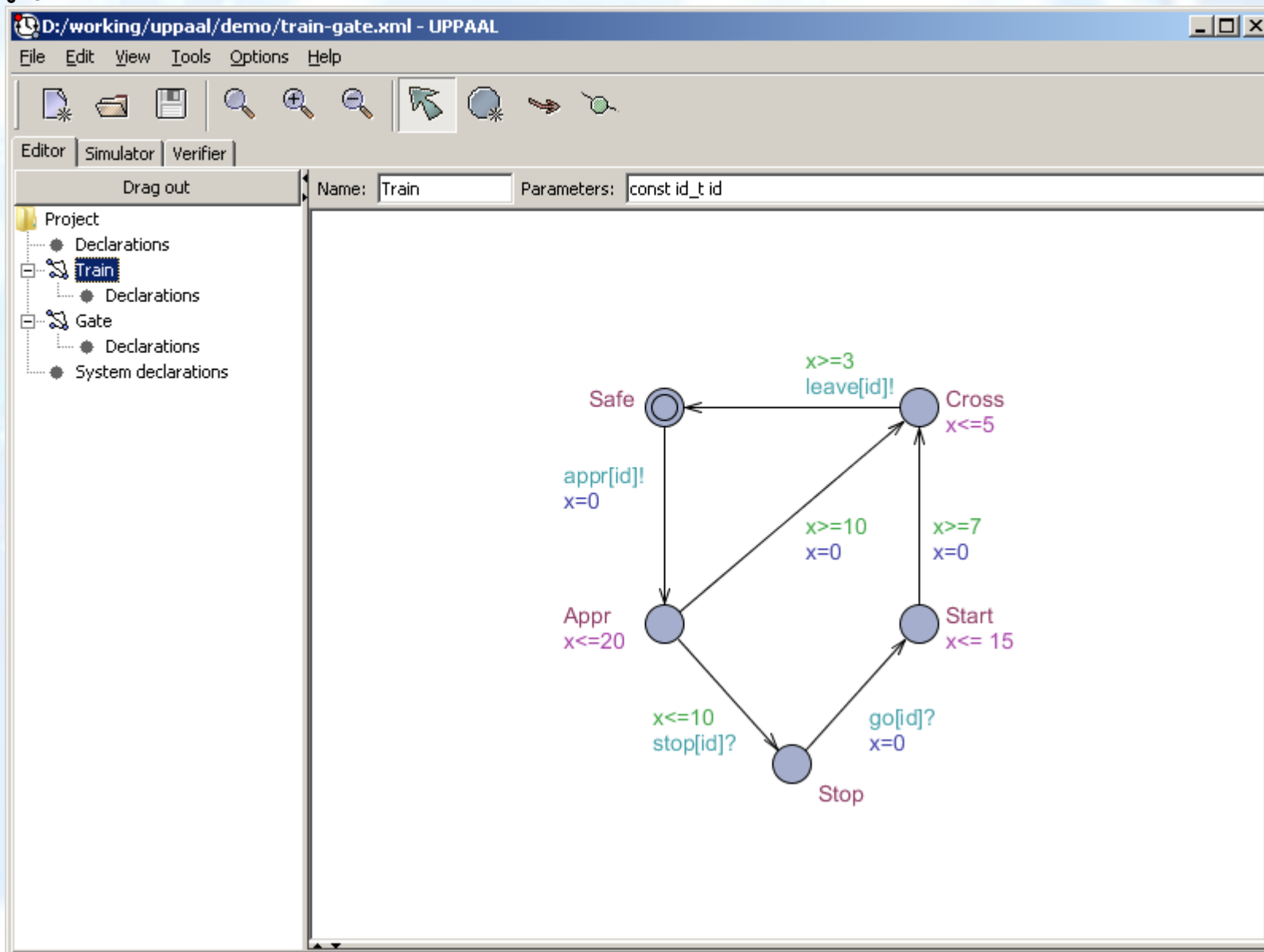
Пример

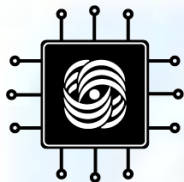
Train Gate

- Система обеспечения доступа к критическому ресурсу
- Требуется обеспечить проход нескольких поездов по одному мосту
- В наличии устройство управления поездами
- Остановка и разгон поезда происходят не мгновенно

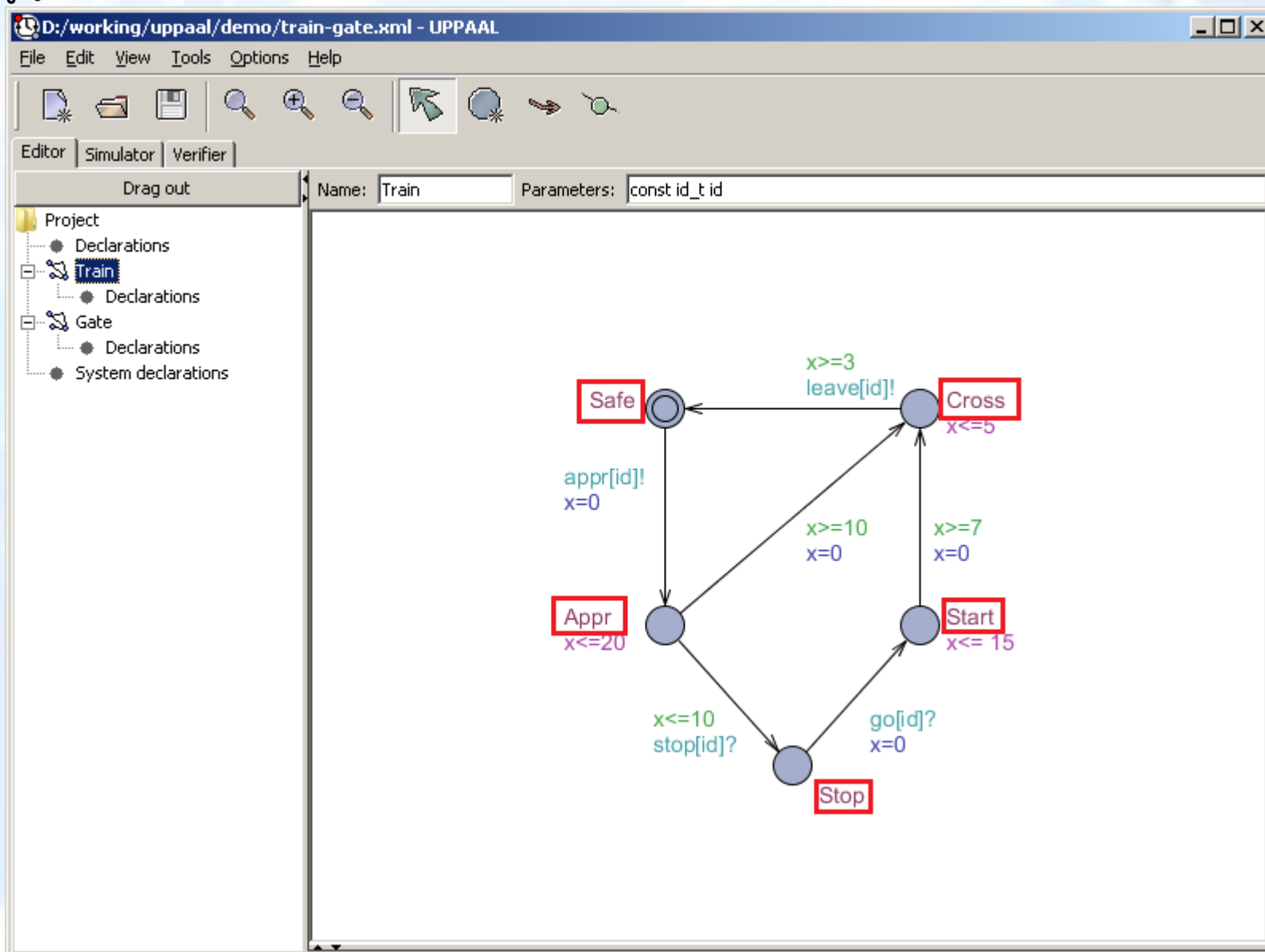


Модуль описания



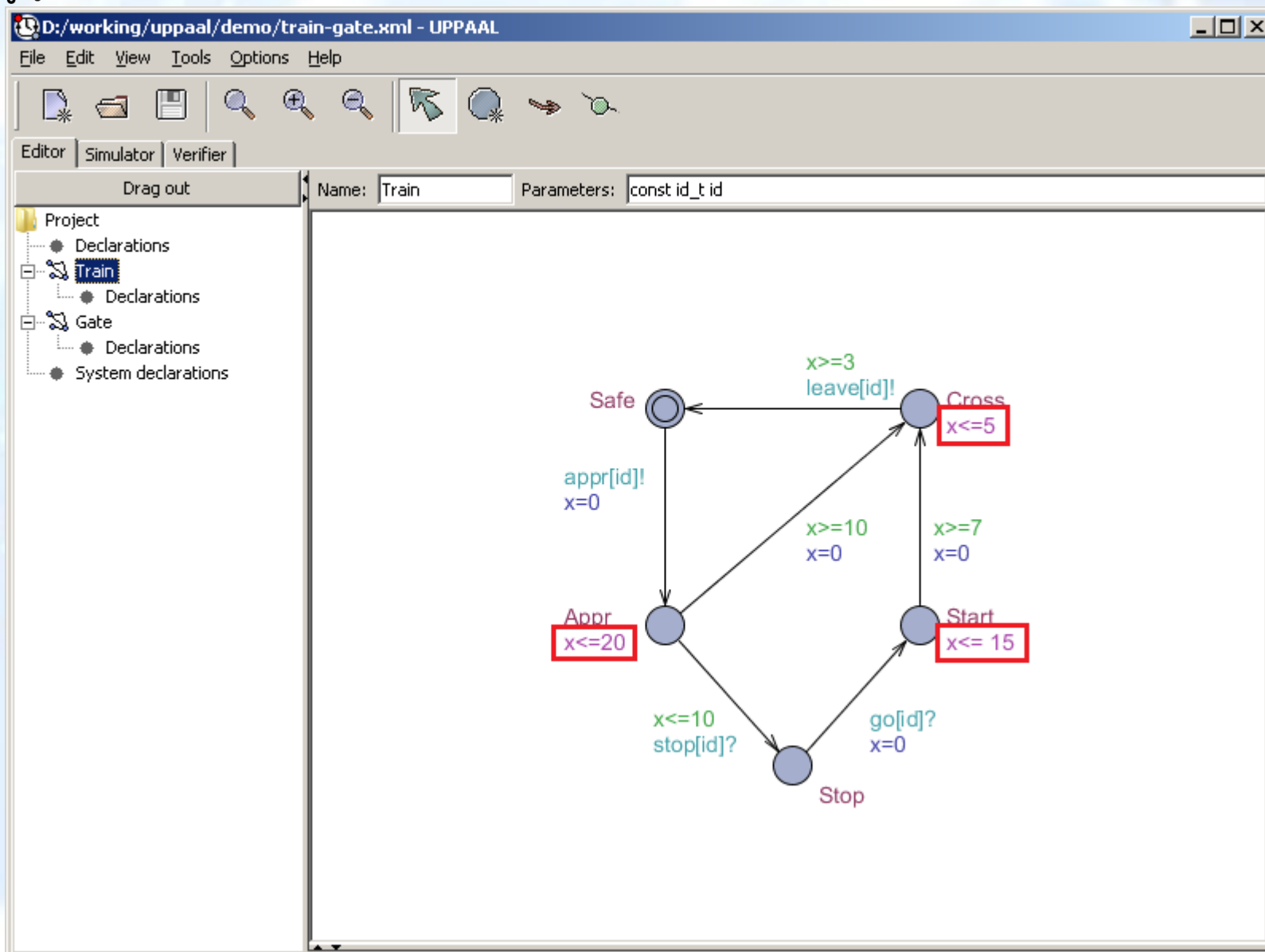


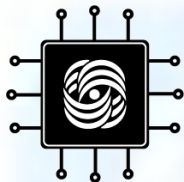
Имена состояний



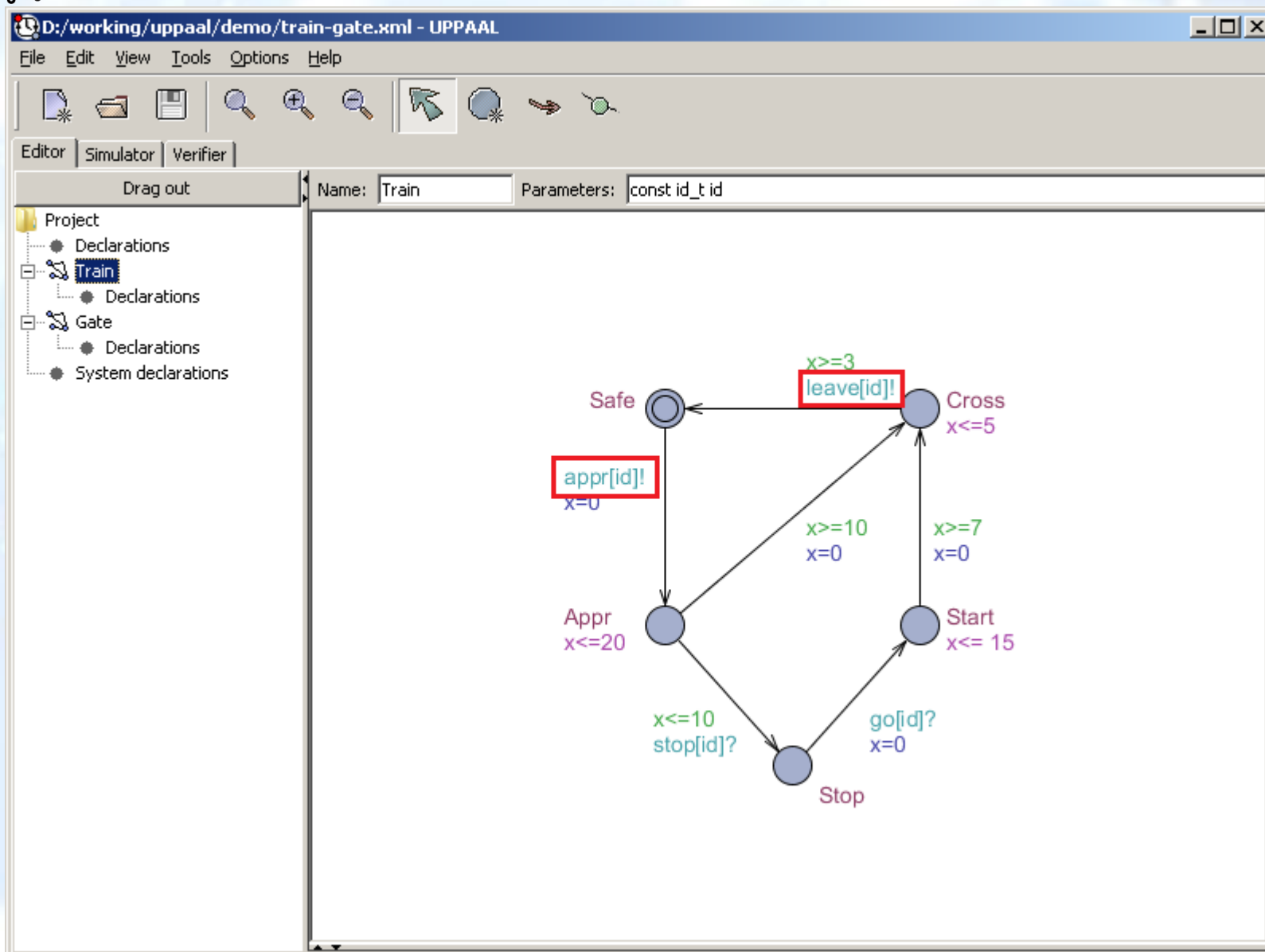


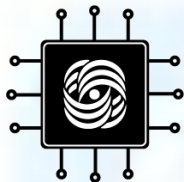
Инварианты



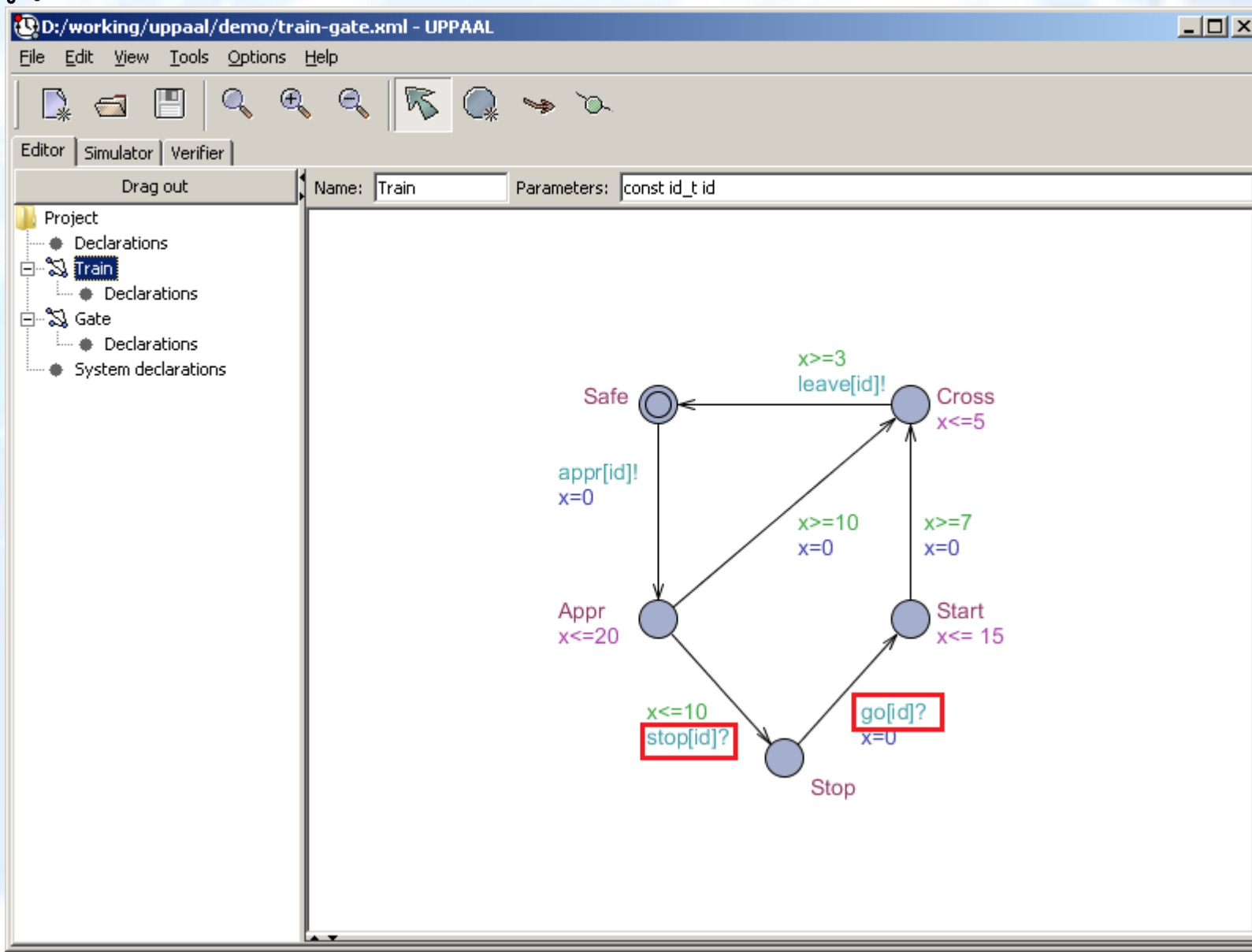


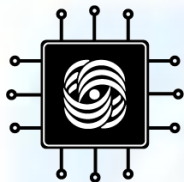
Посылка сигналов



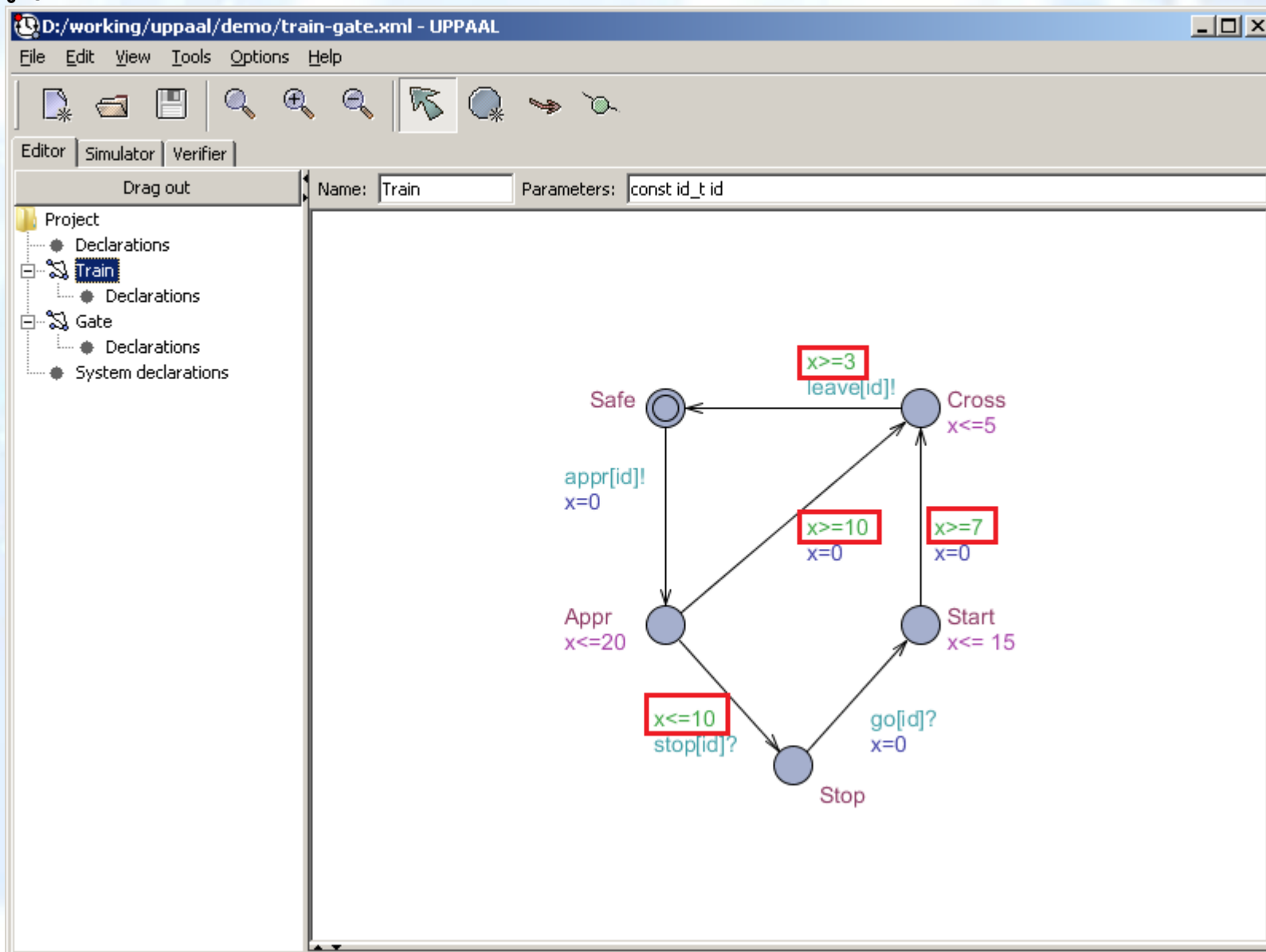


Прием сигналов



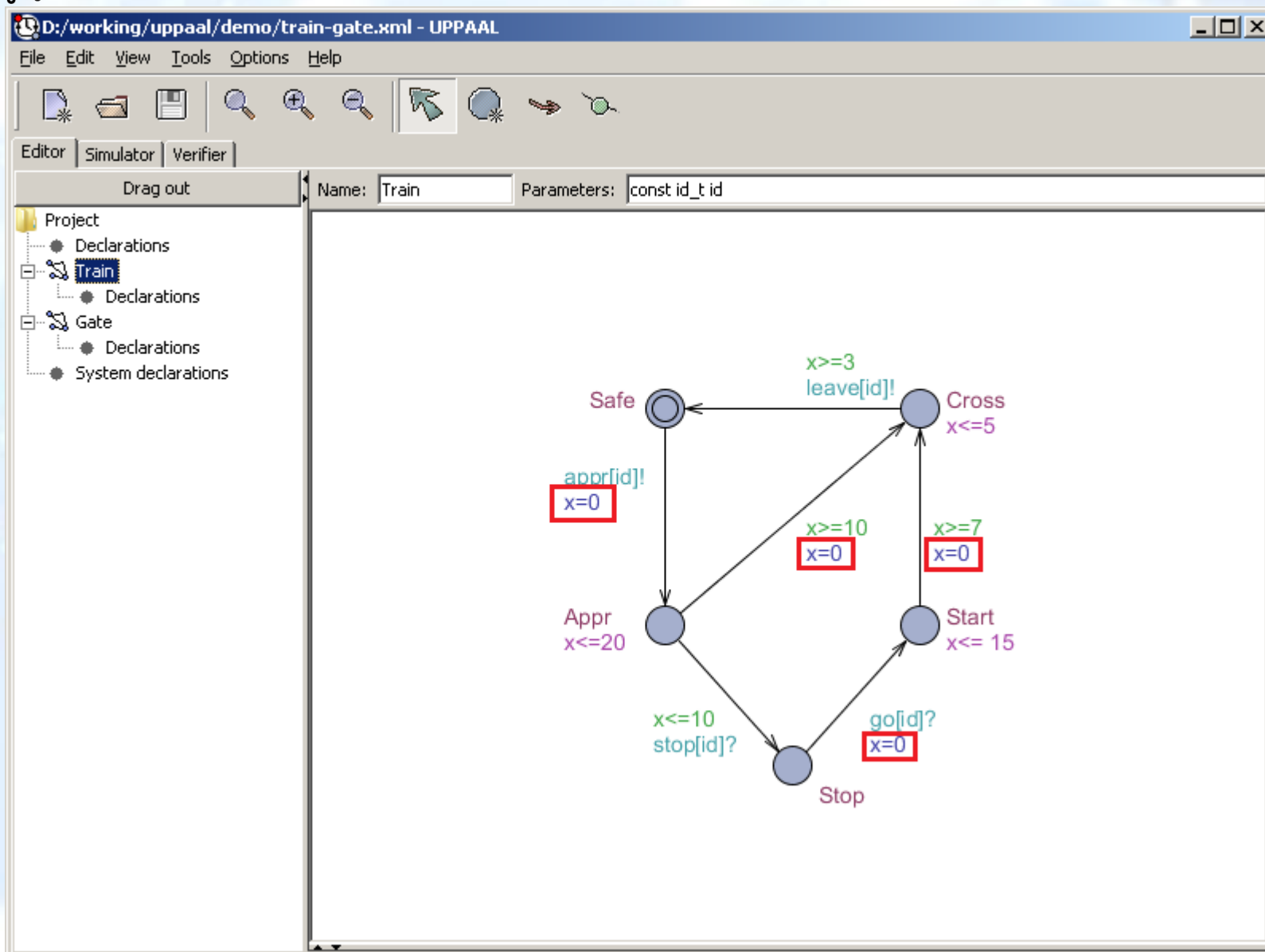


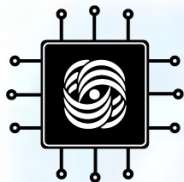
Предусловия



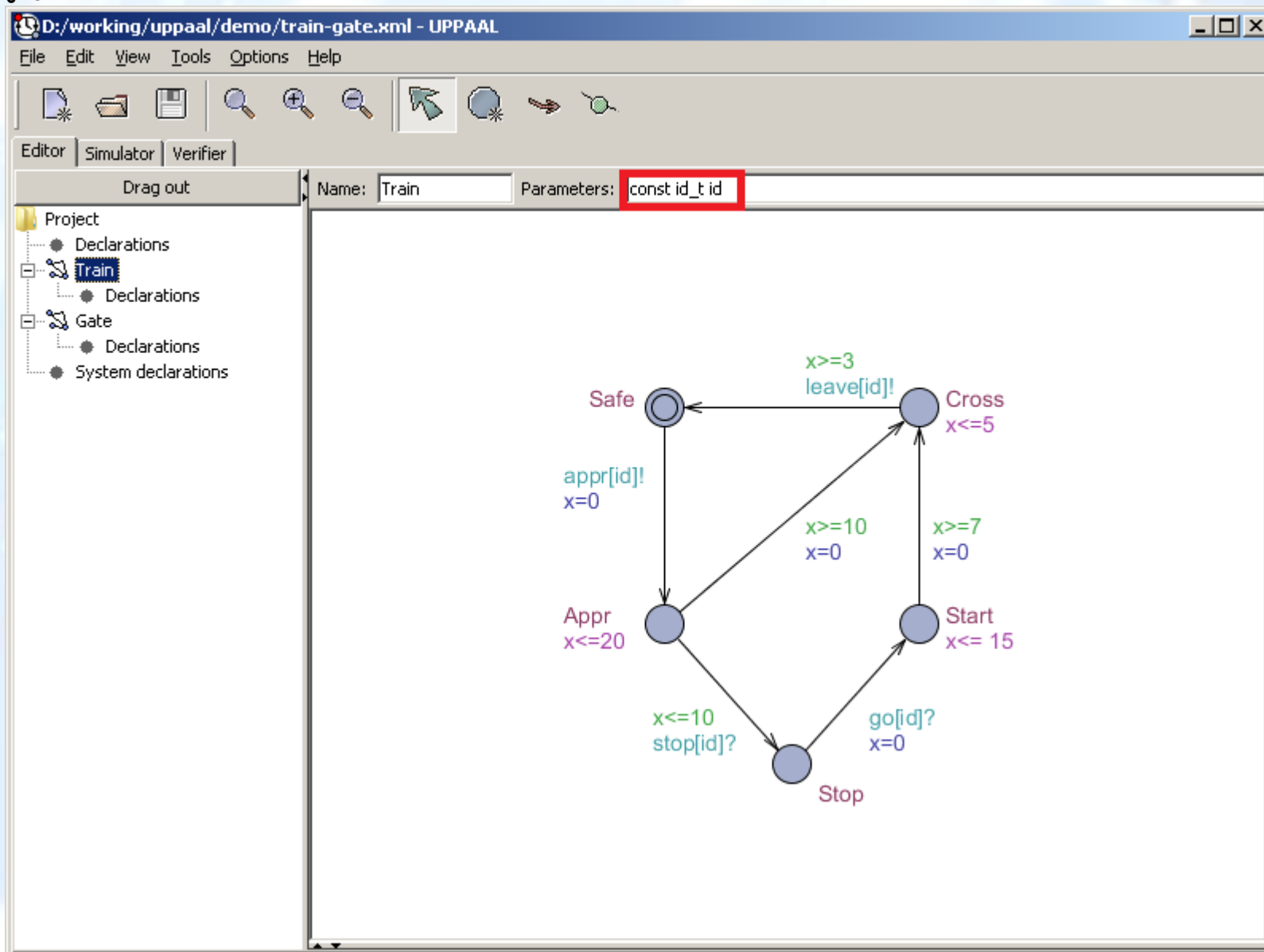


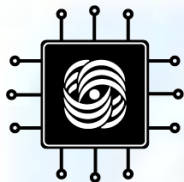
Присваивания



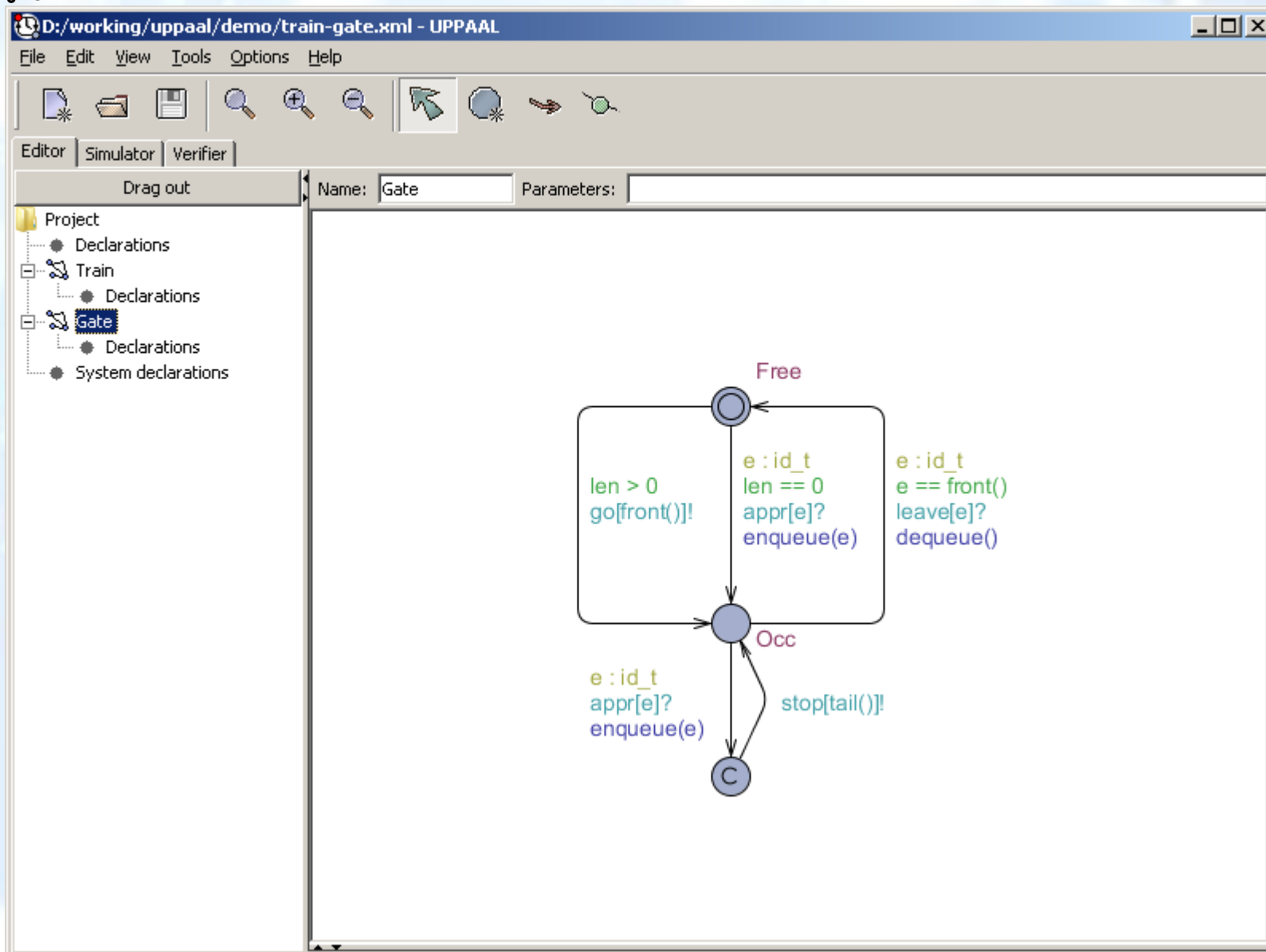


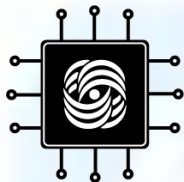
Параметры



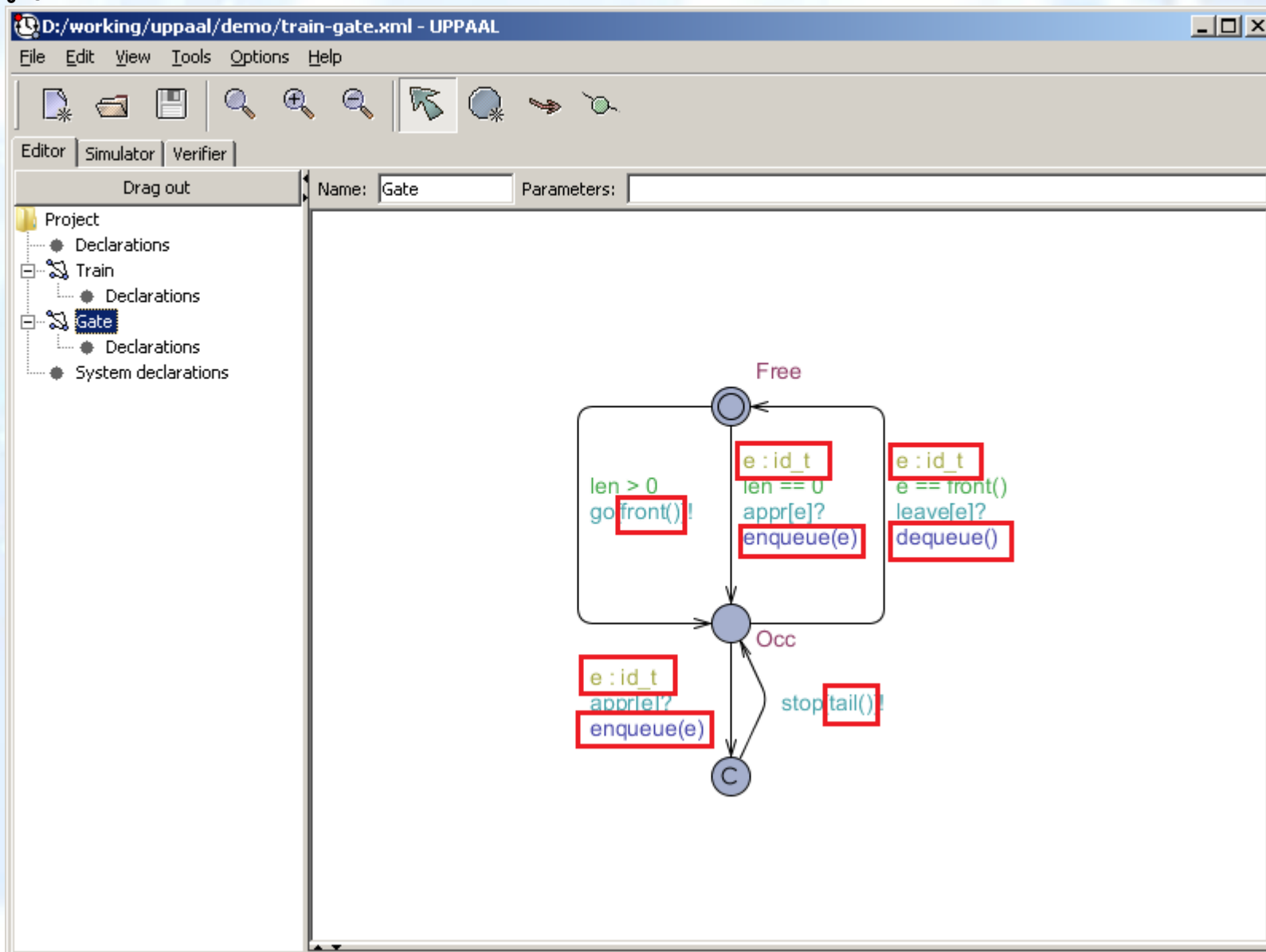


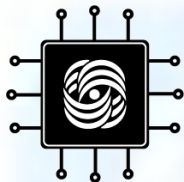
Модуль описания



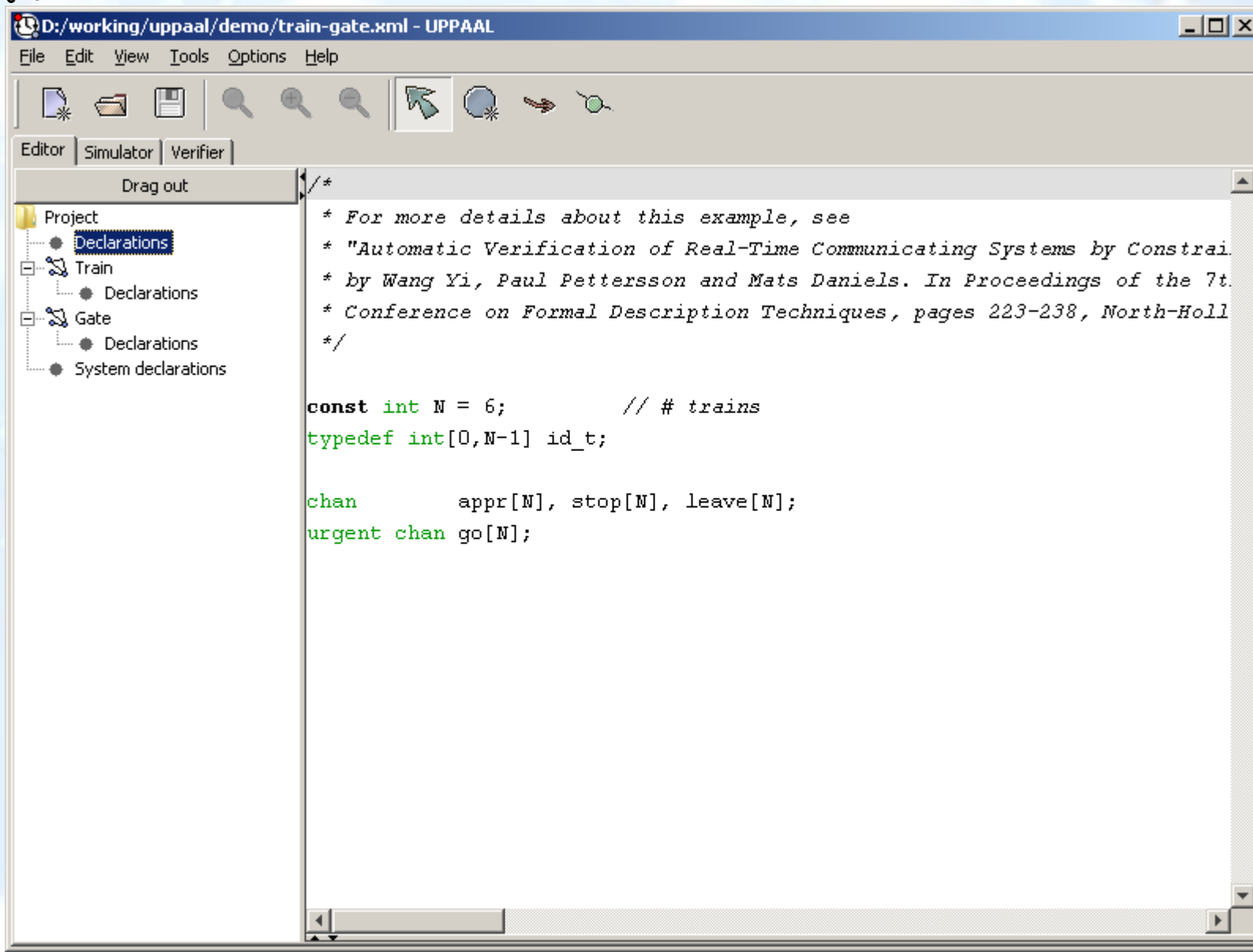


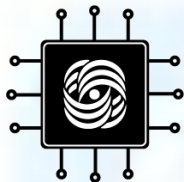
Дополнительные возможности



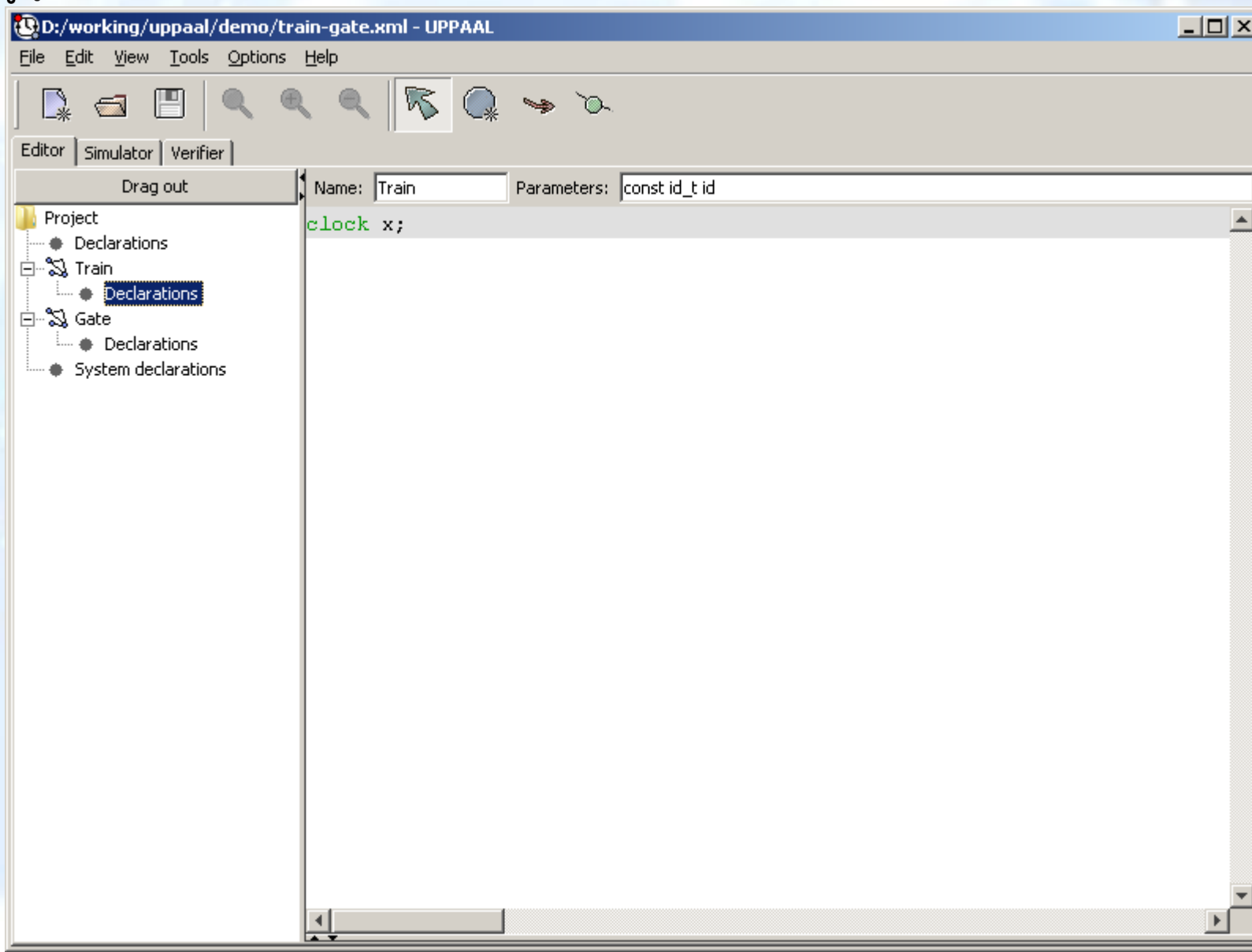


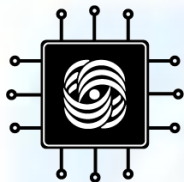
Глобальные описания





Локальные описания





Локальные описания

D:/working/uppaal/demo/train-gate.xml - UPPAAL

File Edit View Tools Options Help

Editor Simulator Verifier

Drag out

Project

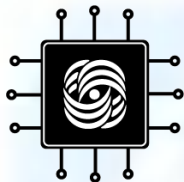
- Declarations
- Train
 - Declarations
- Gate
 - Declarations**
- System declarations

Name: Gate Parameters:

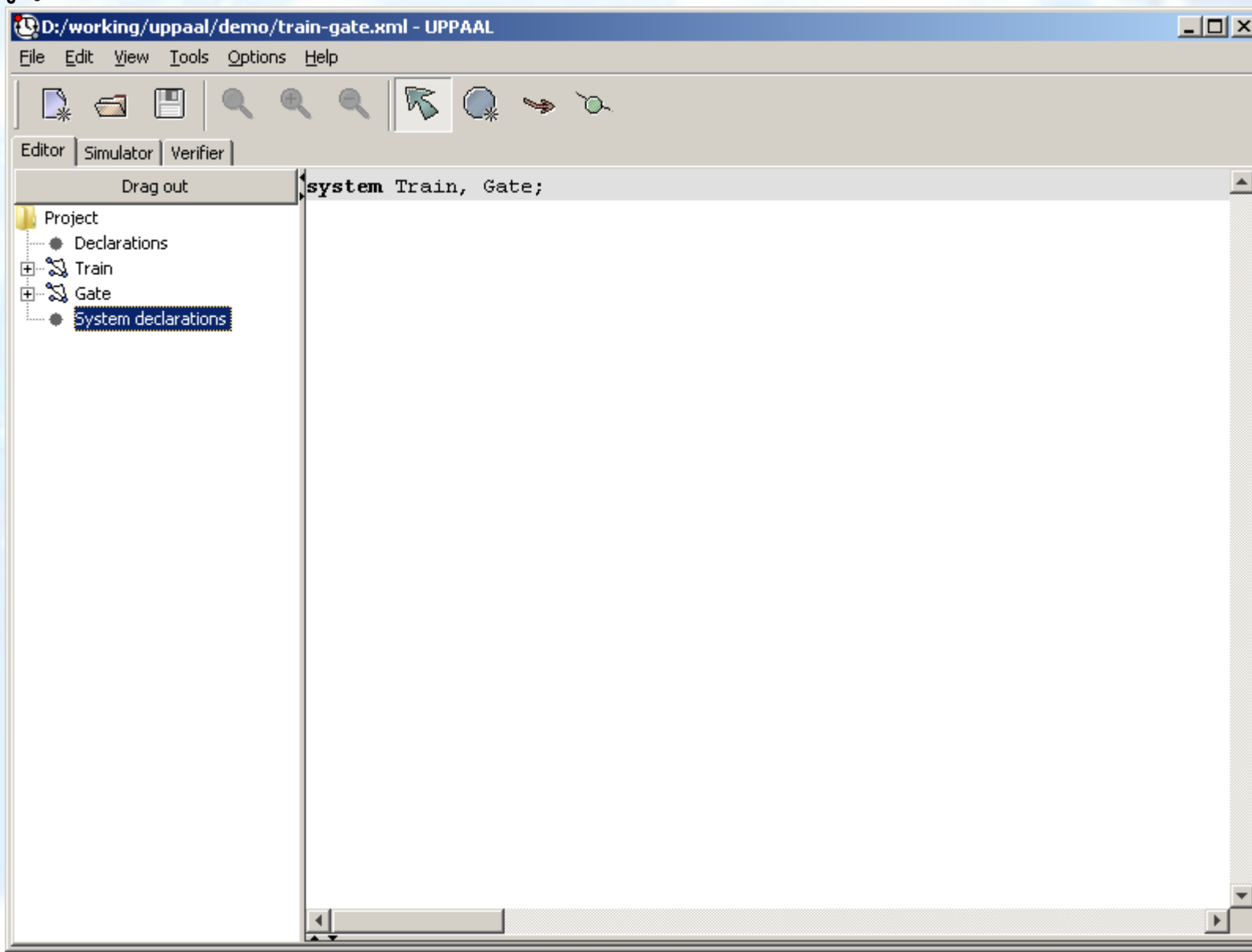
```
id_t list[N+1];
int[0,N] len;

// Put an element at the end of the queue
void enqueue(id_t element)
{
    list[len++] = element;
}

// Remove the front element of the queue
void dequeue()
{
    int i = 0;
    len -= 1;
    while (i < len)
    {
        list[i] = list[i + 1];
        i++;
    }
    list[i] = 0;
}
```



Описание системы

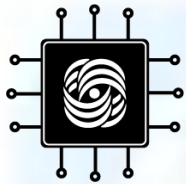




Синтаксис выражений

- Выражение (в стиле языка C)

```
Expr ::= ID | NAT |  
        Expr[Expr] | (Expr) |  
        Expr++ | ++Expr | Expr-- | --Expr |  
        Expr AOp Expr | UOp Expr | Expr BOp Expr |  
        Expr?Expr:Expr | Expr.ID  
UPop ::= - | ! | not  
BOp ::= < | <= | == | != | >= | > | + | - |  
        * | / | % | & | | | ^ | << | >> | && |  
        || | <? | >? | and | or | imply  
AOp ::= := | += | -= | *= | /= | %= | |= |  
        &= | ^= | <<= | >>=
```



Синтаксис выражений

Предусловие

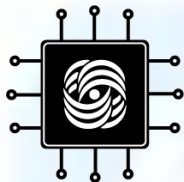
- Булево выражение
- Таймеры (t, t') могут встретиться только в
 $t \text{ rel } \text{Exp} \quad t - t' \text{ rel } \text{Exp}$
- Указанные выше сравнения должны идти в предусловии подряд через конъюнкцию

Присваивания

- Разделенные запятой выражения вида
 $x \text{ AOp } \text{Exp}$

Инвариант

- Булево выражение без отрицаний
- Таймеры (t) могут встретиться только в выражениях вида
 $t \text{ rel } \text{Exp}$



Модуль симуляции

D:\working\uppaal\demo\train-gate.xml - UPPAAL

File Edit View Tools Options Help

Editor Simulator Verifier

Drag out

Enabled Transitions

appr[0]: Train(0) --> Gate
appr[1]: Train(1) --> Gate
appr[2]: Train(2) --> Gate
appr[3]: Train(3) --> Gate
appr[4]: Train(4) --> Gate
appr[5]: Train(5) --> Gate

Next Reset

Simulation Trace

(Safe, Safe, Safe, Safe, Safe, Safe, Free)

Trace File:

Prev Next Replay
Open Save Auto

Slow Fast

Drag out

Gate.list[0] = 0
Gate.list[1] = 0
Gate.list[2] = 0
Gate.list[3] = 0
Gate.list[4] = 0
Gate.list[5] = 0
Gate.list[6] = 0
Gate.len = 0
Train(0).x >= 0
Train(1).x >= 0
Train(2).x >= 0
Train(3).x >= 0
Train(4).x >= 0
Train(5).x >= 0
Train(0).x = Train(1).x
Train(1).x = Train(2).x
Train(2).x = Train(3).x
Train(3).x = Train(4).x
Train(4).x = Train(5).x
Train(5).x = Train(0).x

Train(0)

Safe (red circle) → Cross (blue circle) → Stop (blue circle) → Appr (blue circle) → Safe (red circle)

Transitions for Train(0):
Safe → Cross: x >= 3, leave[0]!
Cross → Stop: x <= 5
Stop → Appr: go[0]?, x = 0
Appr → Safe: x <= 10, stop[0]?
Appr → Cross: x >= 10, x = 0
Cross → Appr: x >= 7, x = 0

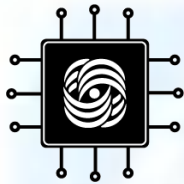
Train(1)

Safe (red circle) → Cross (blue circle) → Stop (blue circle) → Appr (blue circle) → Safe (red circle)

Transitions for Train(1):
Safe → Cross: x >= 3, leave[1]!
Cross → Stop: x <= 5

Train(0) Train(1) Train(2) Train(3) Train(4) Train(5)

Safe Safe Safe Safe Safe Safe



Визуализация активных состояний

D:/working/uppaal/demo/train-gate.xml - UPPAAL

File Edit View Tools Options Help

Editor Simulator Verifier

Drag out

Enabled Transitions

- appr[0]: Train(0) --> Gate
- appr[1]: Train(1) --> Gate
- appr[2]: Train(2) --> Gate
- appr[3]: Train(3) --> Gate
- appr[4]: Train(4) --> Gate

Next Reset

Simulation Trace

(Safe, Safe, Safe, Safe, Safe, Safe, Free)

Trace File:

Prev Next Replay

Open Save Auto

Slow Fast

Drag out

```
Gate.list[0] = 0
Gate.list[1] = 0
Gate.list[2] = 0
Gate.list[3] = 0
Gate.list[4] = 0
Gate.list[5] = 0
Gate.list[6] = 0
Gate.len = 0
Train(0).x >= 0
Train(1).x >= 0
Train(2).x >= 0
Train(3).x >= 0
Train(4).x >= 0
Train(5).x >= 0
Train(0).x = Train(1).x
Train(1).x = Train(2).x
Train(2).x = Train(3).x
Train(3).x = Train(4).x
Train(4).x = Train(5).x
Train(5).x = Train(0).x
```

Safe

Appr
x <= 20

Cross
x <= 5

Start
x <= 15

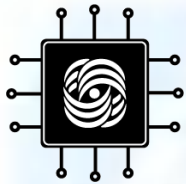
Stop

Gate

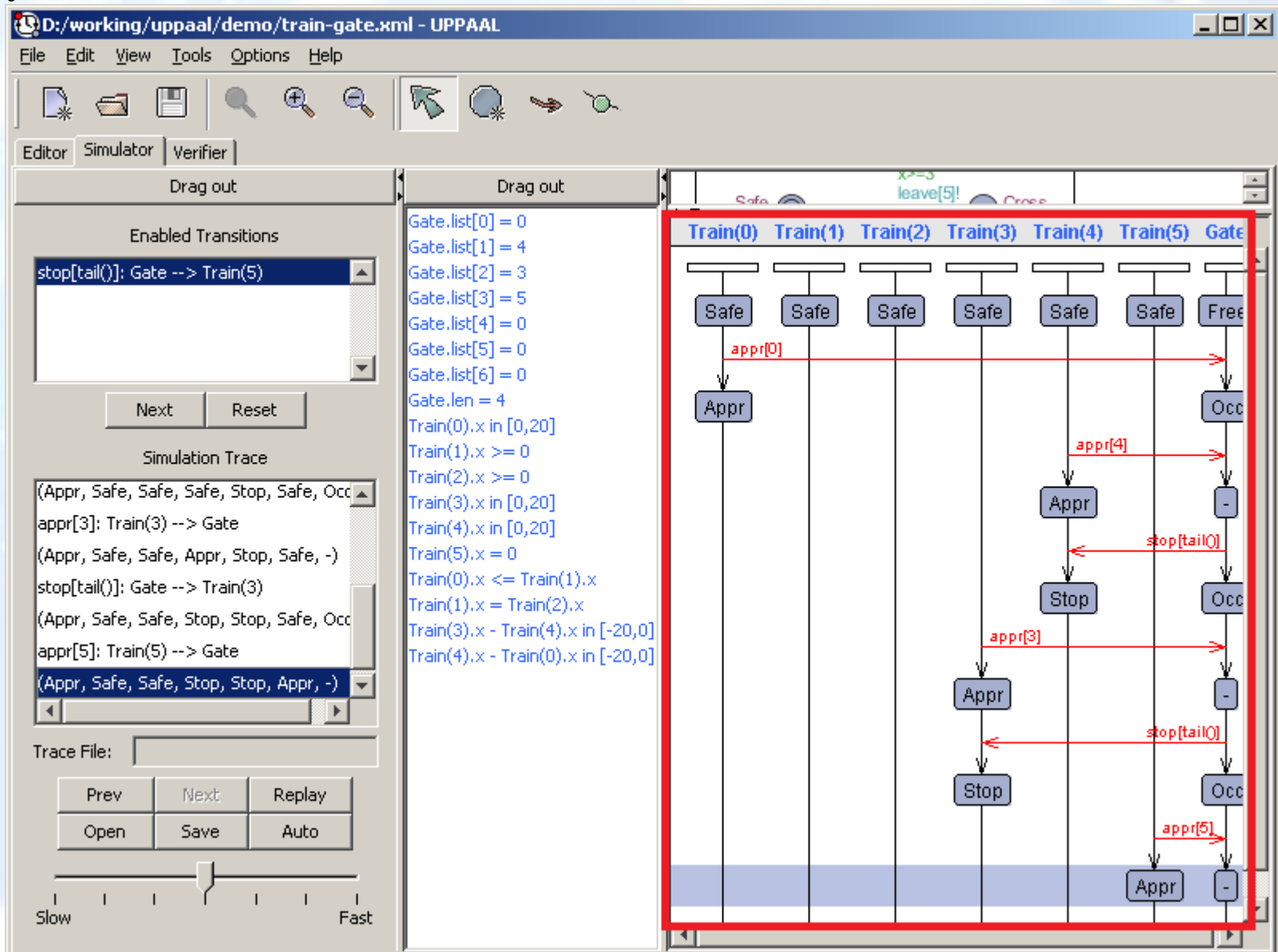
Free

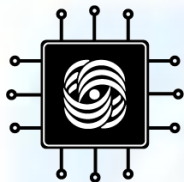
Occ

Train(0) Train(1) Train(2) Train(3) Train(4) Train(5)



Визуализация трассы





Окно текущих значений данных

D:\working\uppaal\demo\train-gate.xml - UPPAAL

File Edit View Tools Options Help

Editor Simulator Verifier

Drag out

Enabled Transitions

stop[tail()]: Gate --> Train(5)

Next Reset

Simulation Trace

(Appr, Safe, Safe, Safe, Stop, Safe, Occ
appr[3]: Train(3) --> Gate
(Appr, Safe, Safe, Appr, Stop, Safe, -)
stop[tail()]: Gate --> Train(3)
(Appr, Safe, Safe, Stop, Stop, Safe, Occ
appr[5]: Train(5) --> Gate
(Appr, Safe, Safe, Stop, Stop, Appr, -)

Trace File:

Prev Next Replay
Open Save Auto

Slow Fast

Drag out

Gate.list[0] = 0
Gate.list[1] = 4
Gate.list[2] = 3
Gate.list[3] = 5
Gate.list[4] = 0
Gate.list[5] = 0
Gate.list[6] = 0
Gate.len = 4
Train(0).x in [0,20]
Train(1).x >= 0
Train(2).x >= 0
Train(3).x in [0,20]
Train(4).x in [0,20]
Train(5).x = 0
Train(0).x <= Train(1).x
Train(1).x = Train(2).x
Train(3).x - Train(4).x in [-20,0]
Train(4).x - Train(0).x in [-20,0]

Train(0) Train(1) Train(2) Train(3) Train(4) Train(5) Gate

Safe Safe Safe Safe Safe Safe Free

appr[0]

Appr

appr[4]

Appr

stop[tail()

Occ

appr[3]

Appr

stop[tail()

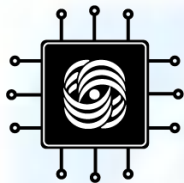
Stop

Occ

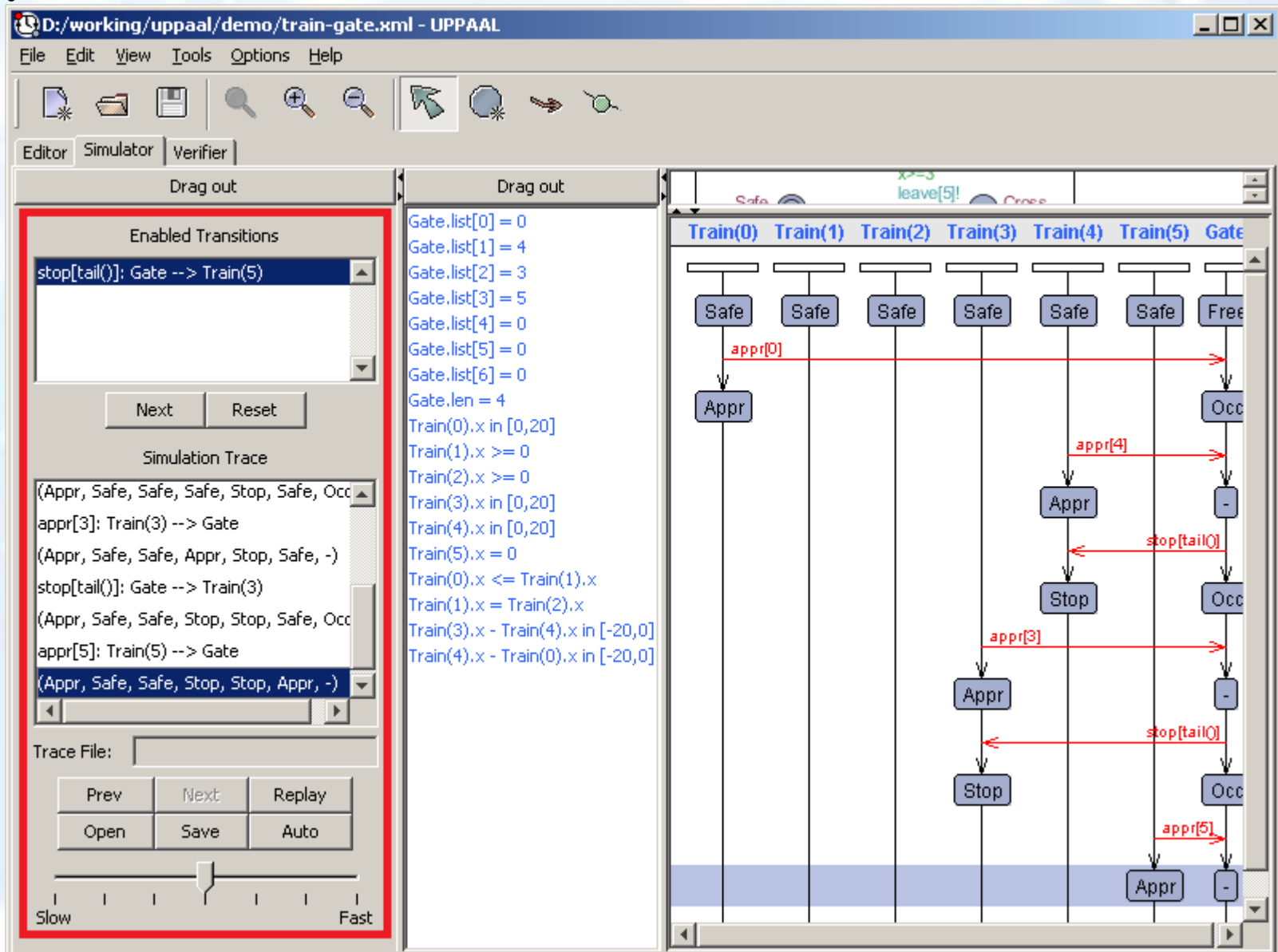
appr[5]

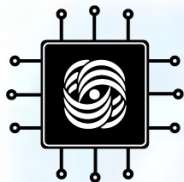
Appr

Occ



Окно генерации трассы





Модуль верификации

D:\working\uppaal\demo\train-gate.xml - UPPAAL

File Edit View Tools Options Help

Editor Simulator Verifier

Overview

A[] Gate.Free	●
E<> Gate.Occ	●
E<> Train(0).Cross	●
E<> Train(1).Cross	●

Check
Insert
Remove
Comments

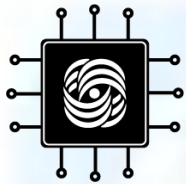
Query

A[] Gate.Free

Comment

Status

Property is satisfied.
Train(4).Appr --> Train(4).Cross
Property is satisfied.
Train(5).Appr --> Train(5).Cross
Property is satisfied.
A[] not deadlock
Property is satisfied.
A[] Gate.Free
Property is not satisfied.



Окно результатов проверки

D:/working/uppaal/demo/train-gate.xml - UPPAAL

File Edit View Tools Options Help

Editor Simulator Verifier

Overview

A[] Gate.Free	Red light	Check Insert Remove Comments
E<> Gate.Occ	Grey light	
E<> Train(0).Cross	Green light	
E<> Train(1).Cross	Green light	

Query

A[] Gate.Free

Comment

Status

Property is satisfied.

Train(4).Appr --> Train(4).Cross

Property is satisfied.

Train(5).Appr --> Train(5).Cross

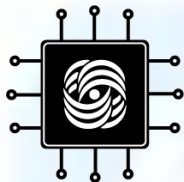
Property is satisfied.

A[] not deadlock

Property is satisfied.

A[] Gate.Free

Property is not satisfied.



Окно текущего запроса

D:/working/uppaal/demo/train-gate.xml - UPPAAL

File Edit View Tools Options Help

Editor Simulator Verifier

Overview

Property	Status
A[] Gate.Free	Not Satisfied
E<> Gate.Occ	Satisfied
E<> Train(0).Cross	Satisfied
E<> Train(1).Cross	Satisfied

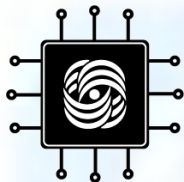
Query

A[] Gate.Free

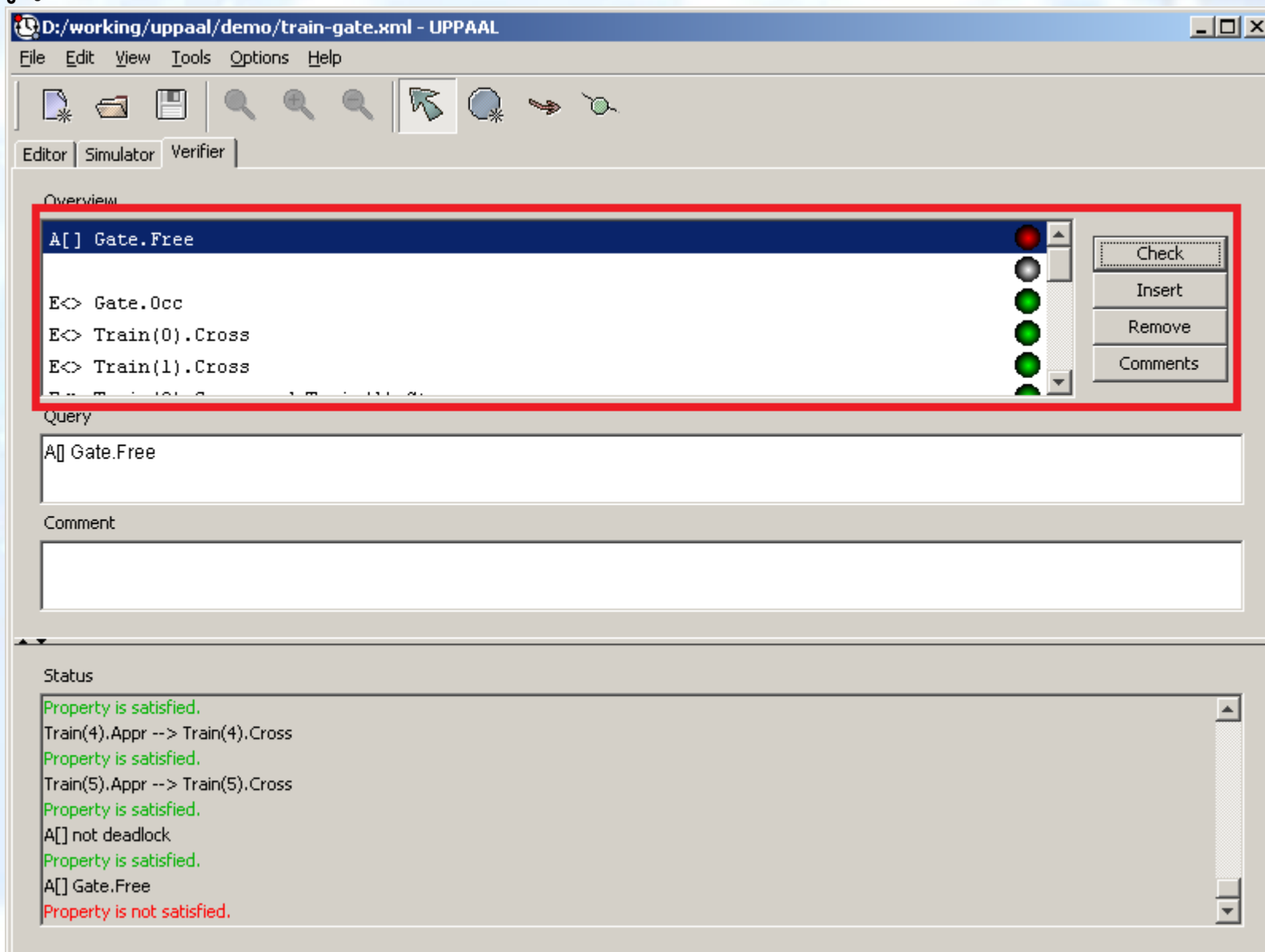
Comment

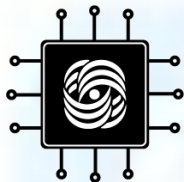
Status

Property is satisfied.
Train(4).Appr --> Train(4).Cross
Property is satisfied.
Train(5).Appr --> Train(5).Cross
Property is satisfied.
A[] not deadlock
Property is satisfied.
A[] Gate.Free
Property is not satisfied.



Окно списка запросов





Примеры запросов

D:/working/uppaal/demo/train-gate.xml - UPPAAL

File Edit View Tools Options Help

Editor Simulator Verifier

Overview

```
E<> Train(0).Cross and Train(1).Stop
E<> Train(0).Cross and (forall (i : id_t) i != 0 imply Train(i).Stop)

A[] forall (i : id_t) forall (j : id_t) Train(i).Cross && Train(j).Cross imply i == j
A[] Gate.list[N] == 0
```

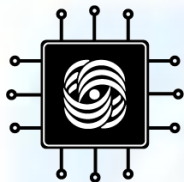
Query

A[] Gate.Free

Comment

Status

```
Property is satisfied.
Train(4).Appr --> Train(4).Cross
Property is satisfied.
Train(5).Appr --> Train(5).Cross
Property is satisfied.
A[] not deadlock
Property is satisfied.
A[] Gate.Free
Property is not satisfied.
```

Примеры запросов

D:/working/uppaal/demo/train-gate.xml - UPPAAL

File Edit View Tools Options Help

Editor Simulator Verifier

Overview

Train(0).Appr --> Train(0).Cross	●
Train(1).Appr --> Train(1).Cross	●
Train(2).Appr --> Train(2).Cross	●
Train(3).Appr --> Train(3).Cross	●
Train(4).Appr --> Train(4).Cross	●
Train(5).Appr --> Train(5).Cross	●

Check
Insert
Remove
Comments

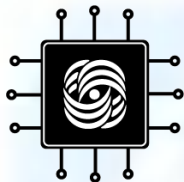
Query

A[] Gate.Free

Comment

Status

Property is satisfied.
Train(4).Appr --> Train(4).Cross
Property is satisfied.
Train(5).Appr --> Train(5).Cross
Property is satisfied.
A[] not deadlock
Property is satisfied.
A[] Gate.Free
Property is not satisfied.



Примеры запросов

D:/working/uppaal/demo/train-gate.xml - UPPAAL

File Edit View Tools Options Help

Editor Simulator Verifier

Overview

Train(2).Appr --> Train(2).Cross	●
Train(3).Appr --> Train(3).Cross	●
Train(4).Appr --> Train(4).Cross	●
Train(5).Appr --> Train(5).Cross	●
A[] not deadlock	●

Check
Insert
Remove
Comments

Query

A[] not deadlock

Comment

The system is deadlock-free.

Status

Property is satisfied.
Train(4).Appr --> Train(4).Cross
Property is satisfied.
Train(5).Appr --> Train(5).Cross
Property is satisfied.
A[] not deadlock
Property is satisfied.
A[] Gate.Free
Property is not satisfied.



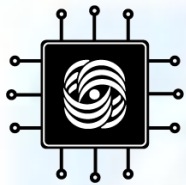
Язык запросов

- Допустимая формула языка запросов

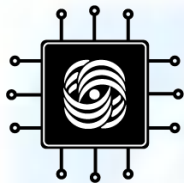
$\Phi ::= A[] \varphi \mid A\langle\rangle \varphi \mid E[] \varphi \mid E\langle\rangle \varphi$
 $\mid \varphi \text{ --> } \varphi$

$\varphi ::= \text{Exp} \mid A.l \mid \text{deadlock}$
 $\mid \varphi \text{ or } \varphi \mid \varphi \text{ and } \varphi \mid \text{not } \varphi \mid (\varphi)$

- `deadlock` можно использовать только с `A[]`, `E<>`
- Семантика определяется на основе семантики формул логики CTL
 - $A.l \Leftrightarrow$ автомат A находится в состоянии l
 - $\text{deadlock} \Leftrightarrow$ вычисление после достижения текущей конфигурации невозможно продолжить
 - $\varphi_1 \text{ --> } \varphi_2 \Leftrightarrow \text{AG}(\neg\varphi_1 \vee \text{AF}(\varphi_2))$
 - Выполнимость выражений определяется естественным образом



Спасибо за внимание



НАДЁЖНОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

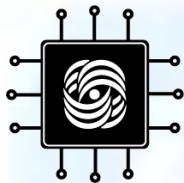
Лекция 7: *Статический анализ*

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



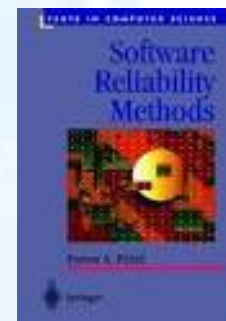
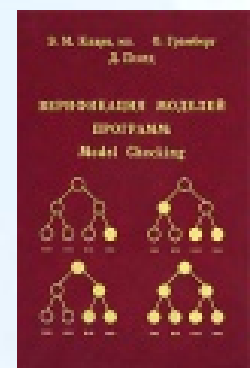
План лекции

- Статический анализ
 - Модели программ
 - Алгоритмы статического анализа
 - Обнаружение дефектов
- Другие статические методы
- Заключение



Литература

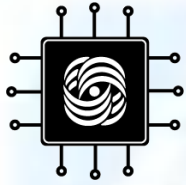
- Карпов Ю.Г. Model Checking. Верификация параллельных и распределенных программных систем, БХВ-Петербург, 2010.
- Кларк, Грумберг, Пелед. Верификация моделей программ: Model checking, МЦНМО, 2002.
- Peled. Software Reliability Methods, Springer, 2001.





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*



Статические методы обеспечения качества

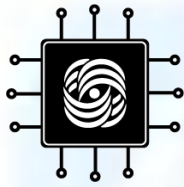
5

- Формальные инспекции, аудит
- Формальная верификация
 - Дедуктивная верификация
 - Model checking
- **Статический анализ**
- Трансформации
 - Рефакторинги
 - Модификации



Статический анализ

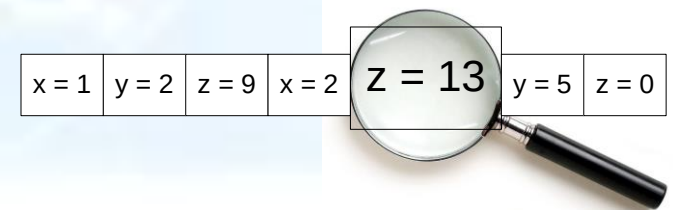
- Использует исходный код ПО для анализа
- Применяется для
 - Форматирования программ
 - Вычисления программных метрик
 - Оптимизации программ
 - Распараллеливания программ
 - Преобразования программ
 - Обфускации программ
 - Деобфускации программ
 - **Обнаружения дефектов**
 - ...

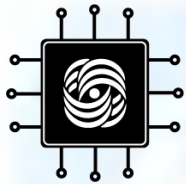


Статический анализ для обнаружения дефектов

7

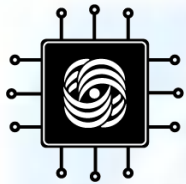
- Использует исходный код ПО ВС для анализа
- Позволяет проанализировать все возможные трассы исполнения
- Позволяет проанализировать все наборы входных данных
- Может быть полностью автоматизирован
- Позволяет обнаружить *нефункциональные* дефекты





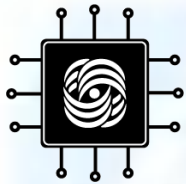
Нефункциональные дефекты

- Ошибки этапа кодирования
- Не являются явным нарушением спецификации
- Являются следствием
 - сложности языков программирования
 - сложности библиотек
 - сложности алгоритмов и структур данных



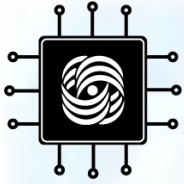
Нефункциональные дефекты

- Выход за границу массива
- Переполнение буфера
- Использование неинициализированных объектов
- Работа с некорректным указателем
- Арифметические ошибки
- Утечки ресурсов
- Ошибки форматных строк
- Ошибки работы с ресурсами
- и т.д.



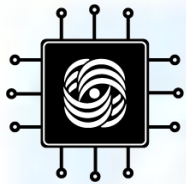
Проявление дефектов

- Без проявления
- Зависания
- Сбои
- Падения
- Деградация производительности
- ...



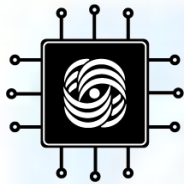
Опасные конструкции

- Язык С
 - Макроопределения
 - Указатели
 - Массивы
 - Адресная арифметика
 - Указатели на функции
 - Приведения типов
 - Объединения (union)



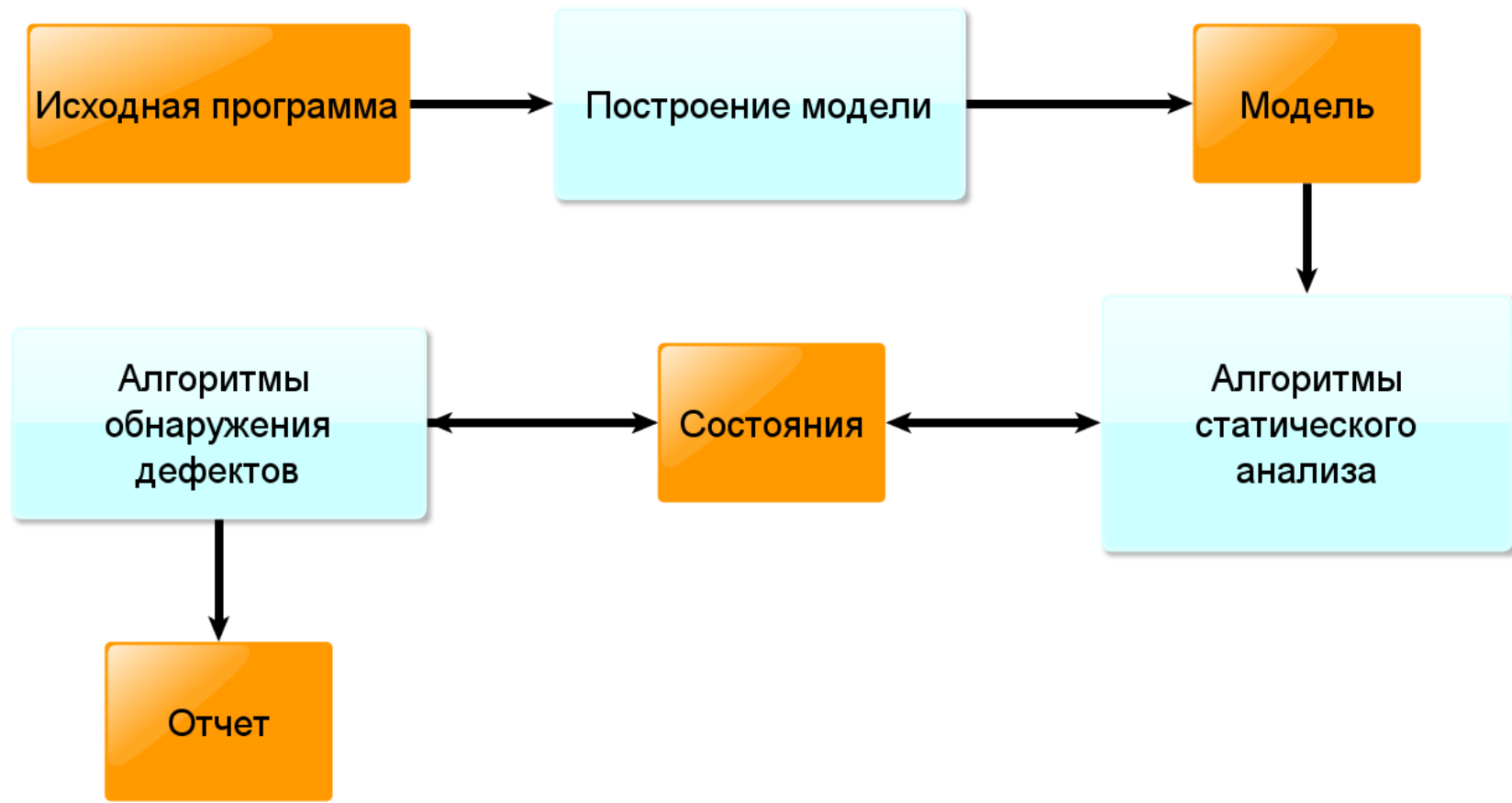
Характеристики анализа

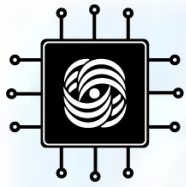
- Полнота – доля найденных дефектов по отношению ко всем дефектам программы
- Точность – доля найденных истинных дефектов по отношению ко всем найденным
- Ресурсоемкость



Общая схема обнаружения дефектов

13

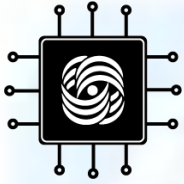




Модели программ. Требования

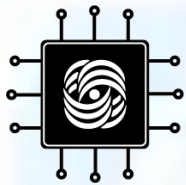
14

- Полнота представления объектов программы
- Восстанавливаемость
 - Идеально: возможность полного восстановления исходного кода
 - Желательно: связь с исходным кодом
- Эффективность поиска объектов



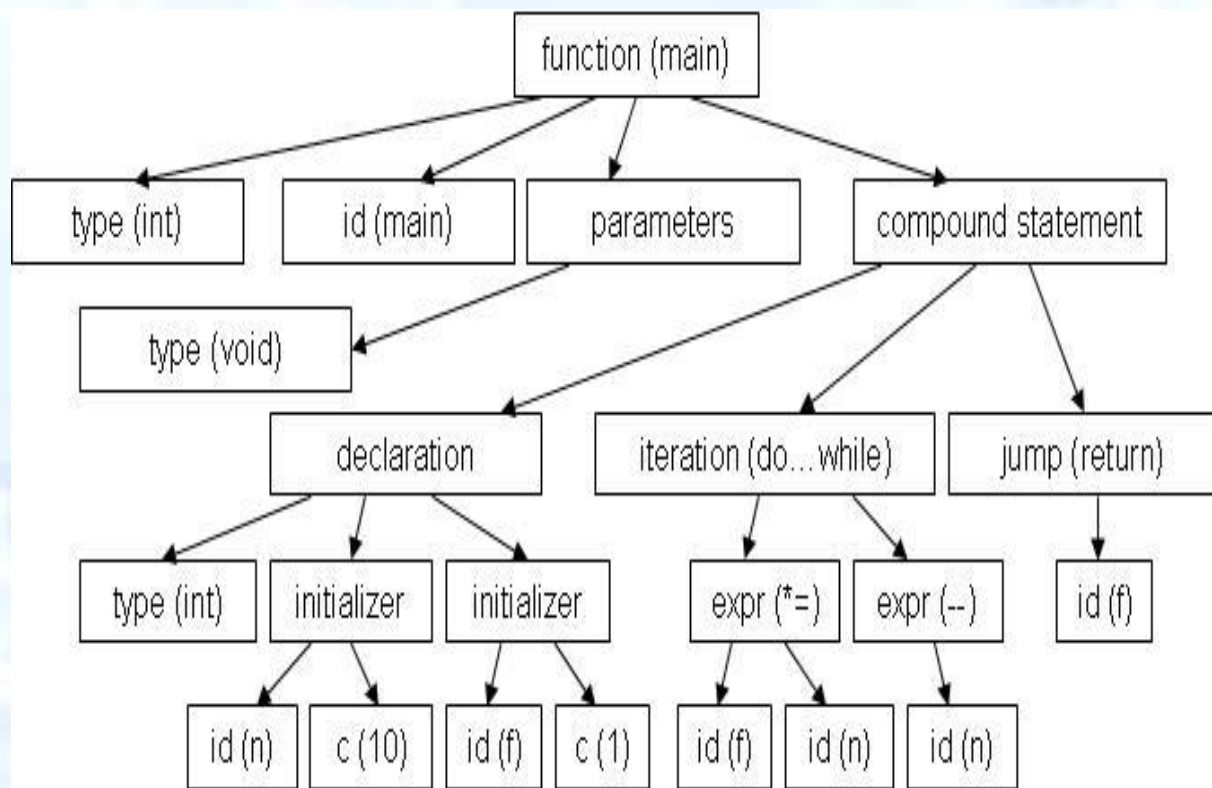
Модели программ. Типы

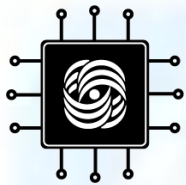
- Структурные
 - Синтаксическое дерево разбора
 - Абстрактное синтаксическое дерево (AST – abstract syntax tree)
- Поведенческие
 - Граф потока управления (CFG – control flow graph)
 - Граф зависимостей по данным (DDG – data dependence graph)
 - SSA - static single assignment
- Гибридные
 - Абстрактный семантический граф (ASG - abstract semantic graph)



Модели программ: AST

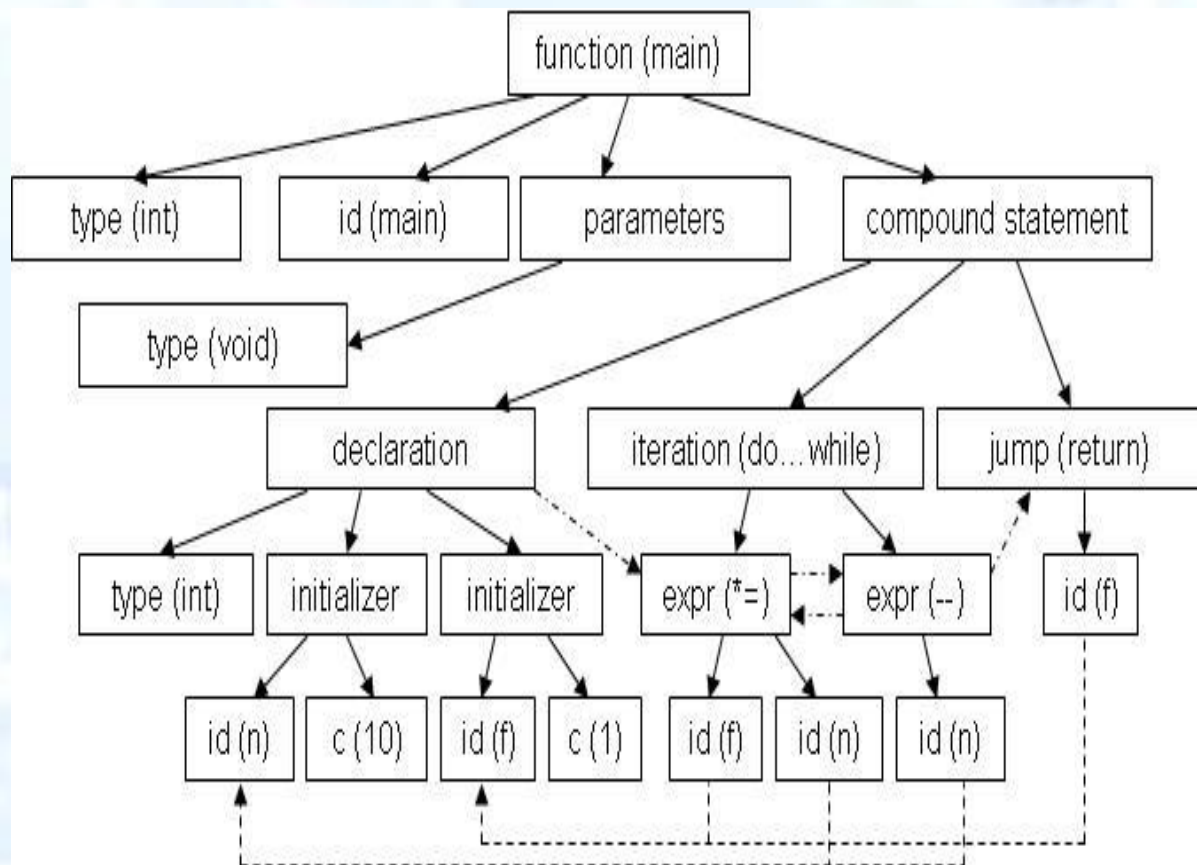
```
int main()  
{  
    int n = 10;  
    int f = 1;  
    while (n > 0)  
    {  
        f *= n;  
        --n;  
    }  
    return f;  
}
```

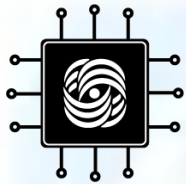




Модели программ: ASG

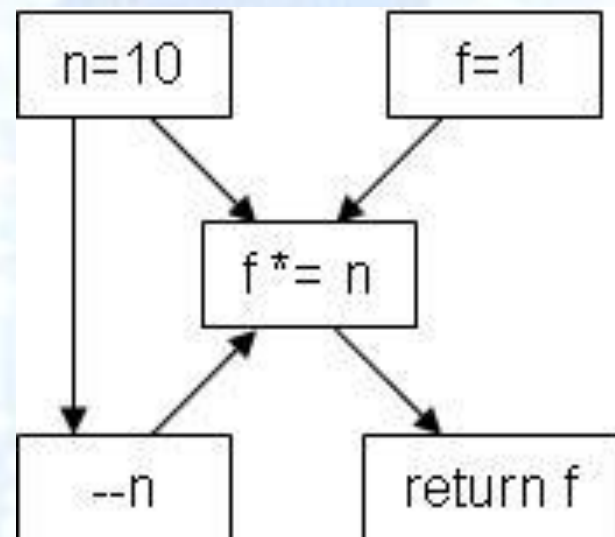
```
int main()  
{  
    int n = 10;  
    int f = 1;  
    while (n > 0)  
    {  
        f *= n;  
        --n;  
    }  
    return f;  
}
```

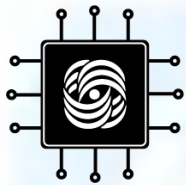




Модели программ: DDG

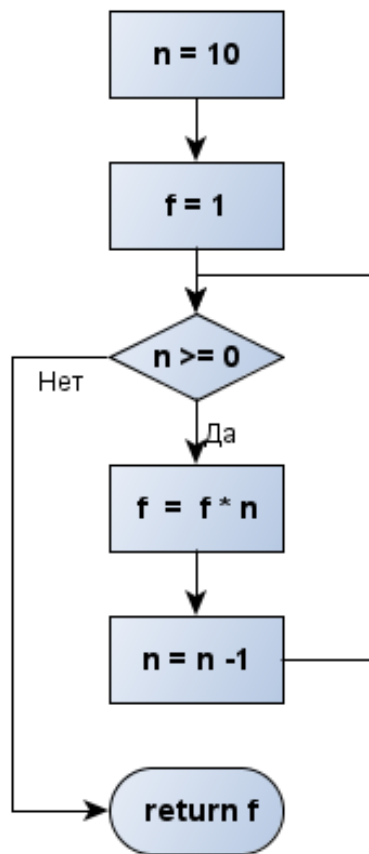
```
int main()  
{  
    int n = 10;  
    int f = 1;  
    while (n >  
        0)  
    {  
        f *= n;  
        --n;  
    }  
    return f;  
}
```

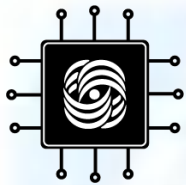




Модели программ: CFG

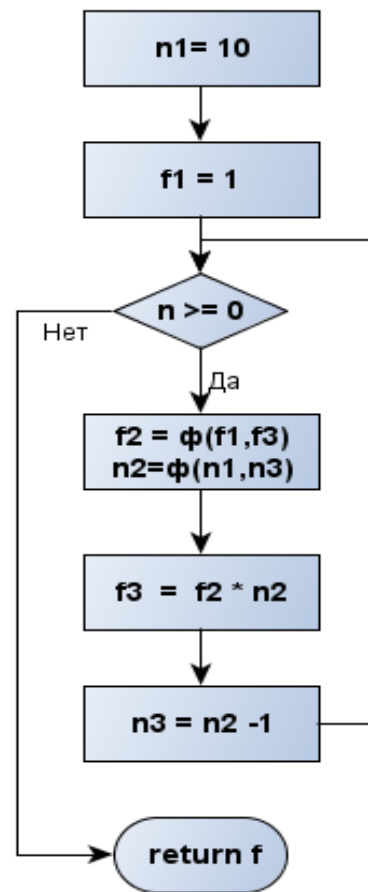
```
int main()  
{  
    int n = 10;  
    int f = 1;  
    while (n >  
        0)  
    {  
        f *= n;  
        --n;  
    }  
    return f;  
}
```

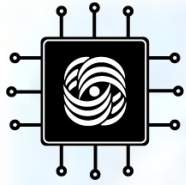




Модели программ: SSA

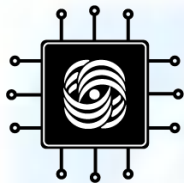
```
int main()  
{  
    int n = 10;  
    int f = 1;  
    while (n >  
        0)  
    {  
        f *= n;  
        --n;  
    }  
    return f;  
}
```





Модели программ. Резюме

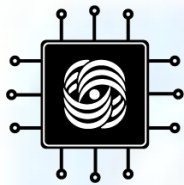
- Для простейшего статического анализа достаточно структурных моделей (AST)
- Для обнаружения дефектов необходим учет семантики:
 - ASG
 - CFG
 - CFG в форме SSA



Модели программ. Построение

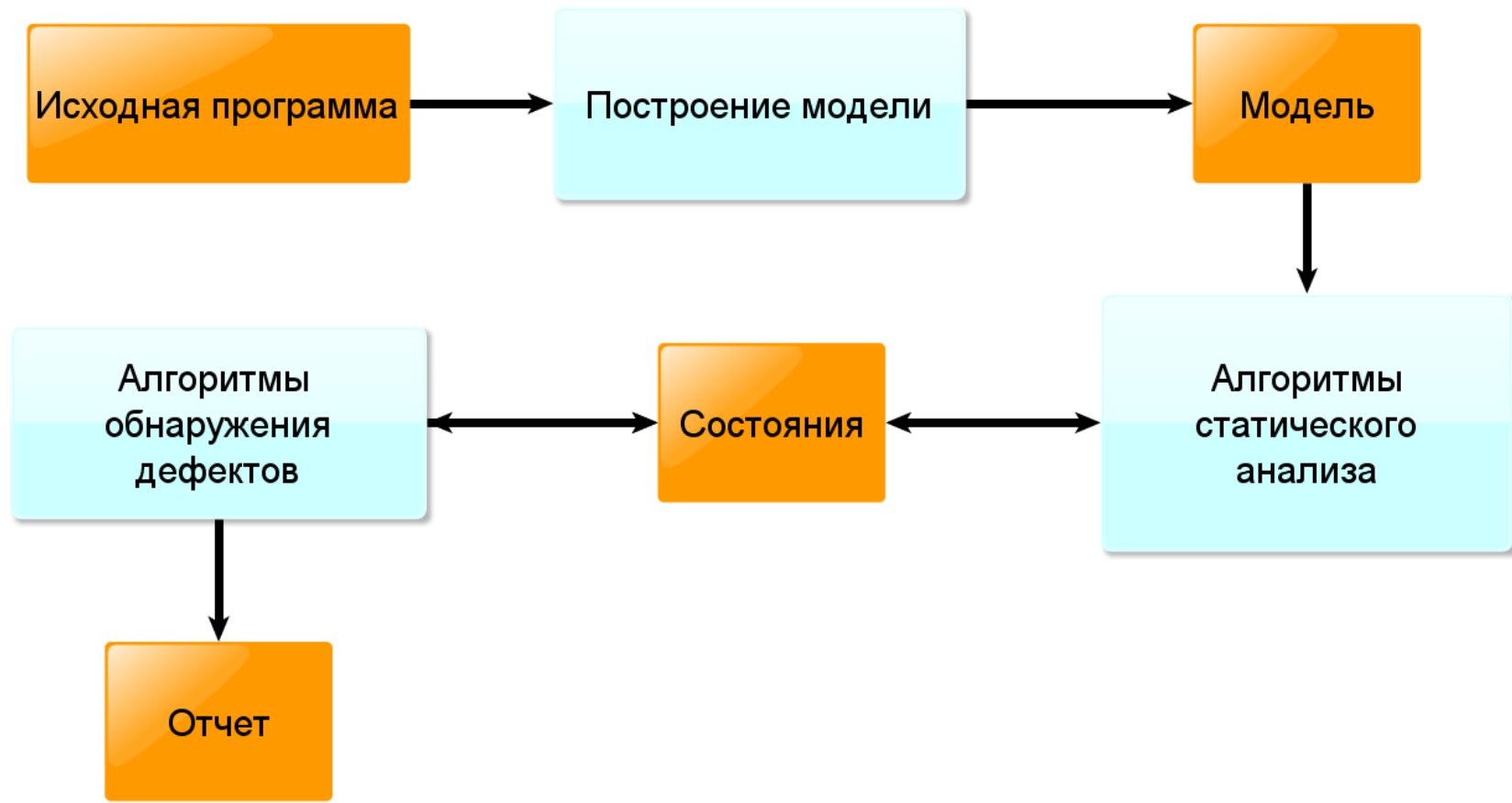
22

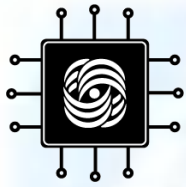
- Построение парсера вручную
- Использование генераторов парсеров (ANTLR, YACC, etc)
- Использование самостоятельных парсеров (Elsa)
- Использование парсеров в составе средств разработки (NetBeans, Eclipse CDT)
- Использование Front-End компиляторов (gcc)
- Использование фреймворков для статического анализа (SUIF, CIL, LLVM, etc)



Общая схема обнаружения дефектов

23

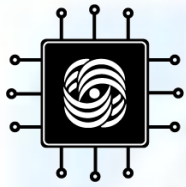




Анализ на основе состояний программы

24

- Состояние программы – совокупность состояний всех объектов программы в какой-либо точке
- Цель – вычислить все возможные состояния программы во всех точках
- Обнаружение дефектов происходит на основе проверки вычисленных состояний



Алгоритмы статического анализа

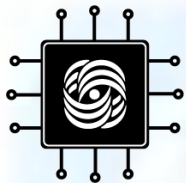
25

- По типу алгоритму
 - Интервальный анализ
 - Анализ указателей
 - Ресурсный анализ
- По масштабу
 - Внутрипроцедурный
 - Межпроцедурный
- По консервативности
 - Пессимистический (консервативный) анализ
 - Оптимистический анализ



Интервальный анализ

- Состояние переменных представляются интервальными значениями
 - $a = [10; 20]$
 - $b = (-40; 20) \cup (30; 100)$
 - $c = [0; \infty)$
- Вычисление состояния после выполнения оператора языка производится на основе интервальной арифметики
- Каждому оператору ставится в соответствие уравнение, связывающее состояние до выполнения и после



Интервальный анализ

Элемент модели

Интервальное правило

int a;

$$S^l = S^{l-1} \cup \langle a, \text{noninit} \rangle$$

a = C;

$$S^l = S^{l-1} \setminus U_i \langle a, i \rangle \cup \langle a, C \rangle$$

a = b;

$$S^l = S^{l-1} \setminus U_i \langle a, i \rangle \cup U_j \langle a, b_j \rangle$$

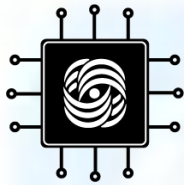
$\varphi(\dots)$

$$S^l = U_i S^i$$

if(cond) S^{true} else S^{false}

$$S^{\text{true}} = S^{l-1} \cap \text{cond}$$

$$S^{\text{false}} = S^{l-1} \cap \neg \text{cond}$$



Интервальный анализ.

Пример

28

```
int a;  
a=10;  
int b = f (...);  
if (b < 100) {  
    ...  
    a = a + b;  
}  
else{  
    ...  
    a = b / 2;  
}
```

$\langle a, \text{noninit} \rangle$

$\langle a, [10:10] \rangle$

$\langle a, [10:10] \rangle, \langle b, (-\infty; \infty) \rangle$

$\langle a, [10:10] \rangle, \langle b, (-\infty; 100) \rangle$

$\langle a, [-\infty+10:110] \rangle, \langle b, (-\infty; 100) \rangle$

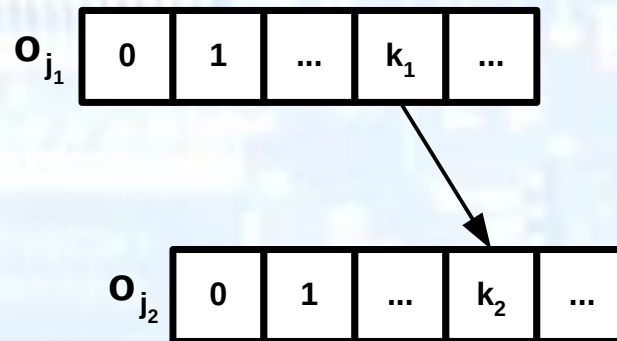
$\langle a, [10:10] \rangle, \langle b, [100; \infty) \rangle$

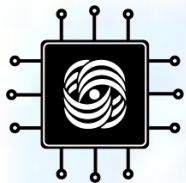
$\langle a, [50: \infty/2] \rangle, \langle b, [100; \infty) \rangle$



Анализ указателей

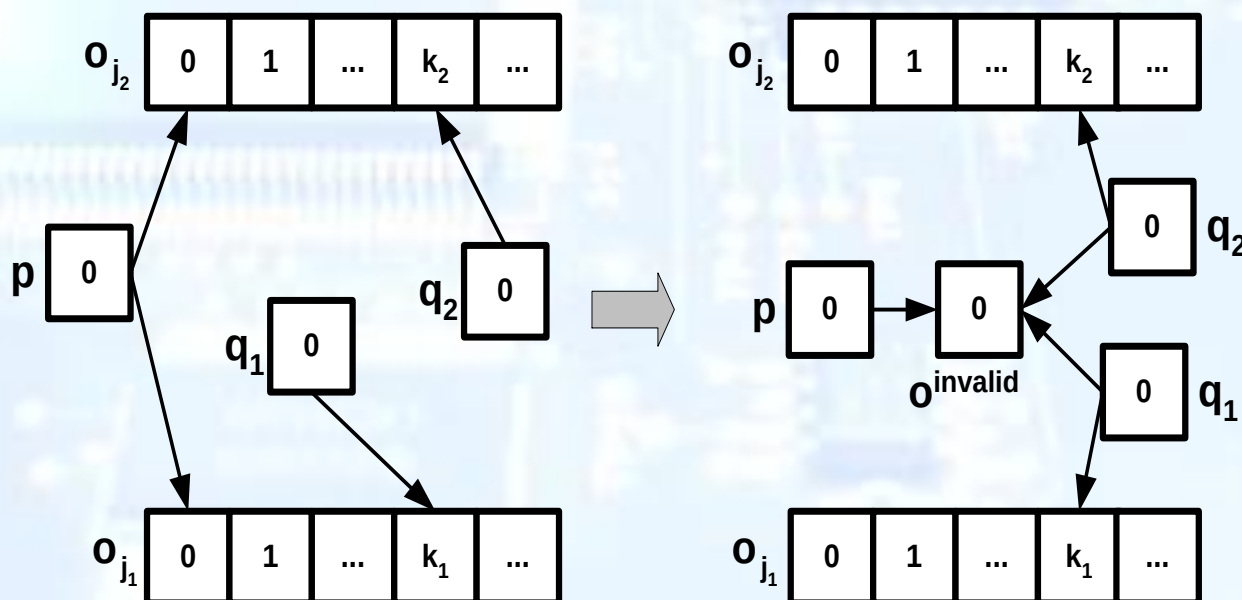
- Отличие от интервального анализа:
 - хранятся не интервалы значений, а списки связи “points-to”
 - значения одних объектов могут влиять на другие

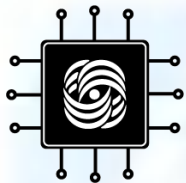




Анализ указателей. Пример

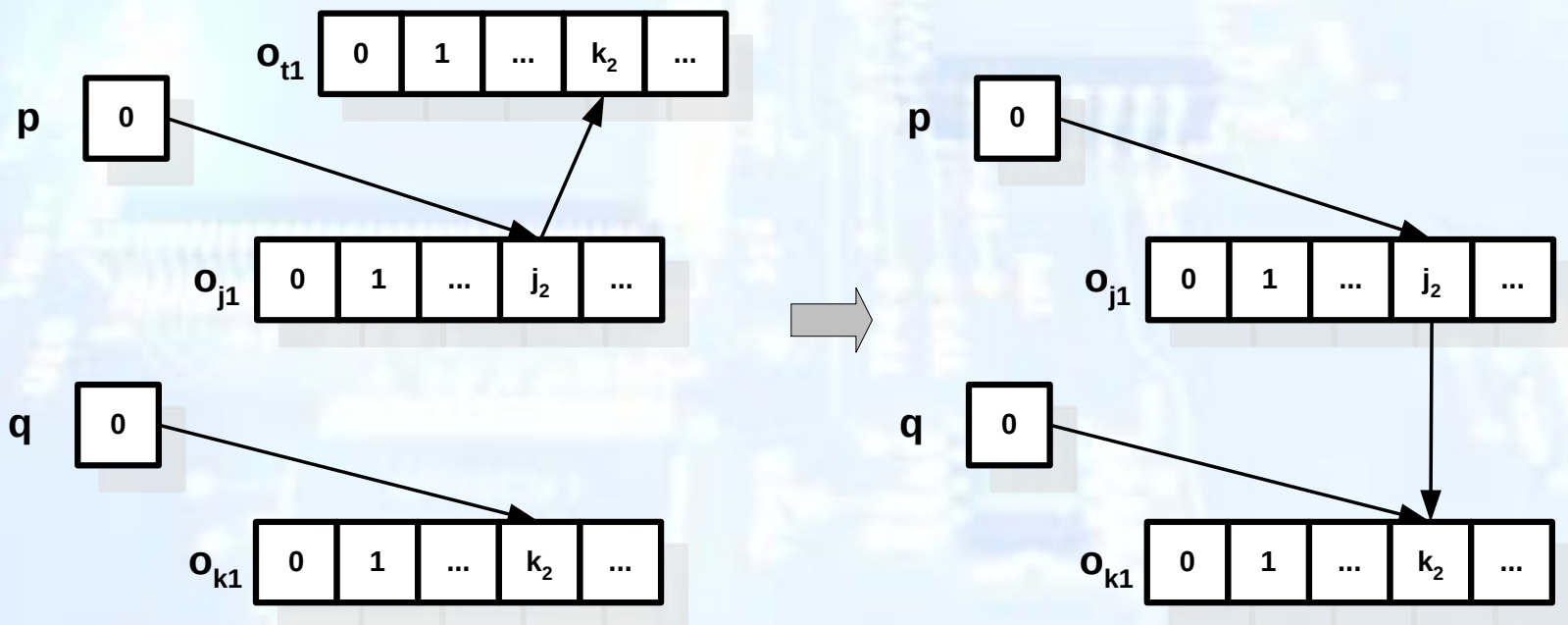
- p , q_1 и q_2 – указатели на динамическую память
- Пример для конструкции `free(p);`

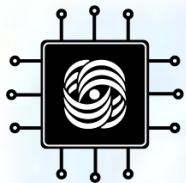




Анализ указателей. Пример

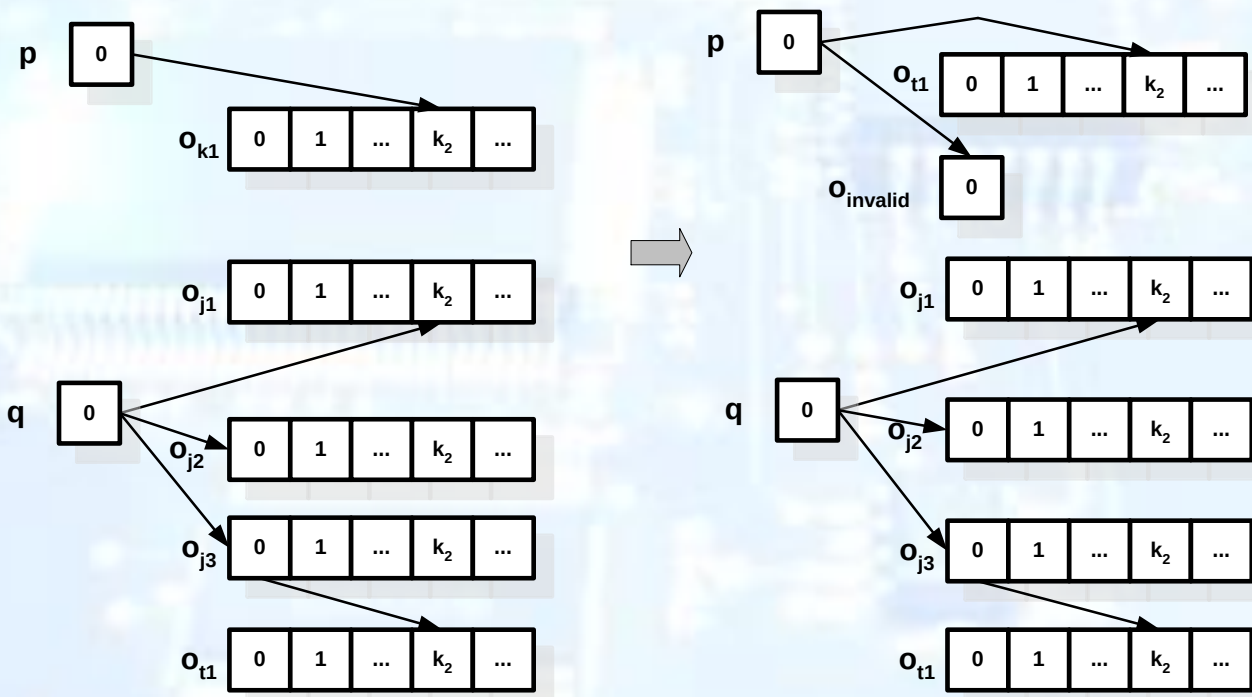
- p, q – указатели
- Пример для конструкции $*p = q$





Анализ указателей. Пример

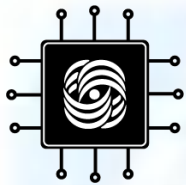
- p, q – указатели
- Пример для конструкции $p = *q$





Ресурсный анализ

- Ресурсы – объекты программы, имеющие свой жизненный цикл
- Примеры ресурсов:
 - Выделенная динамическая память
 - Файлы
 - Семафоры
 - Мьютексы
 - Сокеты
 - Потоки
 - Нити
 - ...
- Обычно ресурсы характеризуются состоянием и атрибутами
- Каждый тип ресурса изменяет свое состояние по определенным правилам

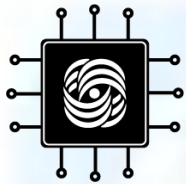


Ресурсный анализ.

Объект в куче

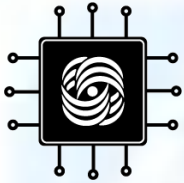
34

- Правила использования
 - Создается (`p = malloc(...)`)
 - Используется (`*p`)
 - Освобождается (`free(p)`)
- Примеры дефектов
 - Использование не созданного объекта
 - Повторное освобождение
 - Утечка



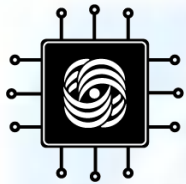
Ресурсный анализ. Файл

- Правила использования
 - Открыть (`f = fopen(..., "r")`)
 - Читать/писать (`fprintf(f, ...)`)
 - Закрыть (`close(f)`)
- Примеры дефектов
 - Чтение из не открытого файла
 - Попытка писать в файл открытый на чтение
 - ...



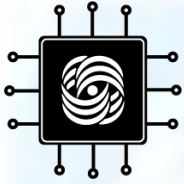
Ресурсный анализ

- Отличие ресурсов от других объектов программы:
 - Управляются операционной системой
 - Управление сменой состояний и атрибутов происходит через функции API
- Необходимо анализировать не конструкции ЯП, а библиотечные вызовы
- Необходима информация о семантике библиотечных вызовов:
 - Семантика зашита в анализатор
 - Семантика описывается через аннотирование библиотек



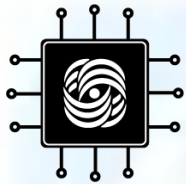
Масштаб анализа

- Внутрипроцедурный анализ
 - Каждая функция анализируется отдельно
 - Информация не передается между функциями
 - Ресурсоемкость относительно низкая
 - Точность и/или полнота – снижаются
 - Реализуется большинством анализаторов



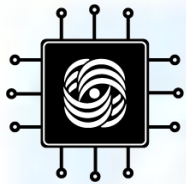
Масштаб анализа

- Межпроцедурный анализ
 - Все функция анализируется совместно
 - Информация передается между функциями
 - Ресурсоемкость очень высокая
 - Точность и/или полнота – выше
 - Реализовано в небольшом числе средств



Консервативность анализа

- Определяется правилами объединения ветвей
- Консервативный (пессимистический) анализ
 - При объединении ветвей объединяет интервалы
 - Сохраняет полноту (обнаруживаем все ошибки)
 - Теряет точность (много ложных обнаружений)
- Оптимистический анализ
 - При объединении ветвей пересекает интервалы
 - Теряет полноту (может пропускать ошибки)
 - Сохраняет точность (мало ложных обнаружений)



Консервативность анализа

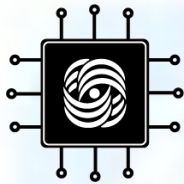
```
int arr1[10];
int arr2[5];
if (...) {
...
} // <i, [1..8]>
else {
...
} // <i, [6..15]>
a = arr1[i]; // возможно
              ошибка
b = arr2[i]; // ошибка
```

- Консервативный анализ:
 - $\langle i, [1..15] \rangle$
 - обнаружит обе ошибки (возможно первая – ложная)
- Оптимистический анализ
 - $\langle i, [6..8] \rangle$
 - обнаружит вторую ошибку (пропустив первую)



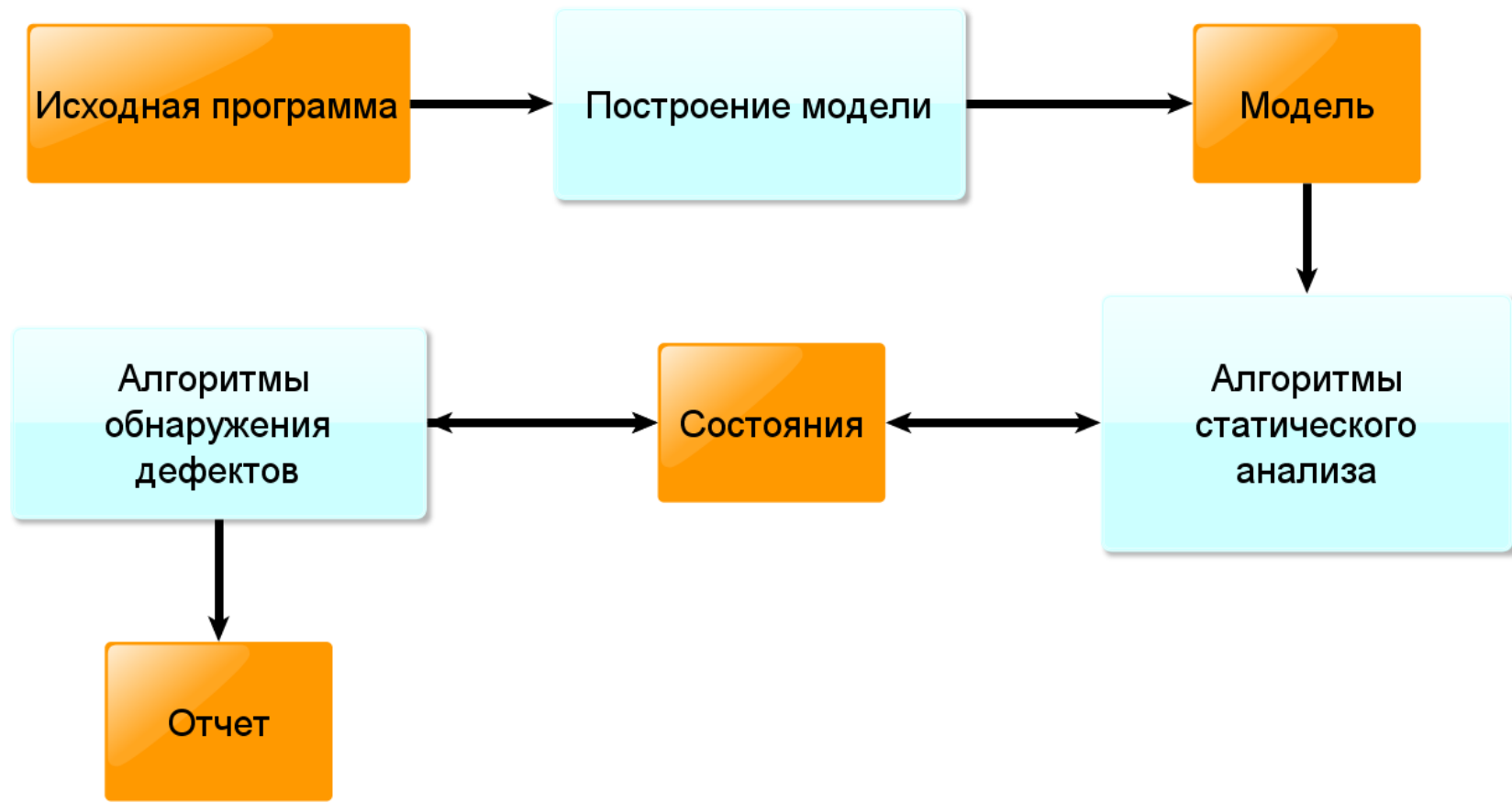
Решение уравнений

- В результате применения алгоритмов в явном или неявном виде строится система уравнений
- Решение системы
 - С использованием теории решеток
 - Методами абстрактной интерпретации
- Завершаемость процесса решения обуславливается
 - Монотонностью уравнений
 - Конечностью системы уравнений



Общая схема обнаружения дефектов

42





Обнаружение дефектов

- На основе анализа допустимости состояний в элементах модели
- Использование неинициализированной переменной:
 - Конструкция: $a = b$
 - Условие: $\langle a, \text{noninit} \rangle \in S^{l-1}$
- Повторное освобождение объекта
 - Конструкция: $\text{free}(p)$
 - Условие: $\langle p, \text{invalid} \rangle \in S^{l-1}$



Обнаружение дефектов

- Разыменование некорректного указателя
 - Конструкция: $a = *p$
 - Условие: $\langle p, \text{invalid} \rangle \in S^{l-1}$
- Арифметические операции над указателями на разные объекты
 - Конструкция: $a = p - q$;
 - Условие: $\langle p, o_1 \rangle, \langle q, o_2 \rangle \in S^{l-1}, o_1 \neq o_2$
- Выход за границы объекта
 - Конструкция: $a = *p$;
 - Условие: $\langle p, o, k \rangle \in S^{l-1}, k \geq \#o^{\text{size}}$



Проблемы статического анализа

45

- Анализ циклов
- Анализ рекурсии и глубокой вложенности вызовов
- Анализ больших типов данных
- Анализ программ с исключениями
- Анализ программ с полиморфизмом и указателями на функции



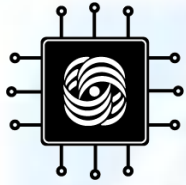
Статический анализ. Резюме

- Достоинства статического анализа
 - Возможность полного обнаружения дефектов определенного класса
 - Полная автоматизация процесса обнаружения
- Недостатки статического анализа
 - Наличие ложных обнаружений (false positives)
 - Высокая ресурсоемкость
 - Невозможность обнаружения функциональных ошибок



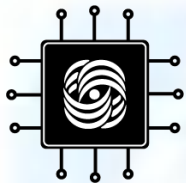
Место статического анализа

- Собственно статический анализ может использоваться
 - В составе сред разработки для оперативной проверки программ
 - В системах Continuous integration для постоянного контроля качества кода и своевременного обнаружения дефектов
- Статический анализ не заменяет тестирование, а дополняет его
- Хорошая практика - использование автоматизированного тестирования вместе со статическим анализом



Другие статические методы

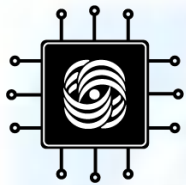
- Инспекции
 - Формальные инспекции
 - Аудит кода
 - Парное программирование
- Дедуктивная верификация
- Model Checking



Применение различных методов обеспечения качества

49

	Хар-ка качества	Проблема	Обеспечение качества
1	Функциональность	Функциональные ошибки, несоответствие спецификации	Тестирование Верификация
2	Надежность	Низкая надежность Наличие уязвимостей	Статический анализ Тестирование
3	Практичность	Сложность использования	Тестирование
4	Эффективность	Проблемы с производительностью, ресурсами	Тестирование Профилирование Динамический/статический анализ
5	Сопровождаемость	Сложность сопровождения, модификации	Рефакторинг Документирование
6	Мобильность	Несоответствие стандартам, Сложность адаптации	Аудит, рефакторинг Статический анализ



Спасибо за внимание!



НАДЁЖНОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Лекция 8: *Надёжность системы*

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



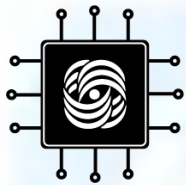
О чём этот курс

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



План лекции

- Надёжность последовательных и параллельных систем
- Блок-схема расчёта надёжности
- Анализ эксплуатационной безопасности
- Анализ типов и последствий отказов
- Анализ дерева неисправностей



Эпиграф

- *В жизни нет гарантий, существуют
одни вероятности*

Том Клэнси





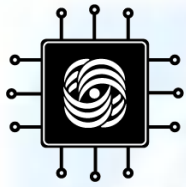
Что такое надежность?

- Надежность - вероятность того, что программная система будет работать по спецификации.
- Дефект – код, приводящий к ошибке.
- Ошибка – проявление дефекта.
- Ошибки приводят к отказам – неверной работе программы.



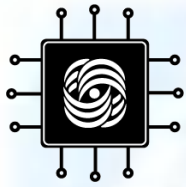
Модели надёжности

- Граф надёжности (Reliability Graph)
- Дерево неисправностей (Fault Tree model)
- Блок-схема расчёта надёжности (Reliability Block Diagram)



Системная Надёжность (1)

- Система состоит из модулей
- Модуль состоит из компонент
- Компоненты могут иметь
 - Разную надёжность
 - Разную зависимость между ними
- Надёжность системы – это функция надёжности компонентов и зависимости между компонентами

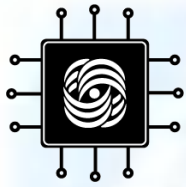


Надежность последовательных систем

- Система состоит из n компонент, соединенных друг с другом
- Отказ каждого компонента приводит к отказу всей системы



- Надежность системы меньше надежности ее компонентов (т.к. $R_k \leq 1$).



Суммирование надежностей

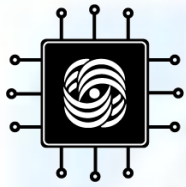
- Надежность системы может быть выражена через надежности ее компонентов, если они отказывают независимо.
- Для систем из отдельных компонентов:

$$R = \prod_{k=1}^{Q_p} R_k$$

Q_p число компонентов

R_k надежность компонента

- Надежности компонентов (R_k) должны быть заданы в одном интервале.



Суммирование надёжностей

- Используя связь между надёжностью и частотой ошибок

$$\lambda = \frac{-\ln R}{\tau} \quad \text{or} \quad R = e^{-\lambda\tau}$$

- Получаем:

$$\lambda = \sum_{k=1}^{Q_p} \lambda_k$$

- Т.е. общая частота есть сумма частот отдельных компонентов



Пример системы

- Система из 4 независимых компонентов

- $R_1 = 0.95$

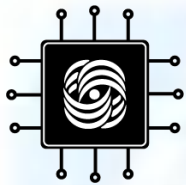
- $R_2 = 0.87$

- $R_3 = 0.82$

- $R_4 = 0.73$

$$R_{system} = 0.95 \times 0.87 \times 0.82 \times 0.73 = 0.4947$$

- Надежность системы меньше надежности каждого из компонентов

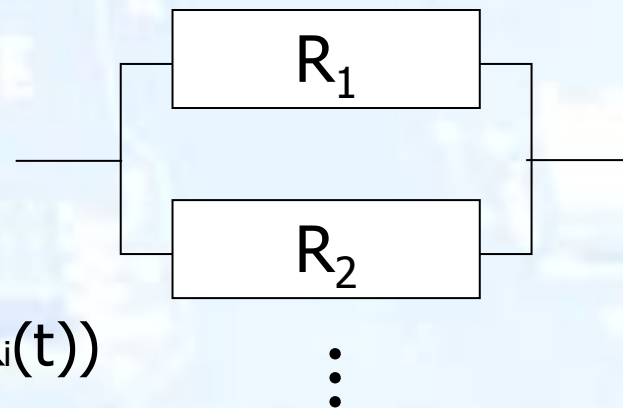


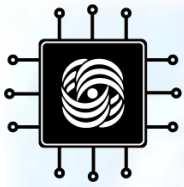
Надежность параллельных систем

- Система состоит из n компонентов, работающих параллельно
- Только отказ всех компонентов приводит к отказу всей системы.

$$\text{Unreliability } F_p(t) = \prod_{i=1}^n F_i(t)$$

$$\text{Reliability } R_p(t) = 1 - \prod_{i=1}^n F_i(t) = 1 - \prod_{i=1}^n (1 - R_i(t))$$





Пример параллельной системы

- Система состоит из 4 параллельно работающих компонентов

- $R_1 = 0.95$

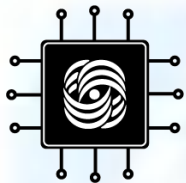
- $R_2 = 0.87$

- $R_3 = 0.82$

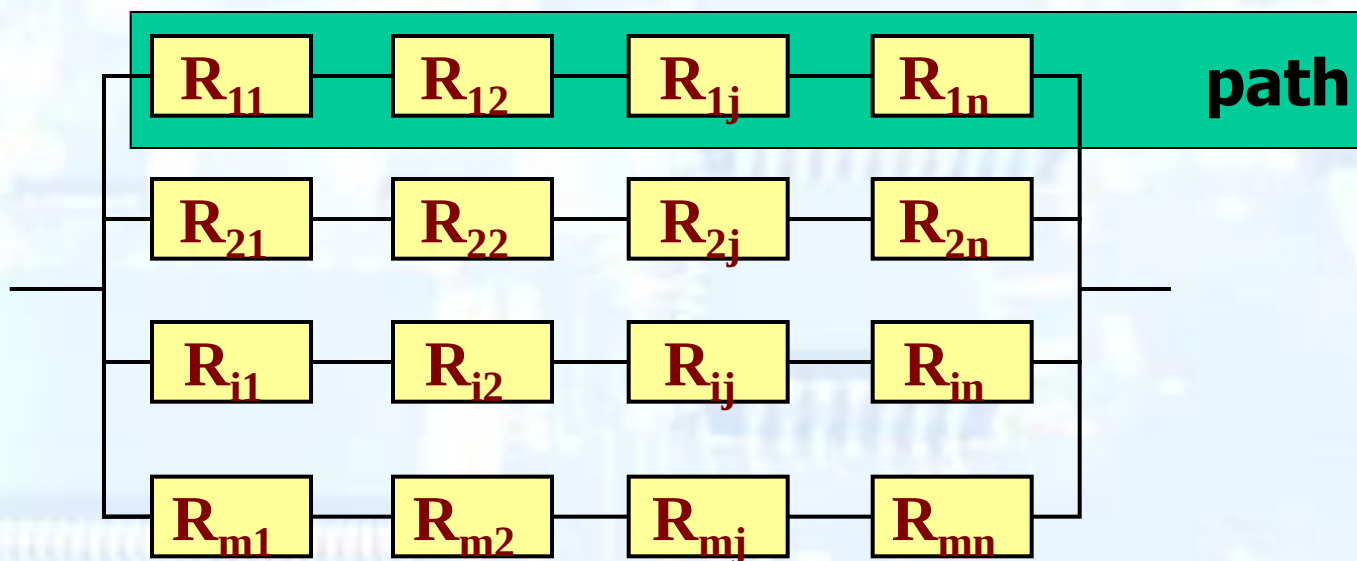
- $R_4 = 0.73$

$$R_{system} = 1 - ((1 - 0.95) \times (1 - 0.87) \times (1 - 0.82) \times (1 - 0.73)) = 0.9996$$

- Надежность системы больше, чем надежности отдельных компонентов.

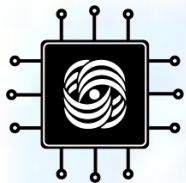


Параллельно-последовательные системы

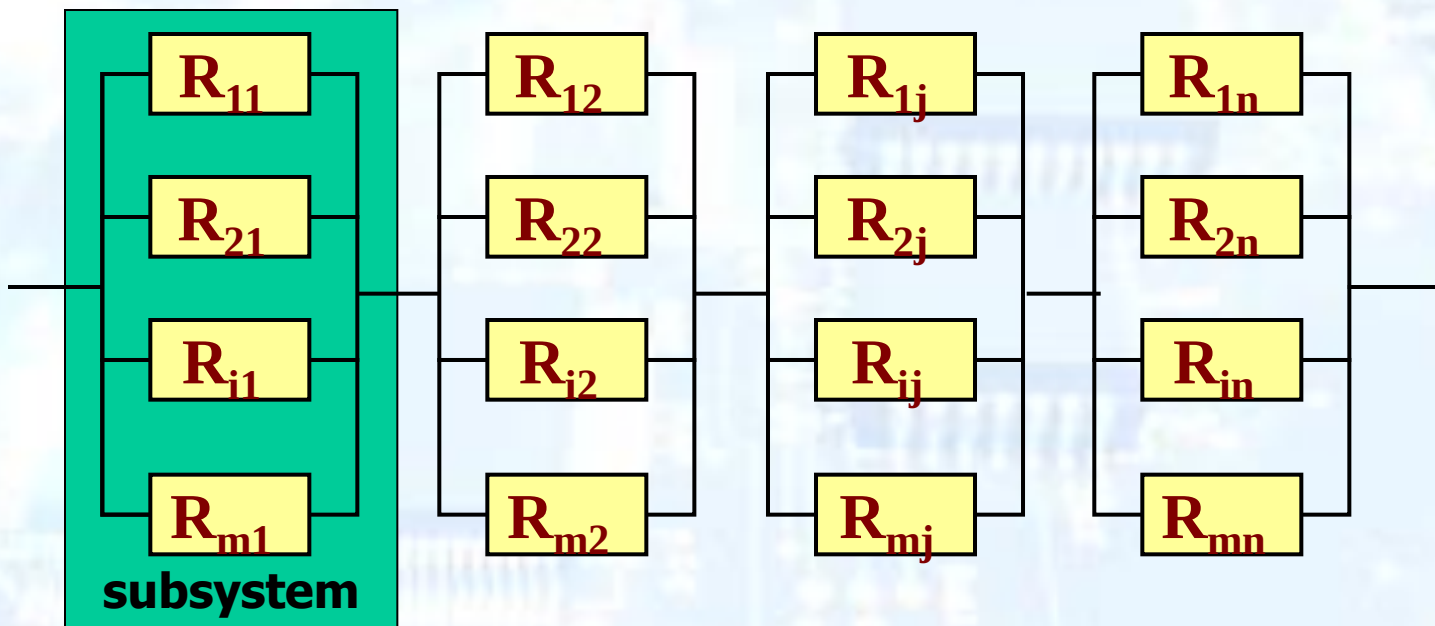


$$\text{Path reliability } R_i = \prod_{j=1}^n R_{ij}$$

$$\text{System reliability } R = 1 - \prod_{i=1}^m (1 - R_i) = 1 - \prod_{i=1}^m \left(1 - \prod_{j=1}^n R_{ij}\right)$$

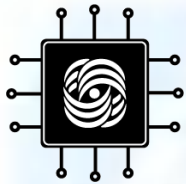


Параллельно-последовательные системы



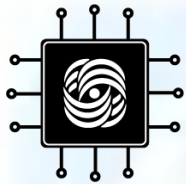
Subsystem reliability $R_j = 1 - \prod_{i=1}^m (1 - R_{ij})$

System reliability $R = \prod_{j=1}^n R_j = \prod_{j=1}^n \left[1 - \prod_{i=1}^m (1 - R_{ij}) \right]$



Другие архитектуры

- Однополосный мост
- Двуполосный мост



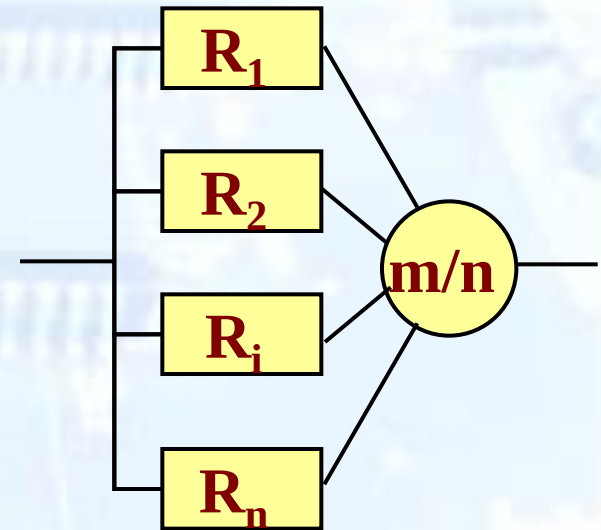
Горячее резервирование

- Использует параллельность.
- Все компоненты работают одновременно
- Каждый компонент достаточен для корректной работы
- Только один компонент необходим для корректной работы
- Каждый компонент удовлетворяет требованиям к надежности
- Система отказывает, если отказывают все компоненты.



Система m – из – n

- Система из n компонентов
- Для корректной работы нужно не меньше m компонент ($m \leq n$).
 - $m=n$: последовательная
 - $m=1$: параллельная
- Например, самолет с 4 моторами может лететь при отказе двух моторов



Предположение:
Все компоненты
имеют
одинаковую надежность

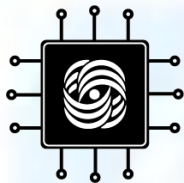
$$R_{\text{sys}}(t) = 1 - \sum_{i=0}^{m-1} \binom{n}{i} R(t)^i (1 - R(t))^{n-i}$$

$$\binom{n}{i} = \frac{n!}{i! (n-i)!}$$

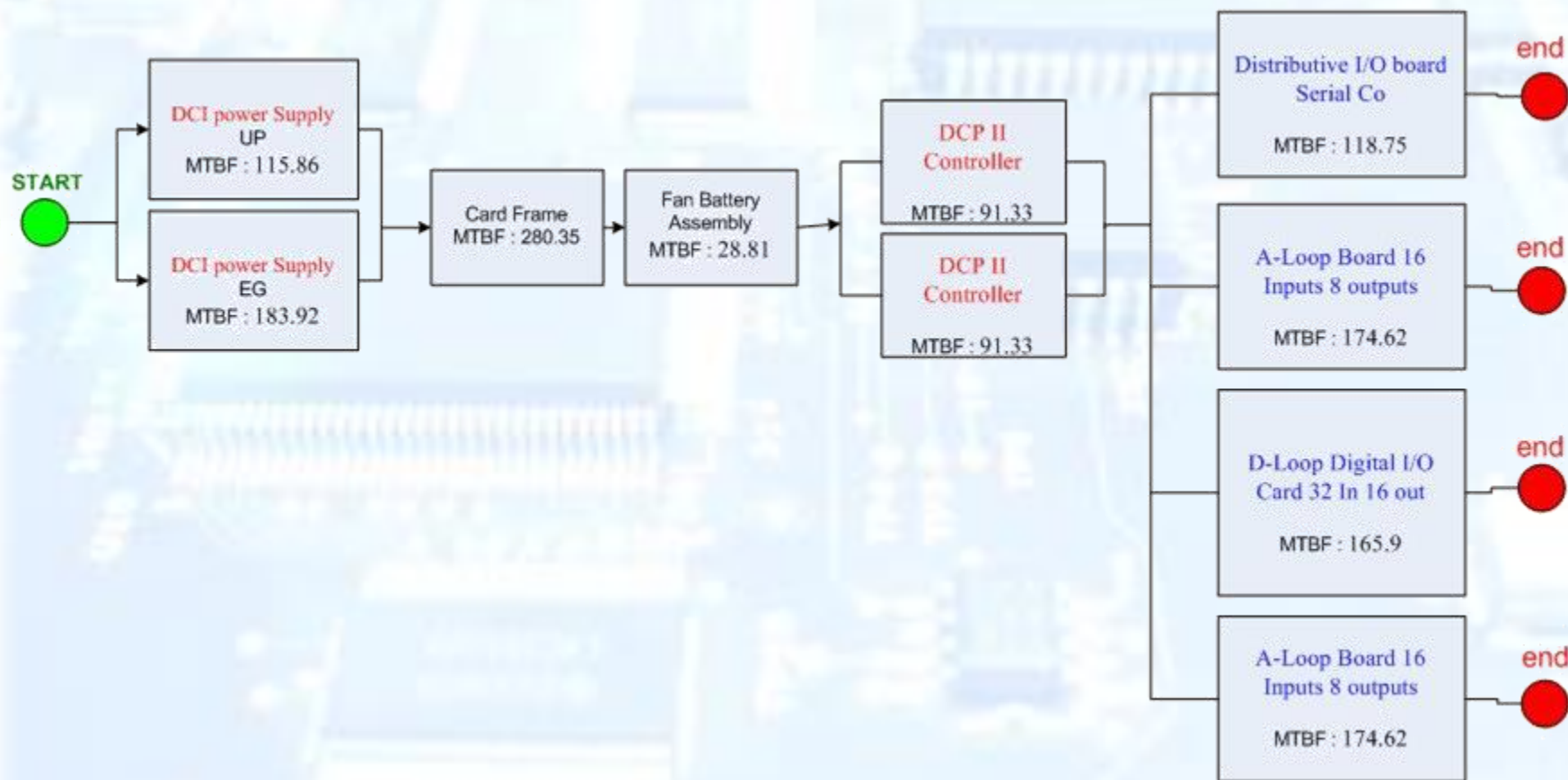


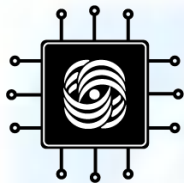
Блок-схема надежности (RBD)

- RBD показывает, как соединены компоненты системы для обеспечения надежности
- Самые частые конфигурации – последовательные и параллельные компоненты.
- В последовательной конфигурации отказ каждого компонента приводит к отказу системы. Общая надежность меньше надежности отдельных компонентов.
- В параллельной конфигурации отказ всех компонентов приводит к отказу системы. Общая надежность больше надежности отдельных компонентов.
- Система содержит некую комбинацию этих конфигураций
- Анализ диаграмм необходим для вычисления надежности систем.

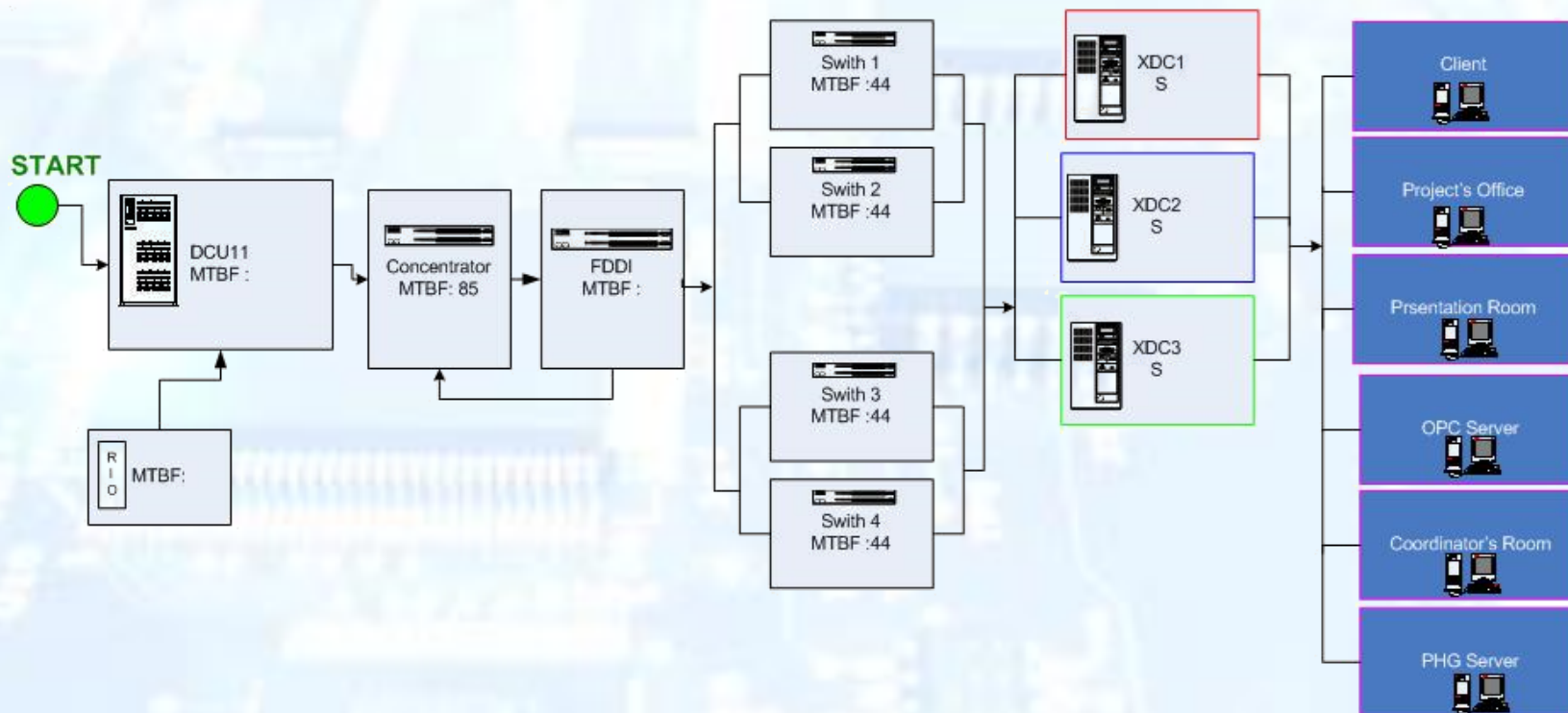


RBD: пример /1





RBD: пример /2





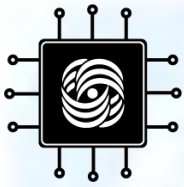
RBD: когда и как?

- Когда использовать RBD?
 - Когда нужно оценить надежность сложной системы и найти в системе уязвимые места
- Как использовать RBD?
 - Нарисовать схему
 - Найти надежность системы
 - Вычислить другие характеристики (доступность, время работы etc)
- Есть автоматизированные средства создания и анализа диаграмм, интегрированные с другими методами, например, с FTA.



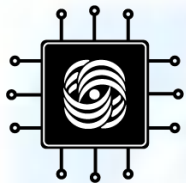
RBD: плюсы и минусы

- RBD – самый простой метод визуализации характеристик надежности.
- Преимущества:
 - Помогает найти цели по надежности
 - Оценивает влияние отказа компонента на работу системы
 - Позволяет простой анализ различных сценариев работы
 - Подсчитывает MTBF системы
 - Уменьшает стоимость исправления проблем в больших системах
 - Позволяет анализировать разные конфигурации
 - Указывает на потенциальные проблемы в системе
 - Определяет чувствительность системы к отказам компонентов
- Недостаток в том, что сложные конфигурации (ожидание, ветвление, распределение нагрузки и.т.д.) не могут быть описаны RBD.



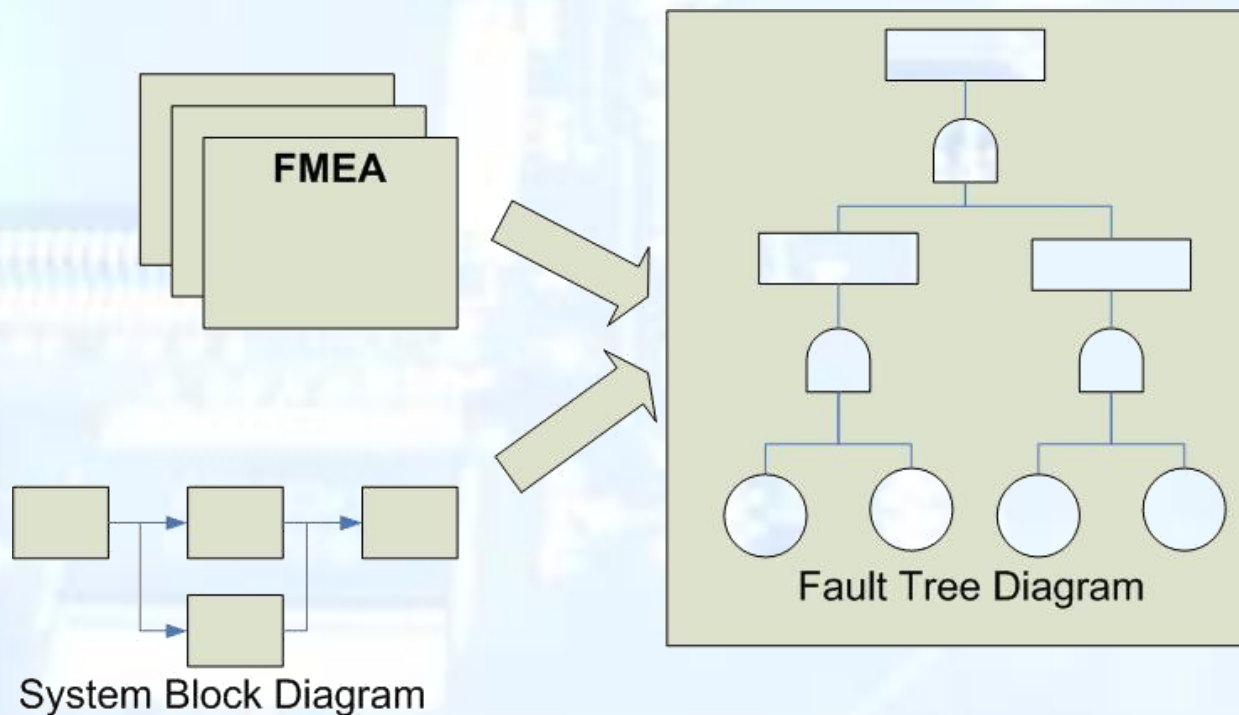
Дерево неисправностей (FTA)

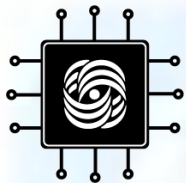
- Дерево неисправностей – графическое представление основных ошибок и отказов в системе, их причин и способов их устранения. Деревья позволяют находить уязвимые места при проектировании новой системы или анализе существующей. Они также помогают находить методы решения проблем.
- Дерево неисправностей можно рассматривать как графическую модель поведения системы, ведущего к нежелательному событию. Поведение определяется событиями и условиями и записывается логическим выражением.
- Анализ деревьев неисправностей полезен как при проектировании новых систем и программных продуктов, так и при поиске недостатков в существующих. Они помогают найти корень проблемы и исправить ее.



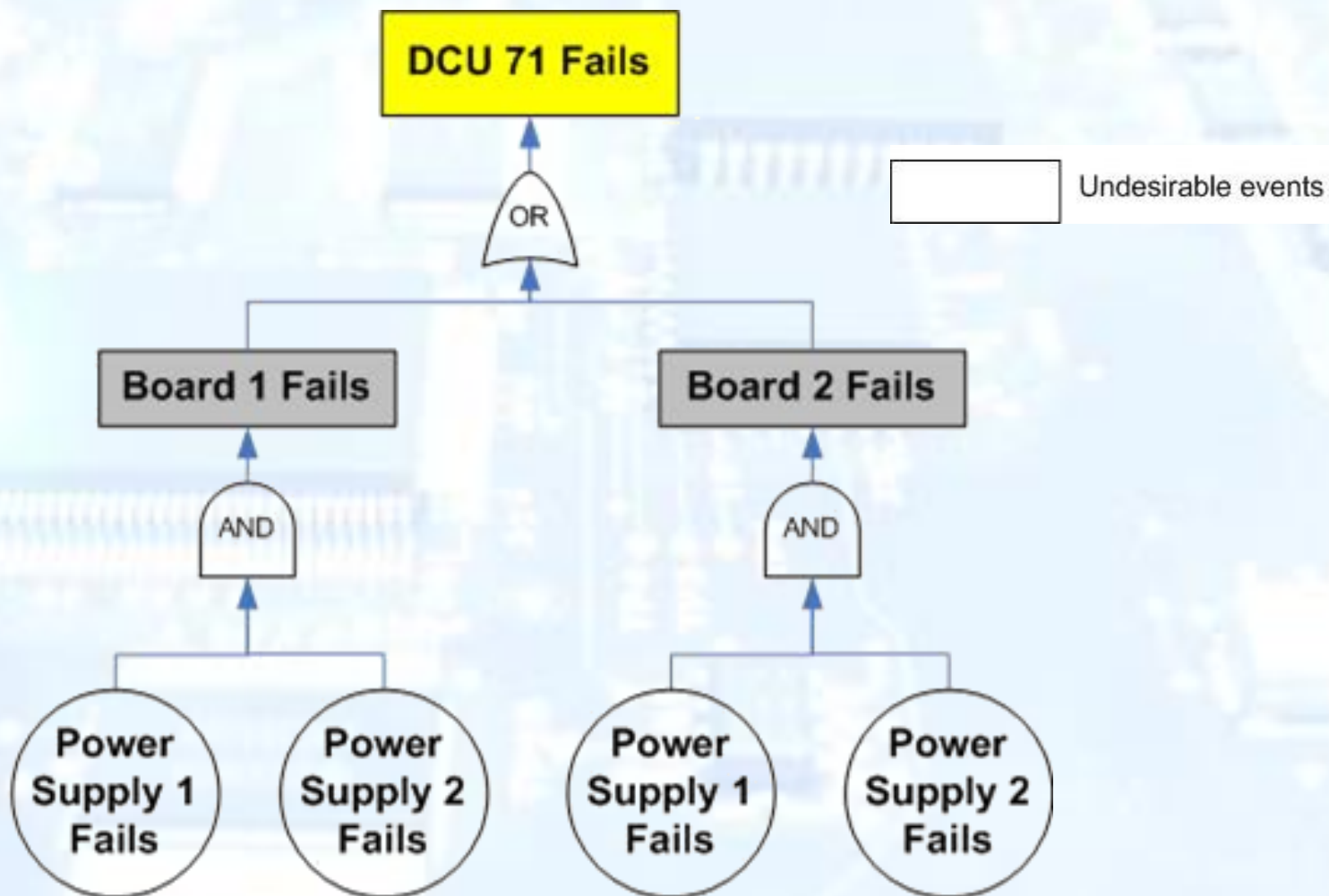
Шаги в FTA

- FMEA определяет условия, приводящие к нежелательным событиям. Затем определяется, какие локальные ошибки могут привести к отказу системы. FMEA отвечает на вопрос, “Что может пойти не так?” даже если спецификация выполнена. **Для работы FTA необходимо иметь FMEA.**





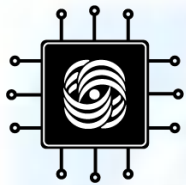
FTA: пример



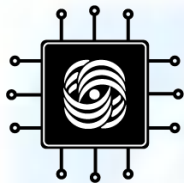


FTA: плюсы и минусы

- **Плюсы:** FTA дают осмысленные данные, позволяющие оценить и улучшить надежность системы. Они также дают оценку необходимости и эффективности резервирования.
- **Минусы:** Ограничение FTA в том, что нежелательное событие необходимо предвидеть и знать основные источники проблемы. Это может быть очень долго и дорого. Успех зависит от способностей аналитика, работающего над этой задачей.



Спасибо за внимание!



НАДЁЖНОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Лекция 9:

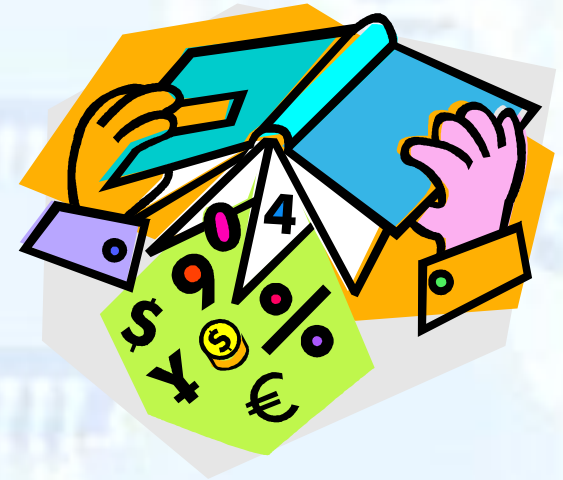
Модели надёжности ПО *Оптимизация надёжности ВС*

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



План лекции

- Основные свойства МНПО
- Пуассоновские модели
- Марковские модели
- Байесовские модели
- Пример выбор модели оценки
- Задача оптимизации надёжности ВС
- Методы обеспечения отказоустойчивости



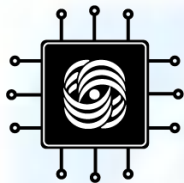


Эпиграф

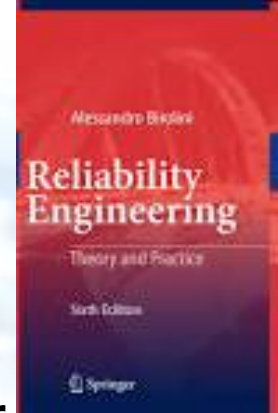
- *Если отладка — процесс удаления ошибок, то программирование должно быть процессом их внесения.*

Эдсгер Дейкстра

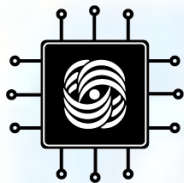




Литература



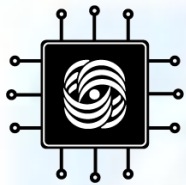
- Birolini A. Reliability engineering. – Berlin : Springer, 2007. – Т. 5. (Глава 2 – Reliability Analysis During the Design Phase)
- Lyu M. R. et al. Handbook of software reliability engineering. – CA : IEEE computer society press, 1996. – Т. 222.(Глава 3 - Software Reliability Modeling Survey) (http://www.cse.cuhk.edu.hk/~lyu/book/reliability/pdf/C hap_3.pdf)
- A.Wood Software Reliability Growth Models. Technical Report, 1996 (<http://www.hpl.hp.com/techreports/tandem/TR-96.1.pdf>)



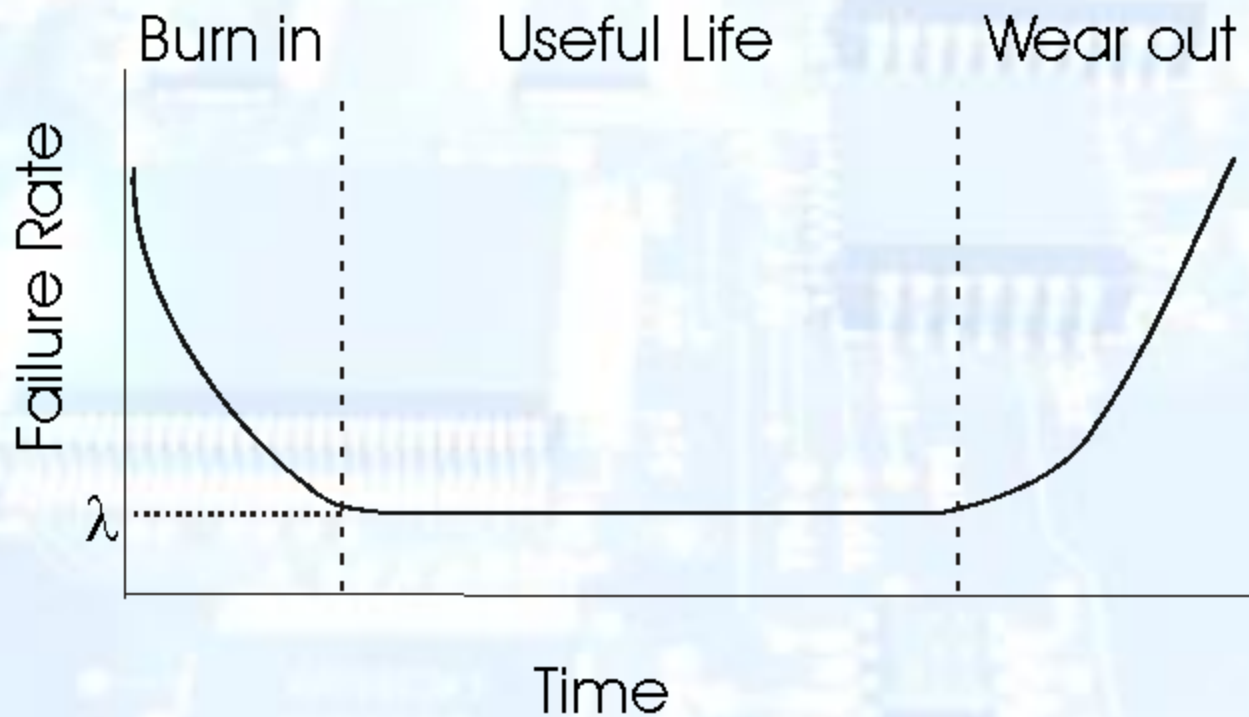
Литература (2)

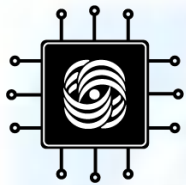
- Kuo W., Wan R. Recent advances in optimal reliability allocation //Computational Intelligence in Reliability Engineering. – Springer Berlin Heidelberg, 2007. – С. 1-36. (http://content.schweitzer-online.de/static/content/catalog/newbooks/978/354/037/9783540373674/9783540373674_Excerpt_001.pdf)
- Pullum L. L. Software fault tolerance techniques and implementation. – Artech House, 2001.



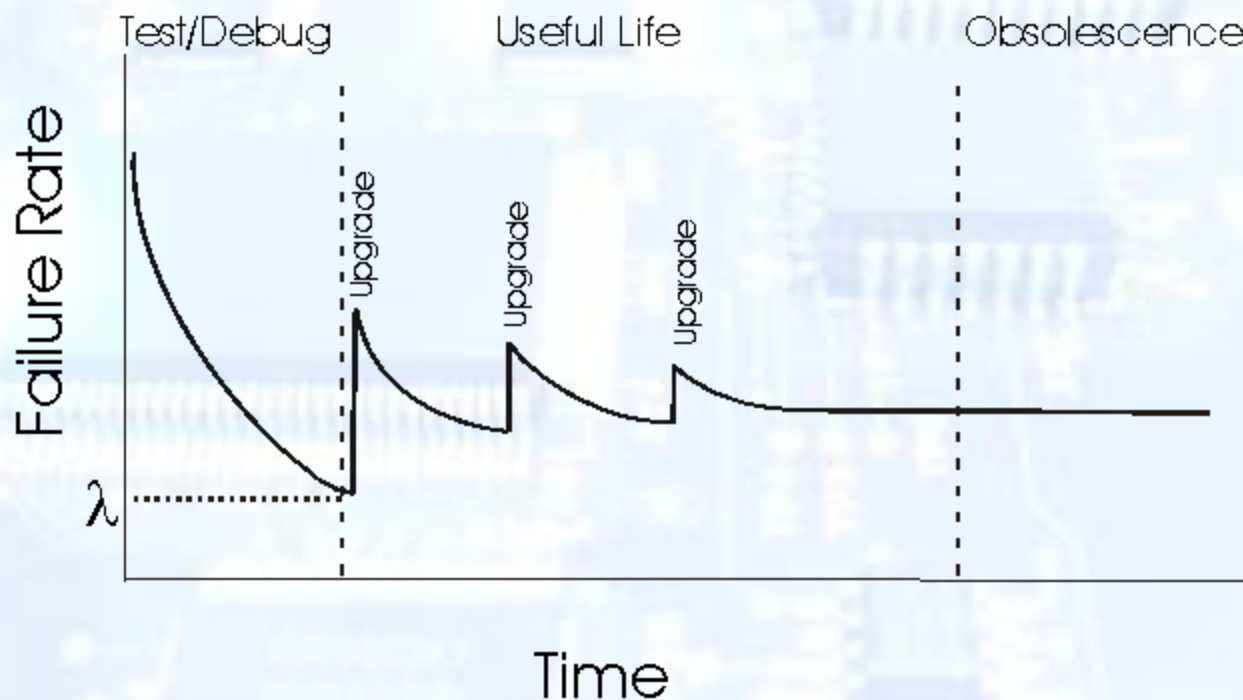


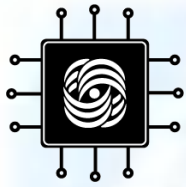
Надёжность аппаратуры





Надёжность ПО





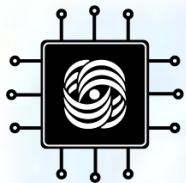
Особенности оценки надёжности ПО

- Оценка надёжности сложная проблема, так как не понятна природа ПО
- Непонятно как измерять размер ПО



Оценка числа ошибок в программе

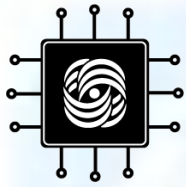
- Задачи:
 - оценить число оставшихся дефектов в текущей версии программы.
 - оценить вероятность отказа компонента или системы
 - оценить среднее время до следующего отказа
- Возможны два пути:
 - Статический анализ кода
 - Анализ статистики тестирования



Пример статистики тестирования

Неделя	Проект 1	Проект 2	Проект 3	Проект 4
1	16	13	6	1
2	27	26	13	8
3	41	40	28	11
4	54	61	48	19
5	69	84	57	27
6	81	95	60	32
7	90	104		36
8	96	112		39
9	99	117		41
10	100			42

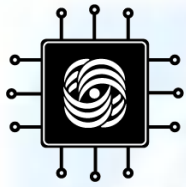
Наша задача – описать процесс обнаружения ошибок математической моделью



Пример статистики тестирования (2)

- Данные:
 - Время отказа
 - Интервал между отказами

Failure no.	Failure times (hours)	Failure interval (hours)
1	10	10
2	19	9
3	32	13
4	43	11
5	58	15
6	70	12
7	88	18
8	103	15
9	125	22
10	150	25
11	169	19
12	199	30
13	231	32
14	256	25
15	296	40



Пример статистики тестирования (3)

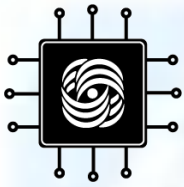
- Данные:
 - Зависимые отказы к заданному времени
 - Отказы во временной интервал

Time(s)	Cumulative Failures	Failures in interval
30	2	2
60	5	3
90	7	2
120	8	1
150	10	2
180	11	1
210	12	1
240	13	1
270	14	1



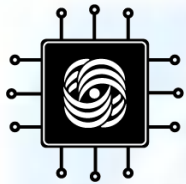
Модели надёжности

- Конечная цель – оценить надёжность программы, основываясь на данных, полученных при тестировании.
- Суть моделей надёжности - подобрать адекватную модель, описывающую количество обнаруженных ошибок с помощью известных вероятностных распределений.



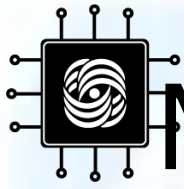
Принцип применения моделей надёжности

- Собрать данные об отказах
- Проверить данные (выбрать вид распределения)
- Выбрать модель
- Оценить параметры модели
- Настроить модель с выбранными параметрами
- Проверка критерия согласия
- Оценить требуемые параметры



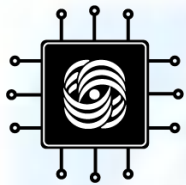
Модели надёжности ПО

- С 1970х годов предложено более 200 моделей но как определить надёжность ПО вопрос открытый
- Одну модель нельзя использовать во всех ситуациях
- Нет полной модели или даже более менее репрезентативной



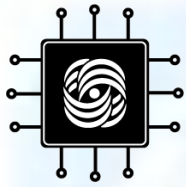
Модели надёжности ПО (2)

- Предположения
- Факторы
- Математическая функция

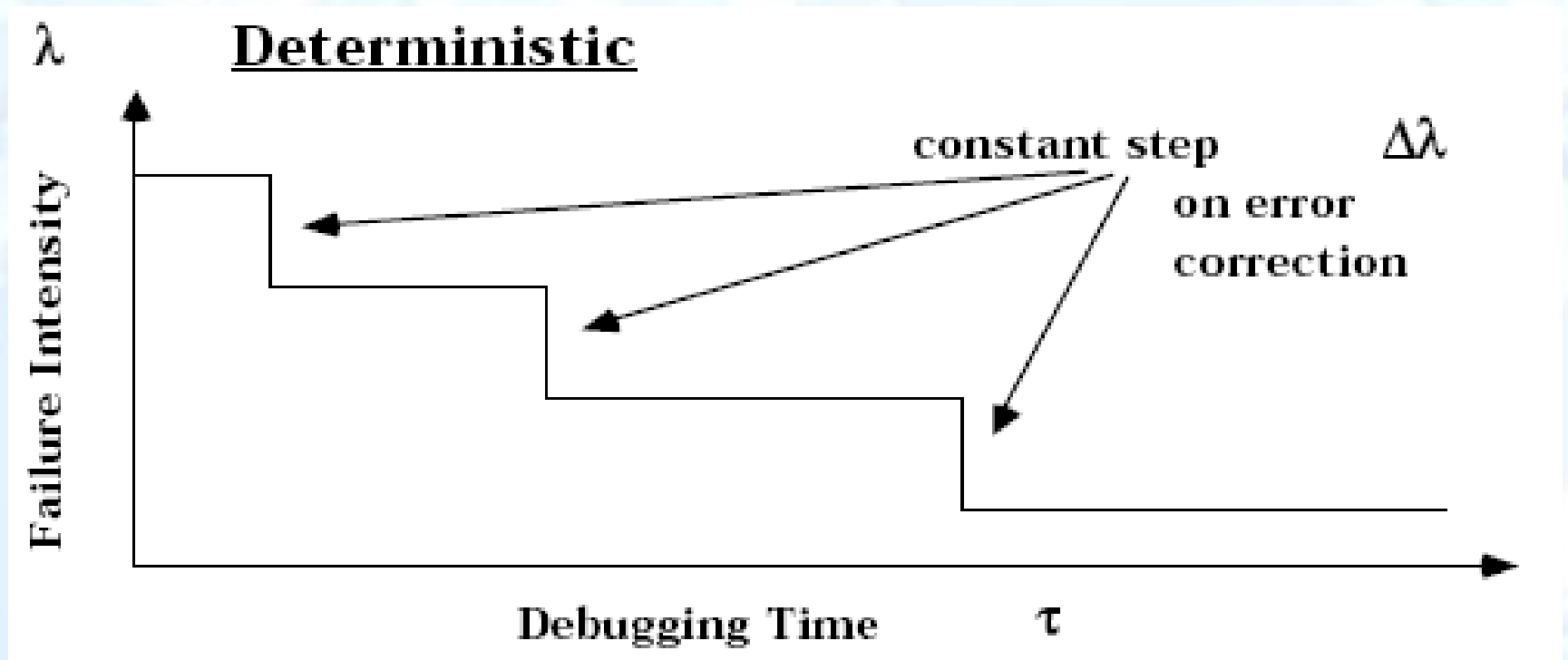


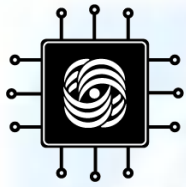
Виды моделей

- Пуассоновские
- Марковские
- Байесовские
- Комбинированные

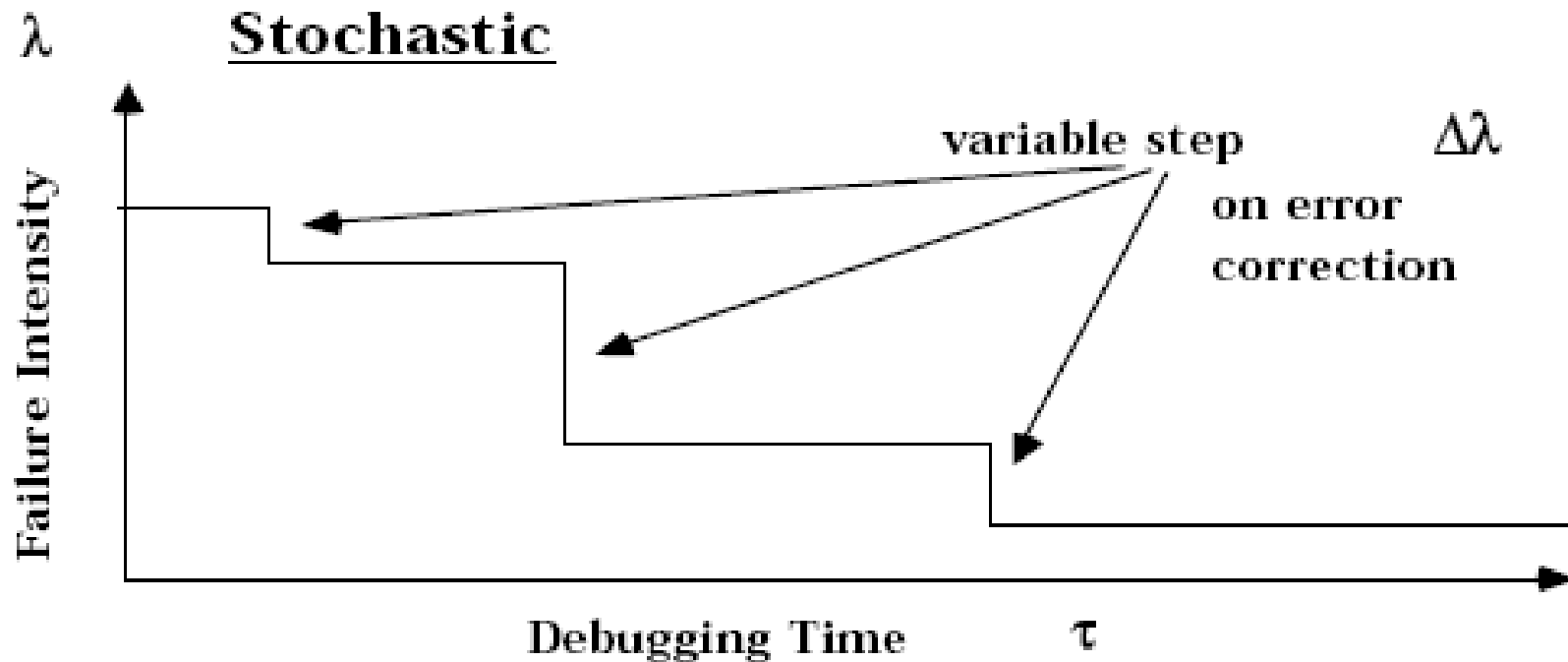


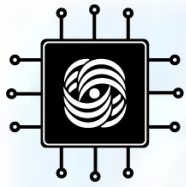
Модели надёжности ПО (3)



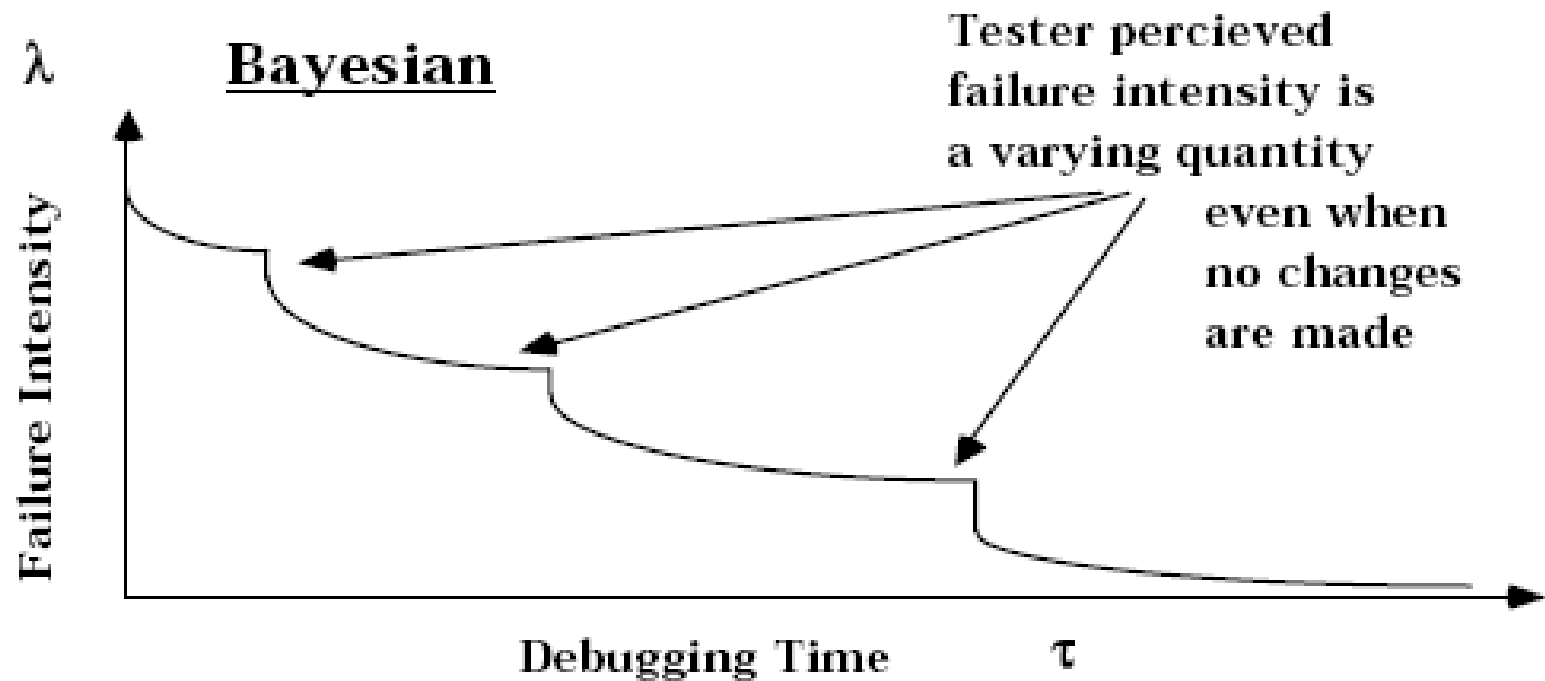


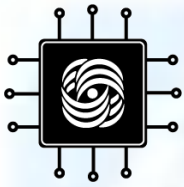
Модели надёжности ПО (4)





Модели надёжности ПО (5)





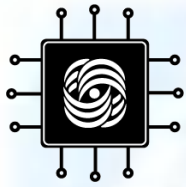
Основная теорема о пуассоновском процессе

- В каждый момент времени

$$P(\xi_t = k) = e^{-\lambda} \frac{\lambda^k}{k!}$$

то есть имеет распределение Пуассона.

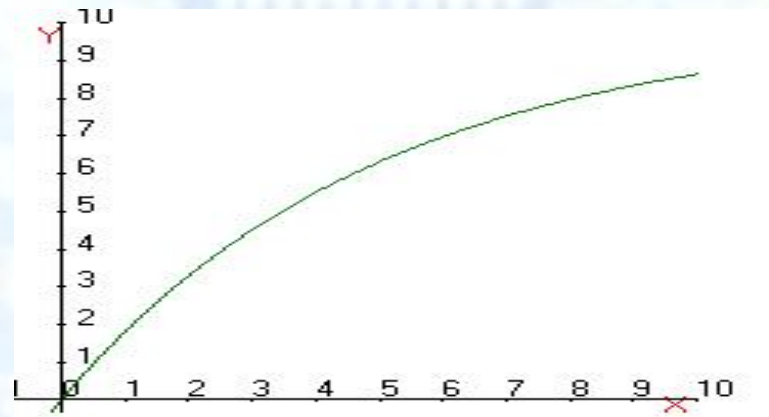
- λ называется интенсивностью потока. В рассматриваемом случае интенсивность есть количество ошибок, найденных к данному моменту.



Экспоненциальная модель

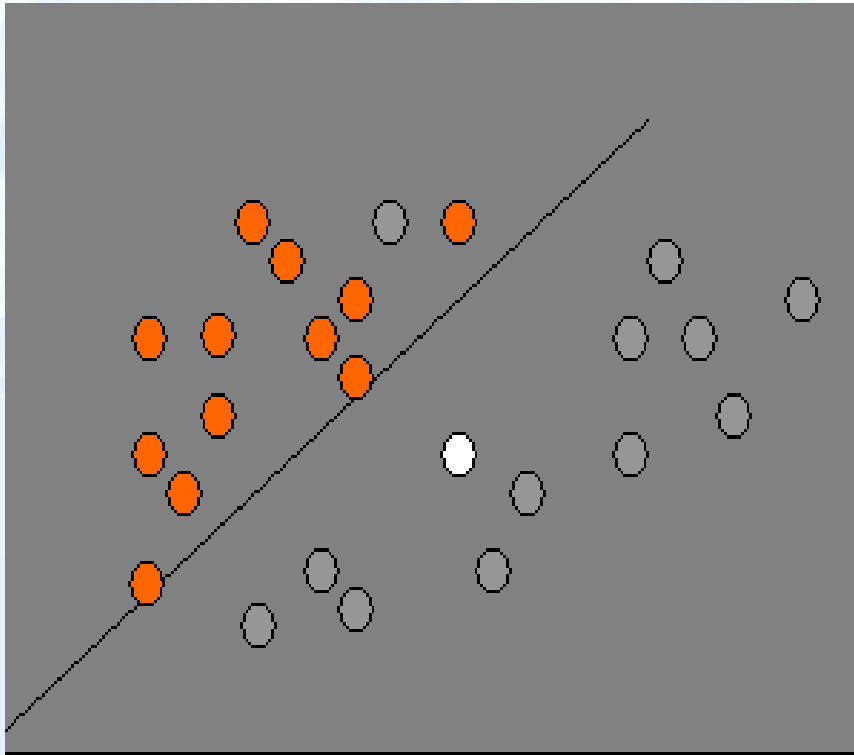
- Наиболее простая из выпуклых моделей

$$m(t) = a(1 - e^{-bt})$$

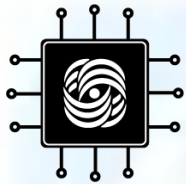




Оценка параметров: линейная регрессия

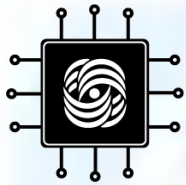


- Происходит минимизация суммы квадратов расстояний до некой прямой.



Ограничения для SRGM

- Ошибки исправляются сразу после обнаружения
- При исправлении новых ошибок не появляется
- В период тестирования не появляется нового кода
- Тестирование централизовано, то есть в каждый момент одна версия программы тестируется одной группой тестеров.
- Ошибки не зависят друг от друга.



Примеры усовершенствованных моделей

- Данная модель позволяет избавиться от ограничения на зависимость ошибок друг от друга

$$m(t) = a * \frac{(1 - e^{-bt})}{1 + \Psi(r)e^{-bt}}$$

$$\Psi(r) = \frac{1 - r}{r}$$

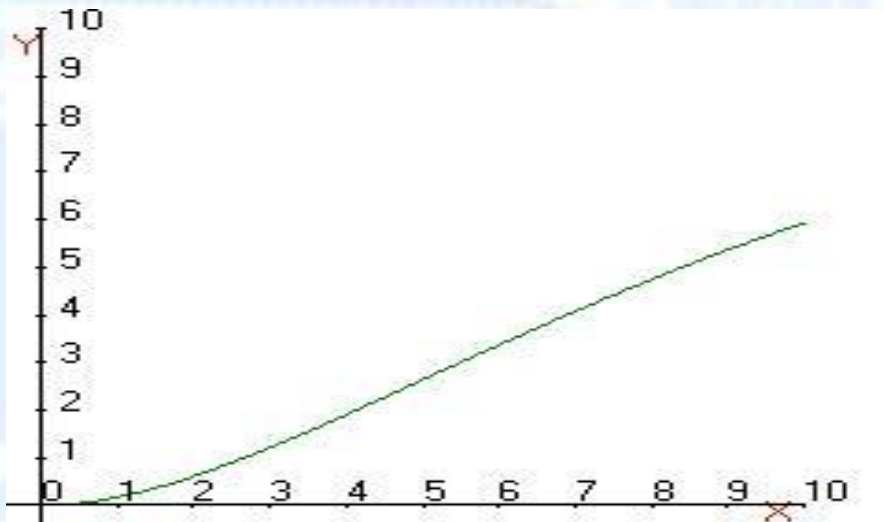
Параметр r есть отношение количества ошибок, которые можно обнаружить в текущий момент, к общему количеству ошибок. При $r = 1$ получаем стандартную экспоненциальную модель.

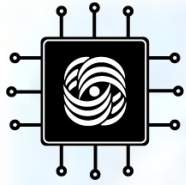


Модель Yamada

- Кривая становится S-изогнутой, и происходящее можно рассматривать как модель для тестирования, при котором умения тестера постепенно увеличиваются.

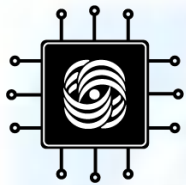
$$m(t) = a(1 - (1 + bt)e^{-bt})$$





Обзор остальных моделей

- Модель Вейбулла – учитывает качество тестирования
- Модель Парето – учитывает, что более серьезные ошибки исправляются раньше
- ...



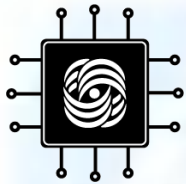
Особенности пуассоновских моделей

- По пуассоновским моделям много работ, их применяли в разных проектах множество раз, и получали адекватные результаты.
- Возможность исследовать несколько параллельных независимых источников ошибок
- Изначально, пуассоновские процессы использовались для моделирования отказов оборудования.



Модель Jelinski-Moranda

- Рассматриваем процесс тестирования как марковскую цепь с неизвестным начальным состоянием. Полагая, что надежность линейно зависит от количества ошибок, можно считать время между обнаружением двух ошибок случайной величиной с экспоненциальным распределением.



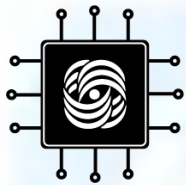
Модель Jelinski-Moranda

- Пусть в программе a ошибок, в начальный момент интенсивность равна $a*b$, где b – положительный параметр. T_i – время между появлением i -й и $i-1$ ошибок имеет показательное распределение с интенсивностью $(a - i + 1)*b$



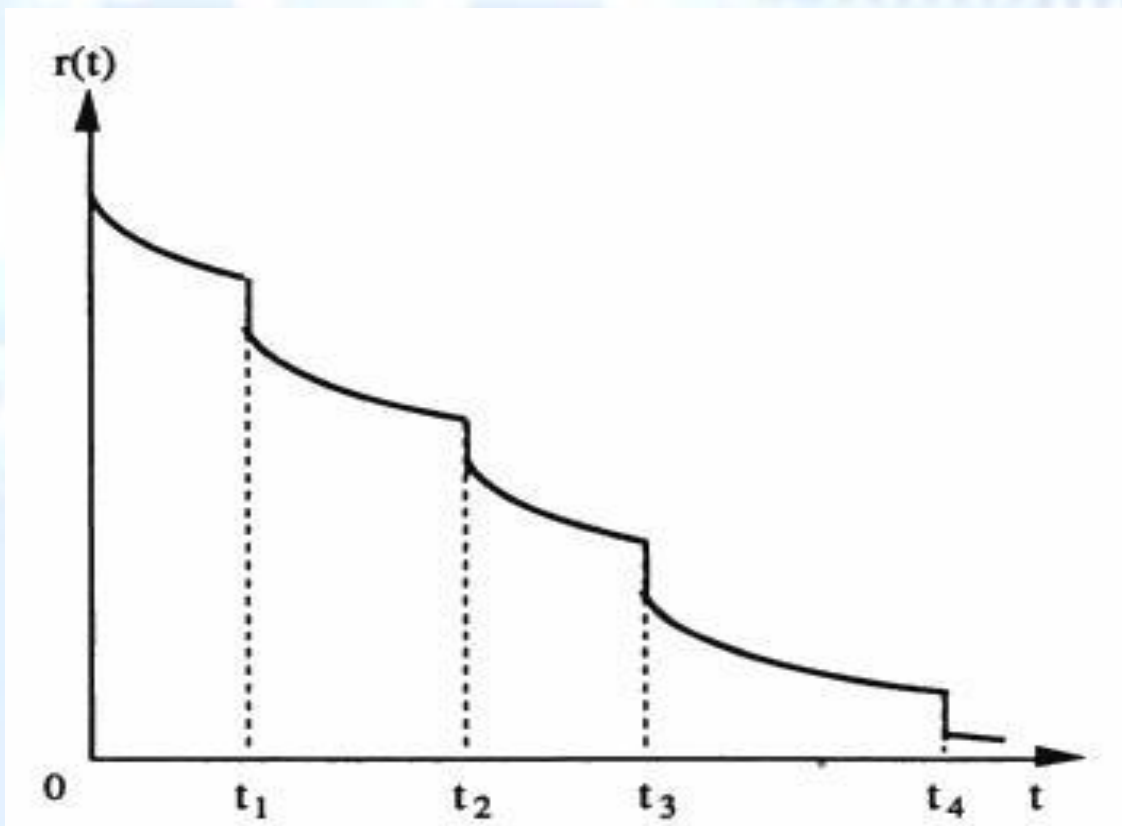
Байесовский подход

- Байесовскими называют различные модели, учитывающие новые данные для корректировки существующих гипотез. Если в описанных выше моделях используются некоторые предположения о характере распределения *a priori*, то при использовании байесовских методов эти параметры меняются в зависимости от новых измерений.



Частота в LV-модели

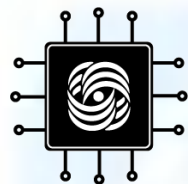
Частота – скорость изменения интенсивности, т.е. $m'(t)$.





Открытый вопрос

- Не решена проблема выбора между существующими SRGM. Критерии применимости той или иной SRGM к конкретному проекту до конца не сформулированы



Пример входных данных

<u>Cum. Failures</u>	<u>Cum. Time</u>	<u>Intensity</u>	<u>Time</u>
5	342	0.014620	171.00
10	571	0.021834	456.50
15	968	0.012594	769.50
20	1984	0.004921	1476.00
25	3098	0.004488	2541.00
30	5049	0.002563	4073.50
35	5324	0.018182	5186.50
40	6380	0.004735	5852.00
45	7644	0.003956	7012.00
50	10089	0.002045	8866.50
55	10982	0.005599	10535.50
60	12559	0.003171	11770.50
65	14708	0.002327	13633.50
70	16185	0.003385	15446.50
75	17758	0.003179	16971.50
80	20567	0.001780	19162.50



ИНТЕНСИВНОСТЬ

$$\text{Intensity (failures/CPU sec)} = \frac{\Delta f}{\Delta t} = \frac{5}{\text{Cum. } T_2 - \text{Cum. } T_1}$$

$$\text{Time (average)} = \text{Cum. } T_1 + \frac{\Delta t}{2}$$

- Две модели "basic execution time model" и "logarithmic Poisson execution time model" (e.g. Musa et al. 1987).



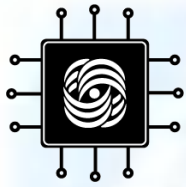
Basic Execution Time Model

- **Basic Execution Time Model**

- Интенсивность отказов $\lambda(\tau)$ за время τ :

$$\lambda(\tau) = \lambda_0 e^{-\frac{\lambda_0}{v_0} \tau}$$

- где λ_0 начальная интенсивность и v_0 ожидаемое количество отказов.

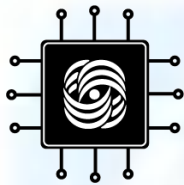


Logarithmic Poisson execution time model

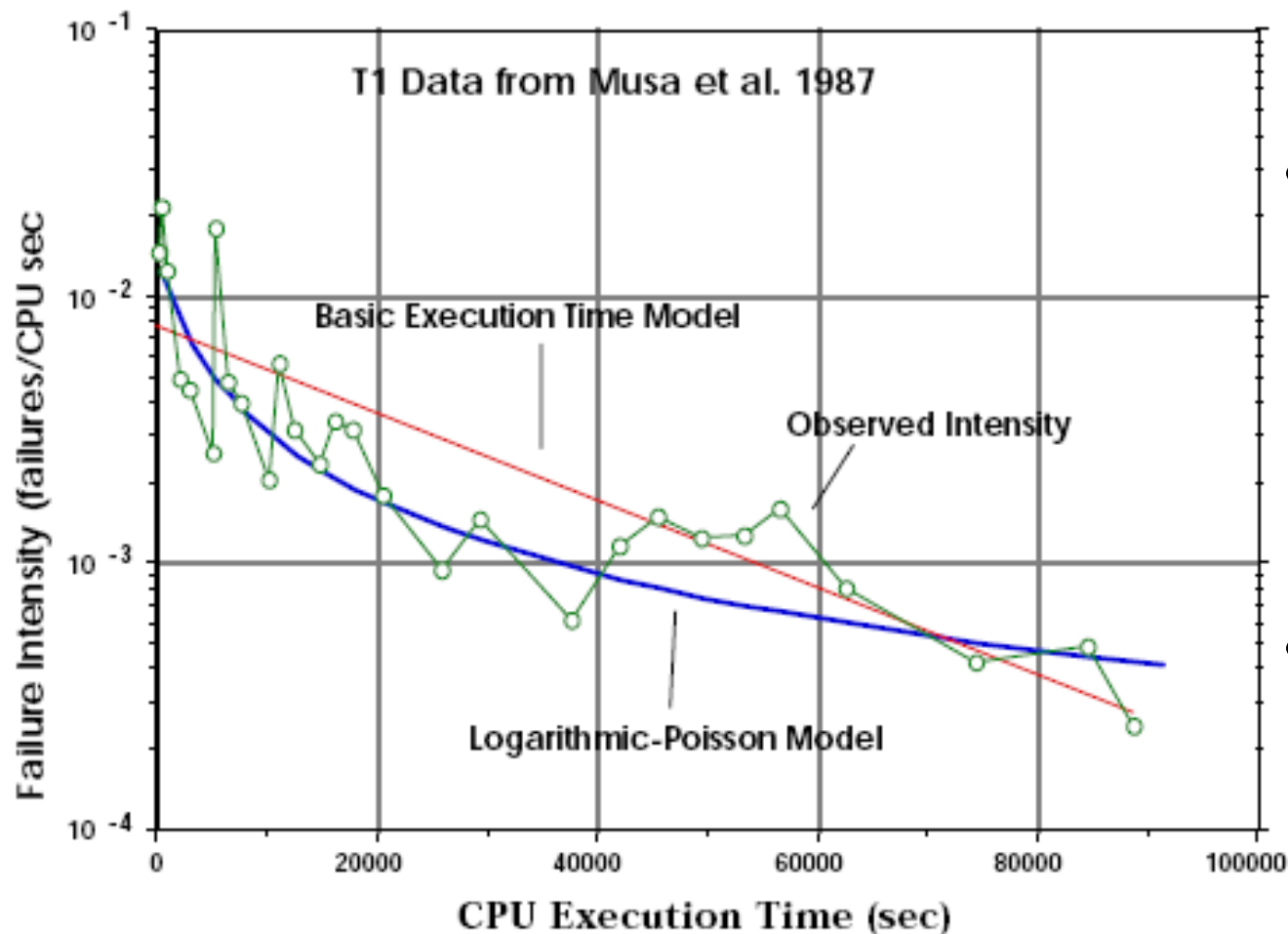
$$\lambda(\tau) = \lambda(\mu) = \lambda_0 \left(1 - \frac{\mu}{v_0}\right)$$

- где $\mu(\tau)$ среднее время между откзами к времени τ .

$$\mu(\tau) = v_0 \left(1 - e^{-\frac{\lambda_0}{v_0} \tau}\right)$$



Пример (2)

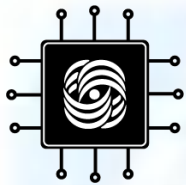


- В этом случае Logarithmic-Poisson Model лучше приближает the Basic Execution Time Model.
- В некоторых других проектах BE model приближает лучше чем LP модель.



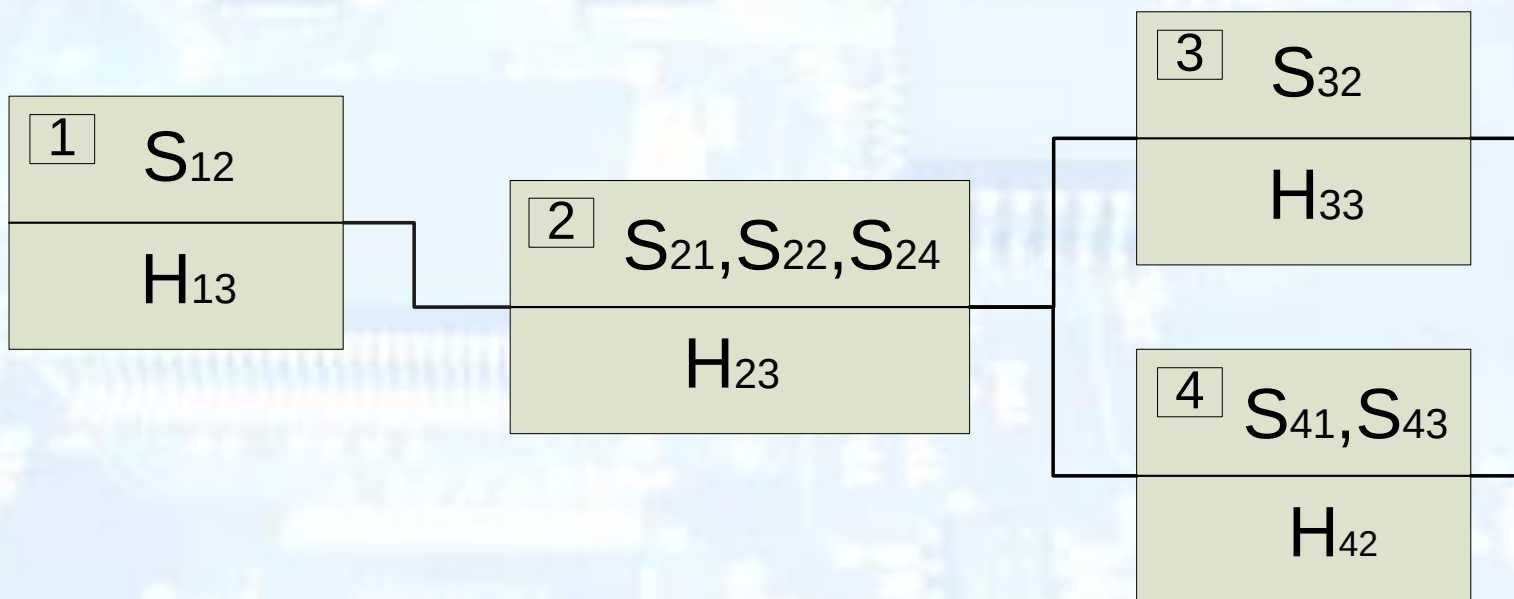
Выводы

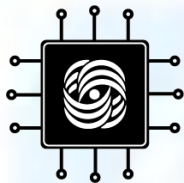
- Нет универсальной модели
- Измерения очень важны для нахождения корректной модели
- Результаты тестирования необходимо использовать и не один раз
- Модели надёжности ПО не так просты как описаны здесь 😊



Неформальная постановка задачи оптимизации надёжности ВС

Пример вычислительной системы:





Классическая постановка задачи оптимизации надёжности ВС

- *Дано:*

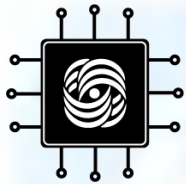
- N – количество подсистем
- Ch_{ij} , Cs_{ij} – стоимость использования аппаратного/программного компонента j для подсистемы i
- Rh_{ij} , Rs_{ij} – надёжность аппаратного/программного компонента j для подсистемы i
- Prv , Pd , $Pall$ – вероятности отказа нескольких версий программных компонентов, вероятность отказа схемы принятия решений, вероятность отказа сразу всех версий программного компонента

- *Необходимо найти:*

- Оптимальный набор компонентов и МОО, на котором $R_{RTES} \rightarrow \max$

- *Ограничения:*

- $C_{RTES} < Cost$



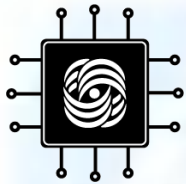
Принятые ограничения

- Все компоненты являются неремонтируемыми
- Моменты появления отказов для аппаратуры статистически независимы
- Все аппаратные компоненты ВС являются активными
- Количество доступных версий для всех компонентов фиксировано
- Интенсивность отказов постоянно



Особенности задачи

- NP-трудная
- Аргументы и значения функции надежности являются дискретными
- Многоэкстремальная область допустимых решений
- Область допустимых решений несвязна
- Функция надежности нелинейна
- Функция стоимости линейна
- Каждый модуль вычислительной системы имеет свой доступный набор механизмов обеспечения отказоустойчивости



Методы решения задачи

- Динамическое программирование
- Точные методы
- Алгоритмы муравьиных колоний
- Генетические алгоритмы
- Имитация отжига
- Иммунные алгоритмы
- Прочие эвристики
- Поиск с отсечением



Цели

- Отказоустойчивая компьютерная система должна предоставлять сервисы в случае ошибок
- Отказы могут возникать, так как дефекты присутствуют в компонентах системы
- Система с отказоустойчивостью надёжней системы без отказоустойчивости, но больше затрачивается ресурсов на обеспечение отказоустойчивости



Проблемы ...

- Традиционные подходы к отказоустойчивости в аппаратных системах основываются на копировании с использованием особенностей отказов физических компонентов.
- Большинство аппаратных методов отказоустойчивости не могут быть применены напрямую в ПО, где почти все ошибки разработчика.



Пример

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^6 x_i \cdot y_i$$

$$\mathbf{x} = (10^{20}, 1223, 10^{24}, 10^{18}, 3, -10^{21})$$

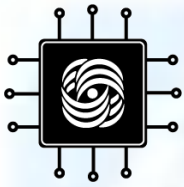
$$\mathbf{y} = (10^{30}, 2, -10^{26}, 10^{22}, 2111, 10^{19})$$

- Правильный ответ должен быть 8779
- Но при обычной реализации возвращается 0



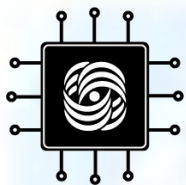
История ...

- **Защитное программирование:**
 - ad-hoc методы минимизирующие ущерб который может возникнуть из-за ошибок
- **Метод двух версий:**
 - Создание двух независимых версий ПО и запуск их. По любому расхождению в версиях включается триггер

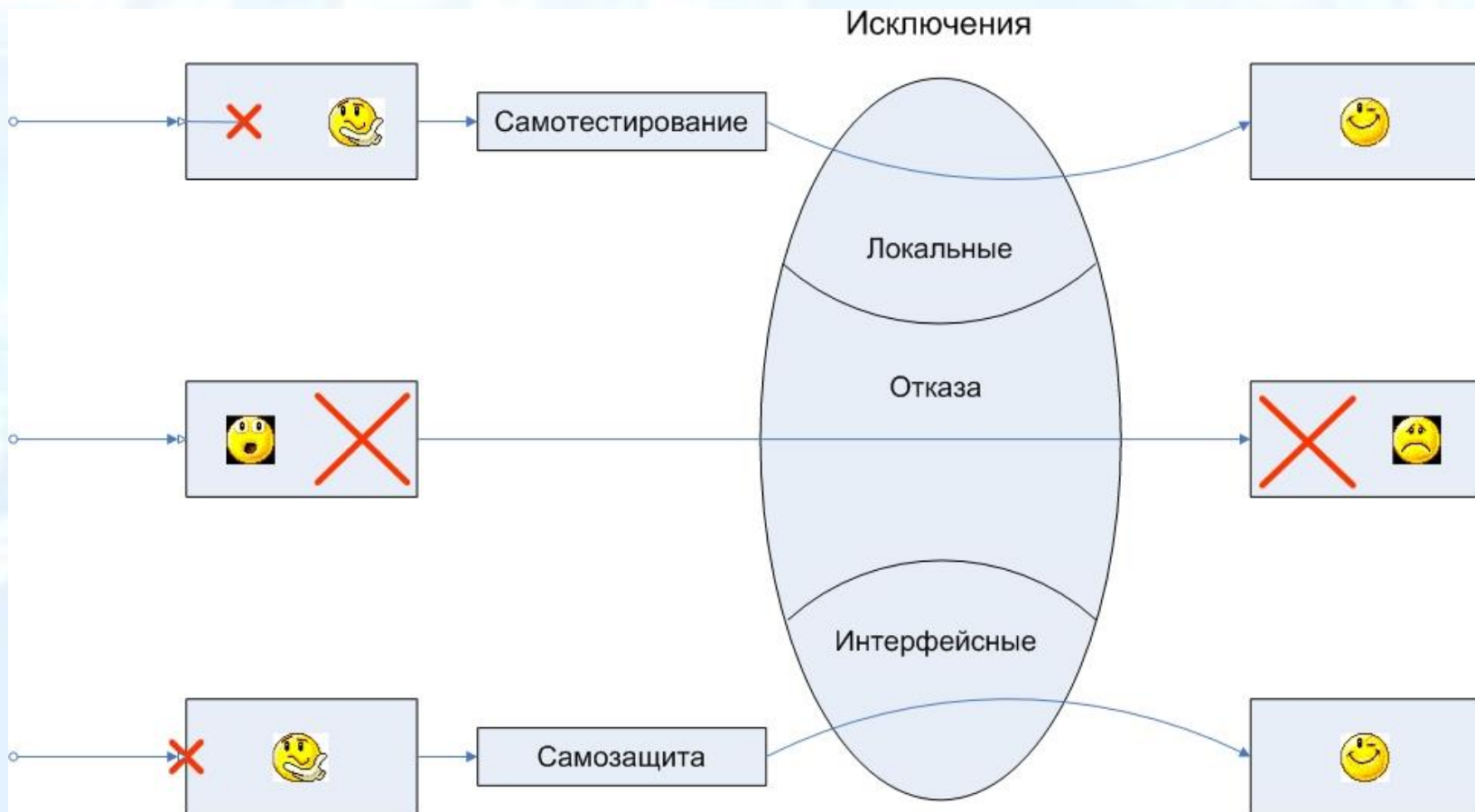


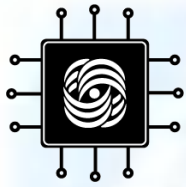
Методы обеспечения отказоустойчивости

- Обнаружение ошибок
- Диагностическое тестирование
- Изоляция ошибок
- Маскировка ошибок
- Корректирование ошибок
- Устранение ошибок



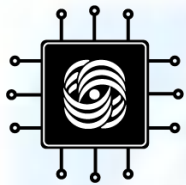
Требования к компонентам





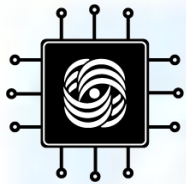
Организация резервирования

- Общие подходы
 - Организация глобального времени
 - Выделение изолированных регионов
 - Отказы в разделяемых компонентах
- Программное резервирование
 - Пространственное резервирование
 - Временное резервирование
 - Функциональный сдвиг во времени
 - Информационный сдвиг во времени
- Аппаратное резервирование
 - Пространственное резервирование
 - Кодирование



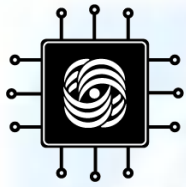
Механизмы обнаружения ошибок

- Программные
 - Приемочные тесты
 - Отказоустойчивые алгоритмы
 - Проверки
 - Временные
 - Кодовые
 - Реверсные
 - Семантические
 - Структурные
- Аппаратные
 - Диагностическое тестирование



Приемочные тесты

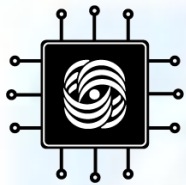




Отказоустойчивые алгоритмы. Попарное тестирование

- Диагностическое тестирование
- Изоляция ошибок
- Маскировка ошибок
- Корректирование ошибок

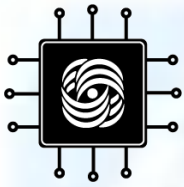




Отказоустойчивые алгоритмы. Голосование

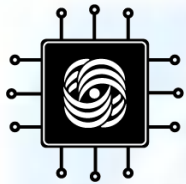
- Диагностическое тестирование
- Изоляция ошибок
- Маскировка ошибок
- Корректирование ошибок





Механизмы устранения ошибок

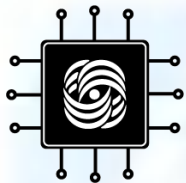
- Программные
 - Одноверсионное программирование
 - Контрольная точка и перезапуск
 - Парные прогоны
 - Многоверсионное программирование
 - Восстановление блоками
 - N-версионное программирование
 - N-самотестируемое программирование
- Аппаратные
 - Резервирование компонент
 - Переконфигурирование системы



Схемы голосования

ГОЛОСОВАТЕЛЬ сравнивает результаты двух и более функционально эквивалентных компонентов ПО и определяет корректный результат

- Схемы голосования:
 - Большинство
 - Консенсус
 - 2-из-N

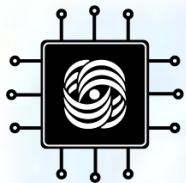


Контрольная точка и перезапуск

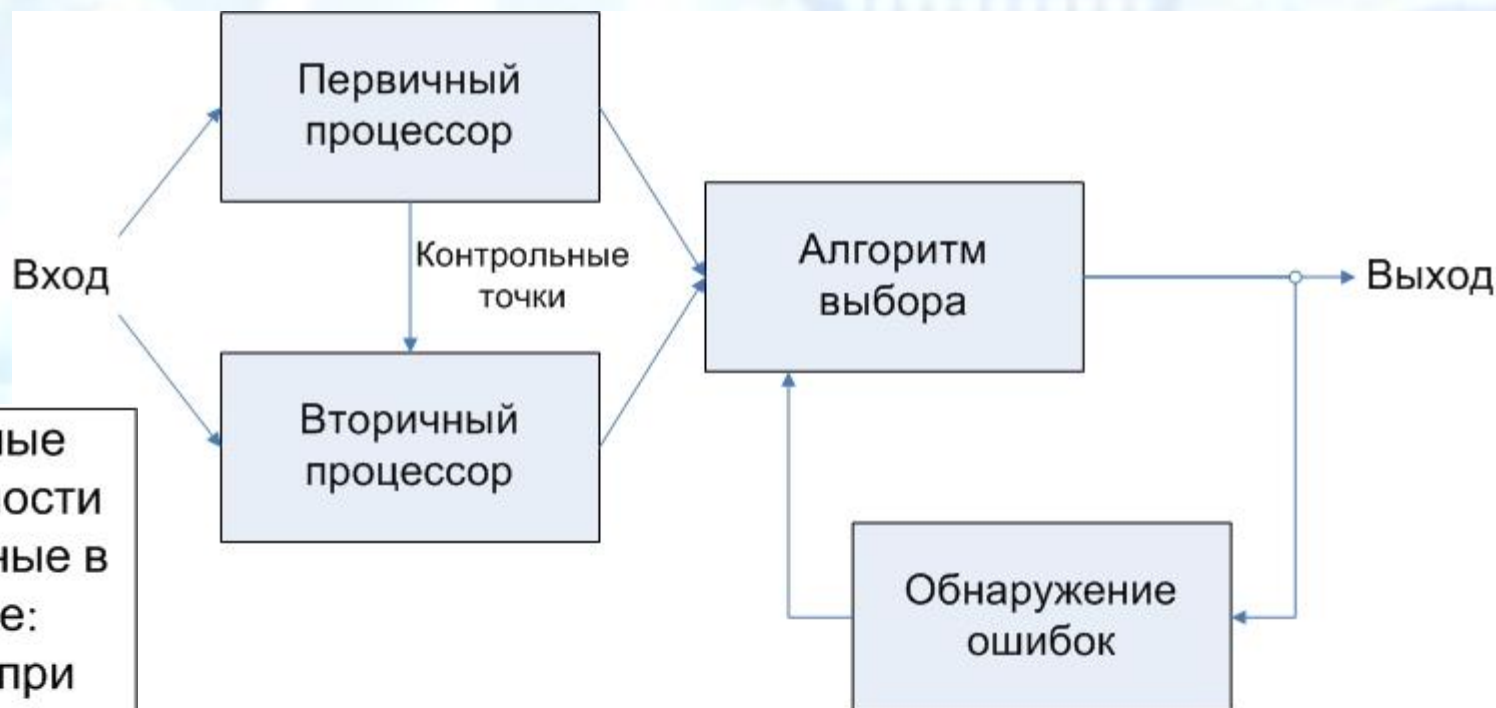


∇ временные неисправности:

- врем. э/м помеха
- приработка устройства

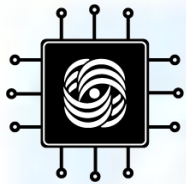


Парные прогоны



∇ временные
неисправности
+ постоянные в
аппаратуре:

- ошибка при
разработке
аппаратуры
- ошибка
ввода

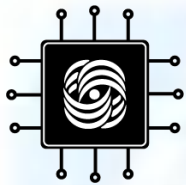


Восстановление блоками

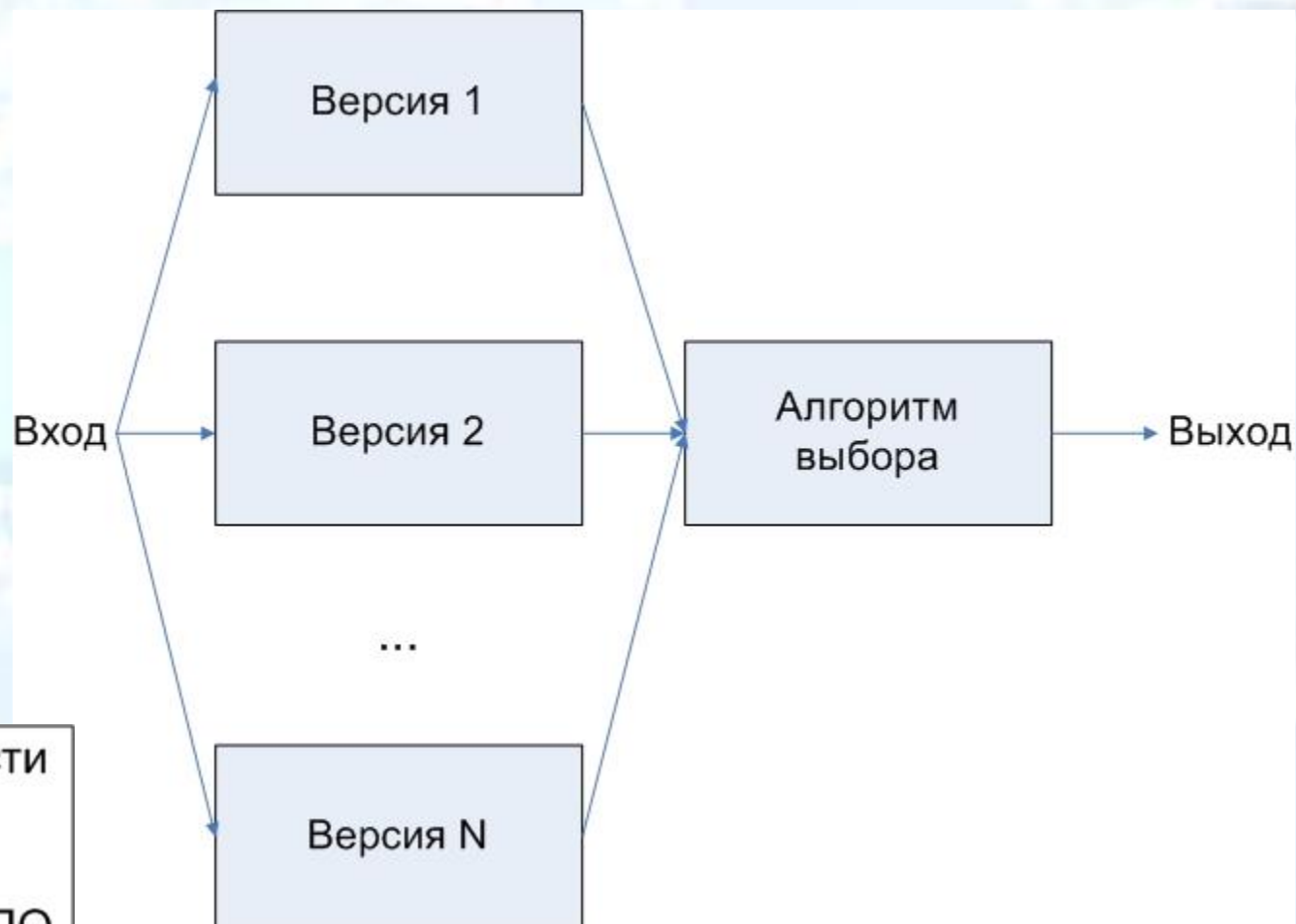


∇ временные неисправности + постоянные в ПО:

- ошибка при разработке ПО
- врем. э/м помеха

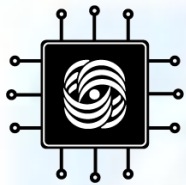


N-версионное программирование

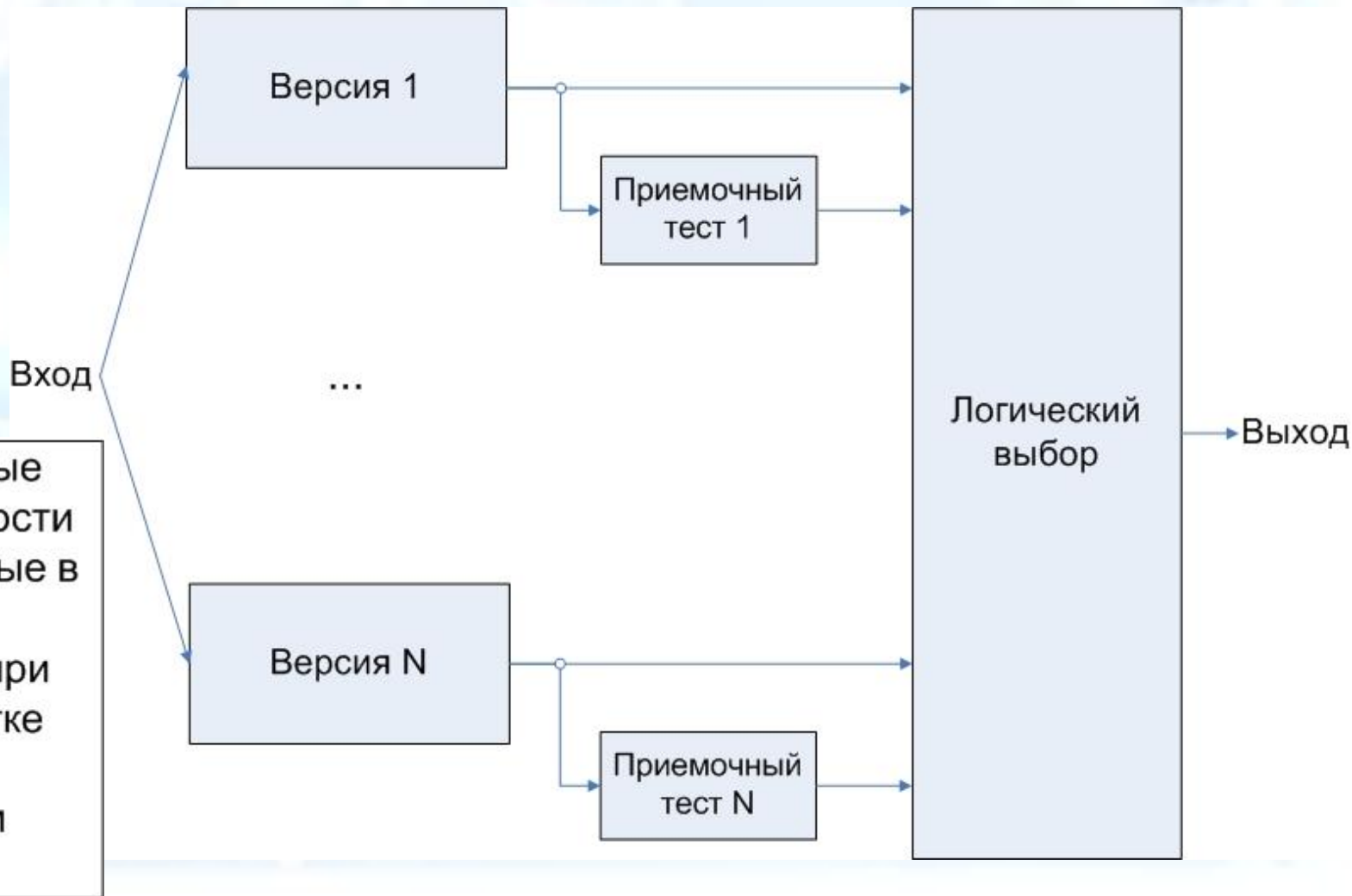


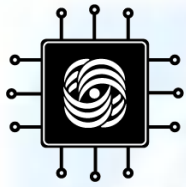
∀ неисправности
в ПО:

- ошибка при разработке ПО
- ошибка ввода



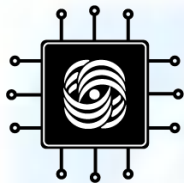
N-самотестируемое программирование



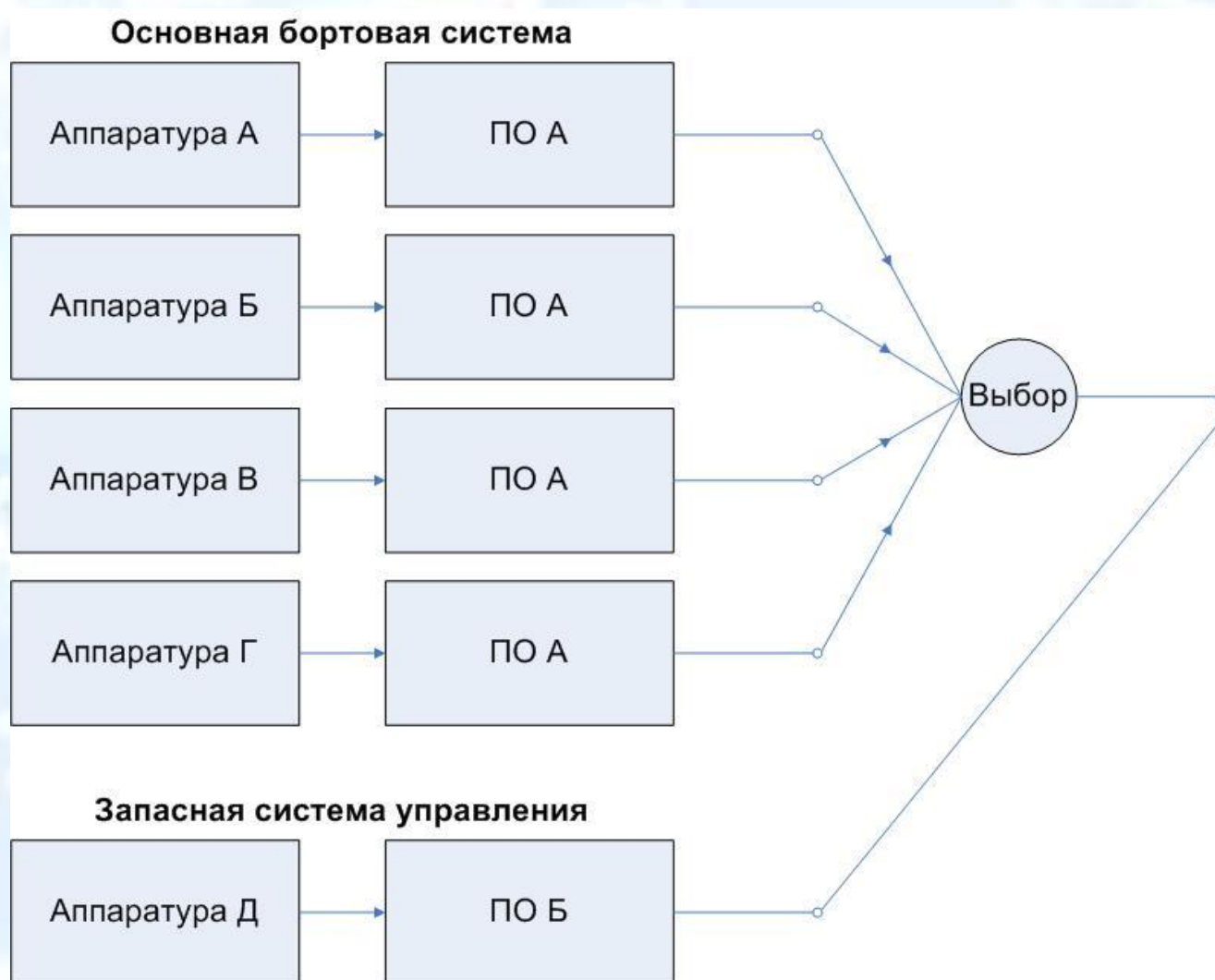


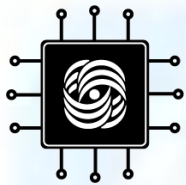
N-версионное программирование. Примеры

Год	Проект и спонсор	Кол-во версий	Требуемое разнообразие	ЯП
1975-76	Текстовый редактор Software Engineering	27	Разработчики	PL/1
1977-78	Решение ДУ Software Engineering	16	Разработчики + 3 алгоритма	PL/1
1979-83	Аэропортовый планировщик NSF	18	Разработчики + 3 спецификации	PL/1
1984-86	Отказоустойчивый датчик NASA	5	Разработчики	Pascal
1986-88	Беспилотная посадка самолета Sperry Flight Systems	6	Разработчики + 6 языков	Pascal, Ada, Modula-2, C, Prolog, T

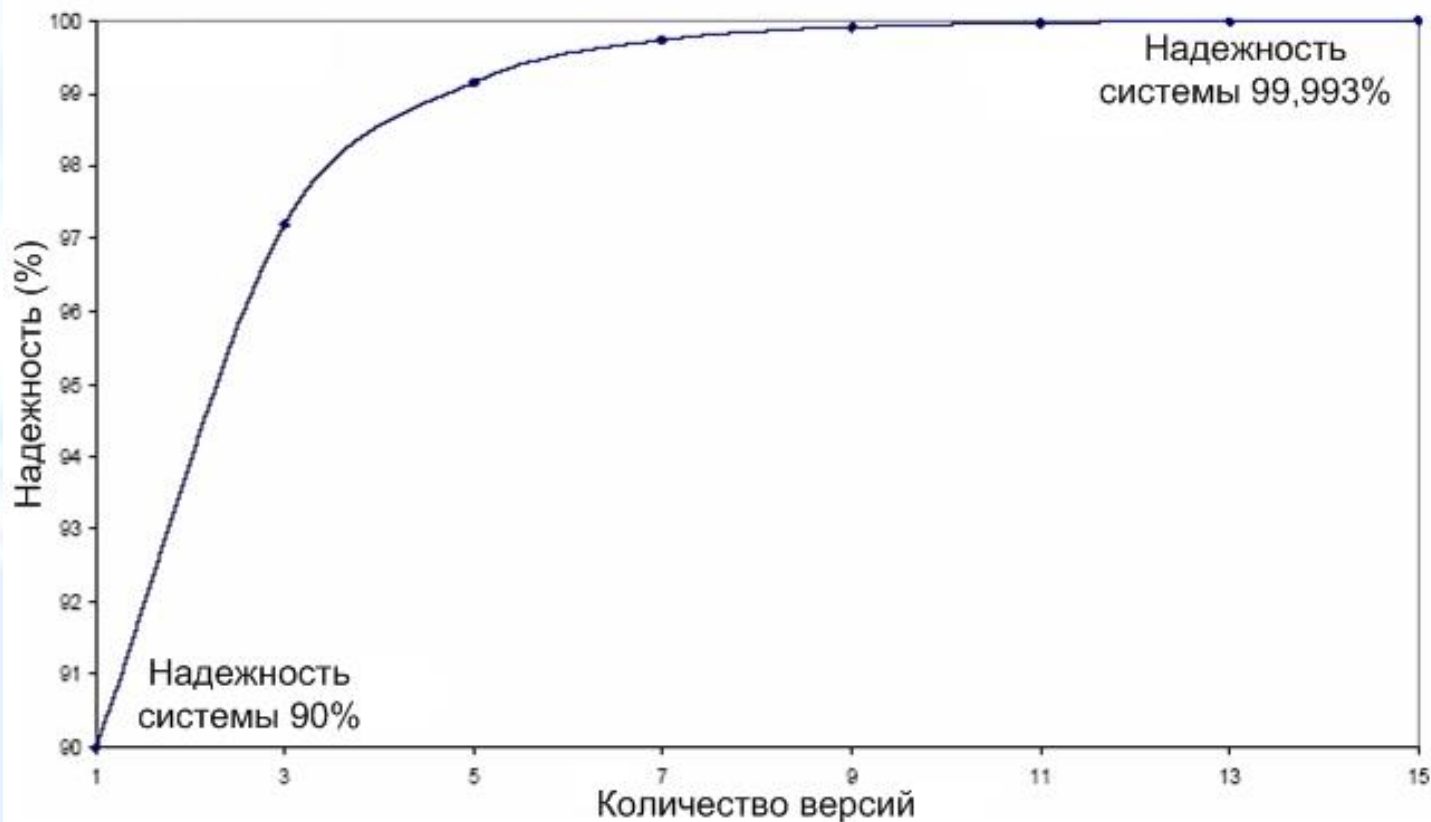


Космический шаттл





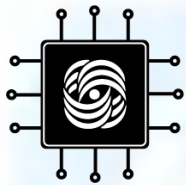
Зависимость надежности от количества версий





Заключение

- Необходимо корректное покрытие приёмочными тестами для обнаружения ошибок
- Часто невозможно быстро проверить корректность процедуры (например, для алгоритма “сортировки” это также сложно как сам алгоритм “сортировки”).
- Проверки часто могут быть очень ресурсоёмкими.
- ***Возможность проявления ненайденных ошибок не должно быть определяющим***



Спасибо за внимание!



НАДЁЖНОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Лекция 10:

Определение необходимого уровня надёжности и построение функционального среза

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

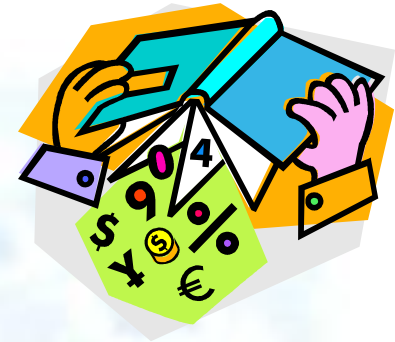


О чём этот курс

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



План лекции



- Определение необходимого уровня надёжности программ
- Классы тяжести отказов
- Интенсивность отказов
- Стратегии уменьшения числа отказов
- Операция, функция, прогон и его типы
- Функциональный срез
- Разработка функционального среза



Эпиграф

- Да, все голосуют за качество, но если оно стоит лишнюю копейку, вы начинаете быстро познавать настоящее отношение к качеству со стороны тех, кто платит.*

***Том Демарко
и Тимоти Листер***

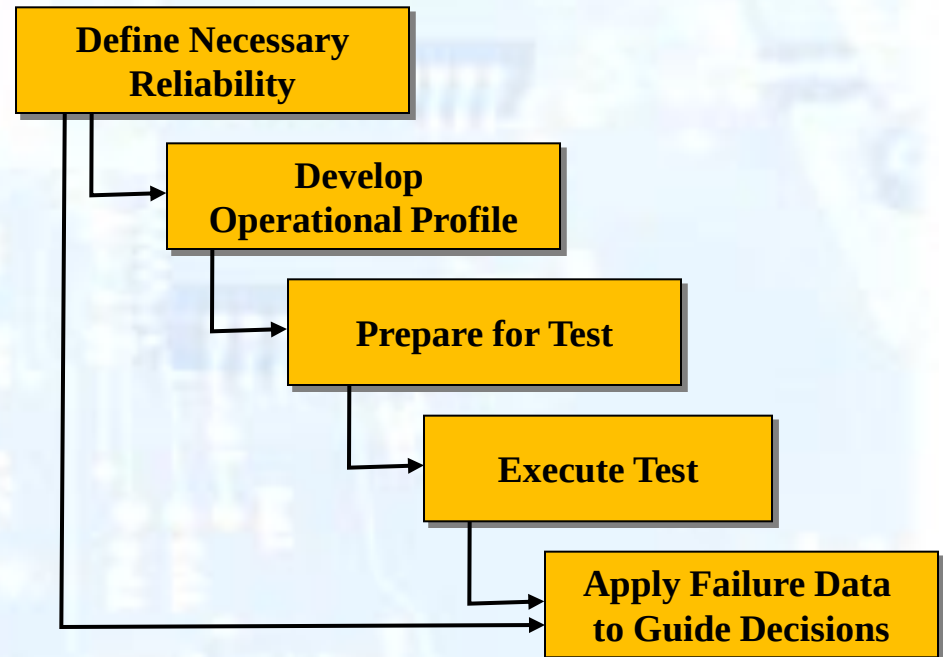




SRE: процесс

- 5 шагов в SRE процессе:

- **Определить необходимый уровень надежности**
- Разработать функциональный срез
- Подготовить для тестирования
- Выполнить тестирование
- Проанализировать данные об ошибках и использовать их для принятия решений



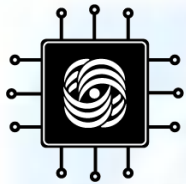


Как определить
требуемый уровень
надёжности программы?



Надёжность и Риск

- Требуемый уровень надёжности зависит от риска. Большой риск требует большего уровня надёжности
 - Литература:
 - Software Testing Fundamentals: Methods and Metrics
 - Marnie L. Hutcheson
- ISBN:047143020X John Wiley & Sons



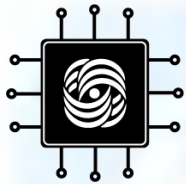
Как найти нужный уровень надёжности:

- 1) Разбить возможные отказы по классам тяжести
- 2) Установить целевую интенсивность отказов для каждой тестируемой системы
- 3) Выбрать единый масштаб для всех тестируемых систем
- 4) Найти текущий показатель интенсивности отказов
- 5) Выбрать стратегию для достижения поставленного уровня интенсивности



1. Классы тяжести отказов

- Отказы различаются между собой по уровню их воздействия на тестируемую систему
- Каждый **класс тяжести отказов** объединяет отказы с одинаковым влиянием на пользователя [по заданным критериям]
- Примеры критериев классификации:
 - Стоимость ущерба, угроза жизни человека
- Тяжесть отказа $\neq$ сложность отказа
- Тяжесть отказа может зависеть от времени его наступления

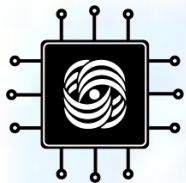


Критерии классификации тяжести отказов

■ **Стоимость**

- Какой ущерб способен нанести отказ в терминах стоимости устранения, восстановления, простоя системы, ...
- Классы тяжести по стоимости ущерба обычно различаются на один порядок
- На практике достаточно использовать 4 группы классов

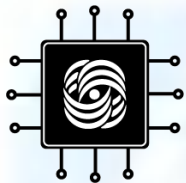
Класс тяжести	Стоимость (\$)
1	> 100,000
2	10,000 – 100,000
3	1,000 – 10,000
4	< 1,000



Критерии классификации тяжести отказов

- **Функционирование системы**
- Включает такие факторы как потерю данных, время простоя системы, возможность её восстановления, ...

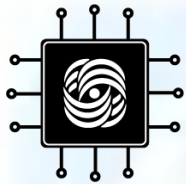
Класс тяжести	Ущерб функционированию
1	Перерыв в работе сервиса
2	Ошибки в работе сервиса
3	Неудобства, требующие срочного исправления
4	Неудобства в работе, исправление которых может подождать



Критерии классификации тяжести отказов

- **Окружающая среда**
- Может привести к нанесению вреда экологии, исчезновению видов растений и животных, ...
- Широко применяется в атомной и химической промышленности

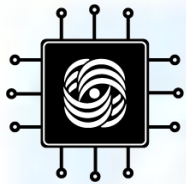
Класс тяжести	Ущерб экологии
1	Значительный и невозстановимый
2	Значительный и частично восстановимый
3	Незначительный, но невозстановимый
4	Незначительный и восстановимый



Критерии классификации тяжести отказов

- **Жизнь людей**
- Отказ может привести к нанесению вреда окружающей среде, потере человеческих жизней
- Применяется в аэрокосмической, медицинской, военной и других отраслях

Класс тяжести	Определение
1	Возможны людские потери
2	Значительные повреждения здоровья людей
3	Незначительные повреждения здоровья людей
4	Восстановимые нарушения здоровья



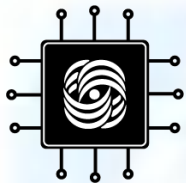
Как определить уровень тяжести отказа?

- На основе опыта: попросить пользователей, заказчиков и разработчиков и сравнить с аналогичными продуктами
- Перечислить всевозможные факторы ущерба, вызванные отказом системы
- Сузить список до наиболее важных факторов
- Некоторые факторы могут быть трудноизмеримыми [репутация компании]



Противоречивые оценки

- Пользователи и разработчики могут иметь разные представления о тяжести отказов
- Мнения отдельных групп должны быть согласованы до установки показателя целевой интенсивности отказов
- Разрешить противоречия в тяжести отказов часто помогает исследование аналогичных продуктов



Документирование уровней тяжести отказов

*Способ
использования*

*Критерии
классификации*

Список возможных отказов

Класс 1

Класс 2

Класс 3

Класс 4

Стоимость

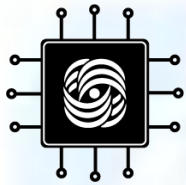
**Функционирование
системы**

**Человеческие
жизни**

Окружающая среда

**Другое (указать
отдельно)**

Классы тяжести отказов определяются независимо по каждому критерию



2. Целевая интенсивность отказов

- Отражает ожидаемое число «багов», оставшихся в продукте после завершения его разработки
- Альтернативный способ измерения надёжности



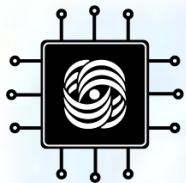
Целевая интенсивность отказов

- Интенсивность отказов обычно определяется как число случившихся отказов системы за заданное время (или аналогичными метриками):
 - 3 сигнала тревоги за 100 часов работы.
 - 5 отказов за время печати 1000 документов.
- Интенсивность отказов системы – сумма интенсивностей отказов всех компонентов системы



Как установить целевую интенсивность отказов?

- Оценка основывается на опыте и субъективно зависит от проекта
- Зависит от компромисса между качеством продукта (временем и стоимостью разработки) и его функциональностью
- **Правило большого пальца:** Устанавливай целевую интенсивность отказов исходя из расчётной итоговой стоимости проекта



Как установить целевую интенсивность отказов?

- Типичные целевые интенсивности отказов для разных проектов:

Стоимость отказа	Целевая интенсивность отказов (λ)	Время между отказами
> 1,000,000,000 \$	1 в 1,000,000,000 часов	114,000 лет
> 1,000,000 \$	1 в 1,000,000 часов	114 лет
> 1,000 \$	1 в 1,000 часов	6 недель
> 100 \$	1 в 100 часов	100 часов
> 10 \$	1 в 10 часов	10 часов
> 1 \$	1 в час	1 час



Целевая интенсивность отказов и надёжность

- Зависимость между целевой интенсивностью и надёжностью

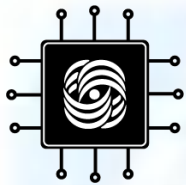
$$\lambda = \frac{-\ln R}{t} \quad \text{или} \quad \lambda \approx \frac{(1-R)}{t} \quad \text{для} \quad R \geq 0.95$$

λ - интенсивность отказов

R - надёжность

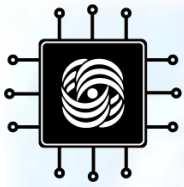
t - единица измерения (время, ...)

- При $R = 0.992$ за 8 часов, $\lambda = 0.001$



Надёжность & Интенсивность отказов

<i>Надёжность для 1 часа работы</i>	<i>Интенсивность отказов</i>
0.36800	1 отказ в час
0.90000	105 отказов в 1000 часов
0.95900	1 отказ в день
0.99000	10 отказов в 1000 часов
0.99400	1 отказ в неделю
0.99860	1 отказ в месяц
0.99900	1 отказ в 1000 часов
0.99989	1 отказ в год



Целевая интенсивность отказов и доступность

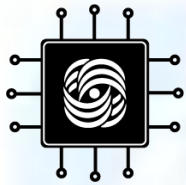
- Зависимость между целевой интенсивностью отказов и доступностью:

$$A(t) = \frac{1}{1 + t_m \lambda(t)} \quad or \quad \lambda(t) = \frac{1 - A(t)}{t_m A(t)}$$

- t_m λ - интенсивность отказов
- время простоя на ошибку

- Если система должна быть доступна 99% времени работы и время простоя 6 минут, тогда целевая интенсивность отказов равна

23 1 отказ в 10 часов



Целевая интенсивность отказов и среднее время безотказной работы

- Ещё одно определение доступности:

$$A = \frac{MTTF}{MTTF + MTTR} = \frac{MTTF}{MTBF}$$

$$\lambda = \frac{1 - \frac{MTTF}{MTTF + MTTR}}{\frac{MTTR \times MTTF}{MTTF + MTTR}} = \frac{1}{MTTF}$$

λ

ИНТЕНСИВНОСТЬ ОТКАЗОВ

$MTTR$

среднее время восстановления после отказа

$MTTF$

среднее время безотказной работы



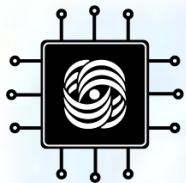
Целевая интенсивность отказов и вероятность первого отказа

- Вероятность первого отказа $z(t)$ – вероятность сбоя в заданном временном интервале при условии отсутствия сбоев до его наступления
- Вероятность первого отказа 0.05 означает, что первый отказ с вероятностью 5% произойдёт в заданном интервале и не раньше его наступления
- Для экспоненциального распределения $z(t) = \lambda$



Целевая интенсивность отказов и прибыль

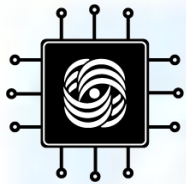
- Расчёт основывается на опыте аналогичных систем с помощью сравнения их основных качественных характеристик и отношению к ним пользователя
- Сравниваются тенденции изменения приоритетов развития компромисса между прибылью и интенсивностью отказов



Пример

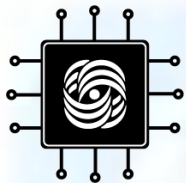
- Совет: выбирайте диапазон на границе высоких прибылей

Интенсивность отказов	Прибыль
1000	0,6
500	1,5
250	5,8
100	17,2
50	21,4
25	19,8
10	16,3



Надёжность vs. Доступность

- Зачем задавать надёжность, если доступность более интуитивна и проще для понимания?
- Доступность субъективна для каждого пользователя и обычно существуют способы изменения доступности системы без изменения её внутренней надёжности
- **Пример:** дублирование серверов DNS повышает доступность системы, но не повышает надёжность серверного ПО



Разрабатываемое ПО

- *Создаваемый продукт – лишь часть системы*





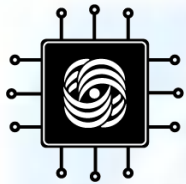
3. Выбор общего масштаба

- Для разных целевых интенсивностей отказов различных частей проекта могут использоваться разные масштабы:
 - Целевая интенсивность отказов системы
 - 30 отказов на 1 миллион операций
 - Среднее время между отказами ОС
 - 3,000 отказов на 10 миллионов операций
 - Среднее время между отказами оборудования
 - 1 на 30 часов работы
- Нужно выбрать общий масштаб



Как найти целевую интенсивность отказов разрабатываемого ПО?

1. Установить общесистемную целевую интенсивность отказов
2. Перевести системы измерения интенсивностей отказов компонентов в единую систему измерения
3. Вычесть из целевой интенсивности системы расчётную интенсивность отказов приобретённых компонентов
4. Вычесть расчётную интенсивность отказов окружения (ОС и интерфейсных систем), в котором будет работать разработанное ПО
5. Остаток – целевая интенсивность для разрабатываемого ПО



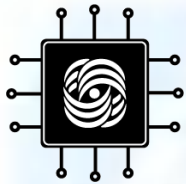
Вычисление целевой интенсивности отказов для разрабатываемого ПО (1)

•Пример 1:

- Общесистемная целевая интенсивность отказов 100 отказов на 1 миллион операций
- Интенсивность отказов «железа» 0.1 отказа в час
- Число отказов ОС на 100,000 операций 0.4 отказа в час

•Целевая интенсивность ПО

- 95 отказов на 1 миллион операций

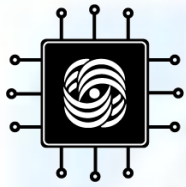


Вычисление целевой интенсивности отказов для разрабатываемого ПО (2)

• **Пример 2: БД на Win 2K:**

- Общая целевая интенсивность отказов
 - 30 отказов на 1 миллион операций
- Среднее время между отказами для Win 2K
 - 3,000 часов на 10 миллионов операций
- Среднее интенсивность отказов «железа»
 - 1 отказ в 30 часов
- Интенсивность отказов других подсистем
 - 9 отказов на 1 миллион операций

• Какова целевая интенсивность отказов для разрабатываемого ПО?



Вычисление целевой интенсивности отказов для разрабатываемого ПО (3)

$$\lambda_{os} = \frac{1}{MTTF} = 1/3000$$

$$\lambda_{hardware} = \frac{1}{30} = 100/3000$$

$$\lambda_{os} + \lambda_{hardware} = 101/3000 \quad \text{for } 10,000,000 \text{ transactions}$$

$$\lambda_{other} = 90 \quad \text{for } 10,000,000 \text{ transactions}$$

$$\lambda_{total} = 191 \quad \text{for } 10,000,000 \text{ transactions}$$

$$\lambda_F = 300 \quad \text{for } 10,000,000 \text{ transactions}$$

therefore

$$\lambda_{developed\_software} = 300 - 191 = 109 \quad \text{for } 10,000,000 \text{ transactions}$$



4. Стратегии уменьшения числа отказов программ

- 4 основных стратегии:
 - Недопущение
 - Устранение
 - Невосприимчивость
 - Прогнозирование



Недопущение отказов

- Избежание отказов **по построению.**
- **Действия:**
 - Перепроверка требований
 - Пересмотр архитектуры и очистка кода
 - Использование стандартов (ISO 9000-3, etc.)
 - Использование CASE-средств со встроенными механизмами контроля
- **Эффективность применения:**
 - Пропорция отказов после выполнения указанных действий не меняется.



Устранение отказов

- Ошибки в коде устраняются с помощью **верификации** и **валидации**.
- **Действия:**
 - Семантическая проверка кода
 - Тестирование
- **Эффективность применения:**
 - Уменьшение числа ошибок в коде.
 - Уменьшение числа отказов ПО.



Тестирование vs. Инспектирование

- Инспекция – детальное изучение спецификаций, архитектуры проекта, его кода, ...
- Инспектирование эффективнее тестирования более чем в 20 раз
- Пересмотр кода позволяет найти в 2 раза больше ошибок, чем его тестирование
- 80% ошибок находятся во время инспектирования
- Инспектирование уменьшает стоимость разработки в 10 раз



Можно ли заменить тестирование инспектированием?

- **НЕТ.** Иногда инспектирование не позволяет получить информацию, доступную с помощью тестирования:
 - Реальное поведения системы
 - Показатели производительности, удобства использования, ...
 - Показатель надёжности системы



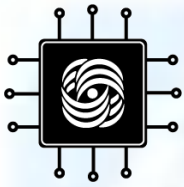
Невосприимчивость к отказам

- Обеспечение сервиса несмотря на отказы ПО с помощью **избыточности**
- **Действия:**
 - Проектирование и обеспечение избыточности
- **Эффективность применения:**
 - Уменьшение числа отказов системы из-за её избыточности



Прогнозирование отказов

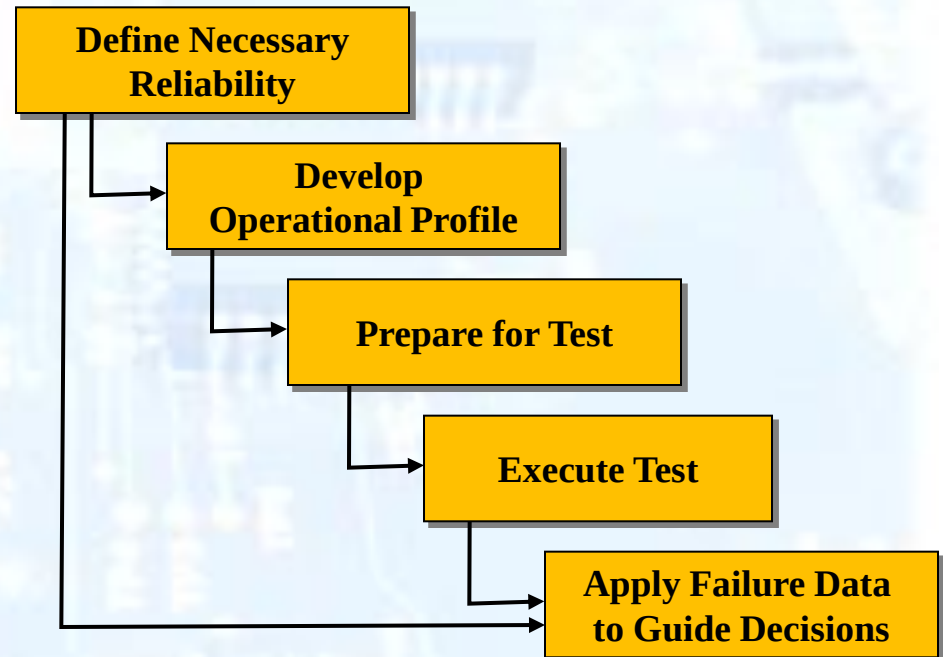
- Построение **оценки** вероятности возникновения отказов
- **Действия:**
 - Построение модели надёжности
 - Сбор информации об отказах
 - Анализ результатов
- **Эффективность применения:**
 - Уменьшение интенсивности возникновения ошибок за счёт управления надёжностью

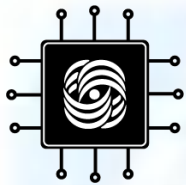


SRE: процесс

- 5 шагов в SRE процессе:

- Определить необходимый уровень надежности
- **Построить функциональный срез**
- Подготовиться к тестированию
- Выполнить тестирование
- Проанализировать данные об ошибках и использовать их для принятия решений

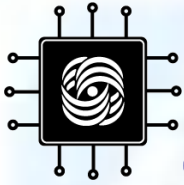




Профиль

- **Профиль** - набор альтернативных сценариев вместе с вероятностями их выполнения
- **Пример:**
 - Если сценарий А выполняется в 30% случаев, сценарий В – в 20%, а С – в 50%, то профиль имеет следующий вид:

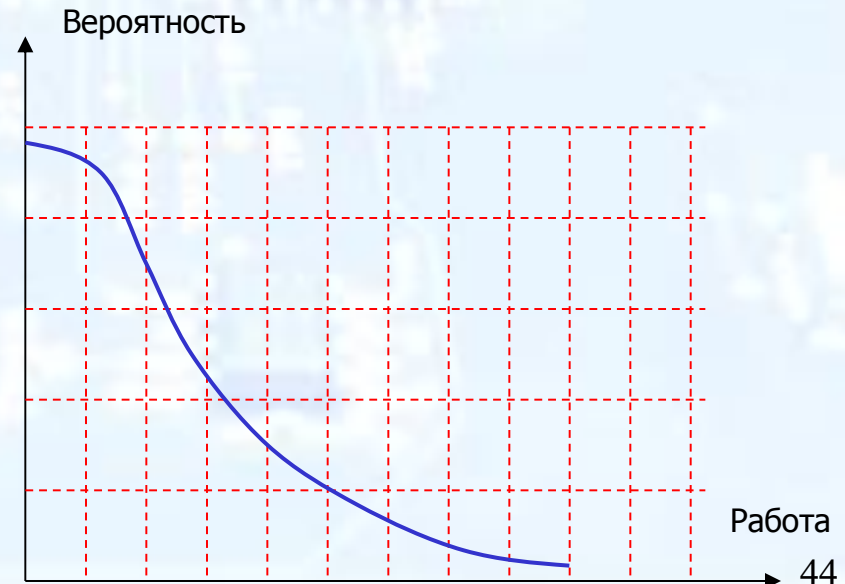
А	0.3
В	0.2
С	0.5

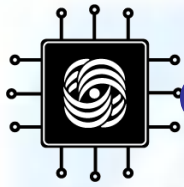


Функциональный срез

- Функциональный срез – полный набор сценариев выполнения вместе с их вероятностями (во время работы заданного ПО)
- Функциональный срез описывает распределение событий, которые происходят во время работы программы
- Функциональный срез программы отражает то, как она будет использоваться на практике

Функциональный срез зависит от пользователя, времени, используемой функциональности т.д.





Функциональный срез

Функциональный срез – полный набор операций (имя и частота) и вероятностей их ИСПОЛЬЗОВАНИЯ

Пример:

Operation	Occurrence rate (operations per hour)
Phone number entry	10,000
Add subscriber	50
Delete subscriber	50
Process voice call, no pager, answer	18,000
Process voice call, no pager, no answer	17,000
Process voice call, pager, answer	17,000
Process voice call, pager, answer on page	12,000
Process voice call, pager, no answer on page	10,000
Process fax call	15,000
Audit section of phone number database	900
Recover from hardware failure	0.1
Total	100,000



Почему это полезно?

- Профиль функционирования программы даёт информацию о том, как её будут использовать, и позволяет разработчикам сфокусироваться на решении действительно актуальных задач
- С помощью профиля разработчики могут:
 - Увеличить эффективность разработки и тестирования.
 - Сделать тесты более реалистичными.
- С точки зрения эффективного управления разработку всегда нужно начинать с изучения приложений программы и составления соответствующего профиля функционирования



Операция (1)

- Неформальное определение:
 - Операция представляет собой задачу, выполняемую программой в заданном окружении. Причём результат выполнения задания должен быть обозримым для пользователя
- Более формальное определение:
 - Операция — **крупное логическое краткосрочное** задание, выполнение которого **значительно отличается** от других заданий



Операция (2)

- *Крупное* означает, что операция должна относиться к функциональным требованиям, предъявляемым к разрабатываемой программе, а не к подзадаче, решаемой в её реализации
- Операция – *логическая* концепция в том смысле, что она может включать набор программных, аппаратных и пользовательских действий



Операция (3)

- *Кратковременность* означает, что при нормальном функционировании программы, она может выполнять тысячи операций в час
- *Значительно отличающееся выполнение* означает, что с высокой долей вероятности отказ внутри операции не произойдёт при выполнении других операций



Прогон и его типы (1)

- **Прогон** - наименьшая часть работы, которая может быть вызвана внешним воздействием
- **Примеры:**
 - Звонок в телекоммуникационной системе
 - Команда в интерактивном приложении
 - Транзакция в интернет-магазине
 - etc.



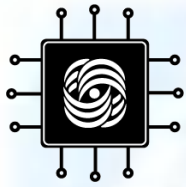
Прогон и его типы (2)

- Прогонны связаны со **входным состоянием**, включающим полный набор **входных переменных** (любых данных, полученных из вне)
- Входные переменные могут быть:
 - Частично неизвестными на стадии проектирования программы
 - Скорее логическими, чем физическими [например адрес в памяти]
- Прогонны с одинаковым входным состоянием принадлежат к прогона одного **типа**



Операции vs. Функции (1)

- **Функция**, заданная требованиями к программе, потенциально может стать **операцией** или набором операций на стадии её разработки
- **Функция** – **группа типов прогона**, заданных требованием к программе и наблюдаемых пользователем
- **Операция** – **группа типов прогона** заданных во время разработки программы и рассматриваемых с точки зрения её проектирования

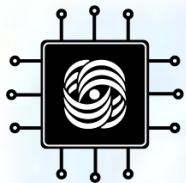


Операции vs. Функции (2)

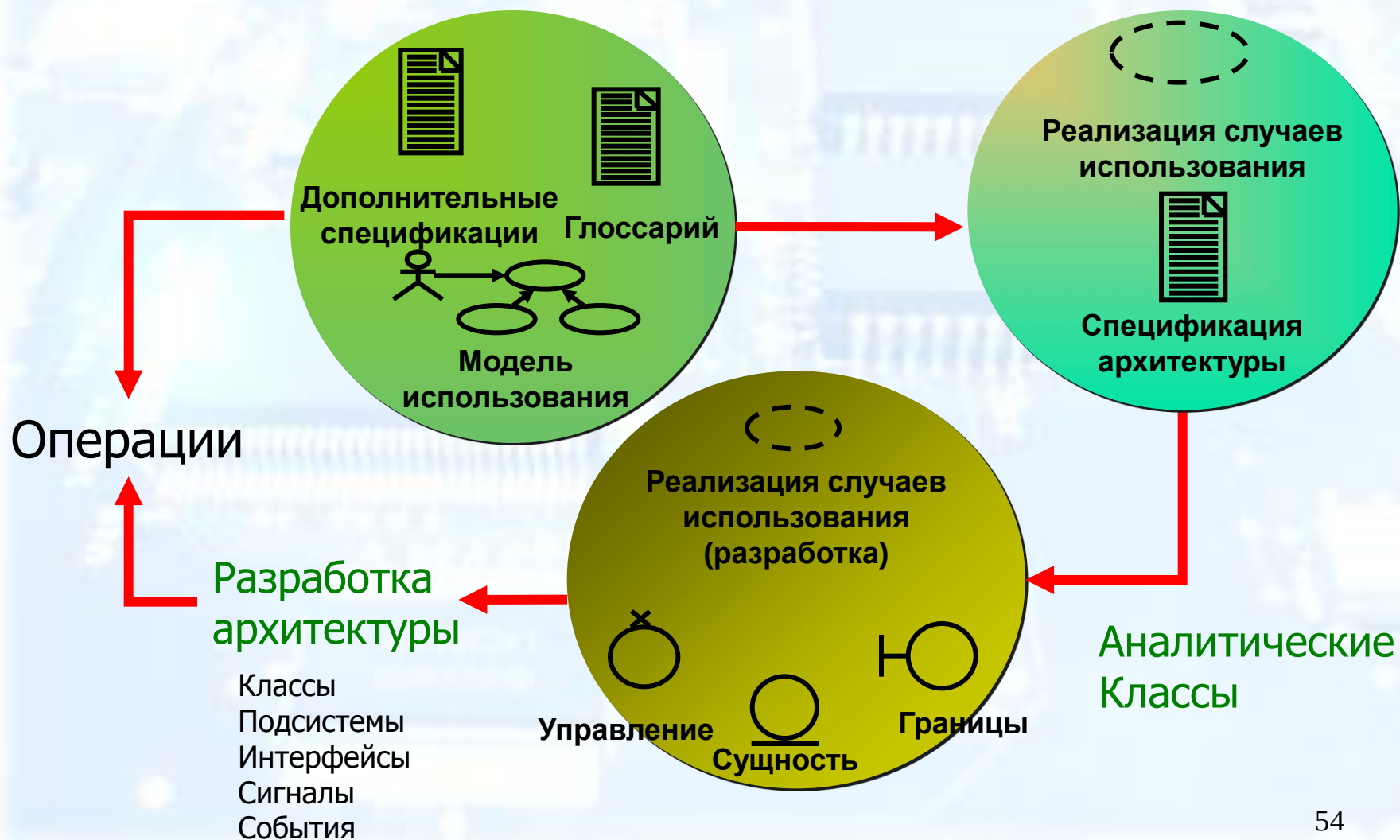
Анализ	Проектирование
Концентрация на понимании проблемы	Концентрация на понимании решения
Идеальный случай	Близко к реальному коду
Поведение	Операции и атрибуты
	Жизненный цикл объектов

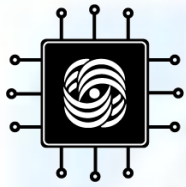
Пример:

Перемещение объекта в памяти (функция) на стадии определения требований может превратиться в операции **создания**, **копирования** и **удаления** на стадии разработки



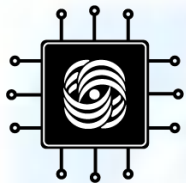
Как искать операции?





Представление

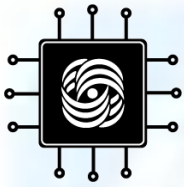
- Описывает функциональный профиль:
 - Табличное представление
 - Графическое представление



Representation: Tabular

- **Табличное представление** включает список имён операций и вероятностям их ВОЗНИКНОВЕНИЯ

Command	Transactions per month	Occurrence probability
relocate	780	0.0153
remove	90	0.0017
add -s staff	70	0.0014
update		0.0010
add -s manager	25	0.0005
add -s secretary	5	0.0001

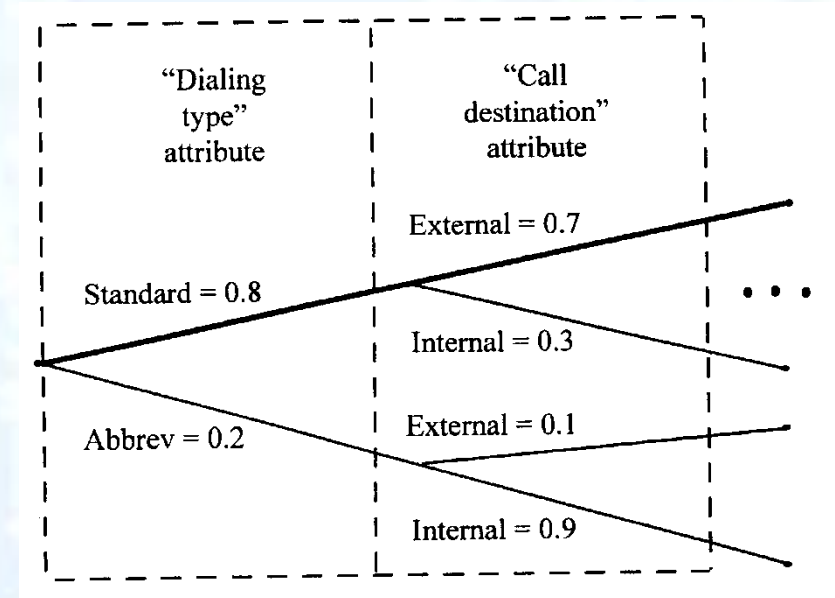


Графическое представление

■ Графическое представление

СОСТОИТ ИЗ:

- Узлов – атрибутов операций
- Ветвей – значения атрибутов
- Вероятность использования: ассоциируется с ветвями



Графическое представление может быть получено из табличного и наоборот.



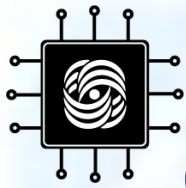
Режим функционирования

- **Режим функционирования** – шаблон использования системы и/или условий окружения которые нужно различать во время тестирования, так как они могут приводить к отказам
- **Пример:** пиковые и свободные режимы работы телекоммуникационной системы
- Одна и та же операция может использоваться в разных режимах функционирования с разными вероятностями
- **Профиль функционирования системы** должен включать все важные режимы функционирования



Сколько это стоит?

- Средняя стоимость построения профиля функционирования для проекта среднего размера (10 разработчиков, 100 тыс. строк кода, интервал разработки в 18 месяцев) составляет около 1 человеко-месяца
- Один и тот же профиль функционирования может быть использован (с минимальными изменениями) разными версиями одной программы без увеличения стоимости



Построение профиля функционирования

1. Выявить режимы функционирования
2. Выявить инициатора операции
3. Выбрать представление
4. Построить список функций и операций
5. Определить частоты использования каждой операции
6. Задать вероятности использования каждой операции, разделив частоты на общее число вызовов



1. Режим функционирования (1)

- Определить все возможные режимы функционирования и выбрать только наиболее важные на основании таких критериев как:
 - Время (время года, день недели, время дня)
 - Тип пользователя (покупатель или пользователь)
 - Опыт пользователя (новичок или эксперт)
 - Условия работы (режим перегрузки, обычный)
 - Ограничения функциональности
 - Критичность (режим остановки атомных станций)



1. Режим функционирования (2)

■ Примеры:

- Время (часы пика и простоя телефонной станции)
- Различные типы пользователей (пользователь, системный администратор, ...)
- Опыт пользователя (использование различных наборов команд новичками и экспертами в системе редактирования видео)
- Ограничения функциональности (такие операции как вызов справки в банкомате или защита соединения в транзакционной системе должны быть доступны всегда)



2. Инициаторы операций

- Возможные инициаторы
(субъекты из схем случаев использования):
 - **Пользователи систем** (любой, кто инициирует событие, включая системного администратора и пользователей)
 - **Внешние системы** (другие системы, взаимодействующие с разрабатываемой системой)
 - **Собственные подсистемы** (задачи поддержки и администрирования: аудит, резервное копирование данных, ...)



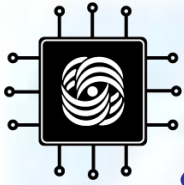
3. Выбор представления

- Используйте табличное представление если каждой операции соответствует небольшое количество атрибутов
- Используйте графическое представление если большинство операций описывается несколькими атрибутами
- На практике может быть удобным использование смешанного представления или конвертации одного представления в другое



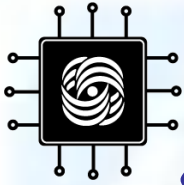
Пример представления

- Пусть для разрабатываемого приложения выполняется следующее:
 - 80% используют систему для личных нужд, 20% - для работы.
 - 20% пользователей подписаны на бесплатные обновления в течение года, 5% - в течение 3х лет. Остальные на обновление не подписаны.
 - 90% пользователей установило обновление безопасности, остальные 10% этого не сделали.
- Каким будет функциональный профиль системы?



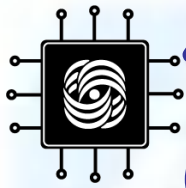
4. Список операций

- **Разбейте каждый режим функционирования на отдельные операции.**
 1. Число операций
 2. Явные и неявные операции
 3. Изначальный список операций
 4. Переменные окружения
 5. Финальные список операций



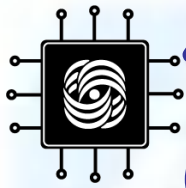
4.1 Число операций

- Стоимость подсчёта надёжности и тестирования прямо пропорциональна общему числу операций
- Обычно от нескольких десятков до сотен
- Число операций увеличивается вместе с размером проекта, числом режимов функционирования, числом ключевых входных переменных, ...
- Обычно лучше оставить только те операции, которые покрывают все значительно отличающиеся виды обработки



4.2 Явные или неявные операции (1)

- **Ключевая входная переменная** - входная переменная сразу для нескольких операций, чьё значение важно для возможности отличать эти операции между собой
- **Неявный профиль** выражается как множество подпрофилей, каждый из которых включает ключевую входную переменную и соответствующую ей вероятность
- **Явный профиль** – множество наборов значений для всех ключевых переменных
- Обычно профиль легче выразить в неявной форме



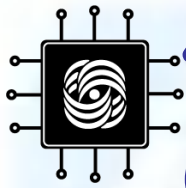
4.2 Явные или неявные операции (2)

■ Пример: неявный профиль

Входная переменная А		Входная переменная В	
значение	вероятность	Значение	Вероятность
A1	0.6	B1	0.7
A2	0.3	B2	0.2
A3	0.1	B3	0.1

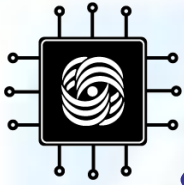
■ Пример: явный профиль:

A1B1	0.42	A1B2	0.12	A1B3	0.06
A2B1	0.21	A2B2	0.06	A2B3	0.03
A3B1	0.07	A3B2	0.02	A3B3	0.01



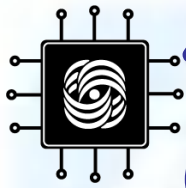
4.3 Изначальный список операций

- Изначальный список операций – пробный список, обычно разделённый по режимам функционирования и инициаторам операций
- Источники, которые могут быть полезны при составлении списка начальных операций:
 - Требования к системе(основной источник)
 - Случаи использования, модели работы
 - Диаграммы процесса работы
 - Черновики инструкций для пользователей
 - Предыдущие версии программы
 - Прототипы



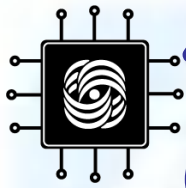
4.4 Переменные окружения

- Переменные окружения описывают условия, влияющие на процесс выполнения программы
- **Пример:** сетевой трафик, конфигурация оборудования и операционной системы
- Некоторые переменные окружения могут отображаться на режимы функционирования
- Перечислите все переменные окружения, которые могут привести к разной реакции программы, а затем выберите из них те, чьё влияние на поведение программы наиболее велико



4.5 Финальный список операций (1)

- Финальный список операций основан на сужении изначального списка операций с помощью следующих критериев:
 - Режим функционирования
 - Переменные окружения
 - Ключевые входные переменные
 - Зависимости между переменными
- Если между отдельными операциями существует связь (одна операция происходит, если происходит другая), то лучше объединить эти операции



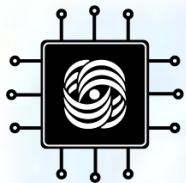
4.5 Финальный список операций (2)

- Между функциями и операциями могут появиться значительные отличия
- Некоторые функции (решающие несвязанные задачи) могут отображаться на одни и те же операции (реализующие разную обработку)
- Пример: Перемещение объекта в памяти (функция) может быть отображена в операции создания, копирования и удаления



5. Частоты операций (1)

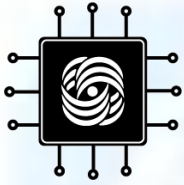
- Частоты использования операций строятся независимо для каждого режима функционирования на основе данных системного журнала, ручной сборки данных и опыта
- Если на этой стадии обнаруживаются связанные операции, их лучше объединить



5. Частоты операций (2)

■ Пример: Система управления звонками

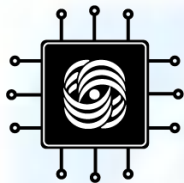
Operation	Occurrence rate (operations per hour)
Phone number entry	10,000
Add subscriber	50
Delete subscriber	50
Process voice call, no pager, answer	18,000
Process voice call, no pager, no answer	17,000
Process voice call, pager, answer	17,000
Process voice call, pager, answer on page	12,000
Process voice call, pager, no answer on page	10,000
Process fax call	15,000
Audit section of phone number database	900
Recover from hardware failure	0.1
Total	100,000



6. Вероятности операций

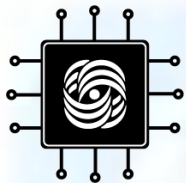
- Вероятности операций определяются с помощью деления индивидуальных частот операций на общее число их вызовов.

Operation	Occurrence probability
Process voice call, no pager, answer	0.18
Process voice call, no pager, no answer	0.17
Process voice call, pager, answer	0.17
Process fax call	0.15
Process voice call, pager, answer on page	0.12
Process voice call, pager, no answer on page	0.10
Phone number entry	0.10
Audit section of phone number database	0.009
Add subscriber	0.0005
Delete subscriber	0.0005
Recover from hardware failure	0.000001
Total	1.0

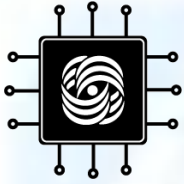


Документация

<i>Режим функционирования</i>	<i>Инициатор / ключевая входная переменная / Переменная окружения</i>	<i>Список операций</i>	<i>Частота / вероятность операции</i>
		1.	
		2.	
		3.	
		1.	
		2.	
		1.	
		2.	
		3.	
		4.	



Спасибо за внимание!



НАДЁЖНОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Лекция 11: *Подготовка к тестированию*

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



План лекции

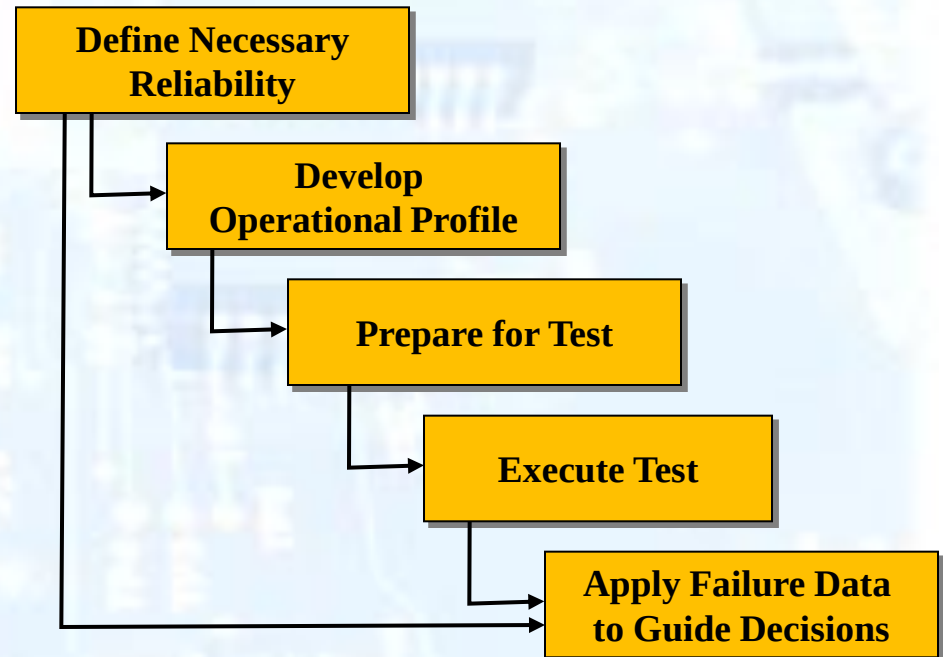
- Прямые и косвенные переменные
- Полевые испытания и регрессионное тестирования
- Что такое сценарий теста?
- Как управлять сценариями теста?
- Процесс тестирования
- Классы эквивалентности и граничные условия
- Как разрабатывать сценарии тестов?
- Подготовка успешного тестирования



SRE: процесс

- 5 шагов в SRE процессе:

- Определить необходимый уровень надежности
- Разработать функциональный разрез
- Подготовка для тестирования
- Выполнить тестирование
- Проанализировать данные об ошибках и использовать их для принятия решений





Основные понятия (1)

- **Прогон:** наименьшее задание, которое может быть вызвана внешним воздействием. Прогон ассоциируется с входным состоянием (набором входных переменных). Прогон с одинаковыми входными состояниями называются прогонами одного типа
- **Операция** – группа типов прогона с точки зрения её архитектуры программы
- **Тестовая процедура** – программа, которая устанавливает переменные окружения и вызывает случайные тестовые сценарии в случайные моменты времени



Основные понятия (2)

Типы тестов

- **Тест на поиск ошибок:** выполнение одной операции, максимально независимое от других операций
- **Нагрузочный тест:** тестирование на реальных данных с подсчётом показателей производительности
- **Регрессионный тест:** проверка функциональности после каждого значительного изменения в коде



Основные понятия (3)

- **Прямые входные переменные** явно управляют поведением программы
 - **Пример:** аргументы, введённые данные
 - Важны во время тестирования функциональности
- **Косвенные входные переменные** могут только влиять на результат операций
 - **Пример:** сетевой трафик, переменные окружения
 - Важны во время нагрузочного тестирования



Пример

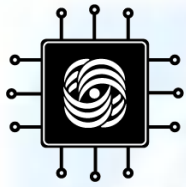
- Прямые и косвенные входные переменные и их эффект на результат операции

Direct	Indirect
Originator = 201 908 5577	Operational mode = prime hours
Forwarder = 908 555 1212	Database state: signified by time
Billing type = per call	Resource state: signified by time
Dialing type = standard	
Screening = yes	



Что такое тестовый сценарий?

- Тестовый сценарий – спецификация прогона в виде перечисления имён входных переменных и их значений
- Тестовый сценарий не зависит от режима функционирования программы
- Один и тот же тестовый сценарий может выполняться в разных режимах функционирования. Поэтому тестовый сценарий может соответствовать сразу нескольким прогонам с разным потенциально возможными отказами



Хороший тестовый сценарий

- Существует разумная вероятность обнаружения ошибки
- Не обладает избыточностью
- Лучше других подходит к сценарию
- Не слишком прост и не слишком сложен
- Делает ошибку в программе очевидной



Пример

- Тестовый сценарий определяется входным состоянием
- В теории (не обязательно на практике) можно провести прогоны для любого входного состояния

Direct
Originator = 201 908 5577
Forwarder = 908 555 1212
Billing type = per call
Dialing type = standard
Screening = yes



Тестовые сценарии и прогоны

- Спецификация косвенных входных переменных даёт тестовому сценарию недостающий контекст для того, чтобы тот стал прогоном
- Во время тестов функциональности влияние косвенных входных переменных на результат должно быть сведено к минимуму, чтобы убедиться в реальной надёжности системы
- Использование косвенных входных переменных важно во время нагрузочных тестов



Процедура

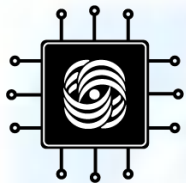
Состоит из двух этапов:

1) Подготовка тестовых сценариев (управление сценариями)

- Использование реальных данных и функциональных профилей

2) Подготовка тестовых процедур

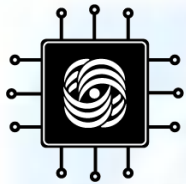
- Создание одной процедуры для каждого режима функционирования, используя его профиль и частоты использования операций



Управление тестовыми сценариями

Подготовка тестовых сценариев включает:

1. Подсчёт числа новых тестовых сценариев, которые нужно создать для данной версии программы
2. Распределение сценариев среди подсистем, которые необходимо тестировать
[уровень системы]
3. Распределение сценариев между новыми операциями внутри каждой подсистемы
[уровень операций]
4. Описание новых сценариев тестирования
5. Добавление новых сценариев



1. Подсчёт числа новых сценариев

■ **Методология:**

- Число тестовых сценариев зависит от:

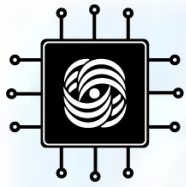
- ***Времени:***

(доступное время × доступные люди) /
(среднее время создания сценария)

- ***Стоимости:***

(доступный бюджет) / (средний расход на создание сценария)

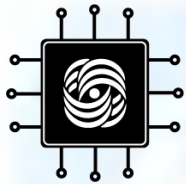
- Число выбирается как минимальное из рассчитанных значений



2. Распределение сценариев между подсистемами (1)

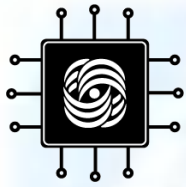
Большинство сценариев нужно ориентировать на поиск ошибок в разрабатываемом продукте





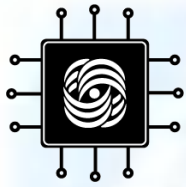
2. Распределение сценариев между подсистемами (2)

- Большинство сценариев нужно ориентировать на поиск ошибок в разрабатываемом продукте
- Создавайте больше сценариев для операций с большей вероятностью использования
- Не создавайте сценариев для приобретённых компонентов, если они уже использовались ранее. В противном случае создайте для них число сценариев, соответствующее риску их использования в контексте надёжности системы
- Если ваша программа значительно зависит от функций операционной системы и её надёжность неизвестна, то на её проверку необходимо выделить 20-30% сценариев



3. Распределение сценариев между операциями (1)

1. Конвертируйте графические представления профилей функционирования в табличное представление
2. Найдите новые **критические операции**, которые редко встречаются, и выделите для каждой из них приблизительное число требуемых сценариев
3. Выделите по одному сценарию для каждой новой редко встречающейся операции
4. Распределите оставшееся число сценариев между остальными новыми операциями



3. Распределение сценариев между операциями (2)

Найдите *пороговую вероятность: 0,5* делённые на число новых сценариев. Выделите по одному тесту для каждой новой нечастой операции

Распределите оставшиеся тестовые сценарии между оставшимися новыми операциями



Выделите по 2-4 сценария для каждой нечастой критической операции



3.1 Критические операции

- **Критическая операция** – операция, получение результата которой принципиально важно, а отказ выполнения ведёт к значительному ущербу (определённому в соответствии с классами тяжести отказа)
- Пример критической операции – остановка реактора атомной электростанции в случае его перегрева. Эта операция используется крайне редко, но очень важна
- Выделяйте только редкие критические операции, так как остальные будут проверены за счёт их использования сразу во многих сценариях выполнения программы



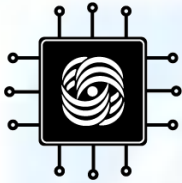
3.2 Нечастые операции

- ***Нечастые операции*** – те которые бы не проверялись в случае равномерного распределения тестовых сценариев из-за очень маленькой вероятности их использования
- Установите порог вероятности использования операции. Хороший выбор его значения равен отношению 0.5 к количеству новых тестовых сценариев
- Выделите по одному тестовому сценарию для всех операций с вероятностью ниже порогового значения



Пример (1)

- Число тестовых сценариев: 500
- Первая версия программы. Все сценарии новые
- Пусть в программе есть одна критическая операция. Выделяем для её проверки 2 сценария
- Рассчитываем пороговое значение вероятности использования: $0.5 / 500 = 0.001$
- Пусть число операций с вероятностью ниже пороговой равно двум. Выделяем по тесту на каждую из них
- Распределяем оставшиеся сценарии $500 - (2+2) = 496$ между остальными операциями в соответствии с их вероятностями использования



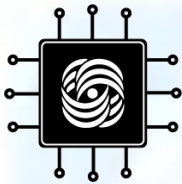
Пример (2)

- Вероятности операций для нормального режима функционирования

Operation	Occurrence probability
Process voice call, no pager, answer	0.18
Process voice call, no pager, no answer	0.17
Process voice call, pager, answer	0.17
Process fax call	0.15
Process voice call, pager, answer on page	0.12
Process voice call, pager, no answer on page	0.10
Phone number entry	0.10
Audit section of phone number database	0.009
Add subscriber	0.0005
Delete subscriber	0.0005
Recover from hardware failure	0.000001
Total	1.0

Нечастые операции с вероятностью ниже пороговой

Критическая операция



Пример (2)

- Вероятности операций для нормального режима функционирования

Operation	Occurrence probability
Process voice call, no pager, answer	0.18
Process voice call, no pager, no answer	0.17
Process voice call, pager, answer	0.17
Process fax call	0.15
Process voice call, pager, answer on page	0.12
Process voice call, pager, no answer on page	0.10
Phone number entry	0.10
Audit section of phone number database	0.009
Add subscriber	0.0005
Delete subscriber	0.0005
Recover from hardware failure	0.000001
Total	1.0

Распределяем
сценарии в
соответствии с
вероятностями

Нечастые
операции
с вероятностью
ниже пороговой

Критическая операция



4. Задание тестовых сценариев (1)

■ Простое определение:

- Тестовый сценарий – спецификация прогона в виде перечисления имён входных переменных и их значений

■ Лучшее определение:

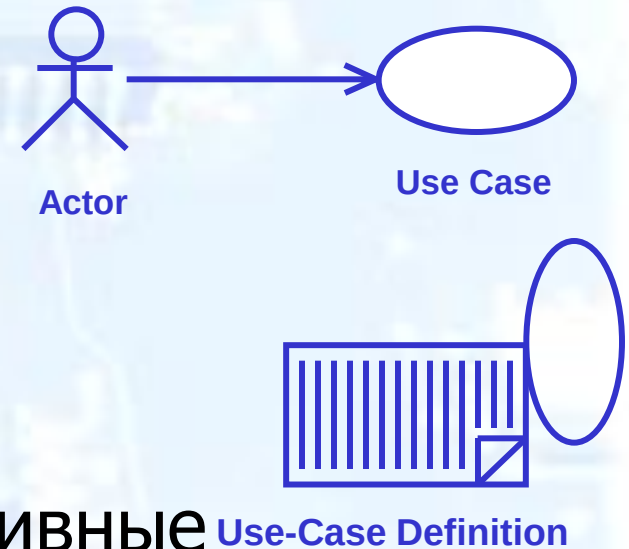
- Тестовый сценарий – случай использования, описанный в терминах тестовых входных данных, условий выполнения и ожидаемых результатов

Случаи использования должны быть описаны до начала разработки тестовых сценариев



4. Задание тестовых сценариев (2)

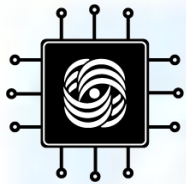
- Обычное описание случая использования включает:
 - Имя случая использования
 - Инициатор
 - Цель
 - Предусловия
 - Результат (Постусловия)
 - Детальное описание
 - Исключения и альтернативные пути выполнения





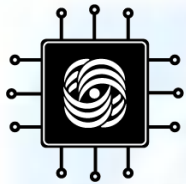
Тестовые сценарии

- Для каждого случая использования должно быть создано не меньше двух сценариев:
 - Один для штатного выполнения, другой – для отказа
- Может создаваться и большее количество сценариев, покрывающих исключения и альтернативные пути выполнения



5. Добавление нового сценария в набор тестов программы (1)

- Набор тестов обычно состоит из старых сценариев (оставшихся после выхода предыдущих версия программ) и новых сценариев (для добавленных функций)
- Для каждого режима функционирования программы необходимо разработать свой **профиль функционирования теста**, отличающийся от профиля программы:
 - Редким операциям уделяется больше внимания
 - Добавлены новые операции



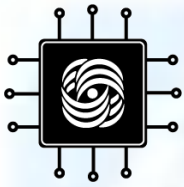
5. Добавление нового сценария в набор тестов программы (2)

- Для описания профиля функционирования теста для заданного режима начните с:
 - Режим и профиля функционирования (для первой версии программы)
 - Режим и профиля функционирования теста (для последующих версий программы)
- Измените вероятности операций с учётом новых редких критических операций
- Измените вероятности вновь добавленных операций с учётом взаимного влияния операций друг на друга



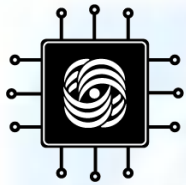
Тестовая процедура

- Тестовая процедура – программа, которая устанавливает переменные окружения и вызывает случайные тестовые сценарии в случайные моменты времени
- Подготавливается по одной тестовой процедуре для каждого режима функционирования



Как описать тестовый сценарий?

- Для заданной операции определите наборы значений прямых входных переменных, для которых программа будет вести себя по одному сценарию
- Тогда тестовый сценарий можно описать как:
 - Комбинацию значений входных переменных
 - Время их передачи на вход программе
- Если число выделенных комбинаций превосходит число доступных сценариев, то сценарии должны создаваться только для их части



Как создать тестовый сценарий

1. На основе пользовательских сценариев и случаев использования
2. На базе классов эквивалентности
 - Подготовка одного теста для класса операций
3. На базе граничных условий
 - Программа, которая не работает на простых условиях, не работает и на граничных условиях
4. На основе диаграммы изменения состояний



1. Классы эквивалентности

Группа сценариев «**эквивалентна**» если:

- Все они проверяют одну операцию
- Если один сценарий может найти ошибку, остальные скорее всего тоже смогут
- Если один сценарий не может найти ошибку, остальные – тоже не смогут
- Они используют одинаковый набор входных переменных и влияют на значения одинакового набора выходных переменных



Замечание (1)

- Не забывайте о классах эквивалентности для некорректных ВХОДНЫХ данных

Пример:

Пусть программа принимает на вход любое целое число между 1 и 99

Тогда существует 4 класса эквивалентности:

- Целые числа между 1 и 99 (корректный ввод)
- Любое целое число меньше 1 (некорректный ввод)
- Любое целое число больше 99 (некорректный ввод)
- Не целое число (некорректный ввод)



Замечание (2)

- Обратите внимание на диапазон значений переменных

Пример:

Пусть программа принимает на вход любое целое число между 1 и 99

Тогда существует 4 класса эквивалентности:

- Сделайте тест для очень больших значений
- Проверьте небольшие отрицательные значения, так как знак может обрабатываться некорректно



Замечание (3)

- Проверьте переполнение стека рекурсивных программ

Пример:

Пусть программа осуществляет рекурсивный обход дерева в глубину

- Проверьте корректность её работы на большом дереве, вытянутом в список



Замечание (4)

- Если входные данные принадлежат к одной группе, то все числа из этой группы должны включаться в один класс эквивалентности

Пример:

Пусть программа просит пользователя ввести имя

- Программа должна корректно принимать любые буквы алфавита в любом регистре
- Попробуйте ввести однобуквенное или очень длинное имя
- Проверьте корректность обработки символов не из таблицы ASCII



Замечание (5)

- Обращайте внимание на зависимости между входными переменными

Пример:

Путь программа принимает на вход значения трёх углов треугольника

- Попробуйте ввести углы, сумма которых не равна 180 градусам



2. Граничные условия

- Проверьте, корректны ли установленные на диапазоны граничные условия
- Проверьте, могут ли нестрогие знаки быть заменены на строгие и наоборот
- Проверьте корректность ограничения числа итераций циклов
- ...



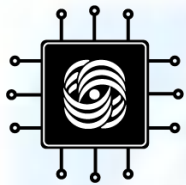
Примеры

Пусть программа ожидает ввода прописной буквы в диапазоне A-Z

- Проверьте корректность её работы на символах непосредственно перед 'A' и сразу после 'Z'

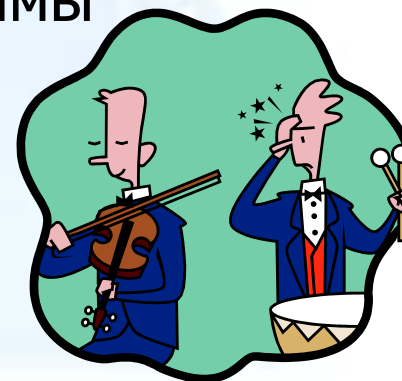
Пусть программа ожидает ввода строго заданного числа параметров

- Попробуйте ввести на один параметр меньше и больше



3. Визуальное изменение состояния программы

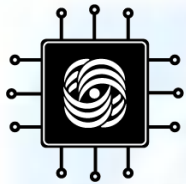
- Каждое взаимодействие с программой (установка входного параметра, выбор пункта из меню, ...) изменяет состояние программы
 - Проверьте все рутинные операции, которые будет совершать пользователь
 - Проверьте любые конфигурации, которые могут как-то повлиять на поведение программы
 - Выберите случайно несколько других операций





Документирование отказа

- Запишите полный набор входящих параметров, на которых программа отказала
- Создайте тестовый сценарий, способный воспроизвести отказ
- Найдите класс эквивалентных тестовых сценариев
- Созданный сценарий впоследствии может быть использован в качестве регрессионного



Что необходимо для успешного тестирования?

- Планирование тестирования
- Технологии тестирования
- Средства тестирования
- Управление тестированием
- Обучение построению тестов
- Эффективность тестов
- Контроль качества тестов
- Взаимодействие с пользователями

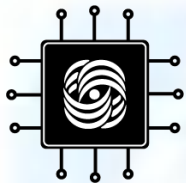
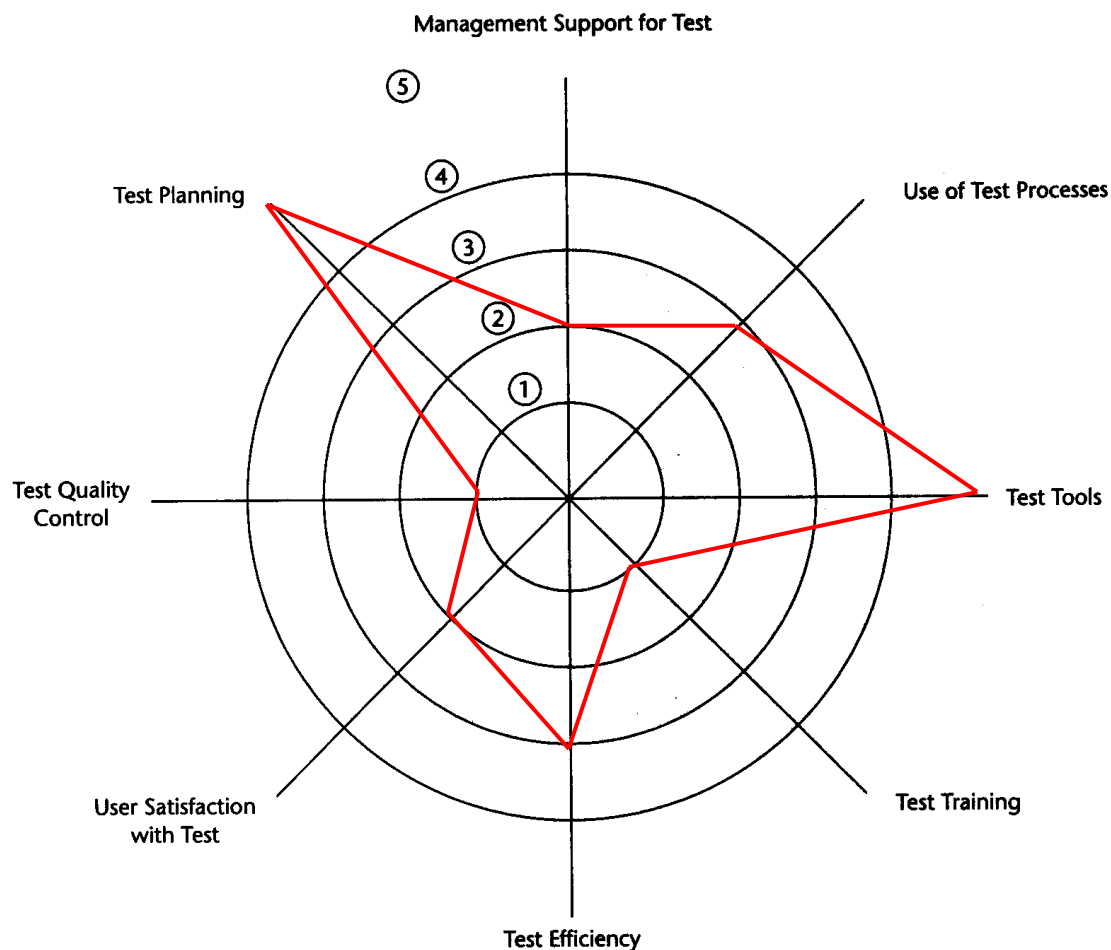
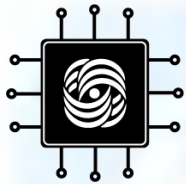


Диаграмма оценки тестирования

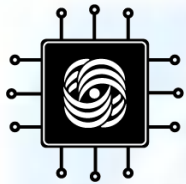
- Успешность тестирования оценивается на основании 8 параметров.





Планирование тестирования

- Соответствует ли план тестирования целям проекта и требованиям, которые к нему предъявляются?
- Вовлечены ли пользователи системы в процесс создания плана тестирования? Включает ли он рекомендации пользователей?
- Изменился ли план тестирования после изменения требований к программе?
- Содержит ли план описание требований к программному обеспечению? Согласны ли пользователи с этими требованиями?



Технологии тестирования

- Существует ли подробная документация по используемым технологиям тестирования? Ведутся ли тесты в соответствии с этой технологией?
- Покрывают ли используемые технологии тестирования весь необходимый спектр мероприятий?
- Ведётся ли разработка механизмов улучшения технологий тестирования? Участвует ли команда в их разработке?



Средства тестирования

- Используются ли автоматизированные средства для генерации и повторного использования тестовых данных?
- Правильно ли выбраны средства тестирования для данного программного обеспечения?
- Умеют ли разработчики тестов пользоваться доступными средствами тестирования?
- Учитываются ли в тестовом плане характеристики средств тестирования?
- Существует ли возможность получения консультации специалистов по работе с конкретными средствами тестирования?



Управление

- Выделено ли достаточное количество ресурсов на выполнение заявленного тестового плана?
- Происходили ли ротации в составе команды тестирования на протяжении разработки проекта?
- Правильно ли распределены ресурсы между отделами разработки ПО и его тестированием?
- Соблюдаются ли командой тестирования существующие технологии?



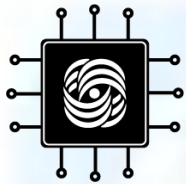
Обучение

- Предусмотрено ли обучение команды тестирования работе с новыми средствами?
- Завершено ли обучение команды до начала процесса тестирования?
- Разбирается ли команда в теоретических основах технологий тестирования? Понимают ли они зачем проводятся те или иные мероприятия?
- Обладают ли члены команды достаточными знаниями для анализа результатов тестирования?



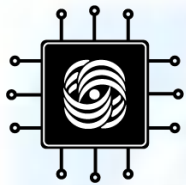
Эффективность

- Учитываются ли классы тяжести отказов во время разработки тестов для их обнаружения?
- Используются ли средства подсчёта статистики результатов тестирования?
- Учитывают ли выделенный бюджет и составленный план возможные отклонения?
- Принимается ли во внимание эффективность использования средств автоматизированного тестирования?
- Учитывается ли вероятность ошибки, подсчитанная на основе предыдущей статистики?



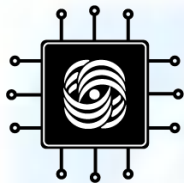
Контроль качества тестирования

- Проводятся ли мероприятия по недопущению повторения совершённых командой тестирования ошибок?
- Проверяется ли разработанный план тестирования на соответствие существующим стандартам?
- Предусмотрены ли способы проверки выполнения плана тестирования?
- Ведётся ли регулярная отчётность?
- Собираются ли резюмирующие внутренние отчёты об общей эффективности тестирования в процессе разработки ПО?

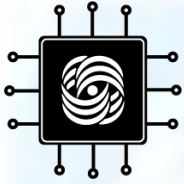


Тестирование и пользователи

- Предоставляется ли пользователям удобный способ выражения своих замечаний по поводу использования продукта?
- Обладают ли пользователи достаточными знаниями для оценки фронта работ по тестированию?
- Ознакомлены ли пользователи с планом тестирования? Одобрен ли ими существующий план?



Спасибо за внимание!



НАДЁЖНОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Лекция 12:

Проведение тестирования.

Анализ данных об отказах для принятия решений

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



План лекции

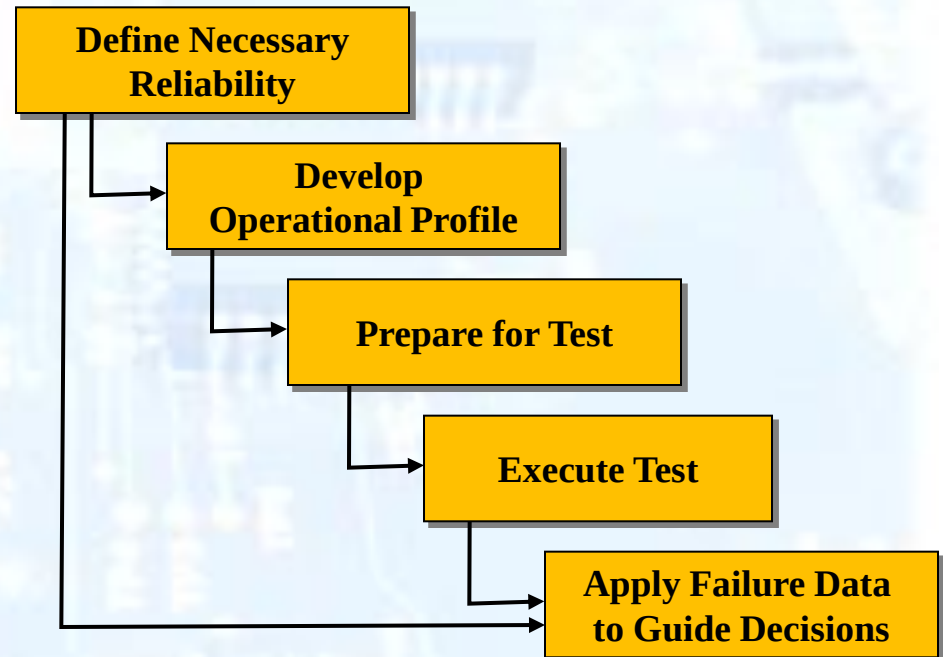
- Распределение ресурсов при тестировании
- Методы тестирования
- Диаграмма надёжности
- Модель RDC
- Критерии выпуска ПО



SRE: процесс

- 5 шагов в SRE процессе:

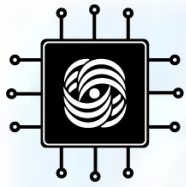
- Определить необходимый уровень надежности
- Разработать функциональный разрез
- Подготовить для тестирования
- **Выполнить тестирование**
- Проанализировать данные об ошибках и использовать их для принятия решений





Как распределить время

- 1) Между тестируемыми системами
- 2) Между функциональным, регрессионным и нагрузочным тестированием для каждой из подсистем
- 3) Между режимами функционирования для нагрузочных тестов каждой из подсистем



1. Распределение время тестирования в масштабах системы

- Выделяйте время на тестирование СВЯЗАННЫХ СИСТЕМ в соответствии с рассчитанным риском
- Распределяйте оставшееся время в соответствии с пропорциями, определёнными для тестовых сценариев

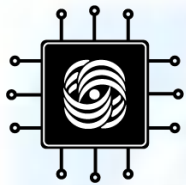
**Интерфейс
для сторонних систем**

**Приобретённые
компоненты**

**Разработанные
компоненты**

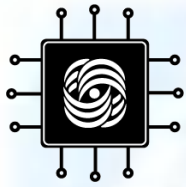
ОС, системное ПО

Аппаратное обеспечение



2. Распределение времени и рост надёжности

- При росте надёжности для каждой подсистемы и каждого нового тестового сценария выделяйте время на:
 - Функциональное тестирование [первая версия программы]
 - Регрессионное тестирование [последующие версии программы]
- При этом гарантируется проверка всех новых критических операций.
- Оставшееся время выделяется на нагрузочное тестирование.



3. Распределение времени нагрузочного тестирования

- Выделяйте время нагрузочного тестирования между режимами функционирования программы в соответствии с числом операций в каждом ИЗ НИХ

•Пример:

Режим	Пропорция операций
Пиковая нагрузка	0.1
Инициализация	0.7
Часы простоя	0.2



Запуск тестовых сценариев (1)

- Последовательность тестирования:
 - 1. Приобретённые компоненты**
 - Проверочные тесты
 - 2. Разрабатываемая система**
 - Сначала функциональные, затем нагрузочные тесты для первой версии программы
 - Сначала функциональные, затем регрессионные тесты для последующих версий программы
 - 3. Интерфейсы других систем**
 - Только нагрузочные тесты
- Допустимо изменение порядка или параллельное тестирование



Запуск тестовых сценариев (2)

- При тестах на увеличение надёжности сначала проводите функциональное, а затем нагрузочное тестирование. Проводите регрессионное тестирование после каждого значительного изменения программы
- Выполнение тестов проводится в случайное время
- При функциональном тестировании новые тесты и регрессионные тесты предыдущей версии программы выполняются в случайном порядке
- При нагрузочном тестировании вызывайте сценарии каждого режима функционирования в соответствии с выделенным для него временем
- Число нагрузочных тестов определяется временем, выделенным для данного режима функционирования



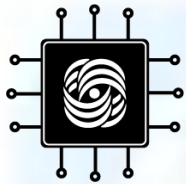
Выбор сценариев (1)

- Для нагрузочных тестов, сценарии могут выбираться повторно. Для функциональных и регрессионных тестов это не так
- При функциональном и регрессионном тестировании результаты прогонов скорее всего будут идентичными, так как влияние косвенных переменных сведено к минимуму
- При нагрузочном тестировании результаты нескольких прогонов скорее всего будут отличаться



Выбор сценариев (2)

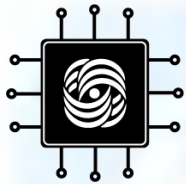
- Зачем повторять одни и те же тесты?
- Одна операция может содержать в себе несколько ошибок. Если выбор производится без перемещения, то операция может быть вызвана только один раз, и вероятность обнаружения других ошибок снижается



Определение отказов системы

Чтобы определить факт отказа:

- 1) Найдите отклонения от нормы в выводе программы
- 2) Определите какие из них вызваны отказами
- 3) Определите время возникновения отказа
- 4) Определите класс тяжести отказа
- 5) ЗадOCUMENTИРУЙТЕ отказ



1. Анализ вывода программы

1. Некоторые отклонения от нормального поведения можно найти с помощью средств отладки (GDB,...)

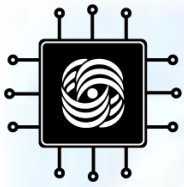
■ Примеры:

- Неверное обращение к памяти
- Взаимная блокировка процессов
- Зависание
- etc.



2. Отклонения и ошибки

- С помощью ручного анализа отклонения можно сделать вывод, что нарушаются требования к программе, следовательно произошёл отказ
- Отказы наибольшей тяжести можно определить без анализа вывода программы:
 - Аварийное завершение программы
 - Неполное выполнение поставленной задачи



3. Время возникновения отказа

- Для определения времени отказа используйте ту же меру, что и при задании желаемой целевой интенсивности отказов
- Для тестов функциональности измеряйте время, когда произошёл отказ
- Для тестов надёжности системы используйте интервальную оценку.
– [5 отказов за 1000 операций]



4. Определение класса тяжести отказа

- Выберите один из описанных ранее классов тяжести отказов, который бы соответствовал найденной ошибке
- Задокументируйте отказы и их классы тяжести
- Начинайте исправлять ошибки с отказов, имеющих больший класс тяжести



Терминология (1)

■ Статический анализ

- Ручная проверка кода
 - Взаимопроверка
 - Формальная проверка
- Методы нахождения отклонений и ошибок
 - Аудит исходного кода
 - Проверка интерфейсов
 - Поблочное тестирование
 - Анализ потоков данных
- Структурный анализ



Терминология (2)

■ **Динамический анализ**

■ Инструментализация

- *Структурный анализ*
- *Измерение производительности: времени выполнения и объёма потребляемых ресурсов*
- *Измерение производительности: анализ сложности алгоритма работы*
- *Вставка динамических проверок (assert)*
- *Интерактивное тестирование и отладка*

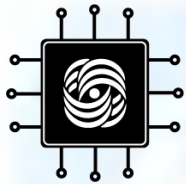
■ **Случайное тестирование**



Терминология (3)

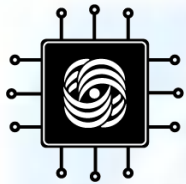
■ **Динамический анализ**

- **Функциональное тестирование**
 - *На основе спецификации*
 - *Построение причинно-следственных связей*
- **Мутационное тестирование**
 - *Мутационный анализ*
 - *Подсев ошибок*
- **Тестирование в реальном времени**
- **Формальное тестирование**



Простейшие технологии тестирования

- Два широко известных метода:
 - **Метод прозрачного ящика**
 - Обнаруживает ошибки на основе анализа внутренней структуры программы
 - **Метод чёрного ящика**
 - Оценивает насколько хорошо программа выполняет предъявляемые к ней требования



1. Метод прозрачного ящика

- Требуется детальное знание реализации программы
- Цель – обеспечить полное **покрытие кода**
 - Как много путей выполнения программы проверены на самом деле?
 - Эффективность теста часто измеряется размером покрытого им кода



2. Метод чёрного ящика

- Предполагает, что требования уже прошли валидацию
- Поиск нереализованной или неправильно реализованной функциональности
- Задаёт входные данные, заранее зная какой результат должен быть получен
- Существует много методов:
 - Тестирование производительности
 - Стресс-тестирование
 - Тестирование надёжности
 - Тестирование безопасности
 - etc.



Уровни тестирования

- Тестирование связано с циклом разработки программ:
 - Блочное
 - Интеграционное
 - Пользователями
 - Установки
 - Регрессионное





1. Блочное тестирование

- Тестирование одного модуля программы методом белого ящика в контролируемом окружении и независимо от других модулей
 - Модуль – функция или небольшой их набор, который может быть полностью покрыт тестовыми сценариями



2. Интеграционное тестирование

- Тестируется комбинация модулей
- Фокус на их взаимодействии
- Методы белого и чёрного ящика
- Существует три основных подхода:
 - Сверху-вниз
 - Снизу вверх
 - Большого взрыва



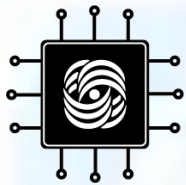
3. Внешнее тестирование

- Метод чёрного ящика
- Проверяет, что система корректно реализует требуемый функционал
- Иногда известно как *альфа* тестирование
- Тестирование имитирует использование системы конечными пользователями



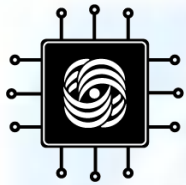
4. Системное тестирование

- Более робастная версия внешнего тестирования
- Различие в тестовой платформе
- Окружение идентично реальному
 - Учитывается «железо», размеры БД, ...
- Позволяет лучше оценить нефункциональные требования (производительность, безопасность,...)



5. Тестирование пользователями

- Также известно как **бета-тестирование**
- Система тестируется конечными пользователями
- Ещё более реалистичное тестирование по сравнению системным
- Валидация на основе пользовательских представлений
- Определяет степень готовности для развёртывания



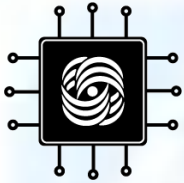
6. Тестирование установки

- Проверка полной, частичной установки/удаления программы и её обновления
- Расплывчато описывается в литературе

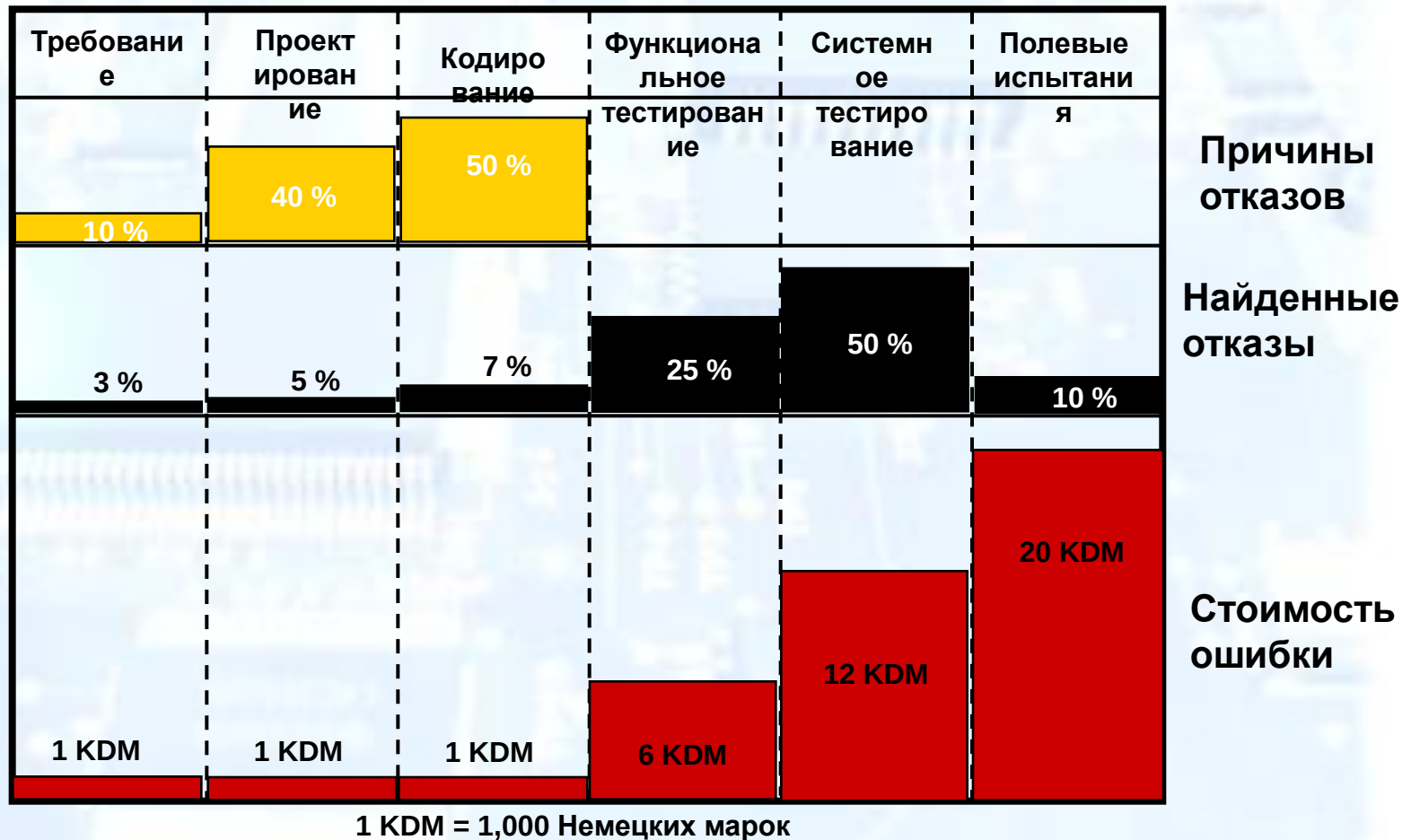


7. Регрессионное тестирование

- Тестирование модифицированной программы
- Проверяет, что внесённые изменения не повлияли на работу остальных подсистем
- Для тестирования выбираются сценарии, на работу которых могут повлиять внесённые правки
- Исправление ошибки может добавить новую ошибку, нарушить структуру или корректность интеграции с другими компонентами



Стоимость ошибки





Тестирование на основе ошибок

- Определяет классы ошибок и входные данные, способные вызвать эти ошибки, если они присутствуют
- **Отправка ошибок** - сознательное добавление ошибок в программу до её тестирования. На основе числа найденных в процессе тестирования ошибок определяется число реальных ошибок в коде
- **Мутационное тестирование** добавляет ошибки в код, чтобы определить оптимальные для тестов данные
- **Добавление ошибок** позволяет определить реакцию всей системы на некорректное поведение отдельных её частей



Нефункциональное тестирование

- Нефункциональные требования описывают отдельные функции системы, а скорее задают общие параметры системы (производительность, безопасность, совместимость, удобство использования, ...)
- На тестирование нефункциональных требований могут быть затрачены большие усилия
- Задание нефункциональных требований позволяет перейти от приложения которое способно выполнить требуемые задачи к приложению, которое нужно реальным пользователям системы



Документирование нефункциональных требований

- Обычно нефункциональные требования задаются двумя способами одновременно:
 - Создаётся единый документ, в котором описываются требования применительно ко всем случаям использования системы
 - Каждое требование к системе содержит раздел «нефункциональные требования», где описываются требования, отличные от общих спецификаций



Тестирование нефункциональных требований (1)

- **Не откладывайте проверку нефункциональных требований**
- Обычно рекомендуют проверять нефункциональные требования одновременно с соответствующими им функциональными требованиями



Тестирование нефункциональных требований (1)

- **Проводите тестирование производительности в реальном окружении**
- Для определения производительности СУБД нужно провести тесты на БД размером в 1, 100, 500, 1,000, 5,000, 10,000, 50,000, и 100,000 записей
- Такой способ тестирования производительности позволяет определить верхнюю границу, после которой приложения использовать нецелесообразно



Тестирование нефункциональных требований (3)

- **Доверяйте тестирование удобства работы с приложением его реальным пользователям**
- Используйте для получения информации сразу несколько методов:
 - Исследуйте существующие средства
 - Наблюдайте за работой пользователя
 - Предлагайте пользователям несколько прототипов интерфейса



Тестирование нефункциональных требований (3)

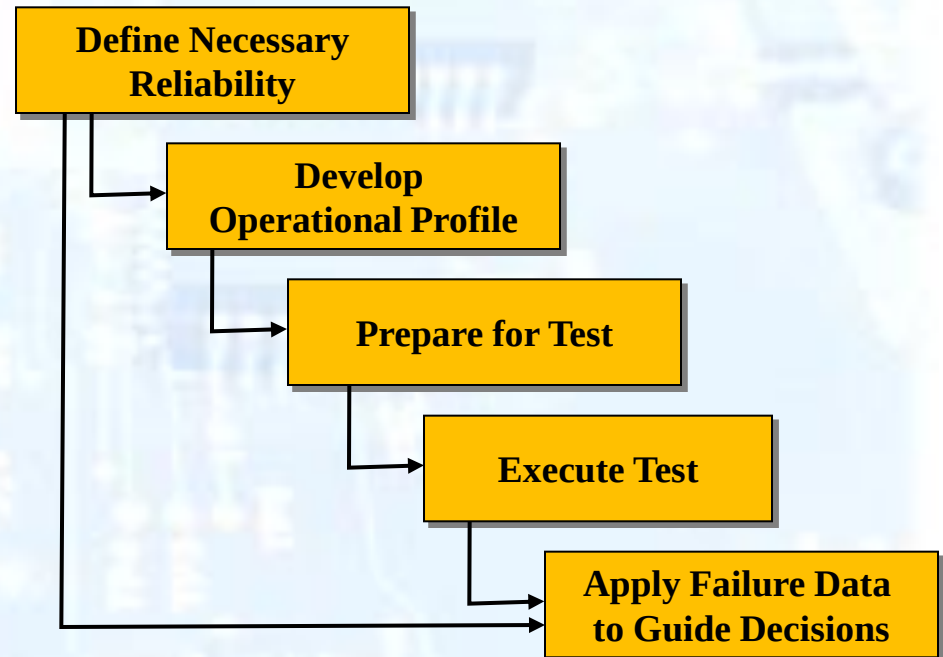
- **Исследуйте вопросы безопасности на как на уровне отдельного требования, так и на общесистемном уровне**
- Каждая функция системы, вероятно, несёт в себе определённые угрозы безопасности
- На основе грамотно составленных требований к безопасности можно построить набор сценариев для их проверки
- Если угрозы безопасности критичны, то для его тестирования стоит использовать сторонние приложения



SRE: процесс

- 5 шагов в SRE процессе:

- Определить необходимый уровень надежности
- Разработать функциональный разрез
- Подготовить для тестирования
- Выполнить тестирование
- Проанализировать данные об ошибках и использовать их для принятия решений





Типы решений

- **Решения, относящиеся к сертификационным испытаниям**
 - Принять/отклонить приобретенный компонент
 - Принять/отклонить подсистему, операционную систему, аппаратное обеспечение и т.п.
- **Решения, относящиеся к испытаниям повышения надежности**
 - Управление процессом разработки ПО
 - Выпуск ПО



Обзор: FIO

- Интенсивность отказов цели (FIO) (λ_F)
 - Интенсивность отказов определяется как отказы в некоторых единицах, например:
 - 3 сигнала за 100 часов функционирования.
 - 5 отказов за 1000 заданий и т.д.
- Средняя наработка до отказа (MTTF)
 - Ожидаемое время до обнаружения следующего отказа.

$$A = \frac{1}{1 + \lambda t_m} = \frac{MTTF}{MTTF + t_m} \quad \text{therefore} \quad MTTF = \frac{1}{\lambda}$$

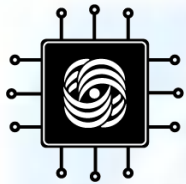


Диаграмма надежности /2

- Эффективный способ проверки выполняется FIO (λ_F) или нет
- Он основан на сборе данных об отказах в конкретные моменты времени
- **Вертикальная ось:** количество отказов (n)
- **Горизонтальная ось:** нормализованные данные об отказах (T_n), т.е. время отказа $\times \lambda_F$

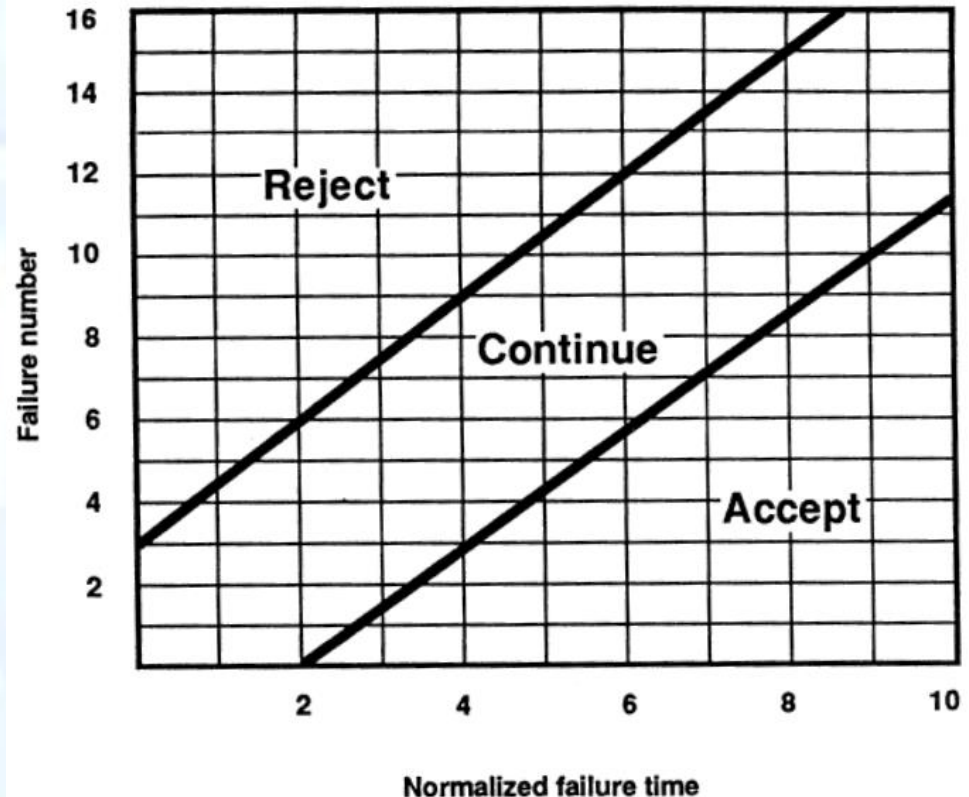


Figure from Musa's Book



Параметры /1

- **Отношение различимости (γ):** Приемлемая ошибка в оценке интенсивности отказов
- **Риск заказчика(β) :** Вероятность того, что разработчик готов принять ложное заявление, что целевая интенсивность отказов выполняется (т.е. принятие), когда это не так
- **Риск разработчика(α) :** Вероятность того, что разработчик готов принять ложно заявив, что целевая интенсивность отказов **не** выполняется (т.е. **отклонение**), когда это так и есть



Параметры /2

Для $\alpha = 10\%$ и $\beta = 10\%$ и $\gamma = 2$

- Существует риск в 10% (β) ошибочного принятия ПО, когда его целевая интенсивность отказов на самом деле равна или больше чем в 2 раза ($\gamma = 2$) превышает целевую интенсивность отказов
- Существует риск в 10% (α) ошибочного отклонения ПО, когда его целевая интенсивность отказов на самом деле равна или менее чем 2 раза ($\gamma = 2$) меньше целевой интенсивности

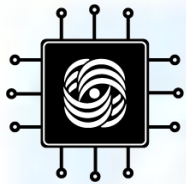


Диаграмма надежности /2

$$A = \ln \frac{\beta}{1 - \alpha} \quad B = \ln \frac{1 - \beta}{\alpha}$$

- A изменяется быстро с *риском заказчика*, но очень незначительно с *риском разработчика* и это определяет отрезок границы **принятия** с горизонтальной линией $n=0$
- B изменяется быстро с *риском разработчика*, но очень незначительно с *риском заказчика* и это определяет отрезок границы **отклонения** с вертикальной линией $T_n=0$

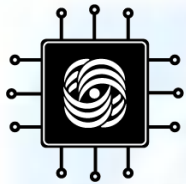


Диаграмма надежности /3

- Граница между областями **принятия** and и **продолжения**

$$T_n = \frac{A}{1-\gamma} - \frac{\ln \gamma}{1-\gamma} n$$

- Граница между областями **отклонения** и **продолжения**

$$T_n = \frac{B}{1-\gamma} - \frac{\ln \gamma}{1-\gamma} n$$

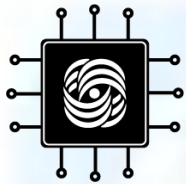


Диаграмма надежности /4

Значения отрезков границ с различными горизонтальными и вертикальными осями

Intercept with	Discrimination ratio γ		
	2	1.5	1.1
Horizontal line at $n = 0$	$-A, -B$	$-2A, -2B$	$-10A, -10B$
Horizontal line at $n = 16$	$-A + 11.1, -B + 11.1$	$-2A + 13.0, -2B + 13.0$	$-10A + 15.2, -10B + 15.2$
Vertical line at $T_N = 0$	$1.44 A, 1.44 B$	$2.47 A, 2.47 B$	$10.5 A, 10.5 B$
Vertical line at $T_N = 16$	$\frac{A + 16}{0.693}, \frac{B + 16}{0.693}$	$\frac{A + 8}{0.405}, \frac{B + 8}{0.405}$	$\frac{A + 1.6}{0.0953}, \frac{B + 1.6}{0.0953}$

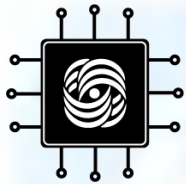


Диаграмма надежности /5

Значения A и B для различных уровней рисков заказчика и разработчика

Supplier risk	Parameter	Consumer risk			
		0.1	0.05	0.01	0.001
0.1	A	-2.20	-2.89	-4.50	-6.80
	B	2.20	2.25	2.29	2.30
0.05	A	-2.25	-2.94	-4.55	-6.86
	B	2.89	2.94	2.99	2.99
0.01	A	-2.29	-2.99	-4.60	-6.90
	B	4.50	4.55	4.60	4.60
0.001	A	-2.30	-2.99	-4.60	-6.91
	B	6.80	6.86	6.90	6.91

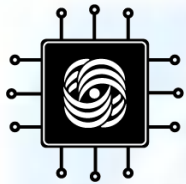


Диаграмма надежности /6

- Когда уровни риска (α и β) уменьшаются, система потребует проведения больше испытаний прежде чем достигнуть областей принятия или отклонения, т.е. необходимо продолжить расширение областей
- Когда отношение различимости (γ) уменьшается, система потребует проведения больше испытаний прежде чем достигнуть областей принятия или отклонения, т.е. необходимо продолжить расширение областей



RDC: Когда и как

- Когда использовать RDC?
 - Когда данные об отказах ограничены несколькими отказами и требуется определить тенденцию в надежности системы
- Как использовать RDC?
 - Собрать данные об отказах (количество отказов и моменты времени)
 - Определить MTTF и ожидаемые уровни доверия
 - Нарисовать точки отказов на графике и проанализировать тенденцию надежности



RDC: Пример/1

- Риск потребителя
 $\beta = 5\%$
- Риск поставщика
 $\alpha = 5\%$
- Отношение
различимости
 $\gamma = 2$

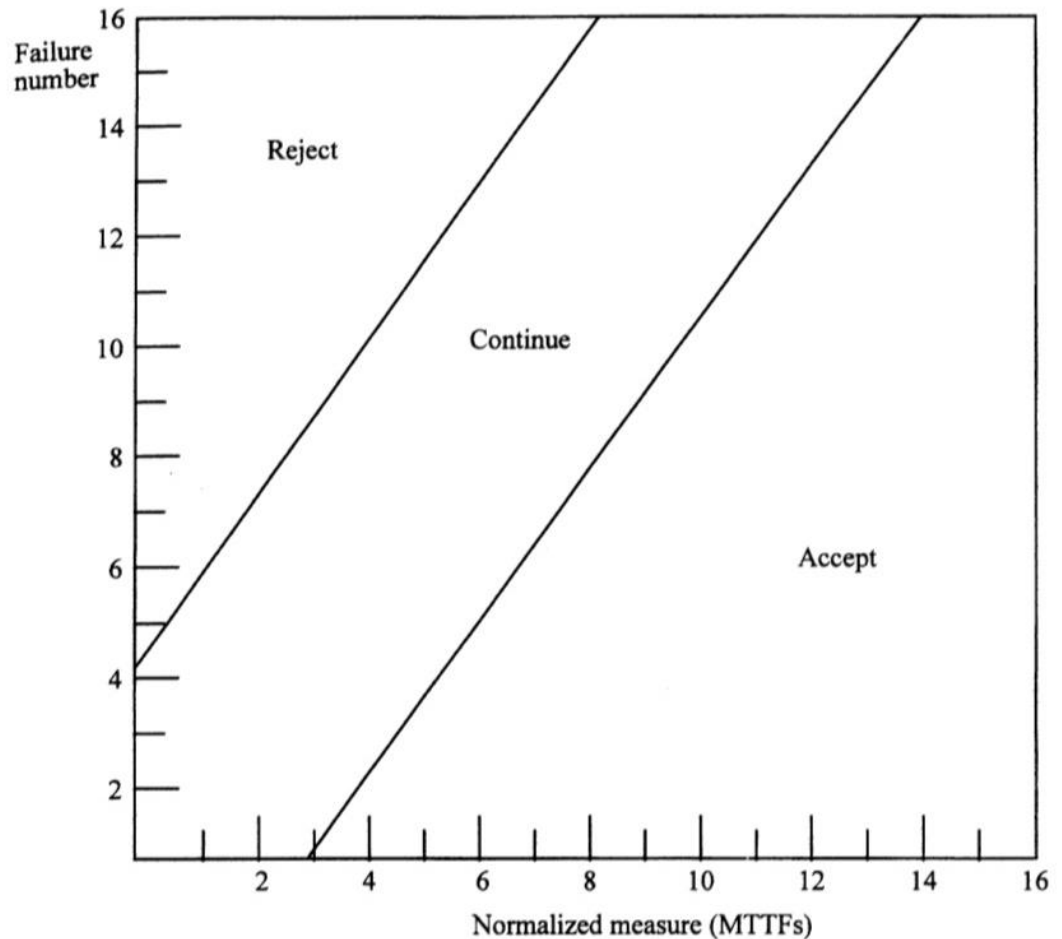
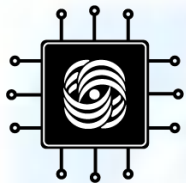


Figure from Musa's Book



RDC: Пример /2

- Риск потребителя
 $\beta = 1\%$
- Риск поставщика
 $\alpha = 1\%$
- Отношение
различимости
 $\gamma = 2$

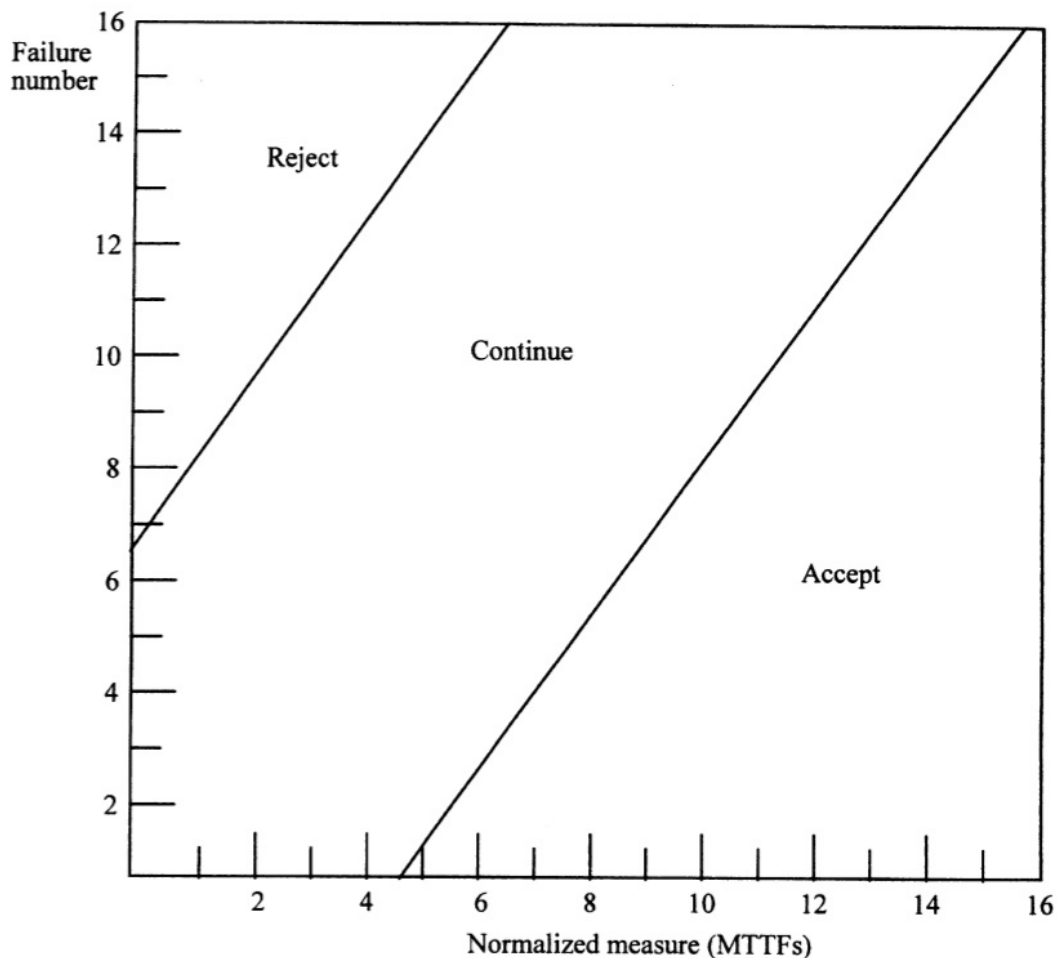
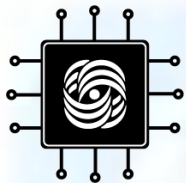
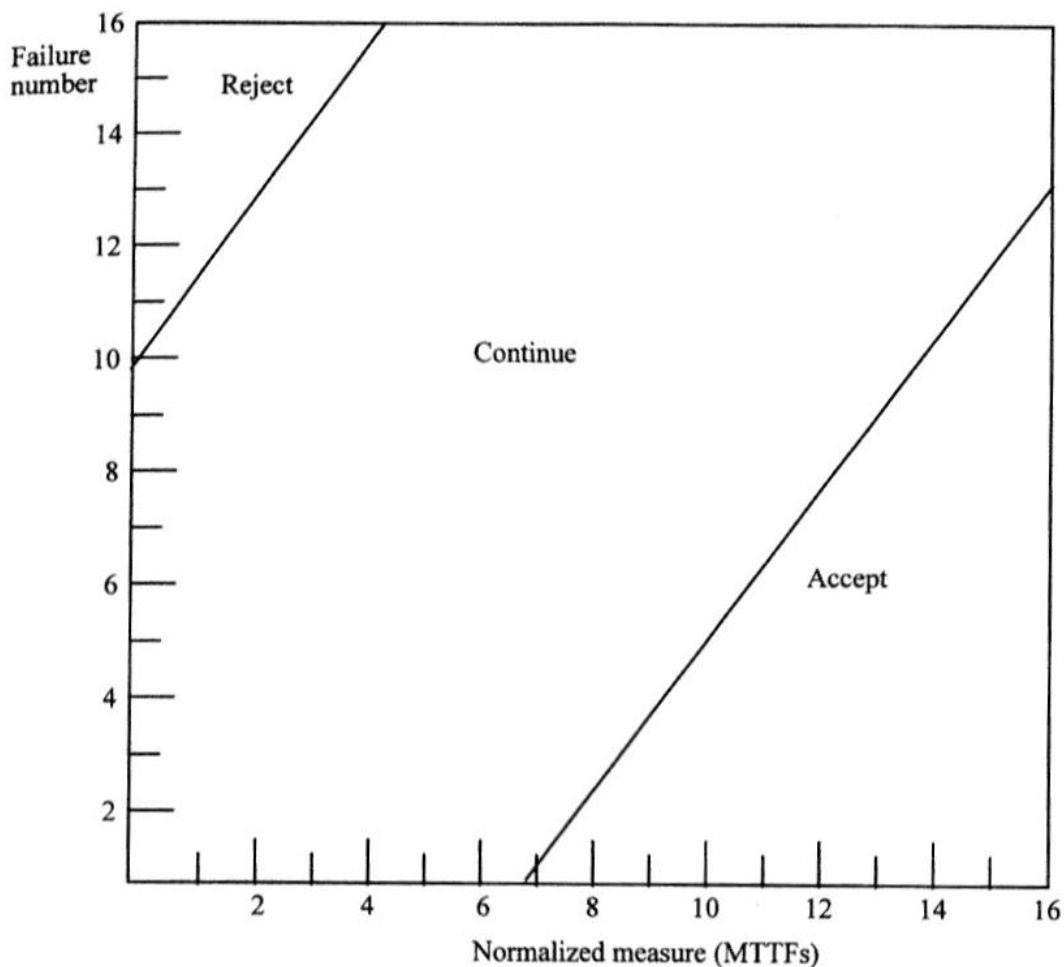


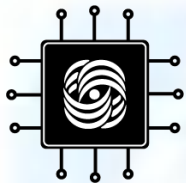
Figure from Musa's Book



RDC: Пример /3

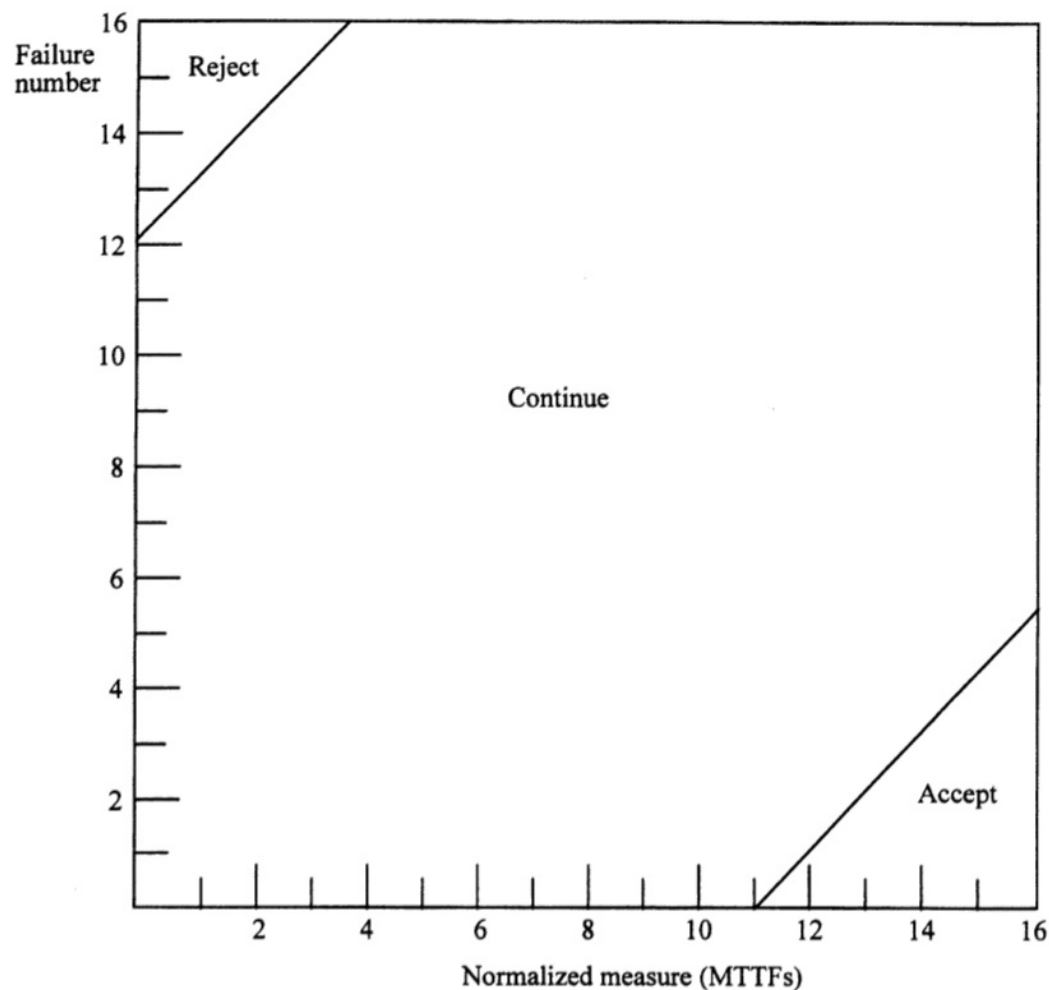
- Риск потребителя
 $\beta = 0,1\%$
- Риск поставщика
 $\alpha = 0,1\%$
- Отношение
различимости
 $\gamma = 2$





RDC: Пример /4

- Риск потребителя
 $\beta = 10\%$
- Риск поставщика
 $\alpha = 10\%$
- Отношение
различимости
 $\gamma = 1.2$





Пример 1

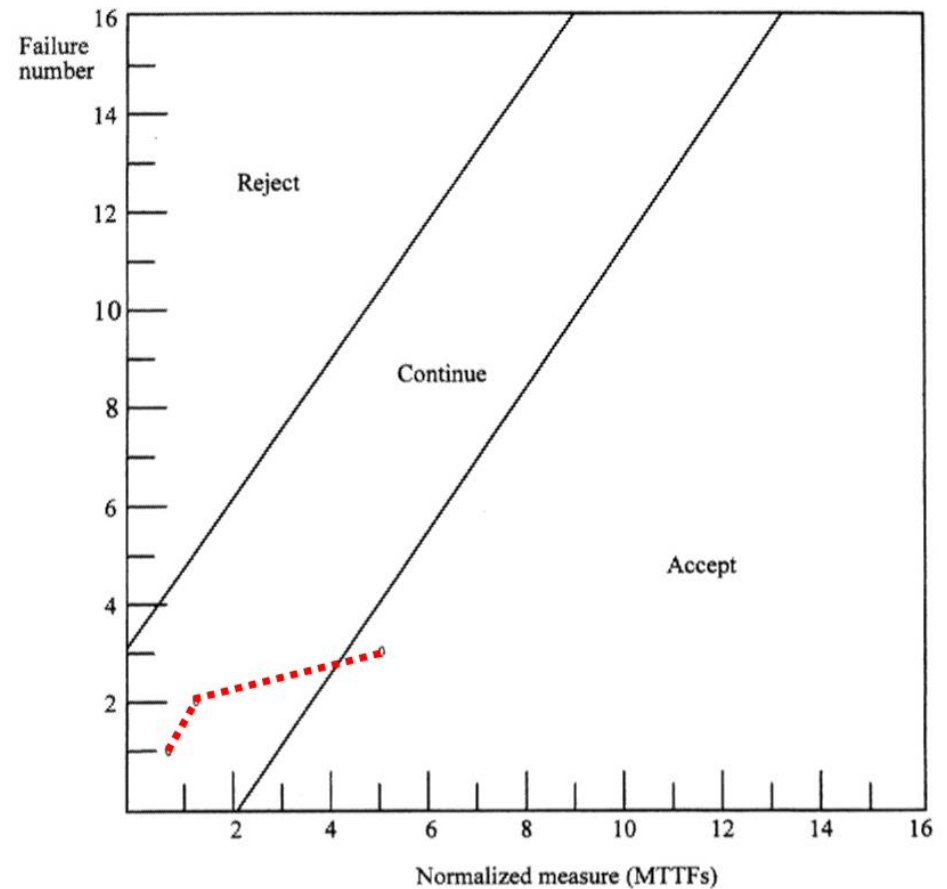
<i>Failure number</i>	<i>Measure (million transactions)</i>	<i>Normalized Measure (MTTF)</i>
1	0.1875	0.75
2	0.3125	1.25
3	1.25	5

$\lambda_F = 4$ failures/million transactions

$\alpha = \%10$

$\beta = \%10$

$\gamma = 2$





Пример 2

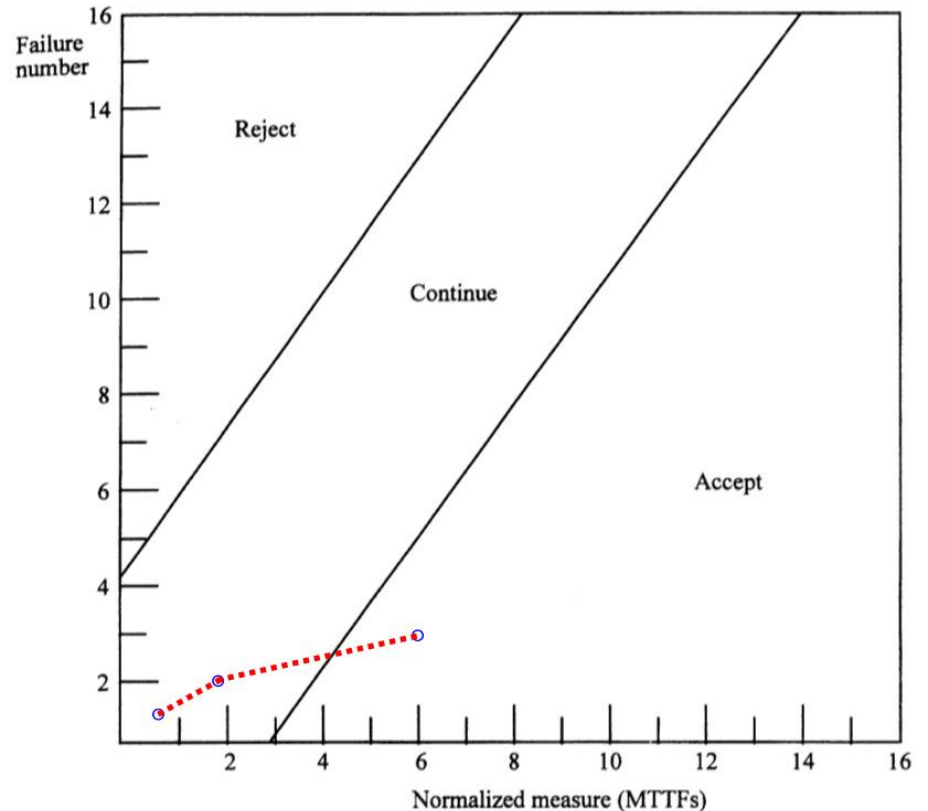
<i>Failure number</i>	<i>Measure (CPU hour)</i>	<i>Normalized Measure (MTTF)</i>
1	8	0.8
2	19	1.9
3	60	6

$\lambda_F = 0.1$ failures/CPU hour

$\alpha = 0.05$

$\beta = 0.05$

$\gamma = 2$





Пример 3 /1

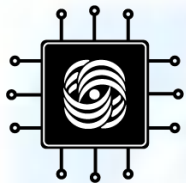
- Мы собрались купить новый цветной лазерный принтер для лаборатории. Мы имеем принтер для проведения тестовых сертификационных испытаний на нем. Инструкция показывает, что необходимо заменить тонер каждые 10000 страниц. Мы хотим, чтобы система работала безотказно между двумя последовательными изменениями тонера или в худшем случае – один отказ
- а) Какая должна быть наша целевая интенсивность отказов для системы?

$$\lambda_F = 1/10000 \text{ pages}$$

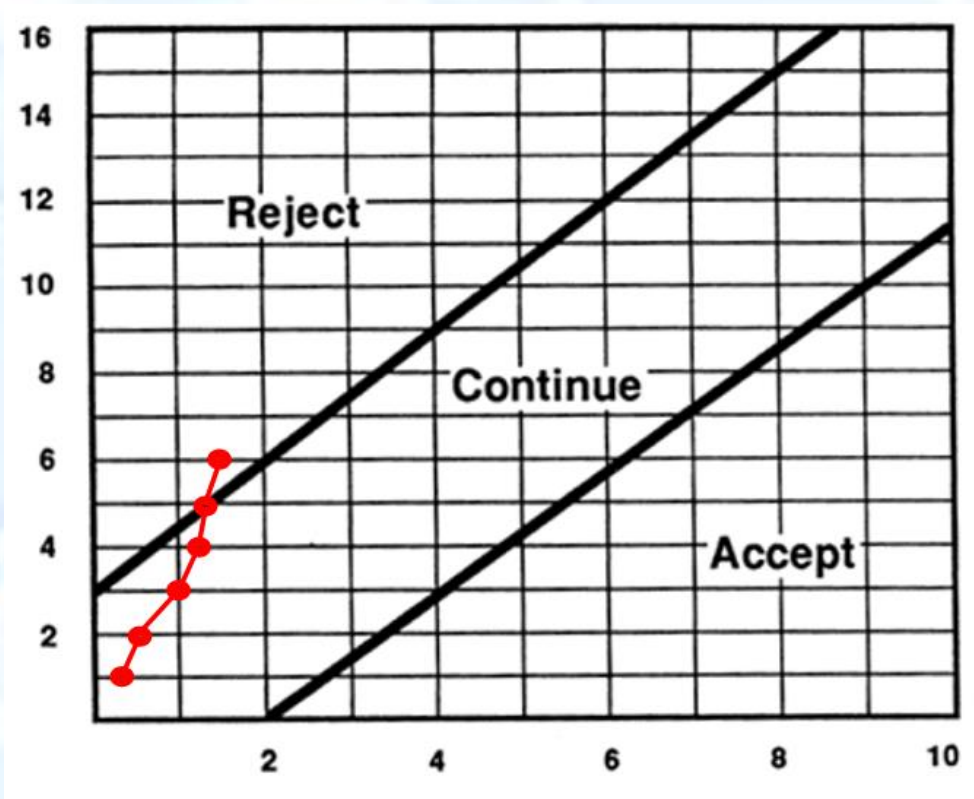


Пример 3 /2

б) Мы наблюдаем, что отказы происходят на 4000 страниц, 6000 страниц, 10000 страниц, 11000 страниц, 12000 страниц и 15000 страниц. Используя демонстрационные диаграммы надежности, какое заключение можно сделать об этом принтере?



Пример 3 /3



- Из-за провала сертификационных испытаний мы отказываемся от этого принтера



RDC: Достоинства и ограничения

- RDC анализ это очень эффективный с точки зрения времени и стоимости способ анализа надежности системы. Он может использоваться для демонстрации надежности системы при принятии решений
- Недостатком RDC анализа является то, что RDC не позволяет дать количественную оценку надежности (или доступность) изучаемой системы. Он может лишь показать тенденции изменений и их влияние на надежность системы
- Другой недостаток: экспериментировать с различными уровнями достоверности и MTTF возможно, но довольно утомительно



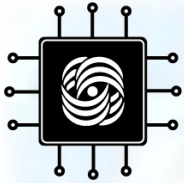
Типы решений

- **Решения, относящиеся к сертификационным испытаниям**
 - Принять/отклонить приобретенный компонент
 - Принять/отклонить подсистему, операционную систему, аппаратное обеспечение и т.п.
- **Решения, относящиеся к испытаниям повышения надежности**
 - Управление процессом разработки ПО
 - Выпуск ПО



Критерий выпуска

- Следующие показатели должны рассматриваться совместно для получения адекватной картины качества продукта:
 - Стабильность, надежность и доступность системы
 - Дефект объёма
 - Выдающиеся критические проблемы
 - Обратная связь с пользователями ранних версий программ
 - Другие атрибуты качества, которые имеют особое значение для конкретного продукта, заказчика и рынка(например, простота использования, производительность, безопасность, мобильность)



Подход: модели

- Использование средств таких как CASRE для построения соотношения $\lambda / \lambda F$ во времени, чтобы проанализировать тенденции
- CASRE использует **базовую экспоненциальную** и **логарифмическую пуассоновскую** и некоторые другие модели
- Базовая экспоненциальная модель предполагает конечное количество отказов за бесконечное время; логарифмическая пуассоновская модель предполагают бесконечное количество отказов
- При оценке соотношения $\lambda / \lambda F$, базовая экспоненциальная модель имеет «оптимистическую» тенденцию; логарифмическая пуассоновская модель имеет «пессимистическую» тенденцию



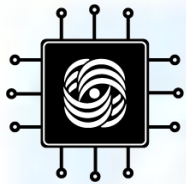
Критерий выпуска /1

- Оценка нормализованной интенсивности отказов имеет степень неопределенности. Она определяется «доверительным интервалом»
- Например, если доверительный интервал 90%, то можно ожидать, что система достигнет своей целевой интенсивности только с вероятностью 90%
- CASRE 2.0 не позволяет вычислять «доверительные интервалы»



Критерий выпуска /2

- Как включить доверительный интервал в оценку?
- Можно использовать следующий приближенный подход:
 - Рассмотрим завершение тестирования для системы, когда нормированная интенсивность отказов снижается до 0,5 или ниже
 - Предположим, что 90% - доверительный интервал для интенсивности отказов вблизи релиза и таким образом интенсивность отказов расширяется от 0.5 до 1.5 раз



Критерий выпуска /3

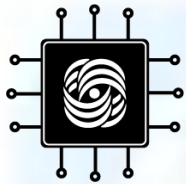
Рассмотрим вопрос о выпуске продукта

1. Тестирование (функций и нагрузки) прекращено удовлетворительно для продукта
2. Тестирование прекращено удовлетворительно для всех вариантов продукта или нормированные значения отношения λ/λ_F для этих вариантов не превышают 0.5
3. Продукт и его варианты проходит все приемочные испытания, запланированные для них
4. Связанные системы проходят все приемосдаточные испытания



Развитие программ /1

- Модели надежности, используемые для оценки интенсивности отказов (λ), предполагают, что программа работает стабильно
- Программа может изменяться из-за:
 - Изменения требований
 - Интеграции новых частей
 - Необходимости адаптироваться к изменяющейся аппаратной и программной среде
 - Необходимости улучшения производительности системы
 - Эволюция как часть процесса разработки



Развитие программ /2

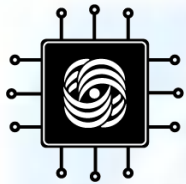
Вероятные сценарии:

- Игнорирование изменений при медленном развитии программы или незначительном по размеру
- Игнорирование старых данных на старой фазе и базы оценок на новой фазе
- Поэтапное развитие
 1. Применение изменений по компонентам
 2. Применение изменений к функциональным группам



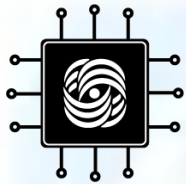
а) Игнорирование изменений

- Эффекты развития могут быть проигнорированы для программ развивающихся медленно и незначительно по размеру (например, менее чем 5% от общего количества кода за неделю)
- **Достоинства:** Не требуется никаких дополнительных испытаний и сбора данных
- **Недостатки:** Оценки параметров модели и $\lambda/\lambda F$ отношения будут отставать от истинного положения и иметь ошибки, но диапазон ошибки может быть приемлемым



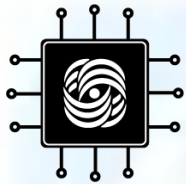
b) Игнорирование старых данных

- Установить диапазон изменения для данных об отказах для того, чтобы включить только последние отказы
- Окно последних отказов должно обычно включать последние 40-50 отказов для приемлемой оценки параметров
- Применять только, когда изменения программы ограничены (например, менее 20% от изменений программы)



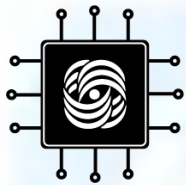
с) Поэтапная эволюция /1

- Подходы поэтапного развития как правило работают лучше *с малым количеством больших изменений*, каждое из которых – результат добавления независимых элементов (подсистем, пакетов и т.п.)
- **В компонентном подходе**, добавляются отдельно интенсивности отказов для отдельных компонентов, чтобы получить интенсивность отказов системы на каждом этапе
- **В подходе с функциональными группами**, добавляются взвешенные интенсивности отказов функциональных групп, чтобы получить интенсивность отказов системы на каждом этапе



с) Поэтапная эволюция /2

- Достоинства и недостатки подхода с поэтапной эволюцией:
- **Достоинства:** Функциональный профиль системы может применяться непосредственно
- **Недостатки:**
 - Требуется дополнительный сбор данных с множества элементов
 - Большая ошибка оценки (или с более поздним достижением заданной степени точности) из-за меньших размеров выборки



Спасибо за внимание!