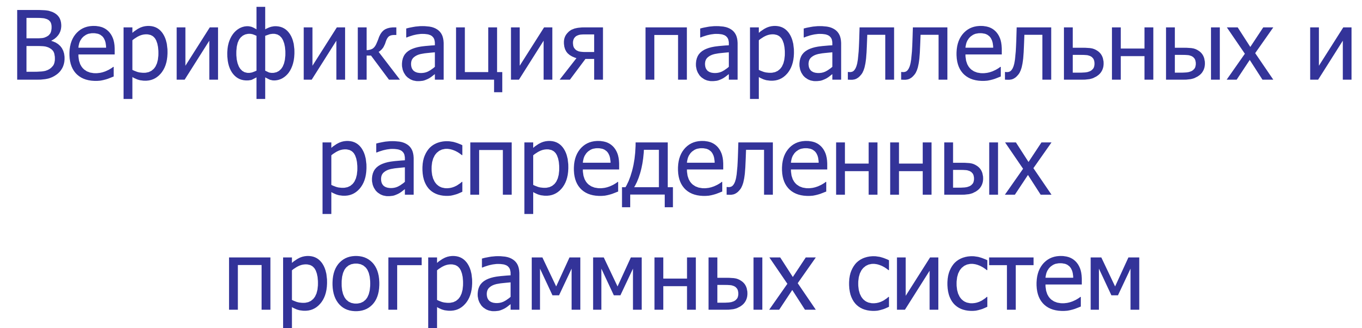


Верификация параллельных и распределенных программных систем

Юрий Глебович Карпов

Ирина Владимировна Шошмина

Алексей Борисович Беляев

Санкт-Петербургский
Политехнический университет

Проблема

- **Компьютеры – повсюду** *Ubiquitous computing* (*вездесущие вычисления*)
  ИТ проникли во все сферы жизни современного общества.
- Раньше компьютер был компьютером, а телефон – телефоном, и любой мог отличить одно от другого. Сейчас и компьютер – не только компьютер, и телефон – не только телефон (A.Tanenbaum)
- **Эра параллельного программирования**
  Последовательные программы имеют очень узкое применение
  Многоядерные чипы, встроенные бортовые СУ -> требуются технологии разработки параллельного ПО, чтобы оно выполняло нужные функции. Multicore processors are bringing parallelism to mainstream of computing
- **Параллельные программы полны ошибок**
  Параллельные программы непостижимы для человека: они очень часто неправильны, содержат ошибки

Примеры ошибок

- В 1994 INTEL выпущен чип с ошибкой во встроенной программе. Замена *миллионов* дефектных процессоров ⇒ потери ~$500 млн.
- 4.06.96 ракета *Ариан 5* взорвалась через 39 с. Ошибка в бортовой программе при преобразовании 64-битового вещ в 16-бит целое. Ущерб > $600 млн.
- Конец 80-х: Therac-25 прибор лучевой терапии. Встроенная программа управления прибором допускала перевод в режимы с огромной дозой облучения. При некоторых (редких) действиях персонала пациенты получали передозировку. Двое умерли, несколько человек стали инвалидами.
- 23.03.2003. Война в Ираке. Ошибка в программе ⇒ система Patriot определила свой бомбардировщик Tornado как приближающуюся ракету и его сбили. Два пилота погибли.
  До 24% потерь в живой силе в I Иракской войне – *"Friendly Fire".*
- Boeing 757, 1995 г (рейс из Майами в Колумбию). Ошибка в Flight Management System привела к катастрофе. Погибли 159 человек. Фирма-разработчик ПО выплатила $300 млн родственникам жертв
- Бортовой компьютер на израильском истребителе в районе Мёртвого моря отказал: деление на 0. Высота над уровнем моря там является отрицательной величиной

СМИ о программных ошибках – каждый день

- Апрель, 2010. Авария в Мексиканском заливе: возможна ли программная ошибка? *Don Shafer, Phillip Laplante. The BP Oil Spill: Could Software be a Culprit? IEEE IT Pro September/October 2010, IEEE, 2010. ...* Не могла ли одной из причин бедствия стать ошибка в программном обеспечении?

- 05.07.2010. Apple: ошибка связи iPhone 4 — программная. Компания Apple признала существование в iPhone 4 проблем, касающихся качества связи.

- 06.12.2010. Спутники ГЛОНАСС и ракету "Протон-М" утопили программисты. Ракета-носитель «Протон-М» со спутниками «Глонасс-М» отклонились от заданного курса из-за допущенных ошибок в математическом обеспечении (основная причина - неправильно написанная формула в документации на заправку кислородом разгонного блока)

- 08.12.2010 — Японский зонд «Акацуки» не смог выйти на орбиту Венеры

  Космический исследовательский аппарат «Акацуки», запущенный Японией для исследования Венеры, не смог выйти на орбиту планеты, сообщает РИА «Новости» со ссылкой на специалистов Японского аэрокосмического агентства JAXA

Тестирование не гарантирует отсутствия ошибок

- Основным методом повышения надежности программ при традиционных методах разработки является тестирование
- Эдсгер Дейкстра:
  "Тестирование не может доказать отсутствия ошибок в программе"

В среднем современные программные системы имеют 10 ошибок на 1000 строк кода,
ПО высокого качества 1-3 ошибки на 1000 строк кода

Современное ПО содержит миллионы строк кода
Уже сданные работающие программы наполнены ошибками

В ОС Windows 95 ~ 5000 ошибок
«Good enough software» недопустимо для критических применений

Ошибки параллельных программ

- ТЕСТИРОВАНИЕ НЕ МОЖЕТ ВЫЯВИТЬ ТИПИЧНЫЕ ОШИБКИ СИНХРОНИЗАЦИИ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ
- Параллельные программы работают правильно “почти всегда”
- Параллельные программы могут годами сохранять ошибки, проявляющиеся после долгой эксплуатации как реакция на возникшую специфическую комбинацию многочисленных факторов, в частности, непредсказуемых скоростей выполнения отдельных процессов в параллельных программах
- Системы программного управления обычно строятся из параллельных взаимодействующих модулей. Ошибки в них часто являются критическими

Конфуз с голосованием на “Эхе Москвы” 24.09.2011

http://www.echo.msk.ru/programs/interception/814452-echo/

Параллельная композиция процессов:

За Путина

```
xП:=0;
while !Stop do
  wait call_1;
  xП ++
od
```

За Медведева

```
xМ:=0;
while !Stop do
  wait call_2;
  xМ ++
od
```

Σ

```
while !Stop do
   xΣ :=xП+xМ
od
```

$\%_П$

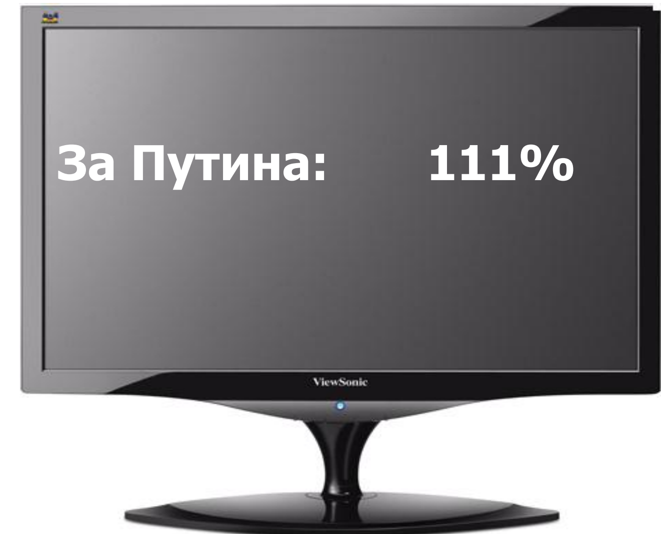
```
while !Stop do
   PП :=xП / xΣ
od
```

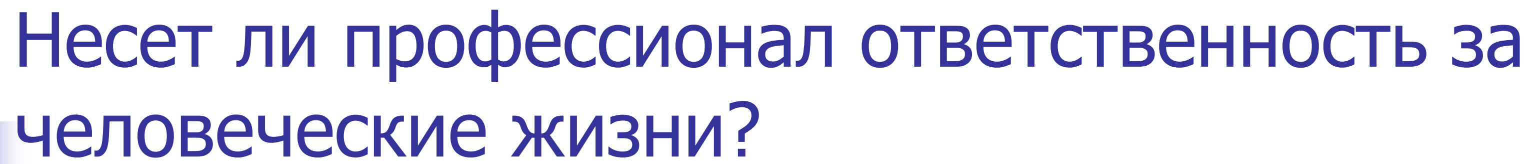
$\%_М$

```
while !Stop do
   PМ :=xМ / xΣ
od
```

- А.В.Венедиктов:

  “*Подвисает система голосования... У меня возникает 111% ...*”





Несет ли профессионал ответственность за человеческие жизни?

Программирование является единственной областью инженерной деятельности,
где разработчик не отвечает за качество своей работы

Программист обычно НЕ ЗНАЕТ (и не хочет знать!)

как определить требования к результату своей работы?

как проверить, что эти требования выполняются?

Важность проблемы валидации

- До 80% средств при разработке встроенного ПО тратится на валидацию – проверку соответствия ПО тому, что нужно потребителю ($60 billion annually for US economy – данные 2005 г.)
- При оставшихся в системах ошибках:
  - Огромные материальные потери
  - Потери жизней
- КТО БУДЕТ ОТВЕЧАТЬ? Страховка?
  - низкая надежность продукта – падает прибыль
  - увеличивается время разработки (time-to-market)
  - иски потребителей ⇒ компания разоряется

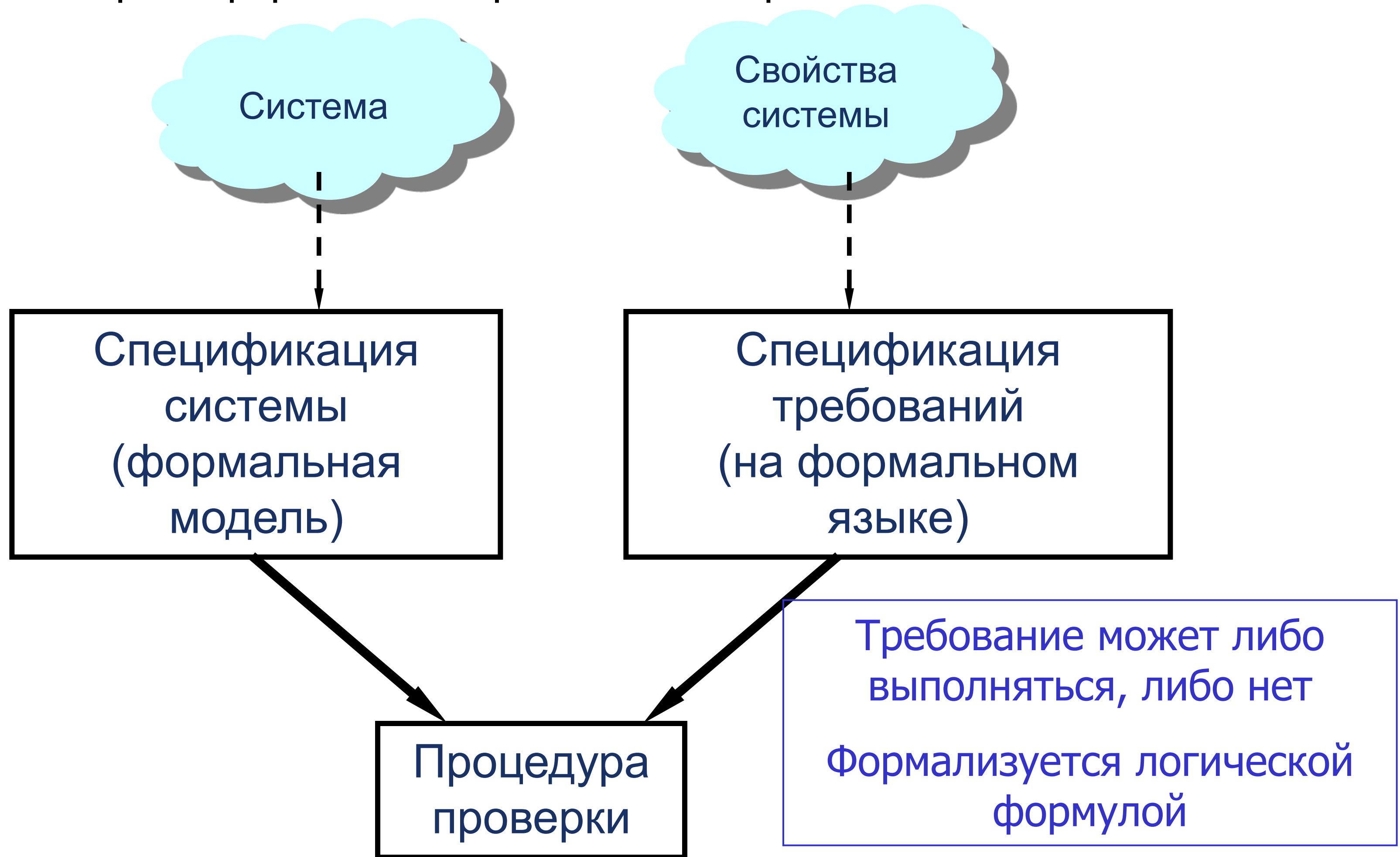
ЧТО ДЕЛАТЬ?

5 декабря 2010. Старт 'Протона-М' со спутниками ГЛОНАСС: ~ $200 млн -> в океан

В последнее десятилетие сложность разрабатываемых компьютерных систем дошла до критического уровня: требуются все более сложные программы, а технология программирования не может их разрабатывать с требуемым качеством

Верификация

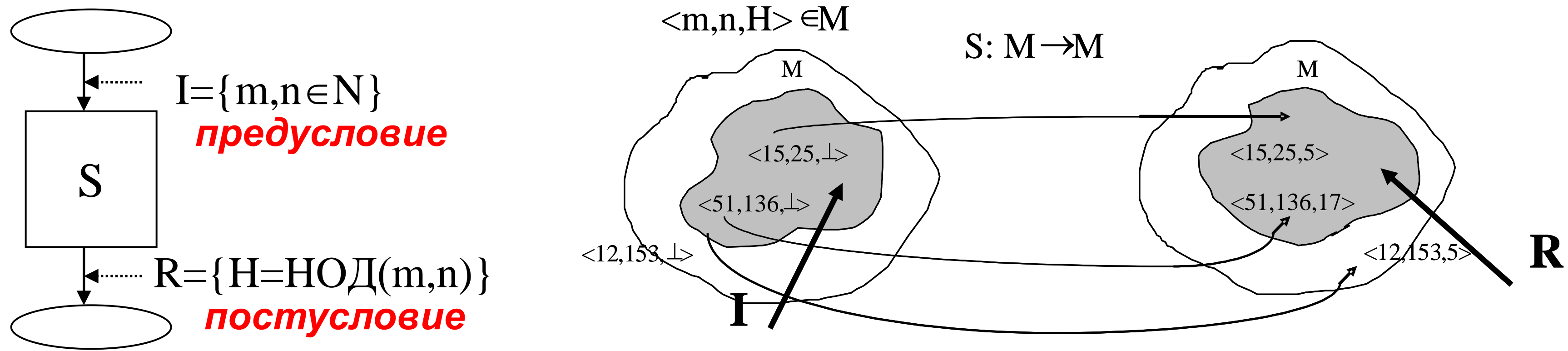
- Верификация - формальное доказательство того, что программная система удовлетворяет формально определенным требованиям



Верификация программ обработки данных

Программу обработки данных можно считать преобразователем ***состояний.*** Такие программы называются ТРАНСФОРМАЦИОННЫМИ

Состояние – вектор значений переменных программы



Как ***конечным образом*** описать все исходные и “правильные” финальные состояния???

Средством описания ***подмножеств*** (бесконечных) множества является **предикат.**

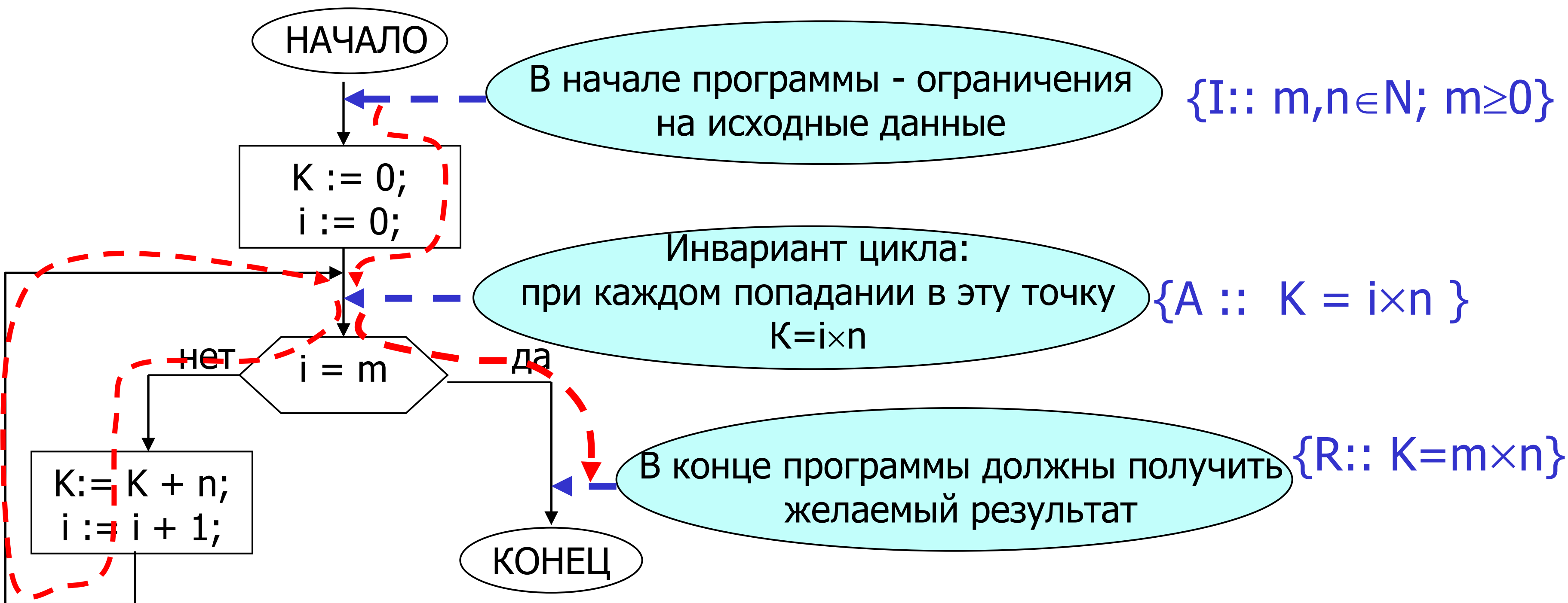
Например: x>y – все те пары x и y, в которых x > y (а их бесконечное число)

z+2x = y - определяет все тройки <x, y, z>, в которых это отношение соблюдается

<51, 136,17> - удовлетворяет предикату H=НОД(m, n), а <12,153,5> - нет

Пример доказательства корректности

Программа находит произведение K=m×n с помощью сложений



Доказательство корректности программы требует доказательства трех теорем

T1: $[\ m,n \in N \wedge\ m>0 \wedge K=0 \wedge i=0\ ] \Rightarrow\ [\ K = i \times n\ ]$

T2: $[\ (\ K=i \times n) \wedge (i= m)\ ]\ \Rightarrow\ [\ K= m \times n\ ]$

T3: $[\ (\ K\text{-}n = (i\ \text{-}1) \times n\ ) \wedge (i\ \text{-}1 \neq m)\ ] \Rightarrow [\ K = i \times n\ ]$

Пример верификации ядра реальной ОС

До последнего времени методы верификации могли использоваться для проверки корректности только небольших систем, но и для них требовались огромные усилия

Сообщение СМИ 01.10.2009

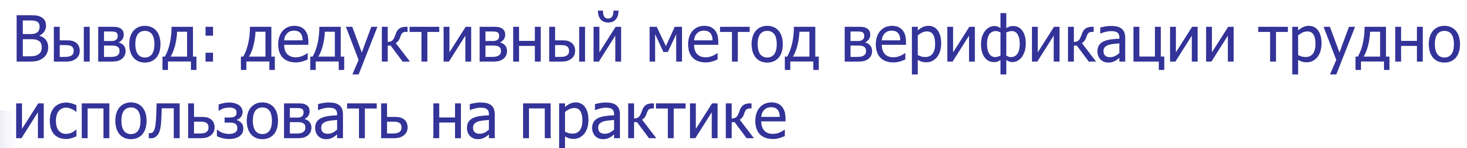
- Research Centre of Excellence компании NISTA (Australia) объявил о завершении работы по формальному доказательству корректности ядра ОС с помощью интерактивной системы доказательства теорем Isabelle
- Доказанная ОС - The Secure Embedded L4 (seL4) microkernel содержит 7,500 строк С code. Было формально доказано 10,000 промежуточных теорем, доказательство заняло 200,000 строк
- Это результат 4-х летнего труда группы из 12 исследователей NISTA под руководством Dr Klein, PhD студентов и нескольких других работников
- Этот экстраординарный результат открывает путь к разработке нового поколения ПО с беспрецедентным уровнем надежности. Это одно из самых длинных из когда-либо выполненных формальных доказательств с помощью формальных средств theorem-proving

Оценка этого результата

Построение доказательства ядра ОС secure embedded L4 потребовало ~ 60 человеко-лет работы, ~720 месяцев, 10 строк кода – 1 чел-месяц работы верификатора

Сложность доказательства ~ в 30 раз выше, чем сложность кода; одна строка кода - ~ 1 страница доказательства

Это согласуется с высказыванием Аллена Эмерсона о дедуктивной верификации: “*мы писали 15-страничный отчет о том, что программа на полстраницы корректна*”



Вывод: дедуктивный метод верификации трудно использовать на практике

- Пока нельзя говорить о широком использовании дедуктивного метода верификации в разработке реальных программ
- Можно доказывать лишь программы в несколько строк

До сих пор практики программирования считают верификацию забавой теоретиков

Model checking - прорыв в верификации

В последнее время – качественный прорыв в области верификации

Разработан метод model checking (проверка модели), основанный на изящных формальных моделях

Премия Тьюринга ACM

21 июня 2008 г премия Тьюринга была вручена трем создателям техники MODEL CHECKING, внесшим наиболее существенный вклад

Edmund M. Clarke (CMU),
E. Allen Emerson (U Texas, Austin),
Joseph Sifakis (Verimag, France)

“*за их роль в превращении метода Model checking в высокоэффективную технологию верификации, широко используемую в индустрии разработки ПО и аппаратных средств*”

ACM President Stuart Feldman :

“*Это великий пример того, как технология, изменившая промышленность, родилась из чисто теоретических исследований*”

Ограниченность классической логики для выражения свойств динамики (процессов, развивающихся во времени)

- Классическая логика
  - Высказывания статичны, неизменны во времени
- Пример - некоммутативность конъюнкции, A&B ≠ B&A:
  - *"Джону стало страшно и он убил"* ⇐?⇒ *"Джон убил и ему стало страшно"*
  - *"Смит умер и его похоронили"* ⇐?⇒ *"Смита похоронили и он умер"*
  - *"Джейн вышла замуж и родила ребенка"* ⇐?⇒ *"Джейн родила и вышла замуж"*
- В классической логике высказываний не формализуются:
  - *Путин - президент России* (истинно только в какой-то период)
  - *Мы не друзья, пока ты не извинишься*
  - *Если m поступило на вход в канал, то m обязательно появится на выходе*
  - *Запрос к лифту с произвольного этажа будет обязательно удовлетворен*

Мы хотим верифицировать системы, развивающиеся во времени, а обычная классическая логика неадекватна для выражения их свойств!

Как ввести понятие времени в логические утверждения?

Использование модальностей (Tense Logic, A.Prior)

Fq – q случится когда-нибудь в будущем

Pq – q случилось когда-то в прошлом

Gq – q всегда будет в будущем

Hq – q всегда было в прошлом

pUq – q случится в будущем, а до него непрерывно будет выполняться p

Xp - p выполнится в следующий момент

Примеры формализации высказываний

- *Джон умер и его похоронили*

  P(Джон_умирает ∧ XF *Джона_хоронят*)

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

  G(P Увидел ⇒ G(Встретил ⇒ X Узнал))

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

  PG Ленин_жив

- *Мы придем к победе коммунистического труда!*

  F Коммунистический_труд_победил!

- *Сегодня он играет джаз, а завтра Родину продаст*

  G(Он_играет_джаз ⇒ FX Он_продает_Родину)

- *Раз Персил – всегда Персил (раз попробовав, будешь всегда)*

  G(Персил ⇒ G Персил)

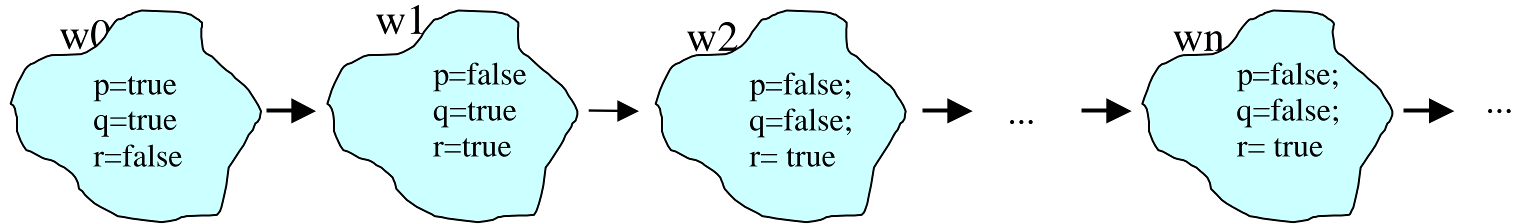
- *Любой запрос к ресурсу будет висеть до подтверждения либо отклонения*

  G(Request ⇒ Request U (Reject ⊕ Confirm))

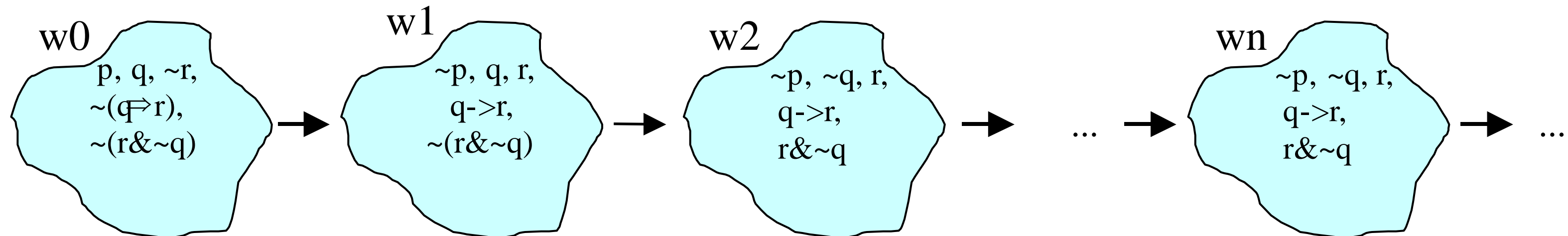
LTL в дискретном времени – А. Пнуэли (1977)

Модель времени – последовательность целых (вчера, сегодня, завтра, …)

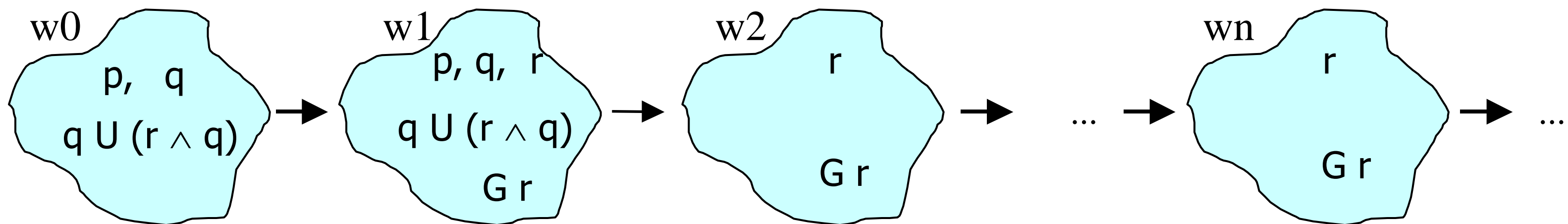
Семантика “возможных миров” Лейбница, в каждом свое понимание истинности:



В каждом мире произвольная логическая формула истинна, либо нет:



Это же справедливо и для произвольной темпоральной формулы:



На всей цепочке выполняются формулы p, q, qU(r∧q), - *они истинны в w0*

Linear Temporal Logic (LTL)

- Формальное определение логики LTL
  - Формула ϕ PLTL это :
    - атомарное утверждение (атомарный предикат) p, q, ...,
    - или Формулы, связанные логическими операторами $\vee$, $\neg$
    - или Формулы, связанные темпоральными операторами U, X
  - Модальных операторов прошлого в LTL нет

Грамматика: $\phi ::= p \mid \phi \vee \phi \mid \neg\phi \mid X\phi \mid \phi \, U \, \phi \mid F\phi \mid G\,\phi$

Выводимые темпоральные операторы :

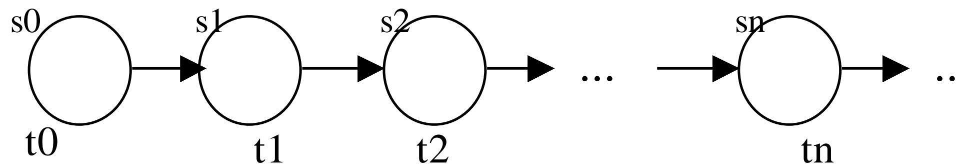
$Fp = true \; U \; p$

$Gp = \neg F \neg p$

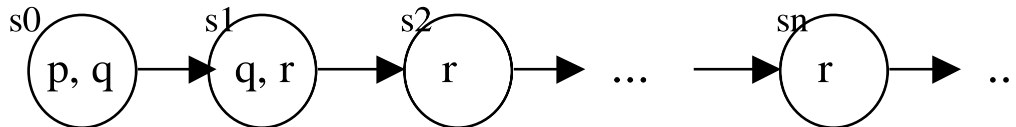
Прошлое при анализе технических систем не важно

TL и анализ дискретных технических систем

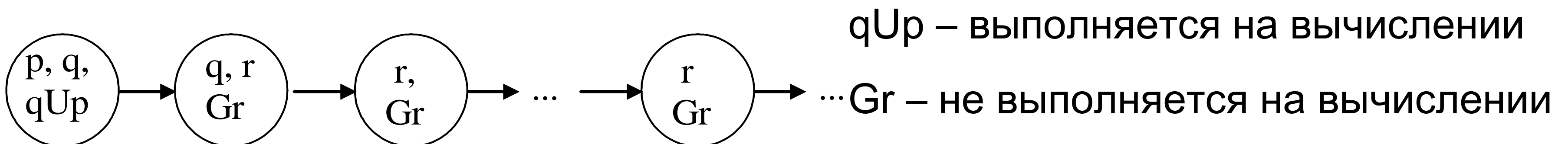
Последовательность "*миров*" в TL можно толковать как бесконечную последовательность состояний дискретной системы, а отношение достижимости – как дискретные переходы системы:



Атомарные предикаты - базисные свойства процесса в состояниях:



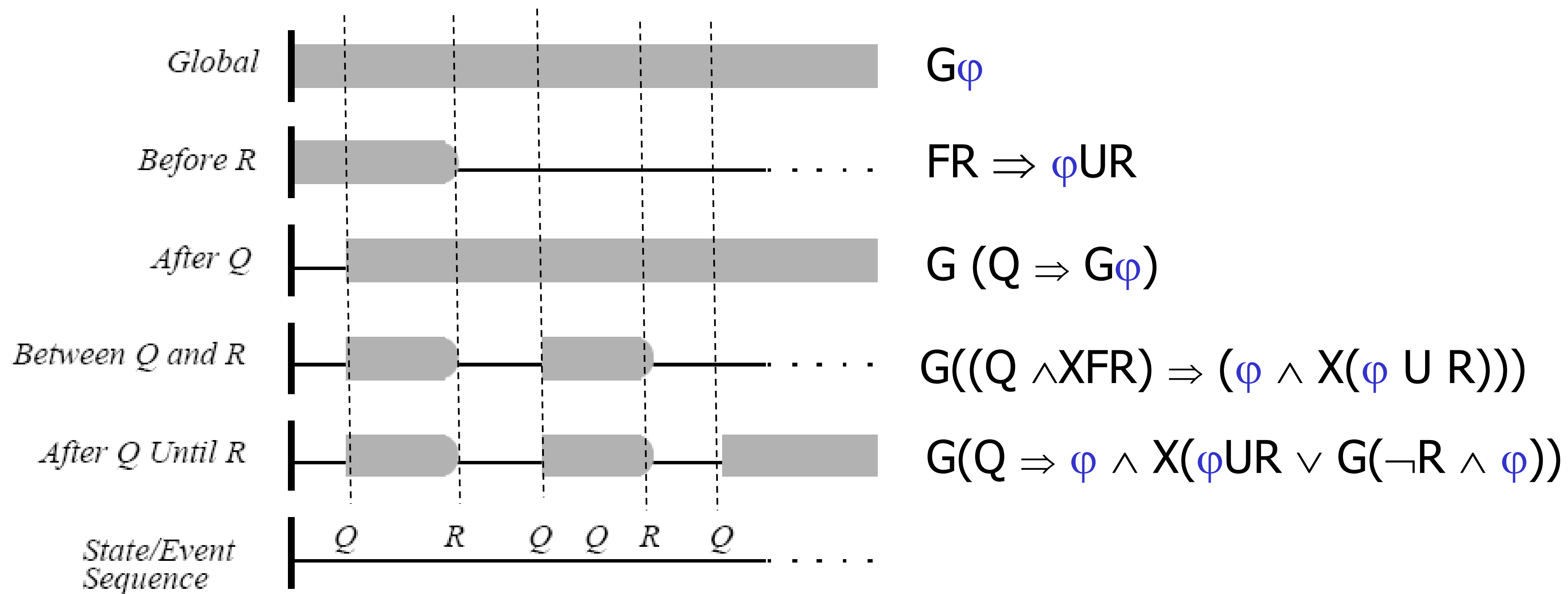
Производные темпоральные формулы в состояниях – это свойства вычисления в будущем, динамики процесса:



qUp – выполняется на вычислении

Gr – не выполняется на вычислении

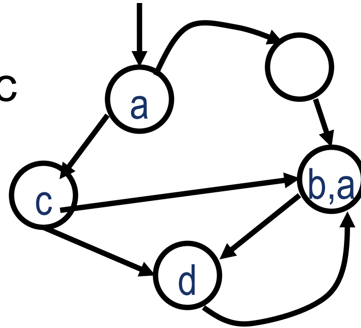
Шаблоны выполнения темпоральных свойств в технических системах



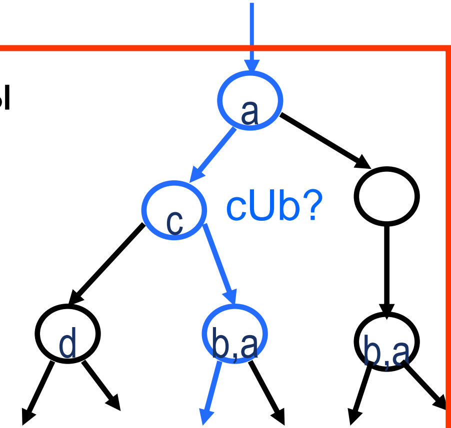
Линейное и ветвящееся время

Как бесконечные вычисления задать конечным образом?

Структура Крипке – система переходов с помеченными состояниями и непомеченными переходами



Развертка структуры Крипке определяет бесконечные цепочки состояний - возможные ВЫЧИСЛЕНИЯ



Каждое состояние может иметь не одну, а множество цепочек – продолжений, и является корнем своего дерева историй (вычислений)

Но как понимать формулы LTL: Gp, pUq, … в состоянии? Для какого пути?

Ввести “*кванторы пути*”

E $\varphi \equiv$ “*существует путь из данного состояния, на котором формула пути* ϕ *истинна*” (**E**xists)

A $\varphi \equiv$ “*для всех путей из данного состояния формула пути* ϕ *истинна*” (**A**ll)

Очевидно, **A** $\varphi \equiv \neg$**E** $\neg\varphi$

Темпоральные логики LTL и CTL

В LTL (Linear Temporal Logic) – темпоральной логике линейного времени, формулы пути предваряются квантором пути A: они должны выполняться для всех вычислений структуры Крипке

В CTL (Computational Tree Logic), в темпоральной логике ветвящегося времени, каждый темпоральный оператор предваряется квантором пути A или E

Формулы LTL:

$AG(p \Rightarrow F\ q)$

$A\ (\neg a \vee Gb\ \&\ (aU\ \neg c))$

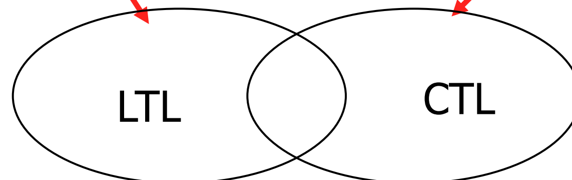
$A\ (a\ U\ \neg b)$

Формулы CTL:

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

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

$A\ (a\ U \neg\ b\ )$

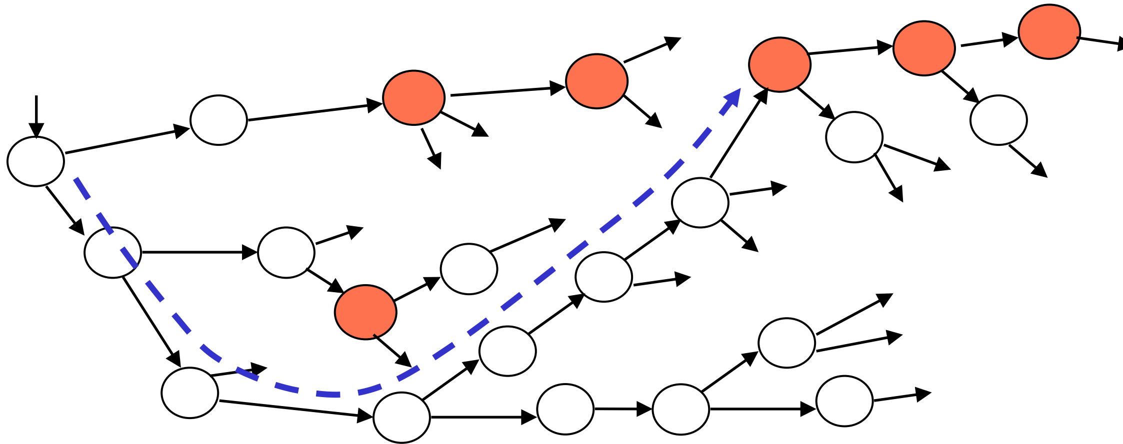


Примеры спецификации требований в CTL

- AGAF Restart
  - из любого состояния при любом функционировании системы обязательно вернемся в состояние рестарта
- $\neg$EF(int >0.01)
  - не существует такого режима работы, при котором интенсивность облучения пациента превысит 0.01 радиан в сек
- AG (req $\Rightarrow$ A (req U ack))
  - во всех режимах после того, как запрос req установится, он никогда не будет снят, пока на него не придет подтверждение
- E[p U A [q U r]]
  - существует режим, в котором условие p будет истинным с начала вычисления до тех пор, пока q не станет непрерывно активно до выполнения r

Пример свойства, выраженного формулой логики CTL

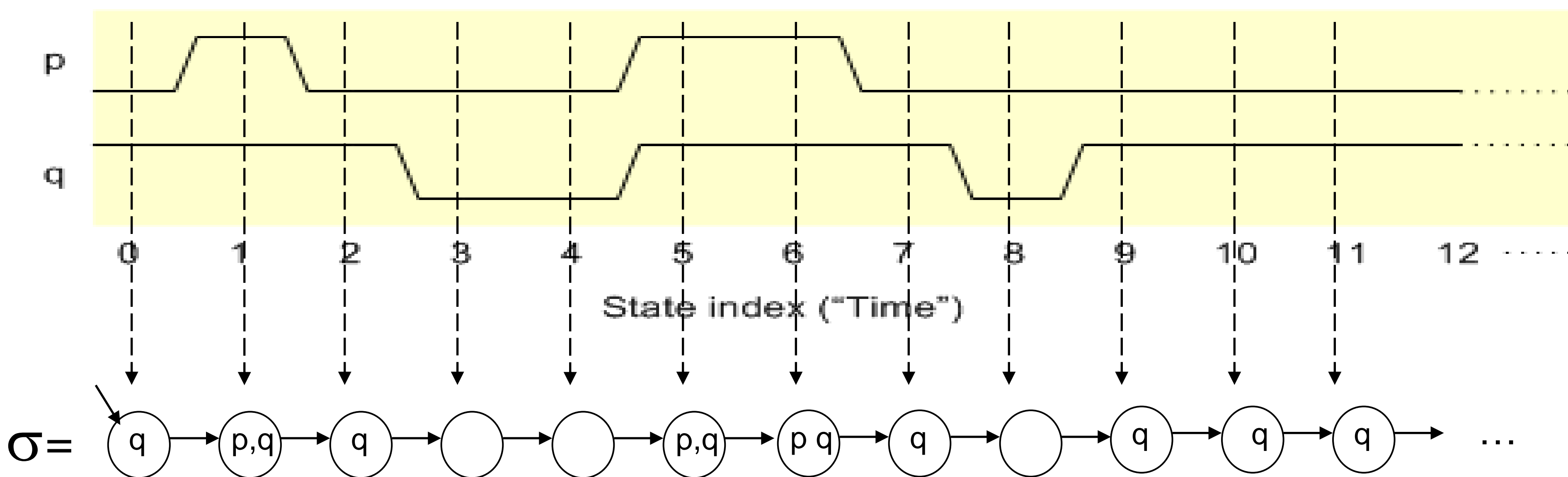
- *Любой грешник всегда имеет шанс вернуться на путь истинной веры*



AG EF EG ‘*истинная вера*’

Всегда, куда бы мы ни попали в нашей жизни (AG), существует такой путь, что на нем в конце концов обязательно попадем (EF) в состояние, с которого идет путь (EG) ‘истинной веры’

Структура Крипке для дискретных схем



$\sigma_0 \vDash \neg p$

$\sigma_0 \vDash F\ \neg q$

$\sigma_0 \vDash G(p \rightarrow q)$

$\sigma_0 \vDash q\ U\ p$

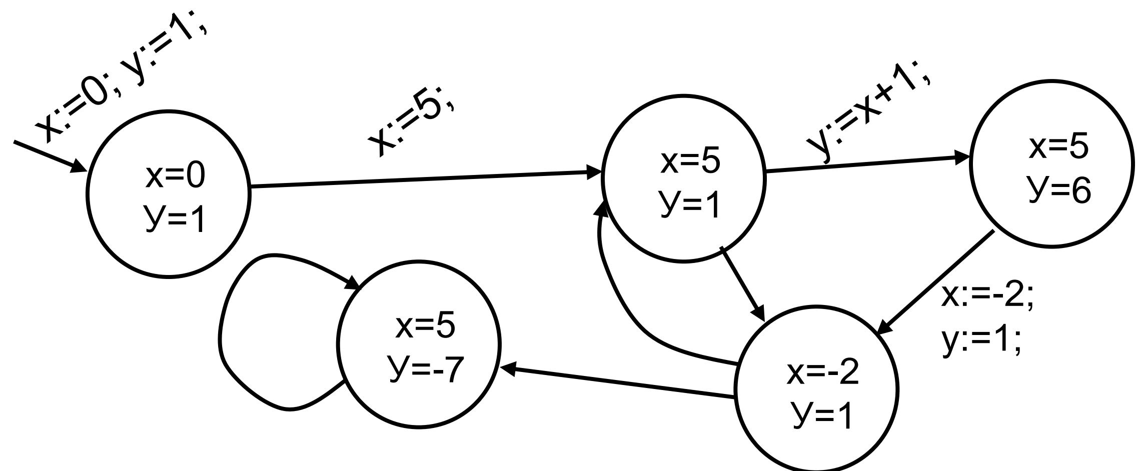
$\sigma_0 \vDash FG(\neg p \wedge q)$

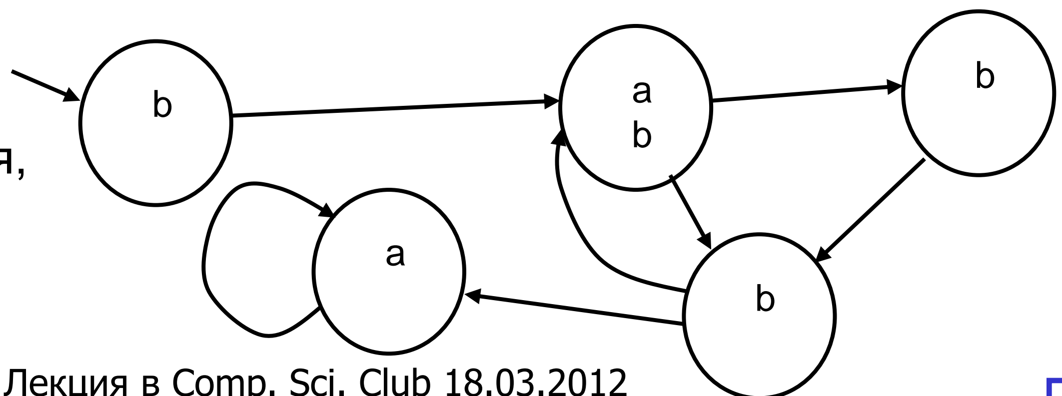
Структура Крипке как модель программы

```
begin
x:=0; y:=1;
while x+z < 5 do
{ x:=5;
if z=1 then y:=x+1;
x:= -2; y:=1;
}
y:= x*y-5; x:=5;
end
```

Состояние программы – вектор значений ее переменных И метки (pc)

Переходы – изменение переменных программы операторами И/ИЛИ только pc:

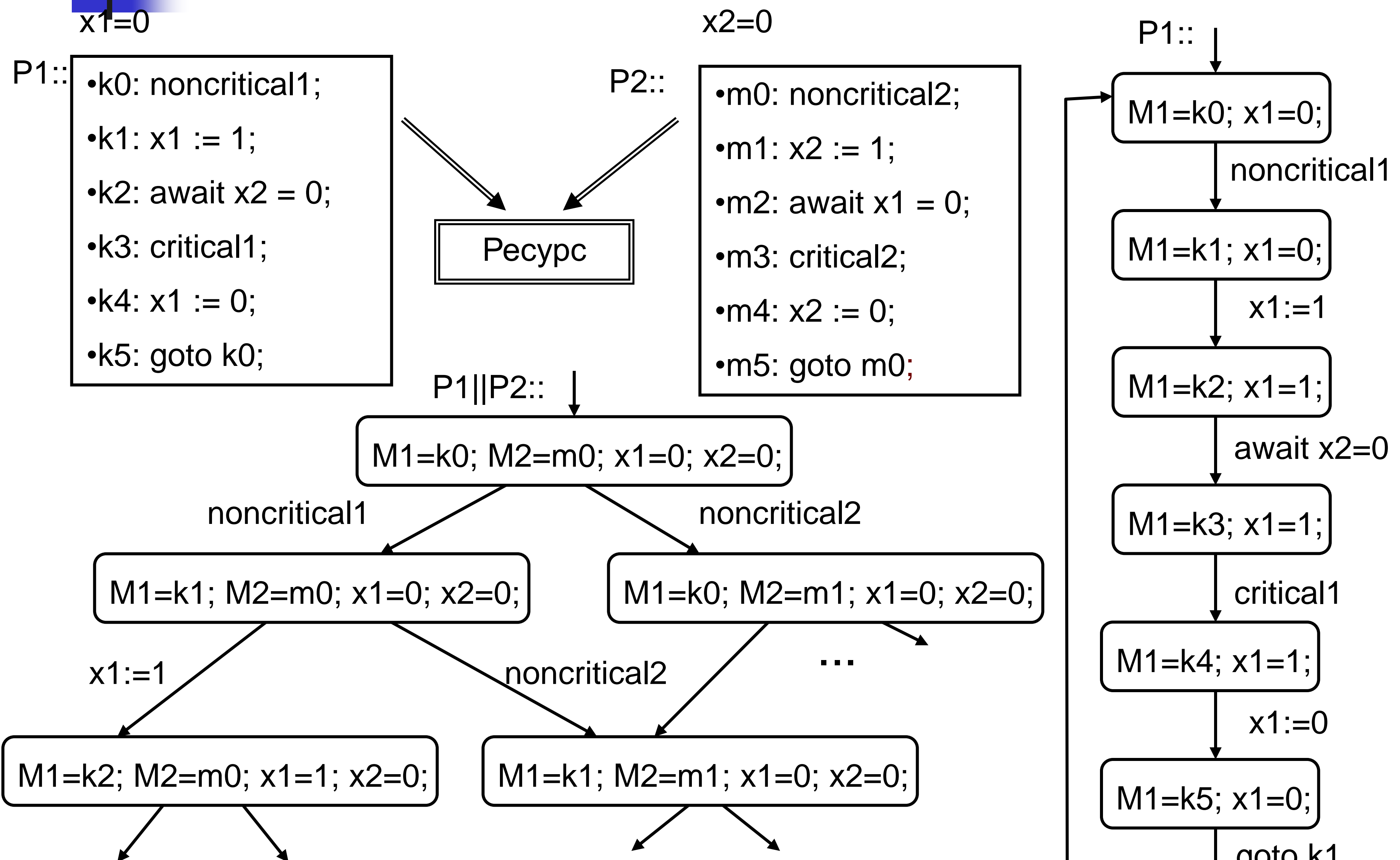




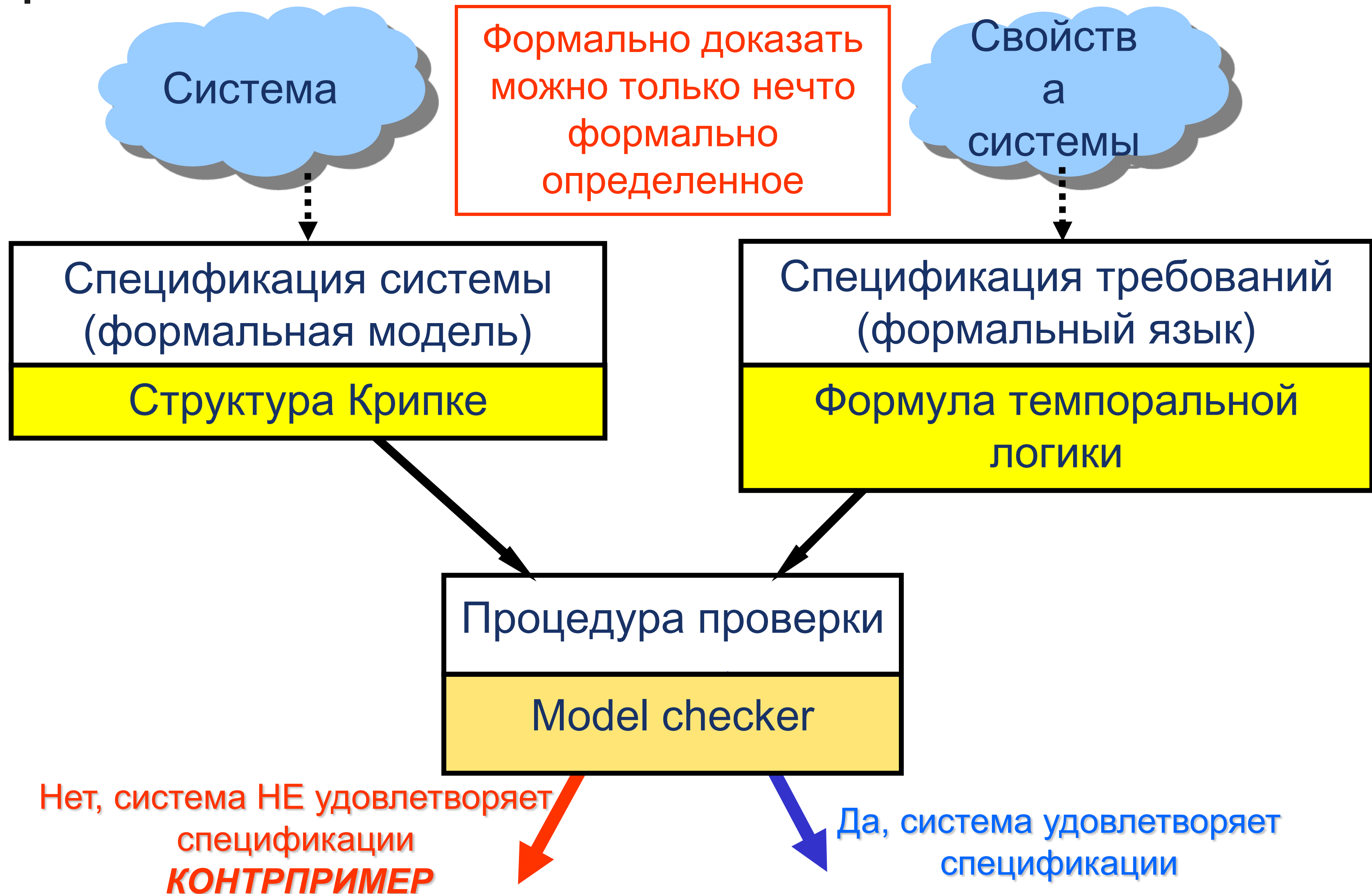
Пусть атомарные утверждения, *ИНТЕРЕСУЮЩИЕ НАС:*

a=x>y; b=|x+y|<3

Структура Крипке - модель параллельной программы



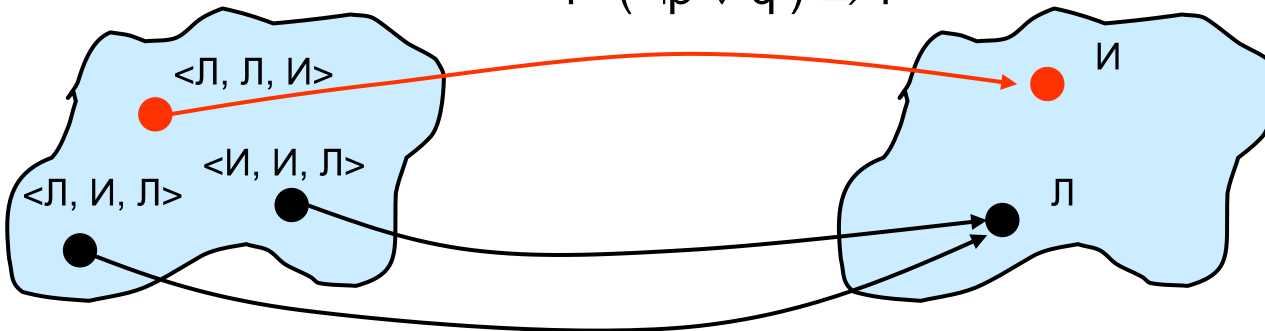
Общая схема верификации Model checking



Model Checking –проверка ~~на~~ модели

Логика высказываний

$\Phi=(\neg p \vee q) \Rightarrow r$

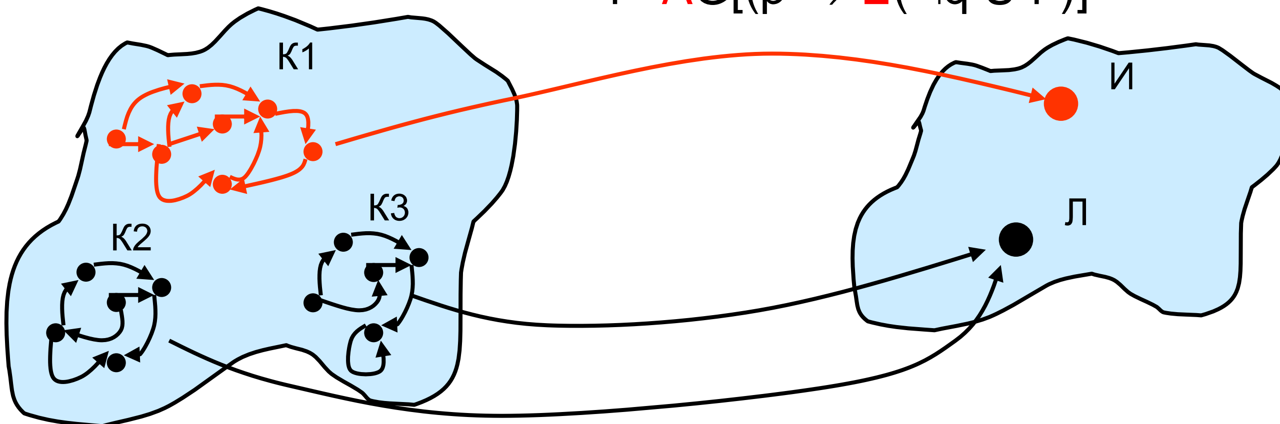


Интерпретации логики высказываний – наборы значений переменных <p, q, r>

Интерпретация <Л, Л, И> - модель формулы Ф

Темпоральная логика

$\Phi=AG[(p \Rightarrow E(\neg q \ U \ r)]$



Интерпретации темпоральной логики – структуры Крипке, в каждом состоянии которых свой набор значений переменных <p, q, r>

Интерпретация К1 - модель формулы Ф

Model Checking для CTL – проверка на развертке (неформально)

LTL CTL

M::

s0 p, q; s1 q, r; s2 r

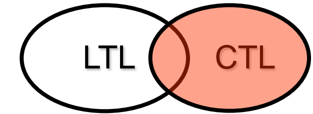
M::

s0 p,q; s1 q,r; s2 r; s0 p,q; s2 r; s2 r; s1 q,r; s2 r; s2 r; …

1. $M, s_0 \models p \wedge q$
2. $M, s_0 \models EX\,(q \wedge r)$
3. $M, s_0 \models \neg AX(q \wedge r)$
4. $M, s_0 \models \neg\, EF(p \wedge r)$
5. $M, s_0 \models \neg\, EGr$
6. $M, s_0 \models AFr$
7. $M, s_0 \models E[\,(p \wedge q)Ur]$
8. $M, s_0 \models A[\,p\ U\ r\,]$
9. $M, s_0 \models EF\ AGr$

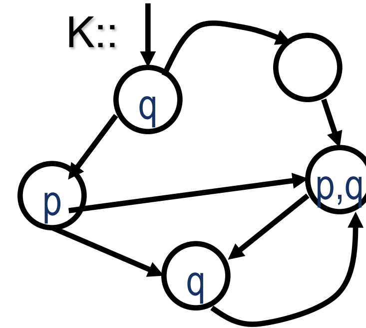
Как построить общий алгоритм, который может проверить выполнимость любой темпоральной формулы на любой структуре Крипке?

Model checking для CTL



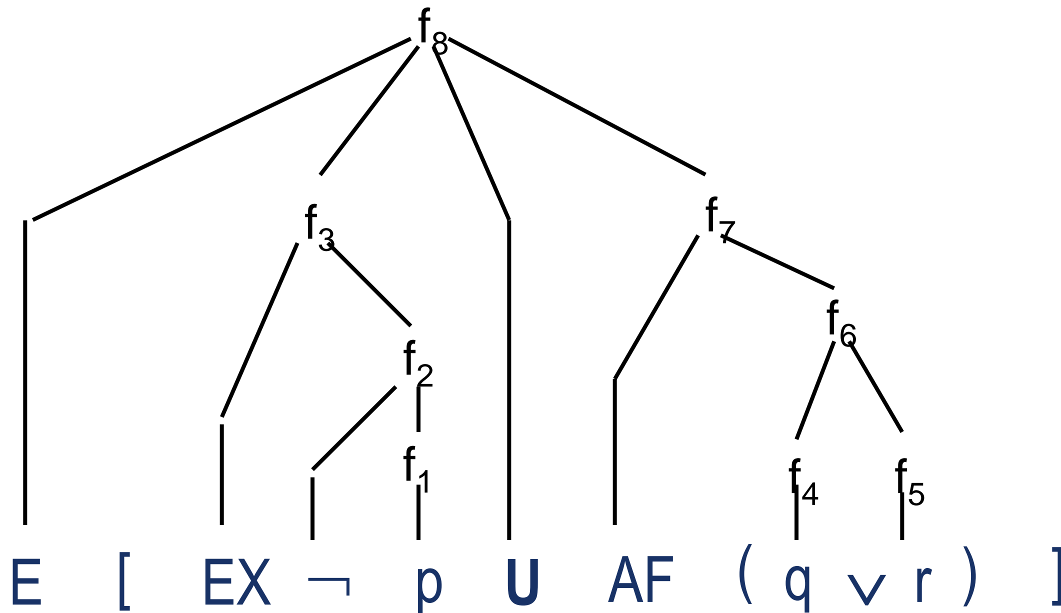
Дано: структура Крипке K и формула Ф CTL.

Построить: множество состояний K, на которых формула Ф истинна.



Алгоритм разметки выполняется по структуре формулы

Пример: $\phi = E\ [\ EX\neg p\ U\ AF\ (q \vee r)\ ]$

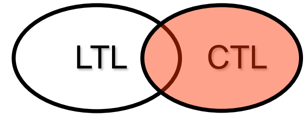


Последовательно снизу вверх:

$f_1 = p;$
$f_2 = \neg f_1;$
$f_3 = EX\ f_2$
$f_4 = q\ ;$
$f_5 = r\ ;$
$f_6 = f_4 \vee f_5$
$f_7 = AF\ f_6$
$f_8 = E[\ f_3\ U\ f_7\ ]$

Обычный синтаксический анализ

Алгоритм разметки состояний для CTL



Грамматика формул CTL (один из базисов):

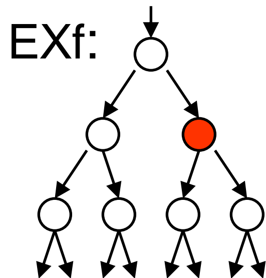
$$f ::= p \mid \neg f \mid f_1 \vee f_2 \mid EX\ f \mid E(f_1 U f_2) \mid AF\ f$$

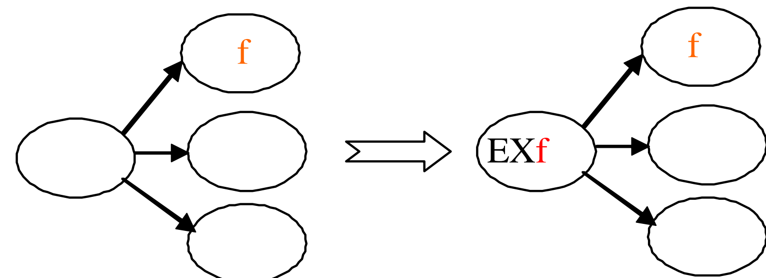
Для Φ = p : множество состояний, помеченных атомарным предикатом, задано

Для Φ = ¬f : помечаем формулой Φ состояния, которые не были помечены f

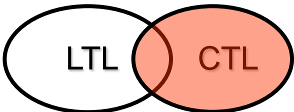
Для Φ = f1∨ f2 : помечаем те состояния, которые были помечены или f1, или f2

Для Φ = EX f: помечаем Φ те состояния, из которых есть хотя бы один переход в состояние, помеченное f

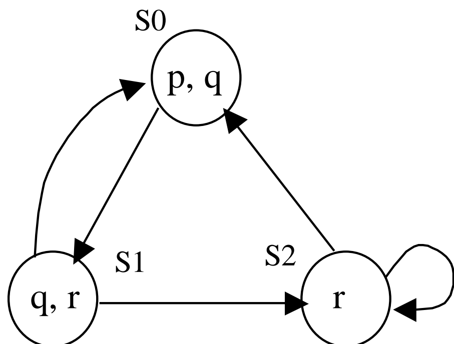




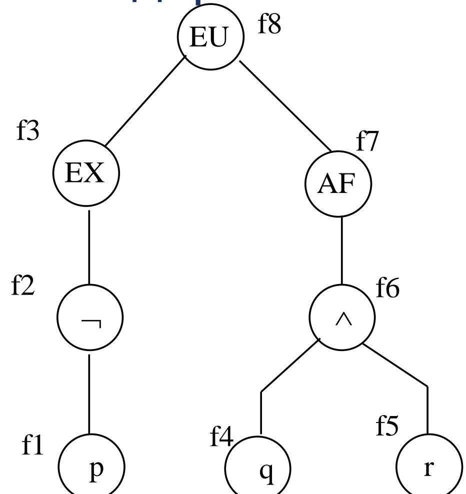
Model Checking – пример анализа



$\phi = E\ \ EX\neg p\ U\ AF\ (q \wedge r)$



Синтаксическое дерево:



$f1 = p$

$f2 = \neg f1$

$f3 = EX\ f2$

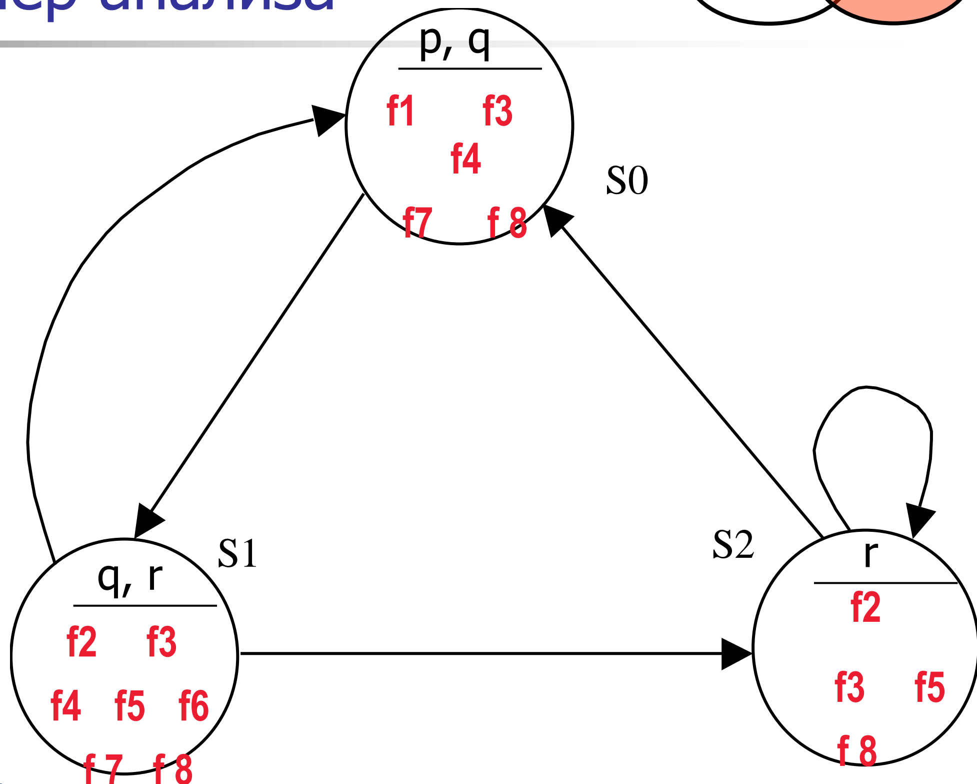
$f4 = q$

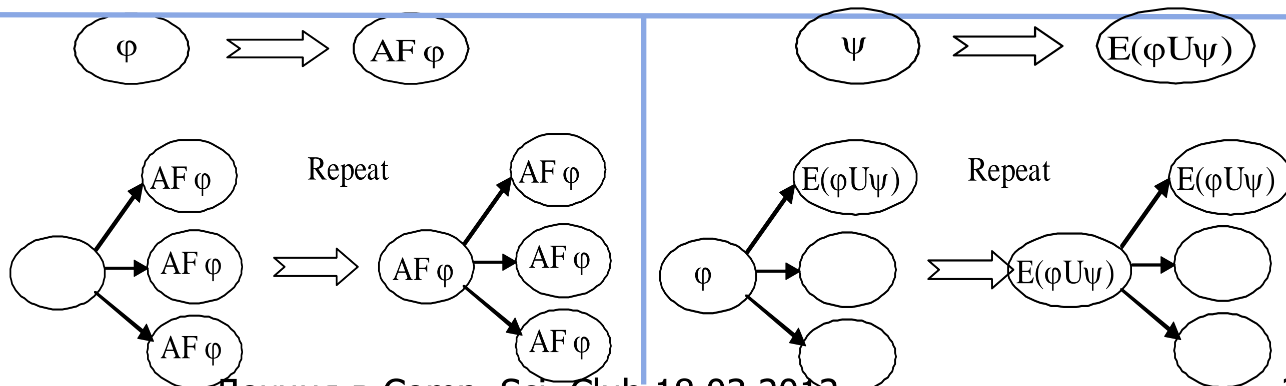
$f5 = r$

$f6 = f4 \wedge f5$

$f7 = AF\ f6$

$f8 = E[\ f3\ U\ f7]$



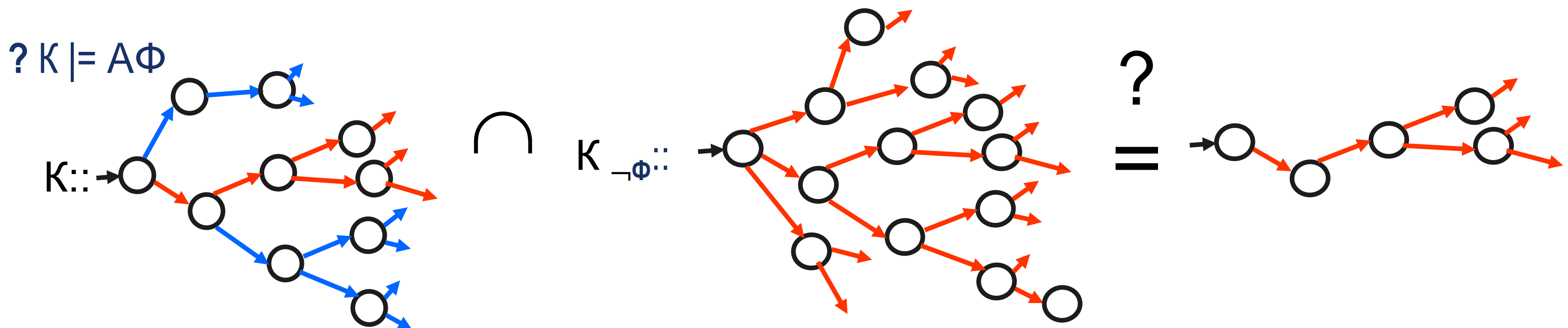


Model Checking для LTL

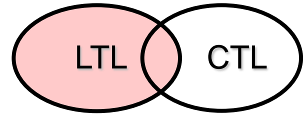
LTL CTL

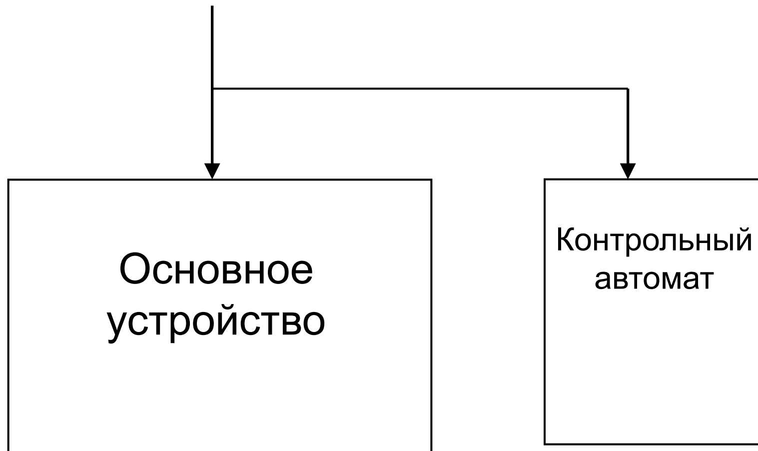
В LTL – формулы пути, нельзя использовать алгоритм маркировки

- Строить алгоритм проверки модели для LTL-формулы, перебирая все пути – бессмысленно
- Другой метод – проверить, есть ли контрпримеры, т.е. проверить, есть ли вычисления, удовлетворяющие $\neg Ф$. Единственный контрпример достаточен, чтобы опровергнуть К |= АФ
- Для этого опишем конечным образом все вычисления, НЕ удовлетворяющие Ф
- Проверим, есть ли пересечения с вычислениями К



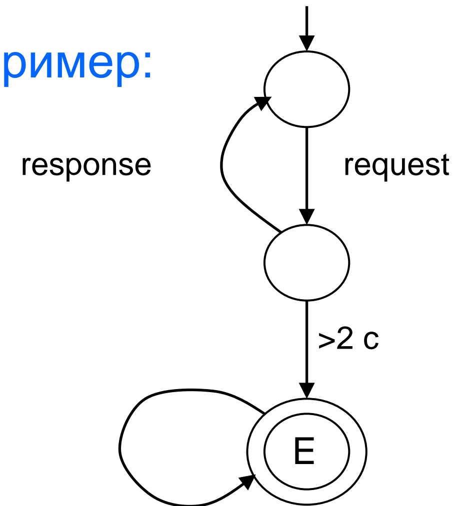
Контрольный автомат - один из методов





Контроль правильности функционирования системы с помощью контрольного автомата (watchdog – сторожевой пес)

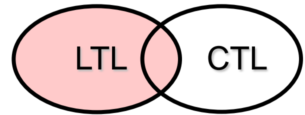
Пример:

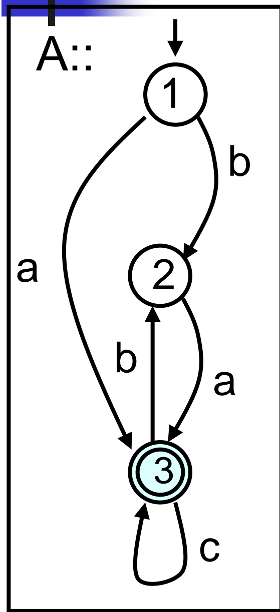


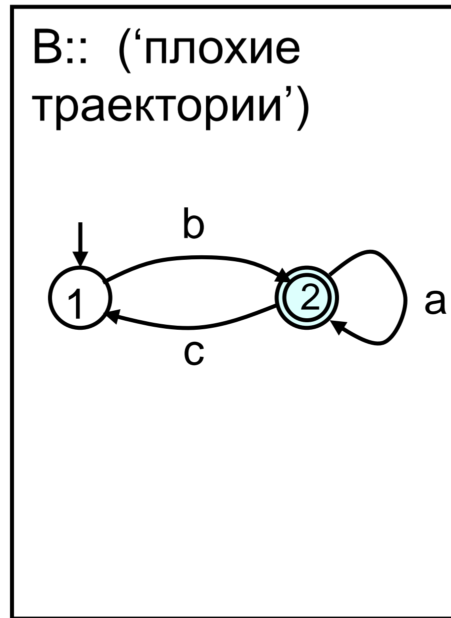
Переход в состояние E, если ответ на любое посланное сообщение придет позже, чем через 2 с

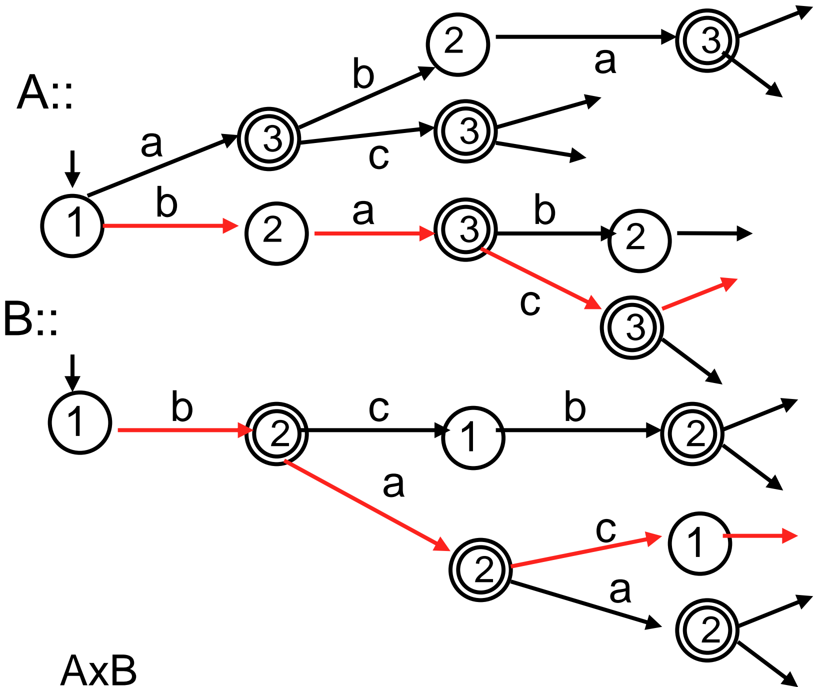
Подобная техника используется для проверки выполнения LTL формул на моделях программ. Этот теоретико-автоматный подход - один из наиболее удобных подходов к верификации LTL

Как найти пересечение автоматных языков?





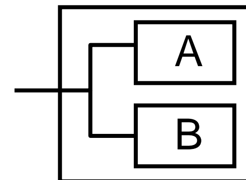




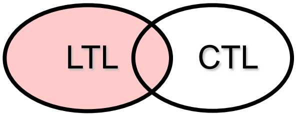
$L_A \cap L_B$ =? – непросто!

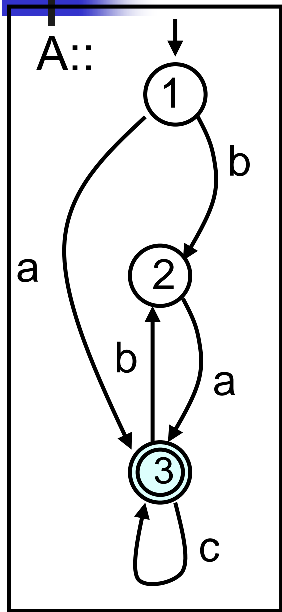
$L_A \cap L_B = L_{AxB}$

AxB

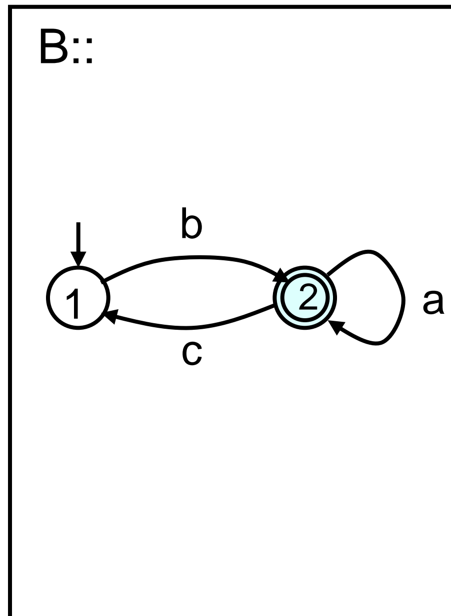


Как найти пересечение автоматных языков?

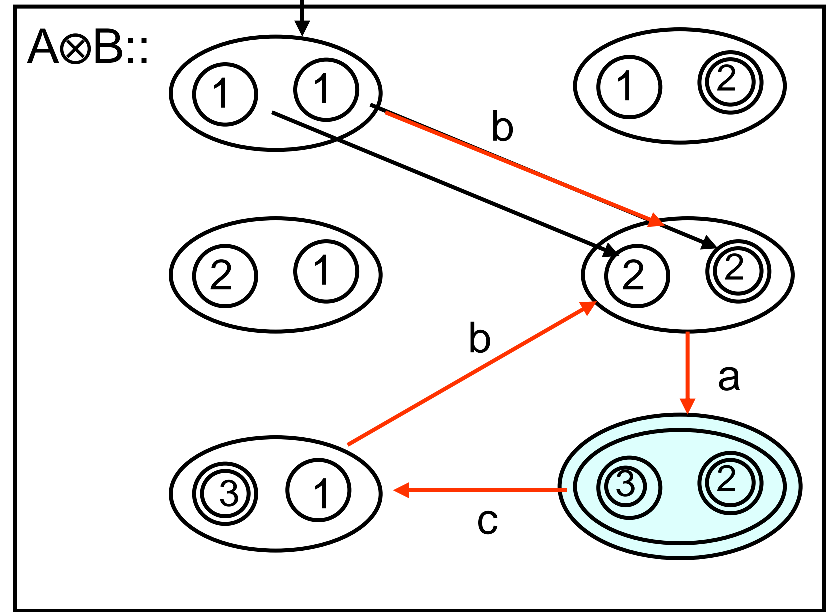




⊗

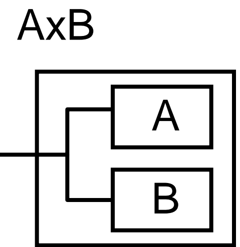


=



$L_A \cap L_B = ?$ – непросто!

$L_A \cap L_B = L_{AxB}$



- Синхронная композиция автоматов – это просто два автомата, стоящих рядом
- Язык, допускаемый автоматом A ⊗ B – это пересечение языков, допускаемых каждым автоматом, и A, и B

Model Checking для LTL: автоматы Бюхи

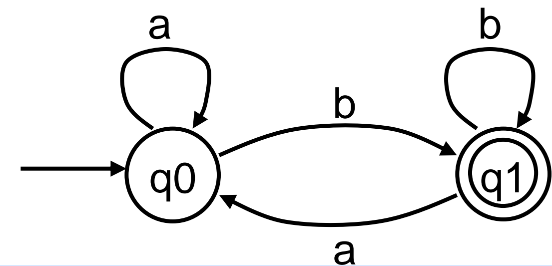
Как описать конечным образом бесконечное множество бесконечных цепочек?

LTL-формула конечным образом описывает множество бесконечных траекторий, ей удовлетворяющих. Конечные автоматы также конечным образом задают множество траекторий (последовательностей цепочек). Но конечные автоматы-распознаватели языков работают с конечными цепочками. Нужен новый тип автомата, распознающего БЕСКОНЕЧНЫЕ последовательности

Такой автомат был введен Richard Buchi

Цепочка допускается автоматом Бюхи B, iff существует заключительное состояние автомата, которое проходится бесконечное число раз при приеме этой цепочки

Теория формальных языков построена для анализа КОНЕЧНЫХ цепочек



Конечный Автомат: допускается любая цепочка, оканчивающаяся на b:

(a+b)*b

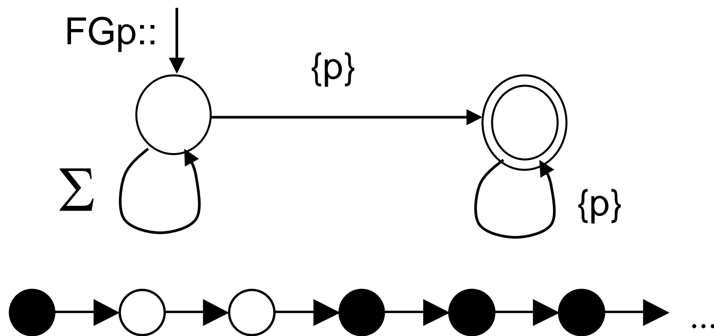
Автомат Бюхи: допускаются цепочки, в которых b встречается бесконечно много раз :

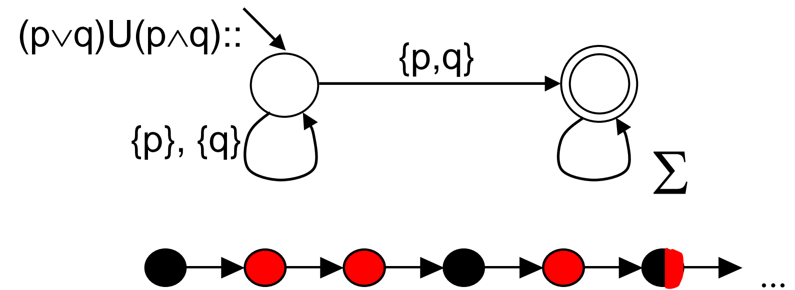
GFb

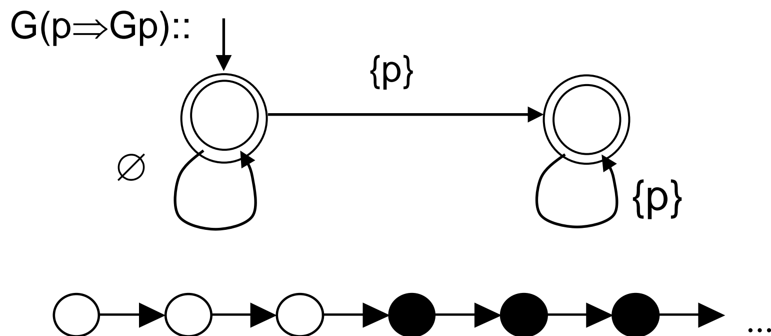
Примеры автоматов Бюхи, допускающих цепочки, удовлетворяющие LTL формулам

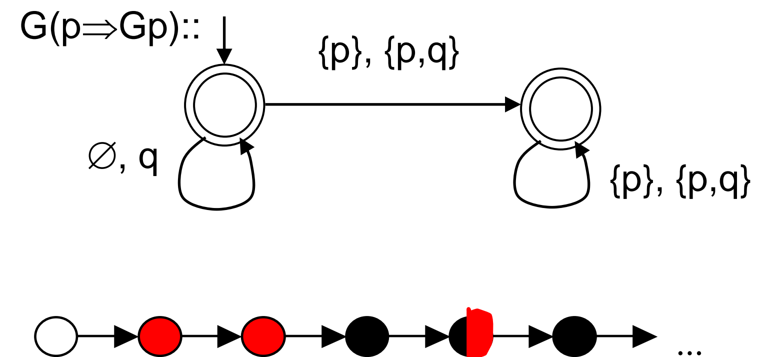
AP={p}; Σ={ ∅, {p} }

AP={p,q}; Σ={ ∅, {p}, {q}, {p,q} }





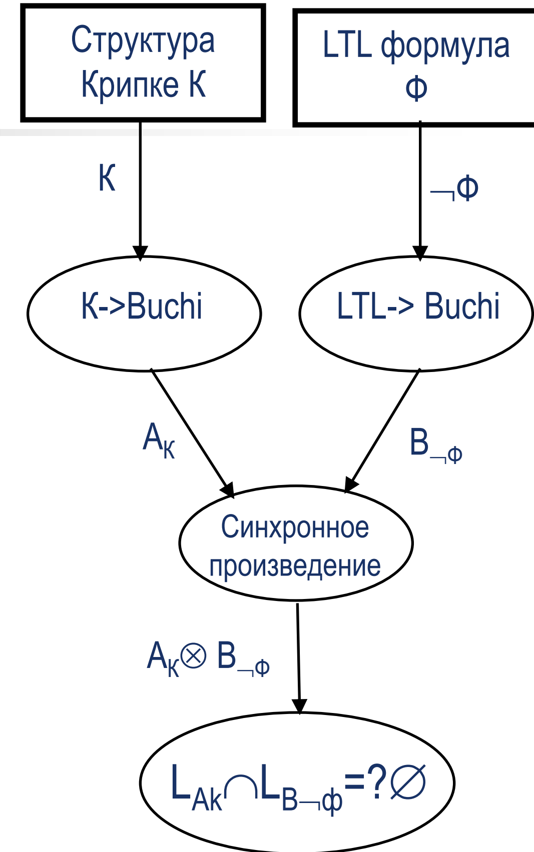




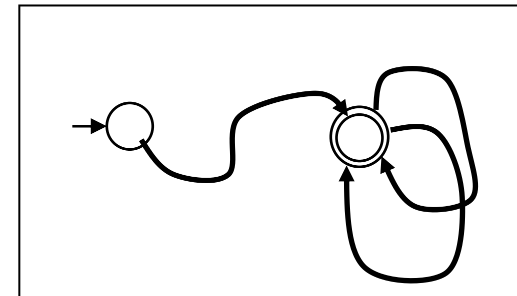
Model Checking для LTL

Общий алгоритм проверки выполнимости формулы LTL для структуры Крипке К:

- 1. По структуре К строится автомат Бюхи A_K. Этот автомат допускает все возможные вычисления структуры К
- 2. По формуле LTL Φ строится автомат Бюхи $B_{\neg\Phi}$, допускающий множество вычислений, которые удовлетворяют $\neg\Phi$
- 3. Строится автомат $A_K \otimes B_{\neg\Phi}$. Этот автомат – синхронная композиция автоматов A_K и $B_{\neg\Phi}$, допускает пересечение языков, допускаемых каждым компонентным автоматом
- 4. Формула Φ выполняется для К, если и только если $A_K \otimes B_{\neg\Phi}$ допускает пустой язык



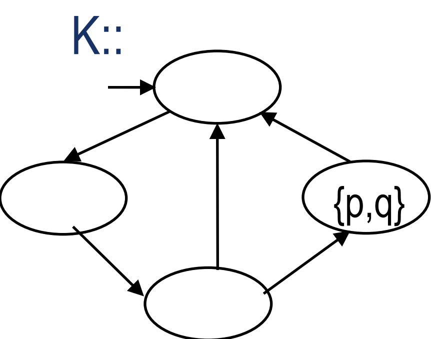
Существует простой алгоритм проверки пустоты языка, распознаваемого автоматом Бюхи



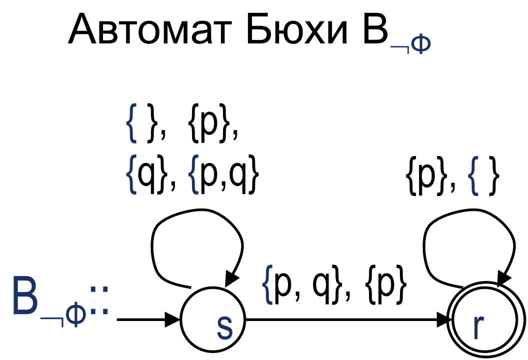
Model Checking для LTL: Пример

Проверим, выполняется ли формула $\Phi = G(p \Rightarrow XFq)$ на структуре Крипке К?

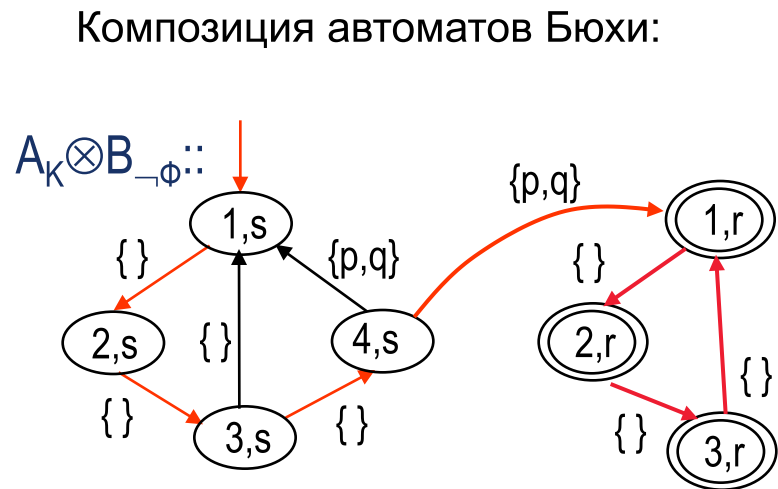
Отрицание Ф: $\neg\Phi = \neg G(p \Rightarrow XFq) = F(p \wedge XG\neg q)$



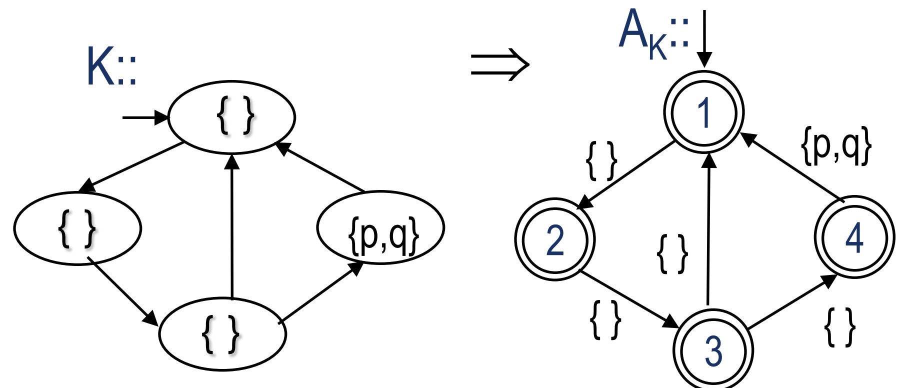
Автомат Бюхи $B_{\neg\Phi}$



Композиция автоматов Бюхи:



Структура Крипке К $\Rightarrow$ автомат Бюхи A_K

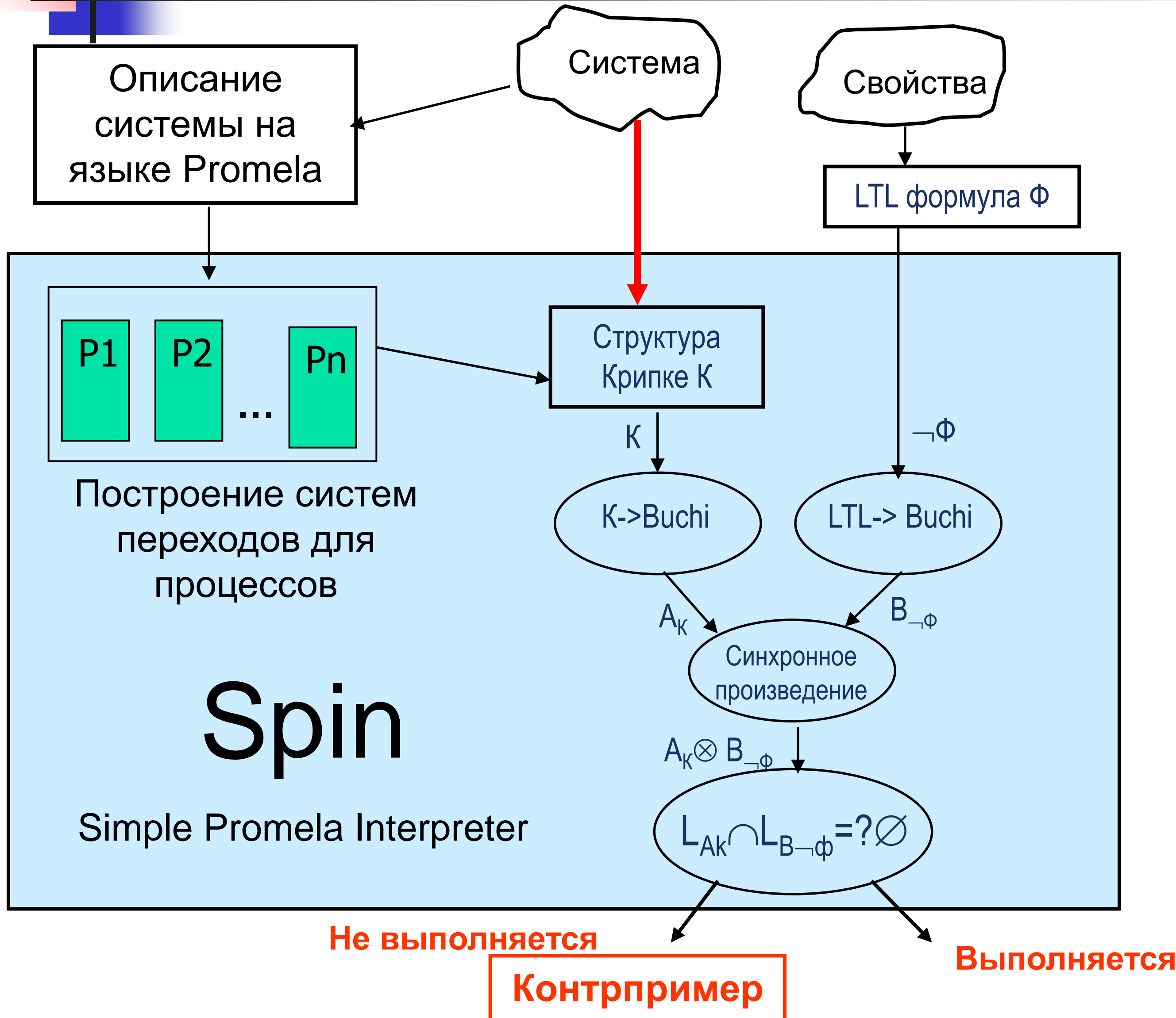


Есть цикл, включающий принимающее состояние $\Rightarrow$ язык $L_{AK \otimes B\neg\Phi}$ непуст.
К не удовлетворяет Ф.
Контрпример: $1234(123)^{\omega}$

Контрпример помогает найти и понять ошибку!!!

Система верификации Spin

ACM System Software Award 2001



Вход в систему Spin:

1. Модель взаимодействующих процессов на Promela (Protocol Meta Language)
2. Атомарные предикаты, выражающие интересующие нас базовые свойства
3. Проверяемое свойство в виде формулы PLTL

"Push-button technique"

Примеры использования подхода

Использование Model checking для верификации программных систем

- Cambridge ring protocol
- IEEE Logical Link Control protocol, LLC 802.2
- фрагменты больших протоколов XTP и TCP/IP
- отказоустойчивые системы, протоколы доступа к шинам, протоколы контроля ошибок в аппаратуре,
- криптографические протоколы
- протокол Ethernet Collision Avoidance
- DeepSpace1 (NASA). Уже после тщательного тестирования и сдачи системы были найдены критические ошибки
- SLAM: Microsoft включил Model checking в Driver Development Kit для Windows

Пример: отчет фирмы Rockwell Collins 2007г.

ADVANCED COMPUTING SYSTEMS

Formal Verification of Flight Critical Software

Dr. Steven P. Miller
Advanced Computing Systems

Elise A. Anderson
Commercial Systems Flight Control

Rockwell Collins
400 Collins Road NE, MS 108-206
Cedar Rapids, Iowa 52498

{spmiller,eaanders}@rockwellcollins.com

Rockwell Collins

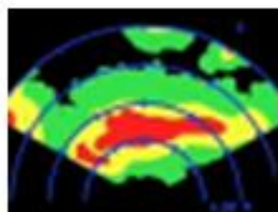
Advanced Technology Center Slide 1

Who Are We?

ADVANCED COMPUTING SYSTEMS

A World Leader In Aviation Electronics And Airborne/ Mobile Communications Systems For Commercial And Military Applications

- Communications
- Navigation
- Automated Flight Control
- Displays / Surveillance
- Aviation Services
- In-Flight Entertainment
- Integrated Aviation Electronics
- Information Management Systems



Rockwell Collins

Advanced Technology Center Slide 4

Methods and Tools for Flight Critical Systems Project

ADVANCED COMPUTING SYSTEMS

- Five Year Project Started in 2001
- Part of NASA's Aviation Safety Program (Contract NCC-01001)
- Funded by the NASA Langley Research Center and Rockwell Collins
- Practical Application of Formal Methods To Modern Avionics Systems

Rockwell Collins

Advanced Technology Center Slide 6

What Are Model Checkers?

ADVANCED COMPUTING SYSTEMS

- Breakthrough Technology of the 1990's
- Widely Used in Hardware Verification (Intel, Motorola, IBM, …)
- Several Different Types of Model Checkers
  - Explicit, Symbolic, Bounded, Infinite Bounded, …
- Exhaustive Search of the Global State Space
  - Consider All Combinations of Inputs and States
  - Equivalent to Exhaustive Testing of the Model
  - Produces a Counter Example if a Property is Not True
- Easy to Use
  - "Push Button" Formal Methods
  - Very Little Human Effort Unless You're at the Tool's Limits
- Limitations
  - State Space Explosion (10^{100} – 10^{300} States)

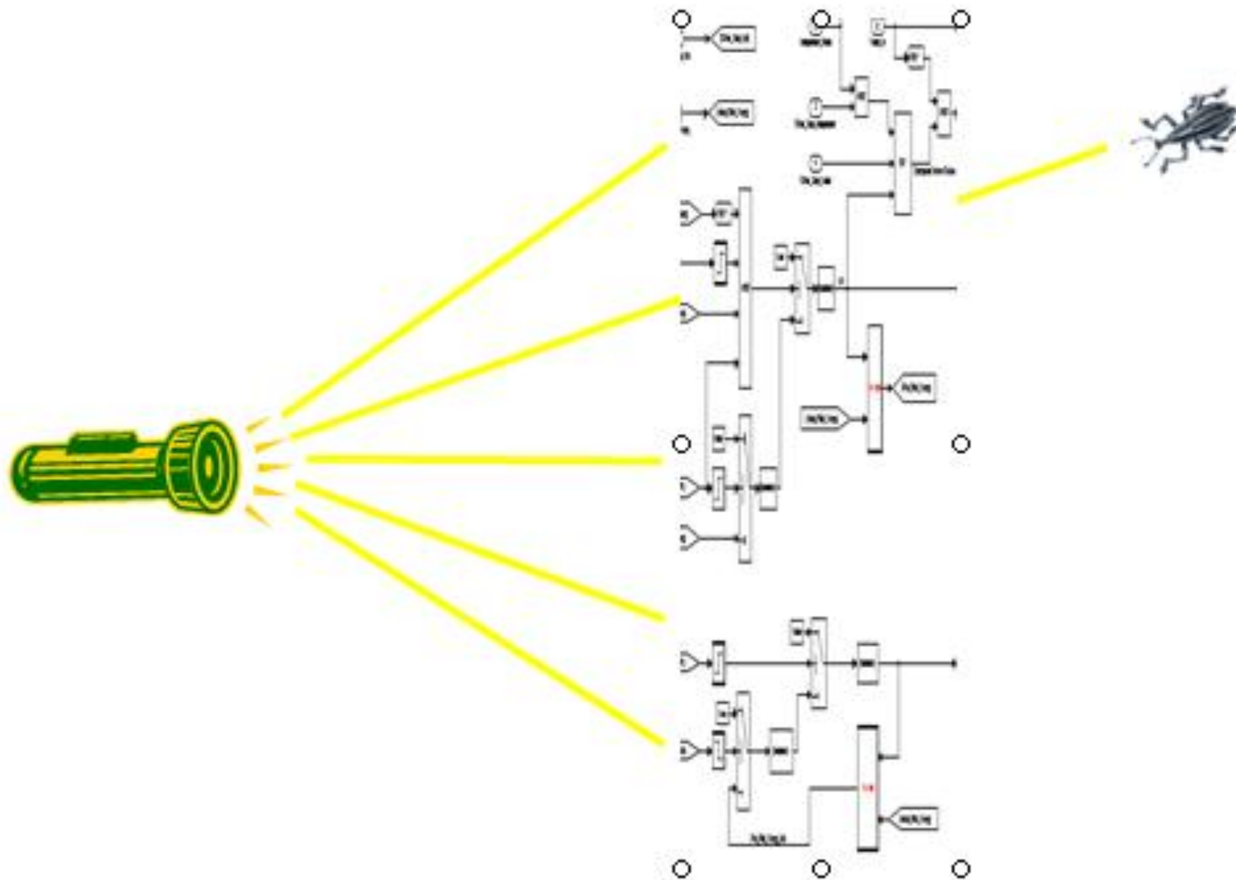
Rockwell Collins

Advantage of Model Checking

ADVANCED COMPUTING SYSTEMS

Testing Checks Only the Values We Select

Even Small Systems Have Trillions (of Trillions) of Possible Tests!



Rockwell Collins

Advantage of Model Checking

ADVANCED COMPUTING SYSTEMS

Model Checker Tries Every Possible Input and State!

Rockwell

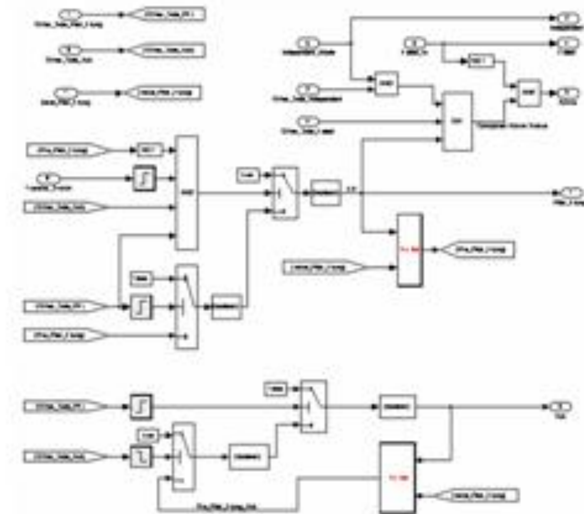
Example - ADGS-2100 Adaptive Display & Guidance System

ADVANCED COMPUTING SYSTEMS

883 Subsystems

9,772 Simulink Blocks

2.9×10^{52} Reachable States



Requirement

Drive the Maximum Number of Display Units Given the Available Graphics Processors

Counterexample Found in 5 Seconds!

Checking 373 Properties
Found Over 60 Errors

Rockwell Collins

Advanced Technology Center — Slide 12

Summary of Errors Found

ADVANCED COMPUTING SYSTEMS

Dectected By	Likelihood of Being Found by Traditional Methods				
	Trivial	Likely	Possible	Unlikely	Total
Inspection			1	2	3
Modeling		5	1		6
Simulation					
Model Checking	2	1	13	1	17
Total	2	6	15	3	26

- **Model-Checking Detected the Majority of Errors**
- **Model-Checking Detected the Most Serious Errors**
- **Found *Early in the Lifecycle* during Requirements Analysis**

Rockwell Collins

Conclusions

ADVANCED COMPUTING SYSTEMS

- Model-Based Development *is* the Industrial Use Formal Specification
- Convergence of Model-Based Development and Formal Verification
  - Engineers are Producing Specifications that Can be Analyzed
  - Formal Verification Tools are Getting More Powerful
- Model Checking is Very Cost Effective
  - Simple and Easy to Use
  - Finds All Exceptions to a Property
  - Used to Find Errors Early in the Lifecycle
- Applied to Models with Only Boolean and Enumerated Types

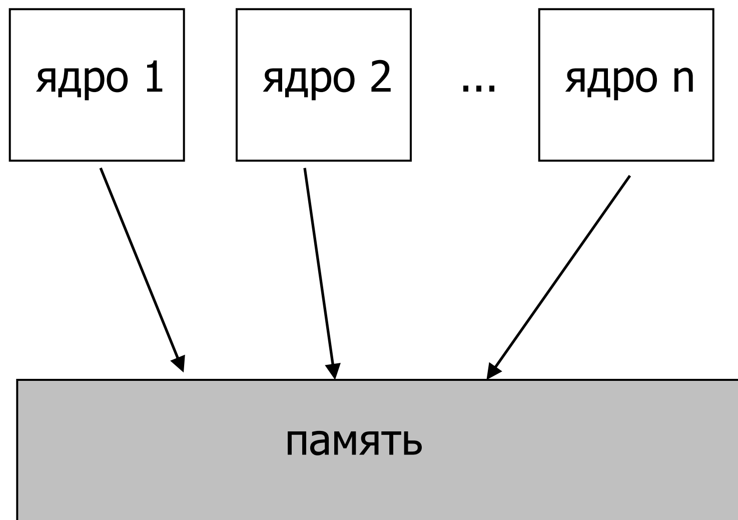
Rockwell Collins

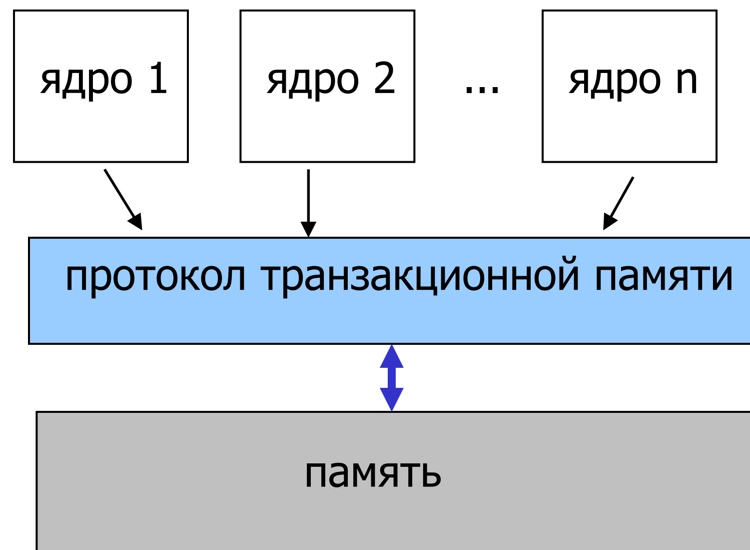
Advanced Technology Center Slide 24

Success Story. Транзакционная память в многоядерных процессорах

- Многоядерные процессоры вывели параллельное программирование и связанные с ним проблемы ошибок в параллельных программах “в народные массы”
- Как решать проблемы ошибок синхронизации параллельных программ в многоядерных процессорах?

Транзакционная память – один из способов, позволяющих избежать проблем синхронизации процессов над общей памятью





“Оконный” алгоритм транзакционной памяти

D.Imbs, и Michel Raynal в 2009 г. предложил оконный алгоритм транзакционной памяти и доказательство его корректности

Michel Raynal has been a professor of computer science since 1981.
At IRISA (CNRS-INRIA-University joint computing research laboratory located in Rennes), he founded a research group on Distributed Algorithms in 1983.

- His research interests include distributed algorithms, distributed computing systems and dependability. His main interest lies in the fundamental principles that underlie the design and the construction of distributed computing systems. He has been Principal Investigator of a number of research grants in these areas
- Professor Michel Raynal has published more than 95 papers in journals (JACM, Acta Informatica, Distributed Computing, etc.); and more than 195 papers in conferences (ACM STOC, ACM PODC, ACM SPAA, IEEE ICDCS, IEEE DSN, DISC, IEEE IPDPS, etc.). He has also written seven books devoted to parallelism, distributed algorithms and systems (MIT Press and Wiley).

Success Story. Ошибка в STM протоколе

- Imbs D.; Raynal M. *Software Transactional Memories: an Approach for Multicore Programming*. LNCS, 5698, 2009
- Эта работа была дана студенту 4 курса Алексею Беляеву для проверки протокола в рамках его НИРС
- Студент выразил требования к протоколу формулой LTL, воспользовался системой верификации SPIN и нашел в протоколе ошибку

michel raynal transaction memory

Результатов: примерно 7 (0,44 сек.)

Software **transactional** memories: an approach for multicore programming

- [Перевести эту страницу]

Damien Imbs · **Michel Raynal**. © Springer Science+Business Media, LLC 2010 ...... thank Professor Yuri **Karpov** and his student Alexey **Belyaev** who found a bug in a ... Imbs D, Raynal M (2009) A versatile STM protocol with invisible read ... Larus J, Kozyrakis Ch (2008) **Transactional memory**: is TM the answer for ...

www.springerlink.com/index/hr4655364l588355.pdf

ВЕРИФИКАЦИЯ АЛГОРИТМА ПОДДЕРЖКИ ТРАНЗАКЦИОННОЙ ПАМЯТИ

А.Б. Беляев
Санкт-Петербургский государственный политехнический университет
E-mail: belyaevab@gmail.com

Аннотация

Широкое внедрение многоядерных процессоров в практику программирования существенно повысило важность проблемы разработки параллельных программ. Одним из подходов, позволяющих существенно упростить параллельное программирование, является технология транзакционной памяти.

Доклад содержит результаты анализа одного из алгоритмов реализации транзакционной памяти для многоядерных процессоров. Этот алгоритм был предложен в работе, представленной на десятой Международной конференции PaCT'09. Работа также содержала аналитическое доказательство корректности алгоритма, в котором в дальнейшем были обнаружены пробелы. В настоящем докладе приводятся результаты автоматической верификации этого алгоритма в системе Spin, которая показала, что алгоритм не удовлетворяет критерию корректности транзакционной памяти. Найденная ошибка очень тонкая: в некоторых специальных случаях транзакция способна выполнить операции с несогласованными данными, что может привести к сбою системы. Алгоритм был исправлен. Верификация исправленной версии алгоритма ошибок не обнаружила.

Введение

Конфуз с голосованием на “Эхе Москвы” 25.09.2011

http://www.echo.msk.ru/programs/interception/814452-echo/

Параллельная композиция процессов:

За Путина

$x_П$:=0;
while !Stop do
  wait call_1;
  $x_П$ ++
od

За Медведева

$x_М$:=0;
while !Stop do
  wait call_2;
  $x_М$ ++
od

Σ

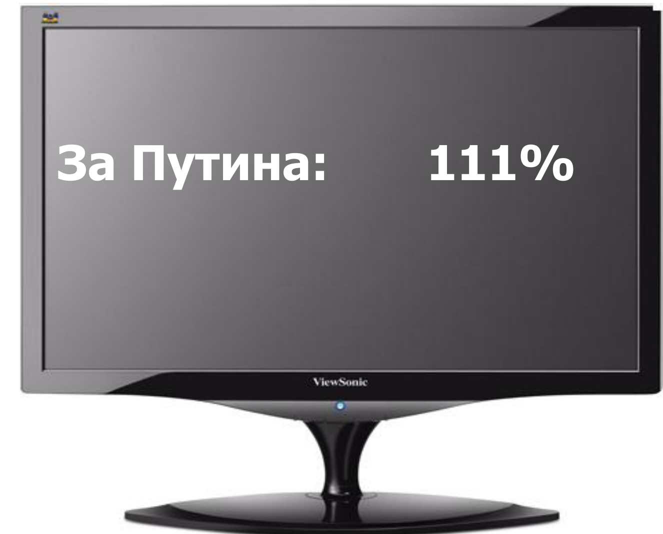
while !Stop do
  $x_\Sigma := x_П + x_М$
od

$\%_П$

while !Stop do
  $P_П := x_П / x_\Sigma$
od

$\%_М$

while !Stop do
  $P_М := x_М / x_\Sigma$
od



- А.В.Венедиктов:

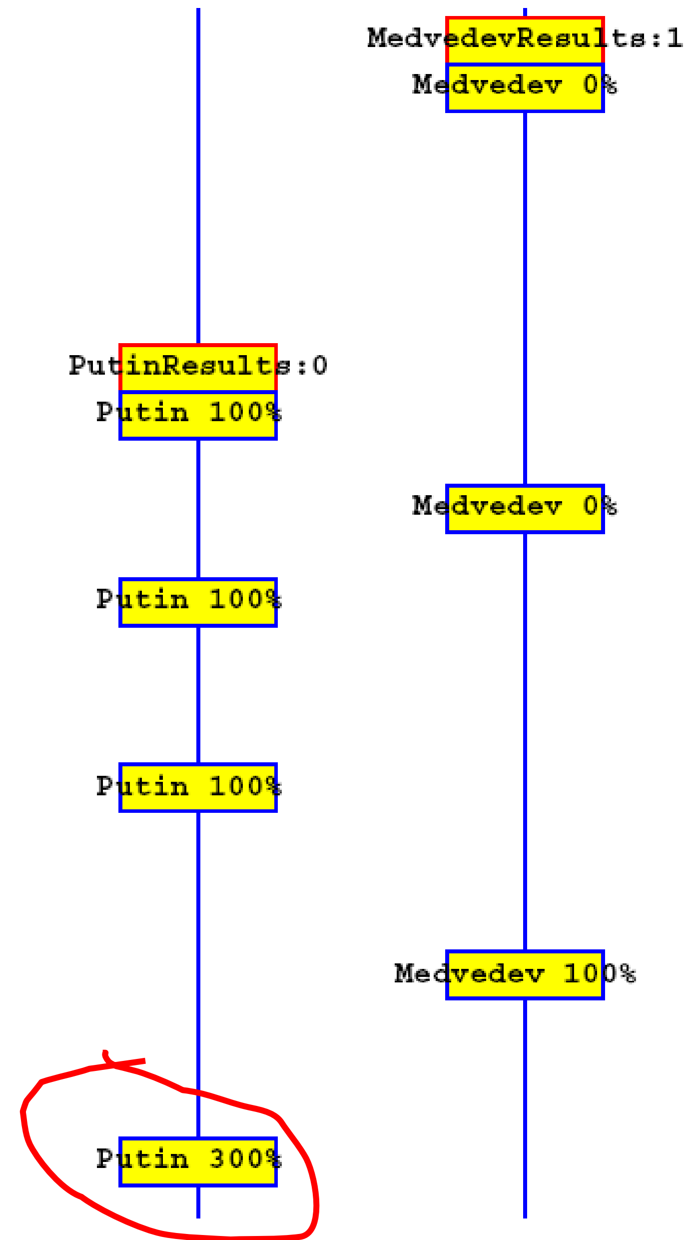
  “*Подвисает система голосования...*
  У меня возникло 111% ...”
  (Эхо Москвы, 24.09.2011)

Пример верификации

- Верификация проблемы Путин-Медведев на системе Spin обнаруживает ошибку

Проверка свойства:

"*За Путина проголосуют 300%*"



Выводы

- В ведущих фирмах методы верификации аппаратуры и программ на основе *Model checking* разработаны до индустриальной технологии
  - Даже опытные профессионалы могут допустить ошибки в разрабатываемых ими параллельных программах
  - Нельзя полагаться на аналитические доказательства корректности программ
  - Даже студент, используя технику Model checking, может найти ошибки в достаточно сложной параллельной программе, разработанной профессионалами
- Изучение веификации методом Model checking даёт студенту
  теорию, алгоритмы,
  элементы технологии,
  инструментарий
  с помощью которых он может проверять корректность нетривиальных параллельных и распределенных программ

Обучение формальным методам

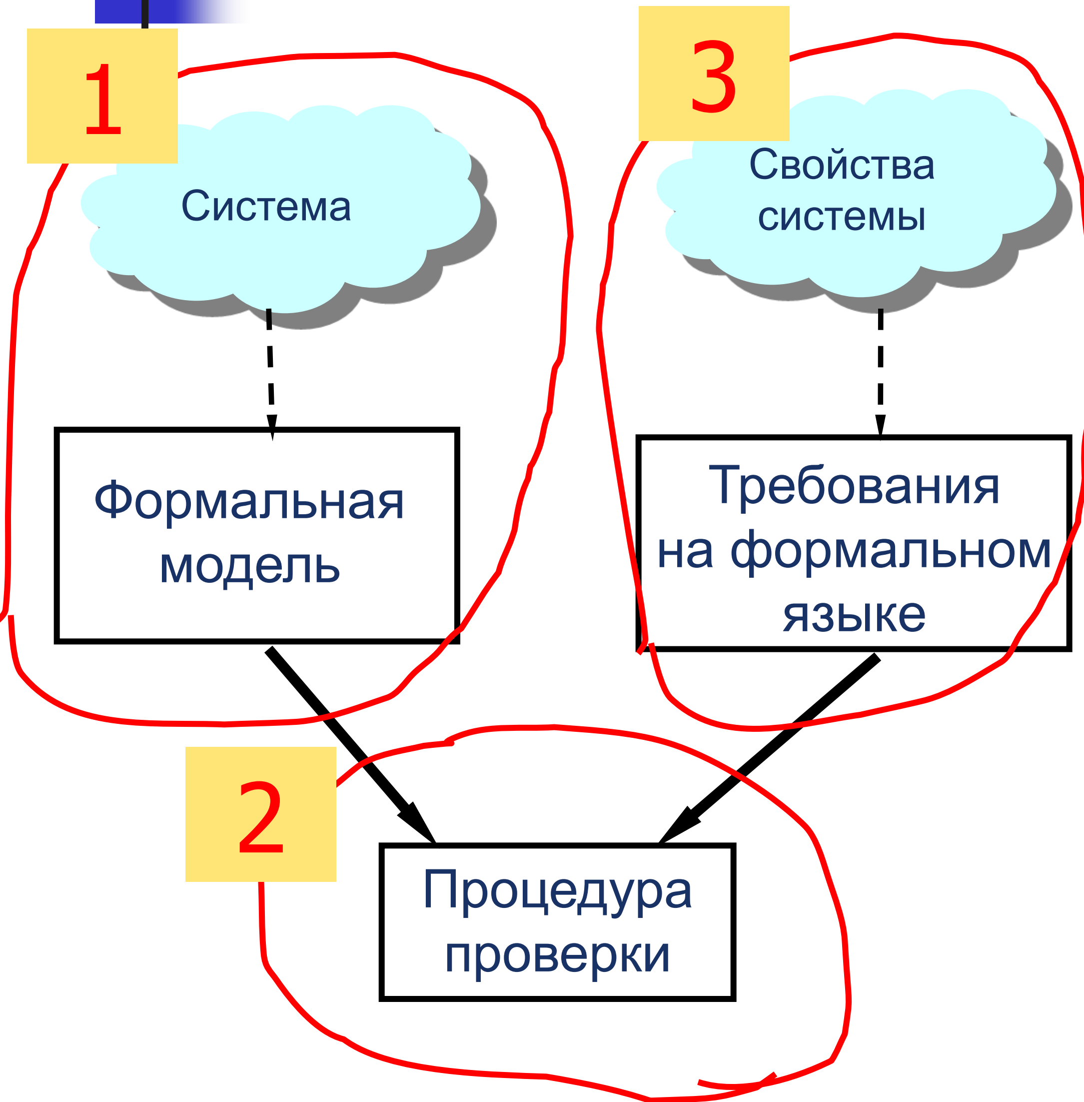
“Формальные методы должны стать частью образования каждого исследователя в области информатики и каждого разработчика ПО так же, как другие ветви прикладной математики являются необходимой частью образования в других инженерных областях.

NASA Report 4673, “*Formal Methods and Their Role in Digital System Validation for Airborne Systems*”

Спасибо за внимание

Проблемы верификации

Проблемы при верификации программных систем

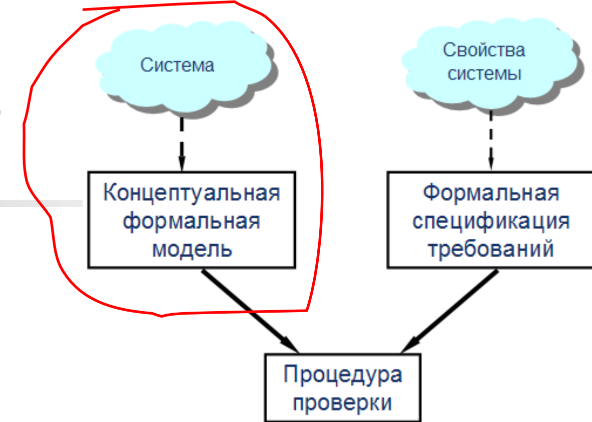


1. Модель из кода или код из модели?

2. “State explosion problem”?

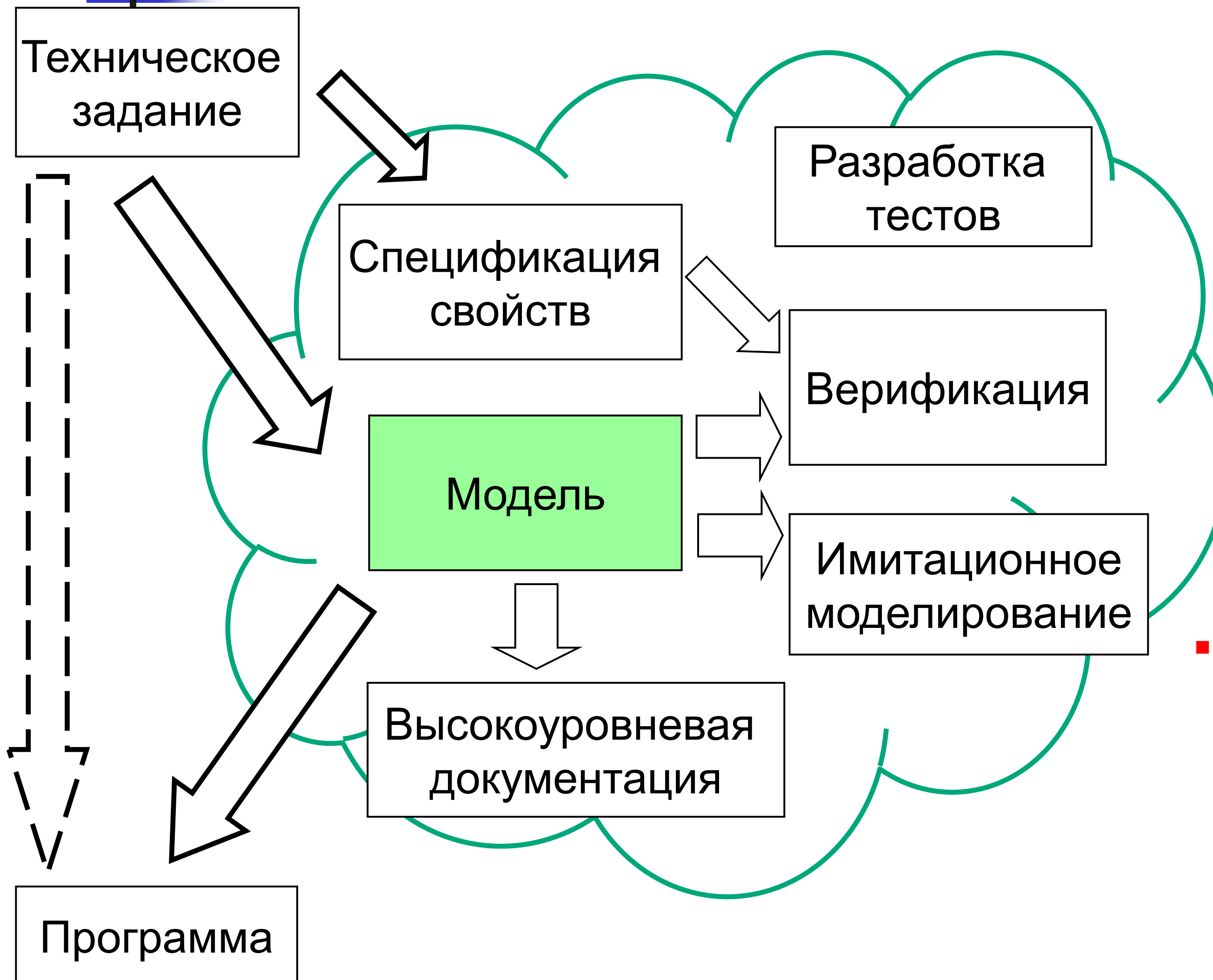
3. Какие требования проверять?

Модель из кода или код из модели?



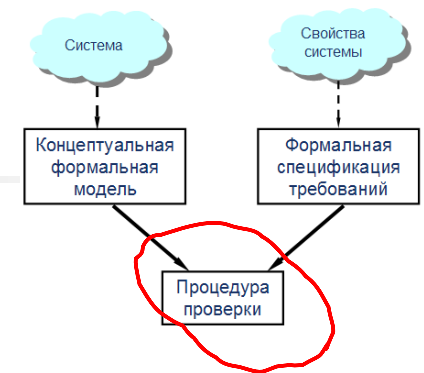
- Абсурдно ставить проблему доказательства программы после того, как программа написана (Н.Н.Непейвода)
- Вместо того, чтобы программы сначала разрабатывать, а ПОТОМ пытаться доказать, что эта они правильны, программы нужно сразу строить корректными (Э.Дейкстра)
- Следует сначала создать "Управляющий скелет", на который впоследствии можно безопасно навесить функциональную обработку (А.Хоар)
- Ясно специфицируйте "Центральную нервную систему", представляющую состояния, переходы, события, условия и время (Д.Харел)
- "Аллен [Эмерсон] и я заметили, что многие параллельные программы имеют свойство, которое мы назвали "синхронизационным скелетом с конечным числом состояний"" (Эдмунд Кларк, CACM 2008, v51)

Проектирование ПО на основе моделей (Model Driven Development)



- Преимущества:
  - модель и есть проект программной системы
  - согласованные тестирование, верификация, кодирование, симуляция
  - анализ на ранней стадии разработки
  - автоматическая генерация кода
  - единый документ, представляющий ясно и полно всю структуру системы
  - сопровождение, поддержка
- Проблемы:
  - адекватность
  - проблема расширения модели не только на reactive systems
  - неэффективность кодогенерации

“State explosion problem”



- Алгоритмы Model checking весьма эффективны: сложность проверки выполнимости формулы CTL Ф на структуре Крипке K пропорциональна числу подформул Ф и сложности K
- При явном представлении структуры Крипке современные скорости процессоров и объемы памяти позволяют верифицировать системы с числом состояний ~10^6 – (скорость обработки – сотни строк/с)

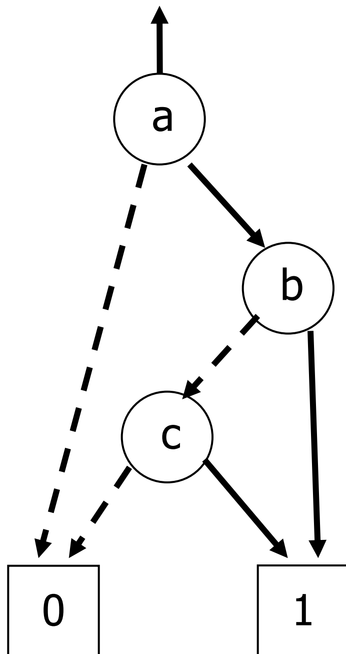
Но этого совершенно недостаточно!

Методы борьбы с проблемой “State explosion”:

1. Символьная верификация
2. Редукция частичных порядков
3. Композициональная верификация

Символьная верификация – использование бинарных решающих диаграмм (Binary Decision Diagrams)

- Эффективная форма представления булевых функций



BDD – это граф принятия решений:
- отсутствуют изоморфные подграфы и избыточные вершины
- фиксированный порядок проверки переменных
- для встречающихся на практике функций рост линеен

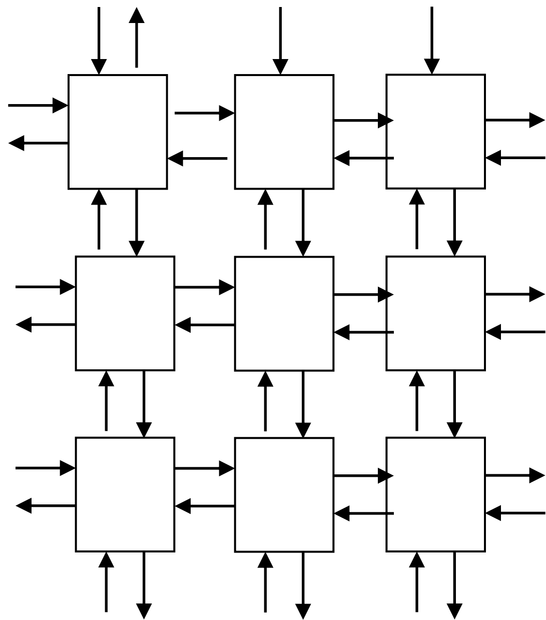
Основные свойства:
- каноническая форма
- сложность зависит от порядка переменных
- эффективное выполнение булевых операций

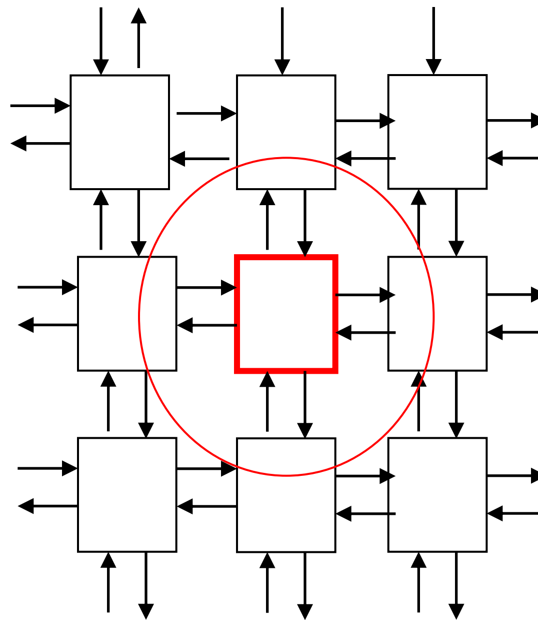
Применения:
- везде, где можно использовать булевы функции
- революция в работе с конечными структурами данных

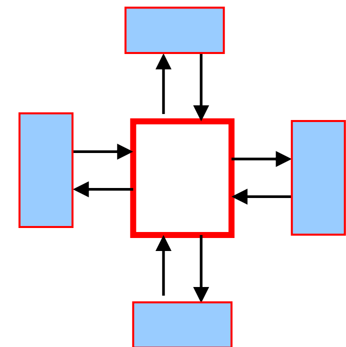
Использование BDD в алгоритмах верификации позволило увеличить сложность верифицируемых систем (число состояний структуры Крипке) в миллиарды миллиардов раз – с 10^6 до 10^{300}

Композициональность: вывод о глобальном поведении, полагаясь только на локальные свойства

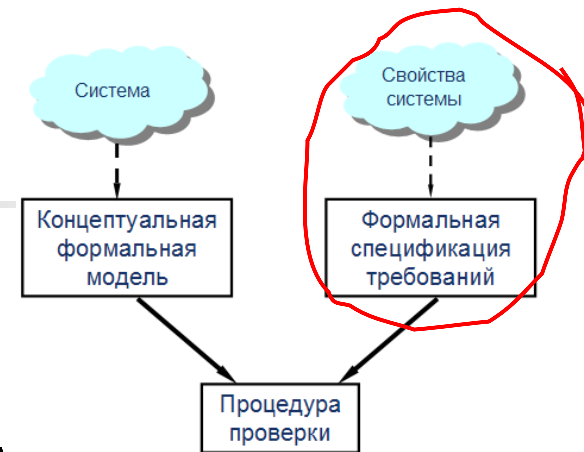
- Модулярная (компонентная) разработка ПО – основа построения сложных систем (Component Based Software Engineering)







Какие требования проверять?



Понятие "полное" множество требований формально определить нельзя

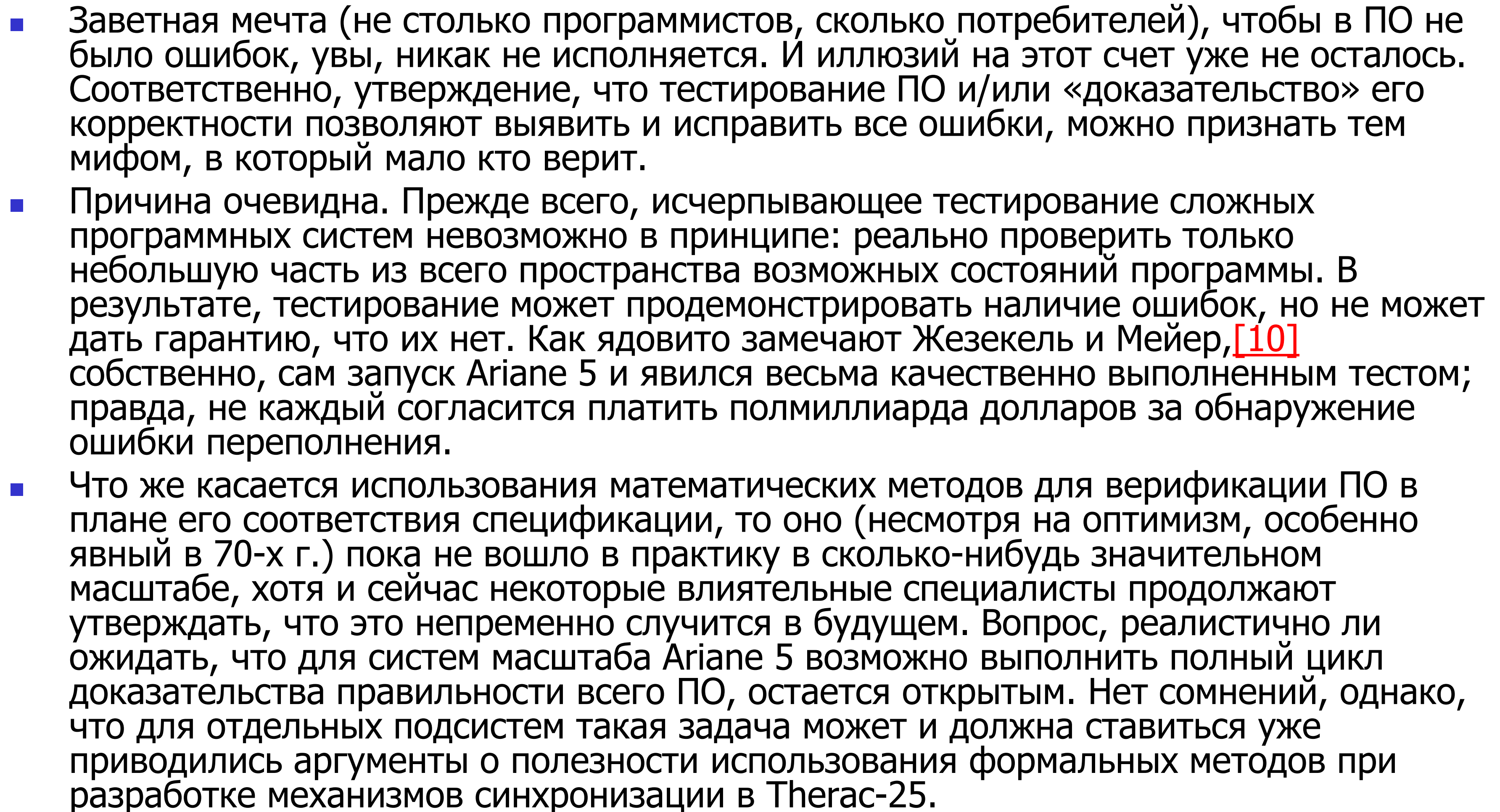
- Три закона робототехники для роботов. Айзек Азимов
  - Первый Закон: *Робот не может причинить вред человеку или своим бездействием допустить, чтобы человеку был причинен вред*
  - Второй Закон: *Робот должен повиноваться командам человека, если эти команды не противоречат Первому Закону*
  - Третий Закон: *Робот должен заботиться о своей безопасности, если это не противоречит Первому и Второму законам*
- Что такое *вред человеку*? Смерть- вред? Эвтаназия (легкая смерть)?
  - - в Нидерландах, Бельгии, Швейцарии, ... – законодательно разрешена
  - - в России – преступление, квалифицируется как умышленное убийство
  - Выполнит ли робот эвтаназию по просьбе человека?
  - В Цюрихской клинике Dignitas здоровой гражданке Канады была проведена эвтаназия. Свою просьбу она мотивировала тем, что хочет уйти вместе со смертельно больным мужем (СМИ, окт 2011)

Подходы в формальной спецификации требований

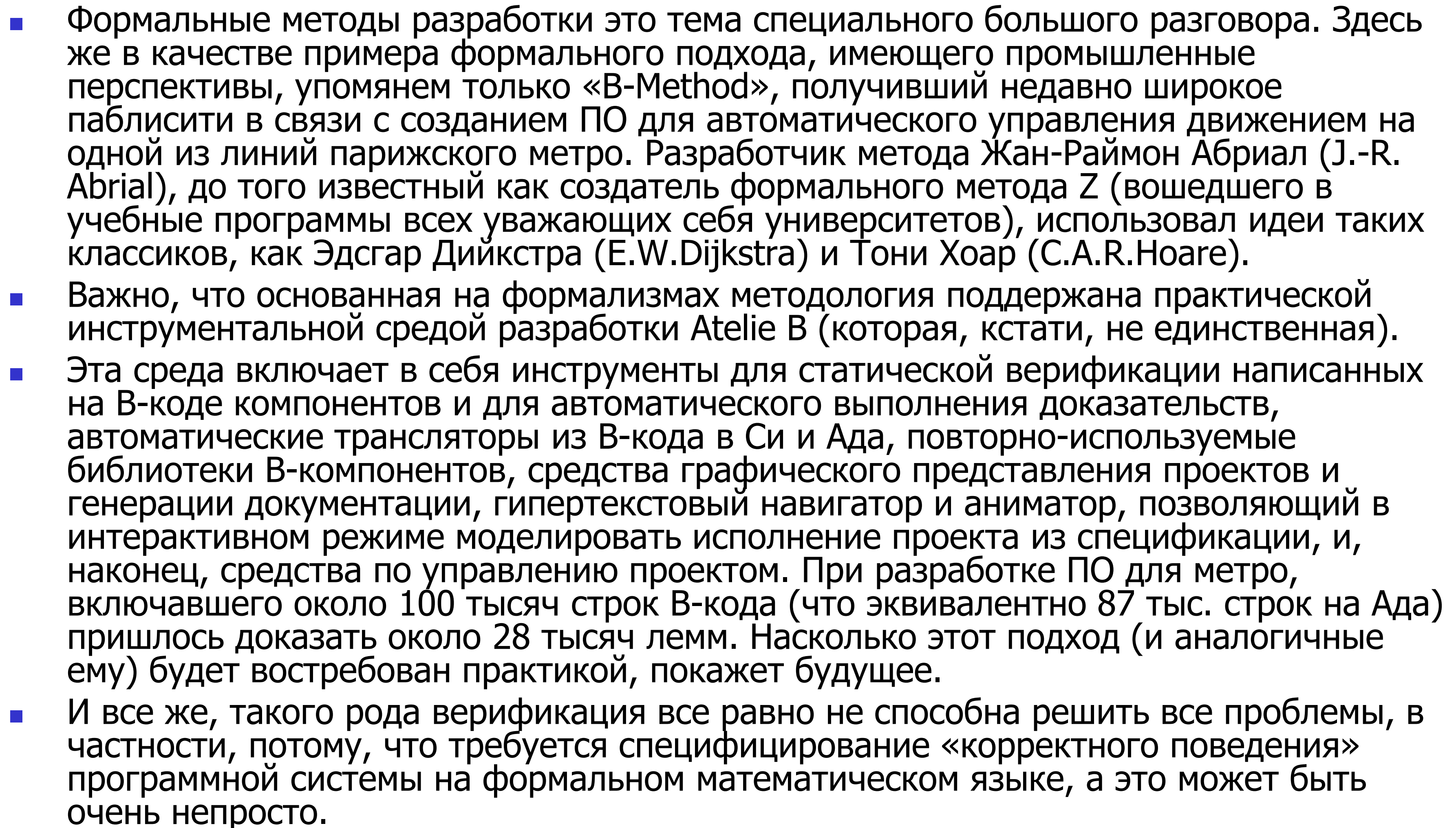
- Обычный метод спецификации требований – из технического задания, написанного на естественном языке
  - Z-спецификация
  - B-спецификация

Заключение

- Вопрос о том, можно ли выполнить полное доказательство правильности реального ПО, наивен. Неформальное понятие правильности формально определить нельзя
- Вопрос о том, можно ли ожидать, что для бортовых систем масштаба AirBus A330 (2M строк) или Boeing 770 (4 M строк) можно выполнить формальное доказательство выполнимости широкого спектра важных критических свойств всего ПО, построенных из спецификации требований, остается открытым
- Однако для отдельных подсистем эта задача разрешима
- Формальные модели, на которых должны основываться спецификация, разработка и верификация ПО, уже входят в стандарты
- DO-178C XXXXXXXXXX



- Заветная мечта (не столько программистов, сколько потребителей), чтобы в ПО не было ошибок, увы, никак не исполняется. И иллюзий на этот счет уже не осталось. Соответственно, утверждение, что тестирование ПО и/или «доказательство» его корректности позволяют выявить и исправить все ошибки, можно признать тем мифом, в который мало кто верит.
- Причина очевидна. Прежде всего, исчерпывающее тестирование сложных программных систем невозможно в принципе: реально проверить только небольшую часть из всего пространства возможных состояний программы. В результате, тестирование может продемонстрировать наличие ошибок, но не может дать гарантию, что их нет. Как ядовито замечают Жезекель и Мейер,[10] собственно, сам запуск Ariane 5 и явился весьма качественно выполненным тестом; правда, не каждый согласится платить полмиллиарда долларов за обнаружение ошибки переполнения.
- Что же касается использования математических методов для верификации ПО в плане его соответствия спецификации, то оно (несмотря на оптимизм, особенно явный в 70-х г.) пока не вошло в практику в сколько-нибудь значительном масштабе, хотя и сейчас некоторые влиятельные специалисты продолжают утверждать, что это непременно случится в будущем. Вопрос, реалистично ли ожидать, что для систем масштаба Ariane 5 возможно выполнить полный цикл доказательства правильности всего ПО, остается открытым. Нет сомнений, однако, что для отдельных подсистем такая задача может и должна ставиться уже приводились аргументы о полезности использования формальных методов при разработке механизмов синхронизации в Therac-25.


Формальные методы разработки это тема специального большого разговора. Здесь же в качестве примера формального подхода, имеющего промышленные перспективы, упомянем только «B-Method», получивший недавно широкое паблисити в связи с созданием ПО для автоматического управления движением на одной из линий парижского метро. Разработчик метода Жан-Раймон Абриал (J.-R. Abrial), до того известный как создатель формального метода Z (вошедшего в учебные программы всех уважающих себя университетов), использовал идеи таких классиков, как Эдсгар Дийкстра (E.W.Dijkstra) и Тони Хоар (C.A.R.Hoare).
Важно, что основанная на формализмах методология поддержана практической инструментальной средой разработки Atelie B (которая, кстати, не единственная).
Эта среда включает в себя инструменты для статической верификации написанных на B-коде компонентов и для автоматического выполнения доказательств, автоматические трансляторы из B-кода в Си и Ада, повторно-используемые библиотеки B-компонентов, средства графического представления проектов и генерации документации, гипертекстовый навигатор и аниматор, позволяющий в интерактивном режиме моделировать исполнение проекта из спецификации, и, наконец, средства по управлению проектом. При разработке ПО для метро, включавшего около 100 тысяч строк B-кода (что эквивалентно 87 тыс. строк на Ада) пришлось доказать около 28 тысяч лемм. Насколько этот подход (и аналогичные ему) будет востребован практикой, покажет будущее.
И все же, такого рода верификация все равно не способна решить все проблемы, в частности, потому, что требуется специфицирование «корректного поведения» программной системы на формальном математическом языке, а это может быть очень непросто.

Новый стандарт сертификации бортового ПО DO-178C

- The Radio Technical Commission for Aeronautics (RTCA) в конце 2010 разработала стандарт DO-178C вместо DO-178B, разработанного в 1992
- Новый стандарт embraces contemporary technologies and methodologies necessary to achieve these aims и в трех основных моментах отличается от DO-178B
- hopes to address software development challenges through DO-178C – a new standard that
- makes little or no allowance for current development and verification technologies and methods, including formal methods, Model-Based Development (MBD), and the use of object-oriented technologies
- The DO-178C standard, due to be finalized in late 2010 by a joint RTCA/EUROCAE committee, will address this shortfall and assist in bringing the certification of avionics software in line with these 21st-century technologies
- DO-178B assumes a traditional software development life cycle that progresses linearly from requirements through design and code to integration and test. Its process model resembles a waterfall or V model in which validated requirements are a given (DO-178B does not mention requirements validation), and there is a *de facto* partitioning of the requirements engineering and software development processes
- Verification in DO-178B comes primarily from top-down testing

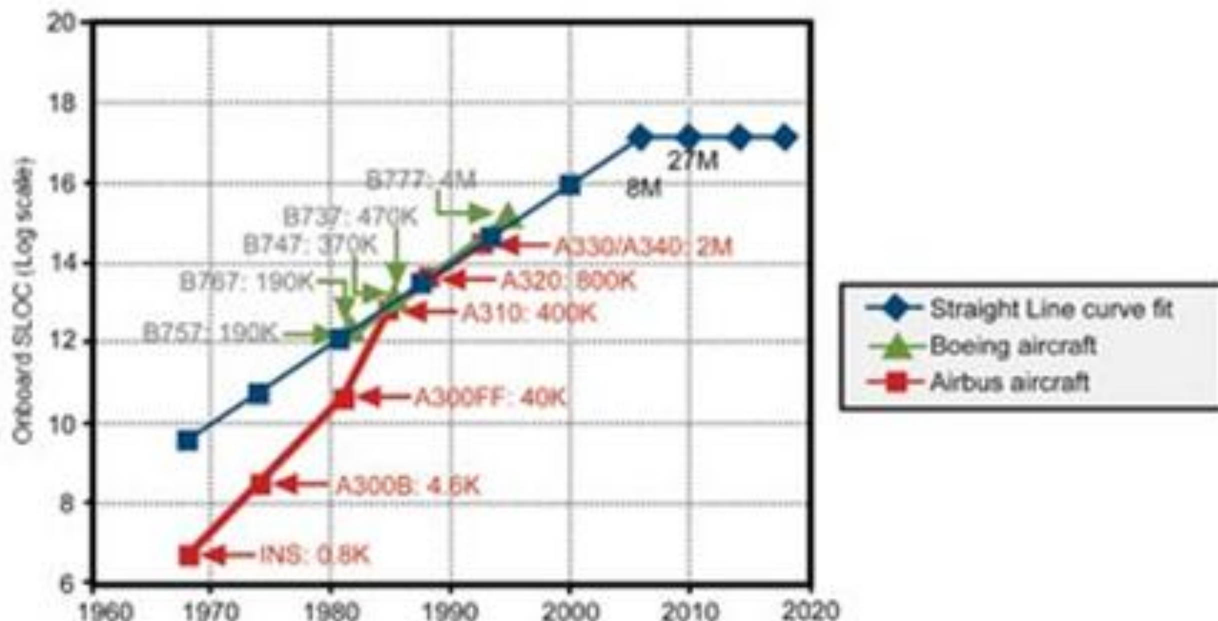
Рост сложности бортового встраиваемого ПО

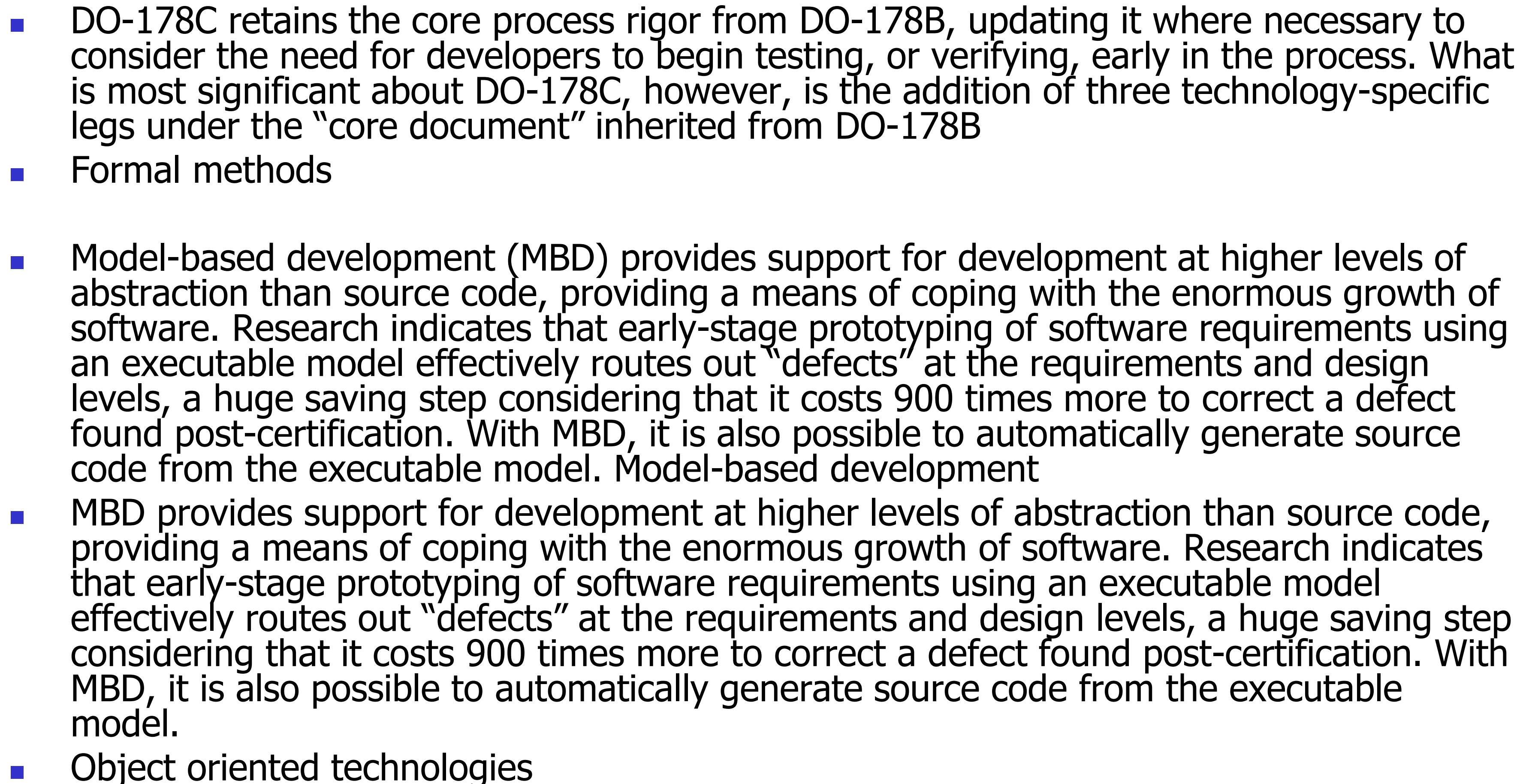
- Boeing и Airbus демонстрируют геометрический рост размеров и сложности бортового ПО

Единственная возможность производителям ПО для критических применений разрабатывать системы вовремя и с близким к 0 уровенем дефектов – использовать новые технологии и методологии – то, что DO-178B не позволял

Эти технологии включают:

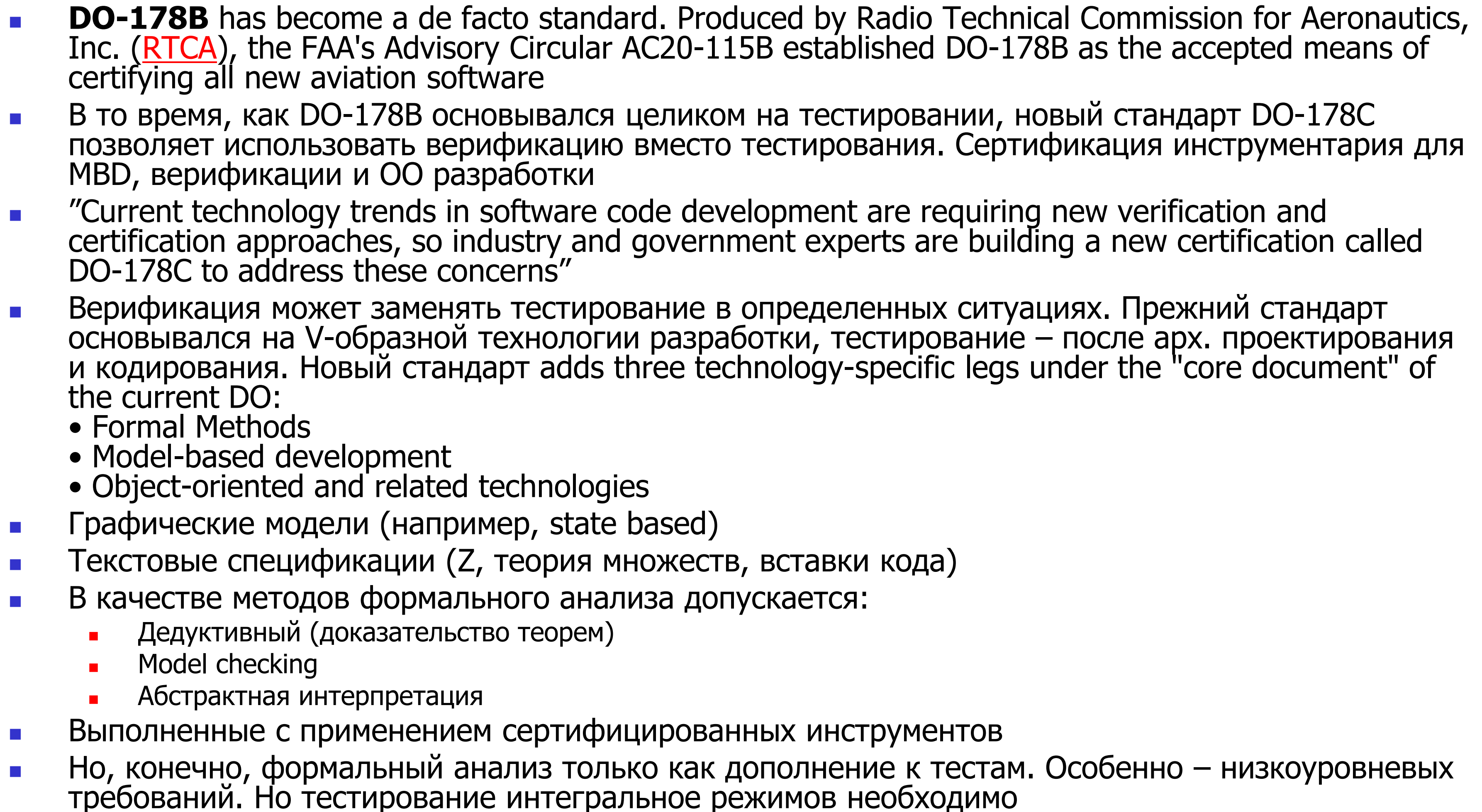
- Model-Based Development
- OOT
- Formal Methods and Tools





- DO-178C retains the core process rigor from DO-178B, updating it where necessary to consider the need for developers to begin testing, or verifying, early in the process. What is most significant about DO-178C, however, is the addition of three technology-specific legs under the “core document” inherited from DO-178B
- Formal methods

- Model-based development (MBD) provides support for development at higher levels of abstraction than source code, providing a means of coping with the enormous growth of software. Research indicates that early-stage prototyping of software requirements using an executable model effectively routes out “defects” at the requirements and design levels, a huge saving step considering that it costs 900 times more to correct a defect found post-certification. With MBD, it is also possible to automatically generate source code from the executable model. Model-based development
- MBD provides support for development at higher levels of abstraction than source code, providing a means of coping with the enormous growth of software. Research indicates that early-stage prototyping of software requirements using an executable model effectively routes out “defects” at the requirements and design levels, a huge saving step considering that it costs 900 times more to correct a defect found post-certification. With MBD, it is also possible to automatically generate source code from the executable model.
- Object oriented technologies



- **DO-178B** has become a de facto standard. Produced by Radio Technical Commission for Aeronautics, Inc. (RTCA), the FAA's Advisory Circular AC20-115B established DO-178B as the accepted means of certifying all new aviation software
- В то время, как DO-178B основывался целиком на тестировании, новый стандарт DO-178C позволяет использовать верификацию вместо тестирования. Сертификация инструментария для MBD, верификации и ОО разработки
- ”Current technology trends in software code development are requiring new verification and certification approaches, so industry and government experts are building a new certification called DO-178C to address these concerns”
- Верификация может заменять тестирование в определенных ситуациях. Прежний стандарт основывался на V-образной технологии разработки, тестирование – после арх. проектирования и кодирования. Новый стандарт adds three technology-specific legs under the "core document" of the current DO:
  • Formal Methods
  • Model-based development
  • Object-oriented and related technologies
- Графические модели (например, state based)
- Текстовые спецификации (Z, теория множеств, вставки кода)
- В качестве методов формального анализа допускается:
  - Дедуктивный (доказательство теорем)
  - Model checking
  - Абстрактная интерпретация
- Выполненные с применением сертифицированных инструментов
- Но, конечно, формальный анализ только как дополнение к тестам. Особенно – низкоуровневых требований. Но тестирование интегральное режимов необходимо

О чем было рассказано

- Уязвимость людей от созданной ими техники. Что делать?
- Верификация: постановка задачи
  - общая постановка задачи
  - дедуктивный метод Isabelle
  - Model checking – что это
  - Теоретические основы Model checking
- Model checking: текущая ситуация
  - Использование верификации в ведущих фирмах: Rockwell Collins
  - Success Story
- Проблемы с внедрением верификации
  - некорректная постановка проблемы верификации: MDD
  - "state explosion problem" и возможные решения
  - что проверять?
- Перспективы разработки безошибочного кода
  - Инициатива Verified Software Initiative
  - стандарт DO178C; сертификация

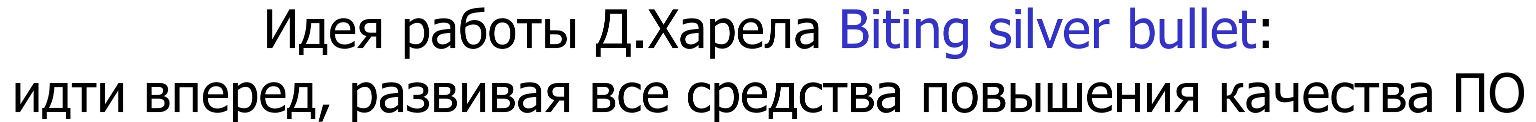
Biting the Silver Bullet

David Harel, IEEE Computer, 1992

Toward a Brighter Future for System Development

Необходимость делать нечто болезненное, но необходимое, несмотря на критику, не теряя мужества и оптимизма

- В 1986 г. Frederic Brooks писал (“No Silver Bullet”) об иллюзиях и надеждах в разработке ПО. Он писал, что, фактически, ни одна из предлагаемых идей не является “серебряной пулей”, которая спасет нас от ужасов, связанных с разработкой больших комплексов ПО. “Nonbullets”: ЯВУ, ООП, верификация, графические языки, ... Весьма пессимистический взгляд
- David Harel: хотя единственного средства (“серебряной пули”) нет, но интегральное использование новых идей, в частности, моделирования, визуального программирования и генерации кода может внести существенный вклад в разработку сложных reactive systems



Идея работы Д.Харела Biting silver bullet:
идти вперед, развивая все средства повышения качества ПО

- Среди наиболее обещающих направлений:
  - автоматическая верификация концептуальных моделей
  - методы генерации качественного кода из модели

Д. Харел (1992):

"*Текущая ситуация и перспективы значительных улучшений показывают, что мы находимся на пороге новой, удивительной эры*"

Современные успехи формальной верификации показывают, что
"БУДУЩЕЕ НАЧИНАЕТСЯ СЕГОДНЯ"

Спасибо за внимание