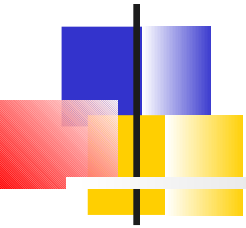
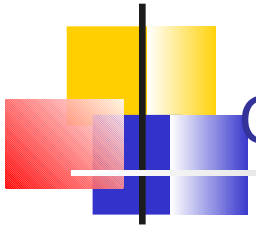


На пути к практическому применению верификации в разработке программных систем

Юрий Глебович Карпов

Ирина Владимировна Шошмина

Санкт-Петербургский
Политехнический университет



СМИ о программных ошибках – каждый день

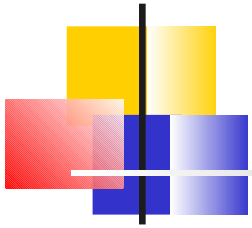
- Апрель, 2010. Авария в Мексиканском заливе: возможна ли программная ошибка? *Don Shafer, Phillip Laplante. The BP Oil Spill: Could Software be a Culprit? IEEE IT Pro September/October 2010, IEEE, 2010. ...* Не могла ли одной из причин бедствия стать ошибка в программном обеспечении?

- 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



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

Конфуз с голосованием на “Эхе Москвы” 24.09.2011

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_Σ :=x_П+x_М
od
```

%_П

```
while ! Stop
do
   P_П :=x_П / x_Σ
od
```

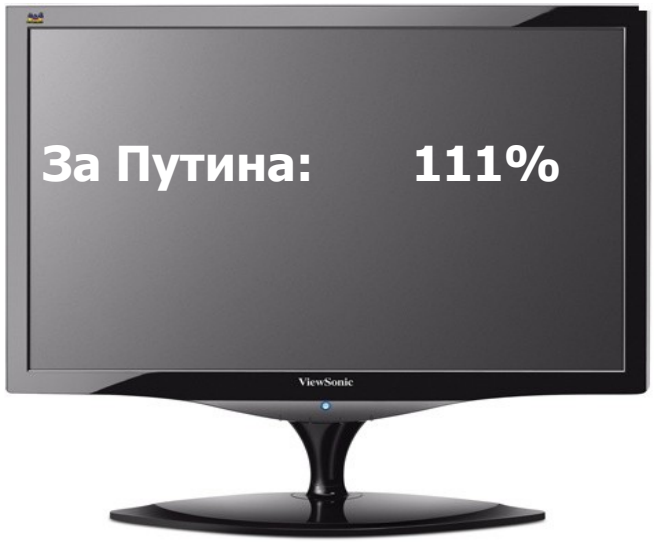
%_М

```
while ! Stop
do
   P_М :=x_М / x_Σ
od
```

- А.В.Венедиктов:

  “*Подвисает система голосования...*
  У меня возникло 111% ... .
  Нет, все, теперь выровнялось. ... ”

(Эхо Москвы, 24.09.2011)



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 по ликвидации ⇒ компания разоряется

ЧТО ДЕЛАТЬ?

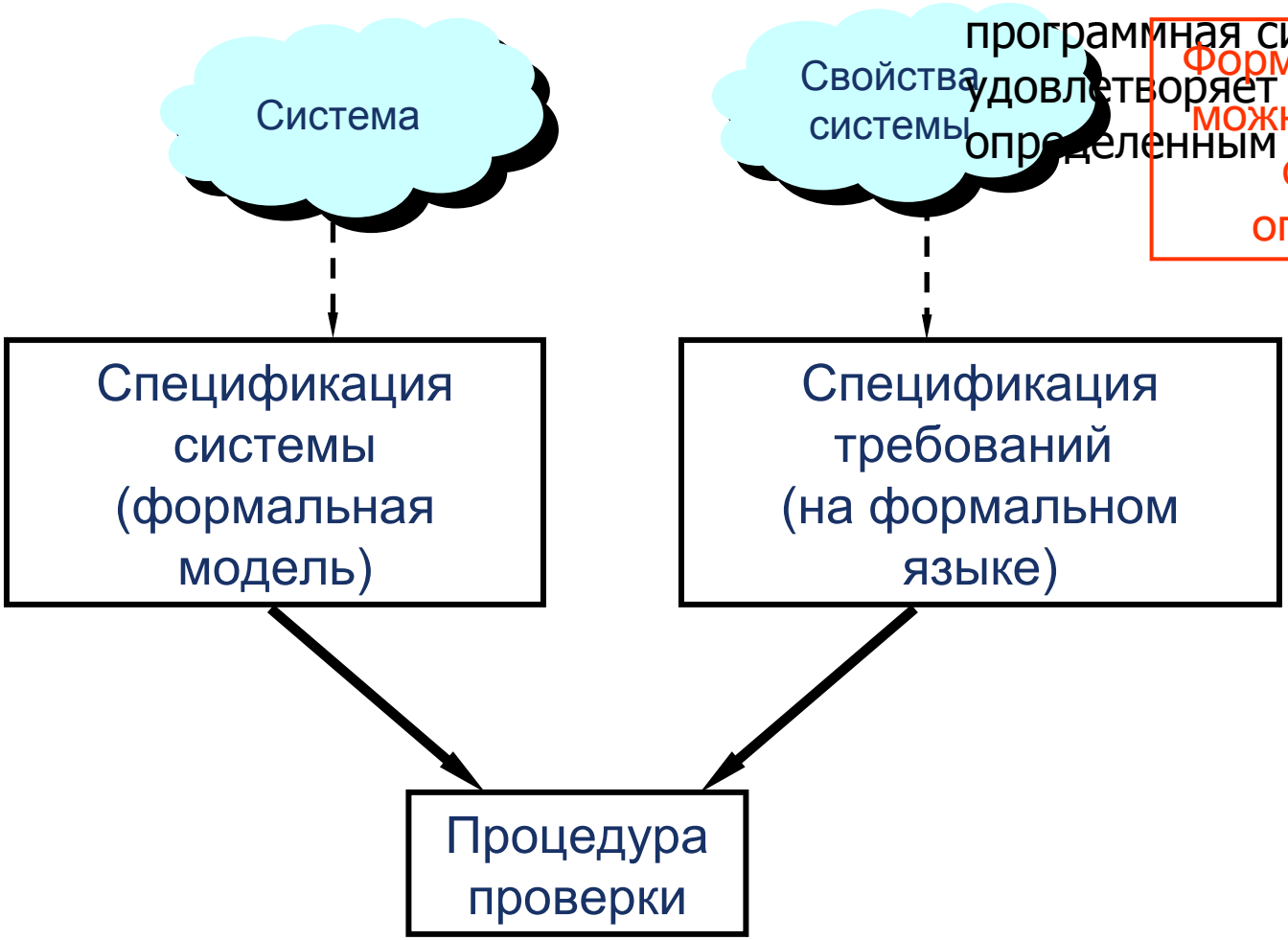
5 декабря 2010.
Старт 'Протона-М' со спутниками ГЛОНАСС:
~ $200 млн -> в океан

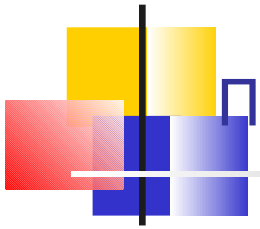
В последнее десятилетие сложность разрабатываемых компьютерных систем дошла до критического уровня: требуются все более сложные программы, а технология программирования не может их разрабатывать с требуемым качеством

Верификация – еще один метод повышения качества программ

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

Формально доказать можно только нечто формально определенное



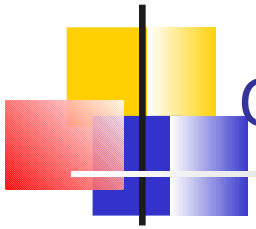


Пример дедуктивной верификации ядра реальной ОС

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 C 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
- Dr. Klein: "*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*"

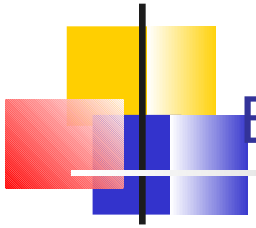


Оценка этого результата

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

Аллен Эмерсон о дедуктивной верификации: "*мы писали 15-страничный отчет о том, что программа на полстраницы корректна*"

Сложность доказательства в 30 раз выше сложности кода;
одна строка кода - ~ 1 страница доказательства



Вывод

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

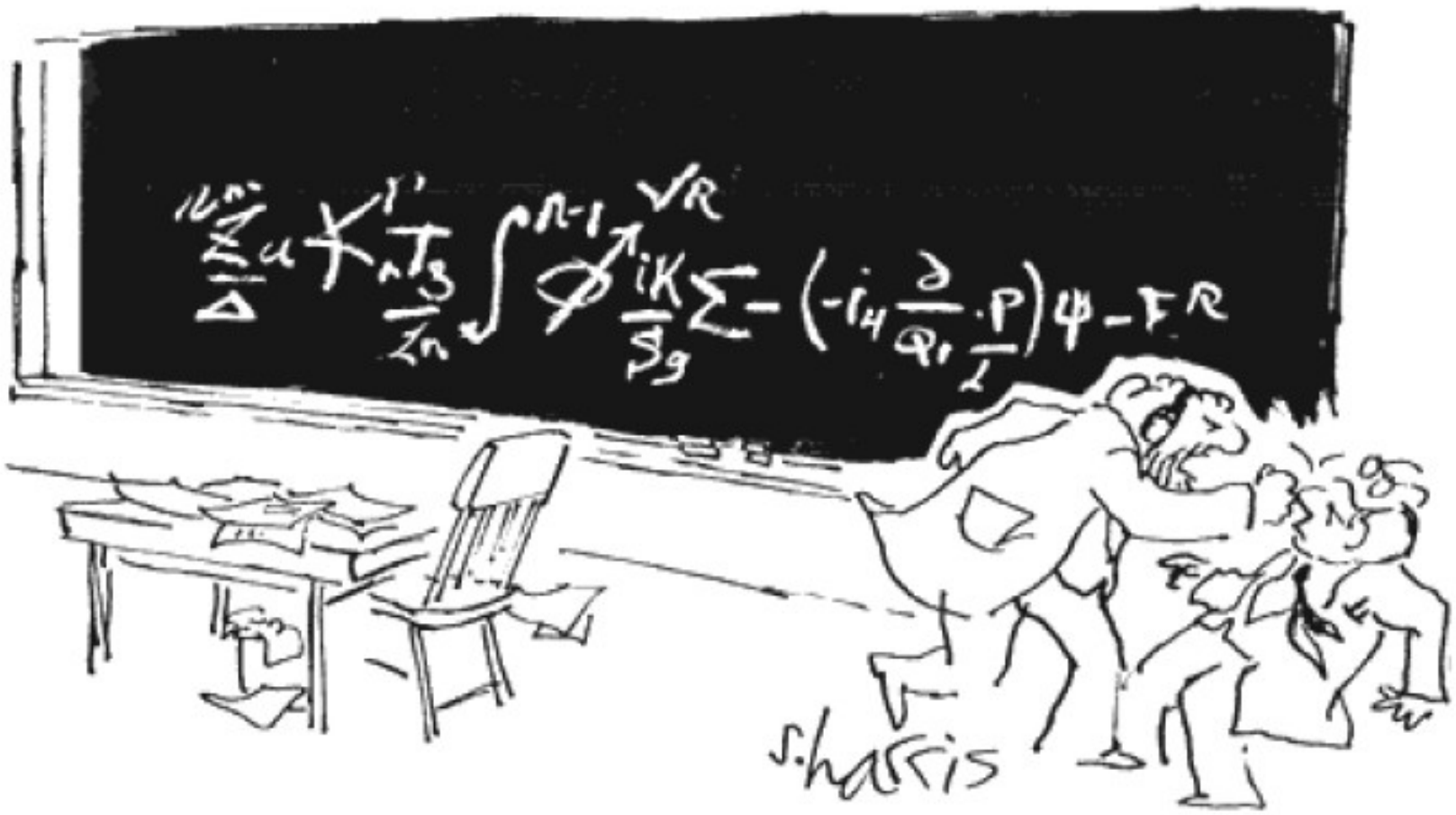
"A chasm: academics see formal methods as inevitable,

while practitioners see formal methods as irrelevant"
[Robert L. Glass, 1996]

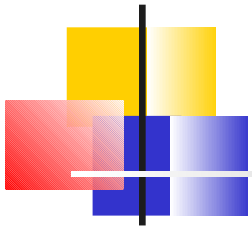
"Practitioner! Beware of formal methods zealots. If they can't make their case on the merit, they may well resort to chicanery"

" Практики! Опасайтесь фанатиков формальных методов ..."
[Robert L. Glass, CACM, 2004, (редактор Journal of Systems and Software)]

До сих пор практики программирования считают верификацию забавой теоретиков



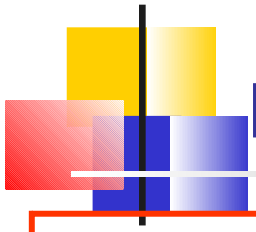
"You want proof? I'll give you proof!"



Model checking - прорыв в верификации

В последнее время – качественный прорыв в области верификации дискретных динамических систем

Разработан метод model checking, основанный на изящных формальных моделях



Премия Тьюринга ACM

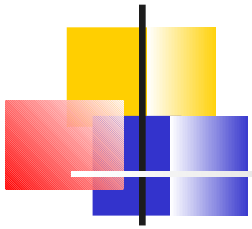
21 июня 2008 г премия Тьюринга была вручена трем создателям техники MODEL CHECKING, внесшим наиболее существенный вклад

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*"



Что такое Model checking?

Логика для формализации требований к ПО

- Классическая логика высказываний (propositional logic) неадекватна
  - Высказывания статичны, неизменны во времени
- Пример - некоммутативность конъюнкции, A&B ≠ B&A:

  "Джону стало страшно и он убил" $\Leftarrow ? \Rightarrow$ "Джон убил и ему стало страшно"

  "Смит умер и его похоронили" $\Leftarrow ? \Rightarrow$ "Смита похоронили и он умер"

  "Джейн вышла замуж и родила ребенка" $\Leftarrow ? \Rightarrow$ "Джейн родила и вышла замуж"
- В классической логике высказываний не формализуются:

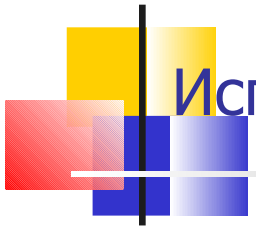
  Путин - президент России **(истинно только в какой-то период)**

  Мы не друзья, пока ты не извинишься

  Любой запрос к лифту с произвольного этажа, поступивший в любой момент времени, 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 1950-e)

Fq – q случится когда-нибудь в будущем

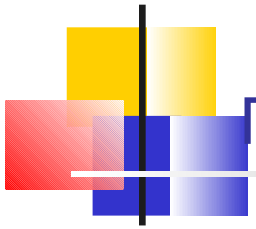
Pq – q случилось когда-то в прошлом

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

- *Смит умер и его похоронили*

  P(Смит_умирает $\wedge$ XF *Смита_хоронят*)

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

  G(P Увидел $\Rightarrow$ G(Встретил $\Rightarrow$ 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 (В.Бахнов)*

  G(Он_играет_джаз $\Rightarrow$ FX Он_продает_Родину)

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

  G(Персил $\Rightarrow$ G Персил)

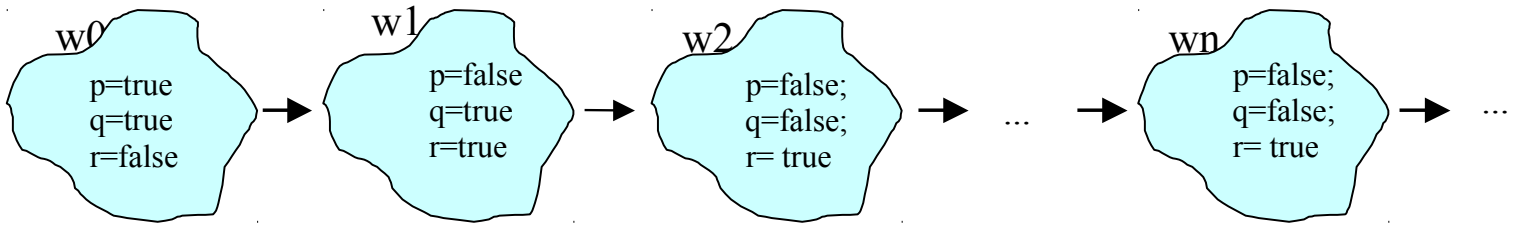
- *Любой запрос к ресурсу будет висеть до его подтверждения либо отклонения*

  G(Request $\Rightarrow$ Request U (Reject $\oplus$ Confirm))

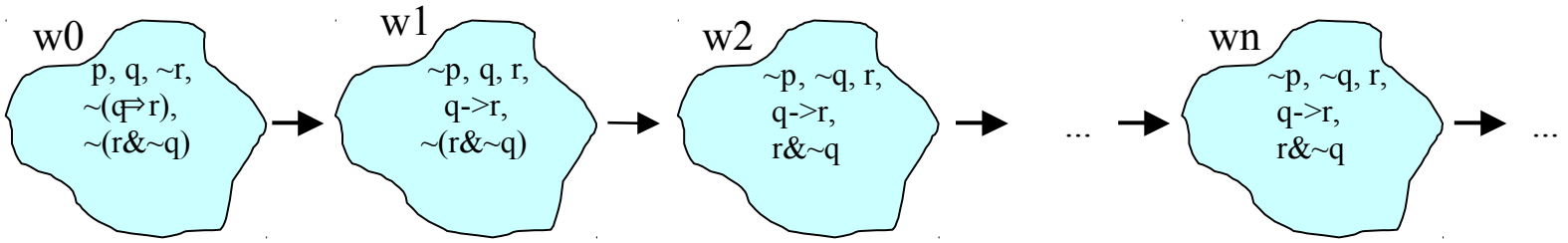
LTL в дискретном времени – А. Пнуэли (1977)

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

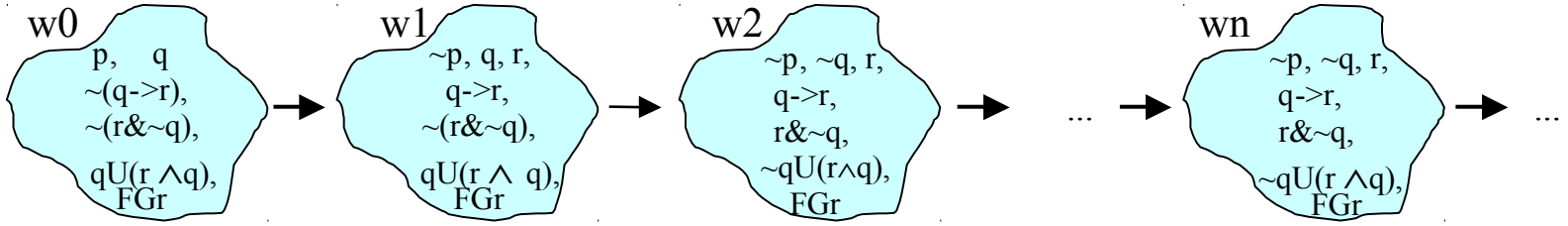
Семантика “возможных миров” Лейбница, в каждом свое понимание истинности:



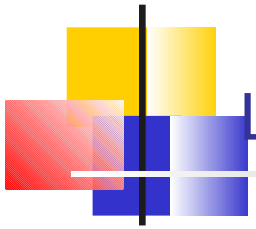
В каждом мире произвольная логическая формула истинна, либо нет:



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)

- Логика LTL – темпоральная логика линейного времени – утверждения о свойствах бесконечных вычислений
  - Формула φ LTL это :
    - атомарное утверждение (атомарный предикат) p, q, ...,
    - или Формулы, связанные логическими операторами ∨, ¬
    - или Формулы, связанные темпоральными операторами U, X
  - Модальностей прошлого в LTL нет

Грамматика: $\varphi ::= p \mid \varphi \vee \varphi \mid \neg \varphi \mid X\,\varphi \mid \varphi\, U\, \varphi$

Базис LTL = {¬, ∨, U, X }

Выводимые темпоральные операторы :

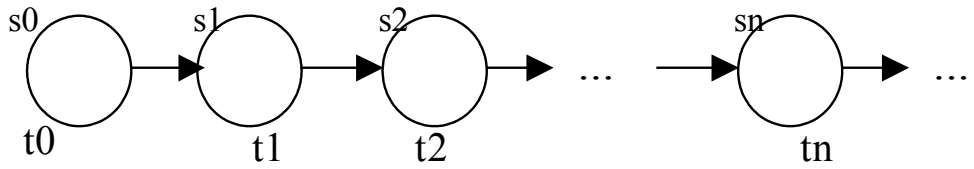
$F\,\varphi = true\ U\ \varphi$

$G\,\varphi = \neg F \neg \varphi$

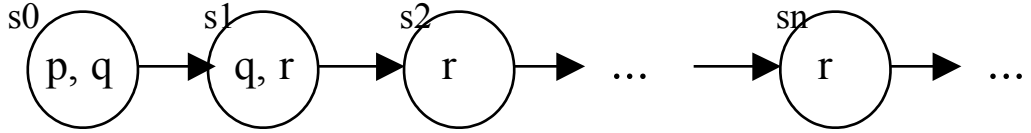
LTL
к логике высказываний
добавлены только два
модальных оператора:
Until и NextTime

LTL и спецификация свойств дискретных систем

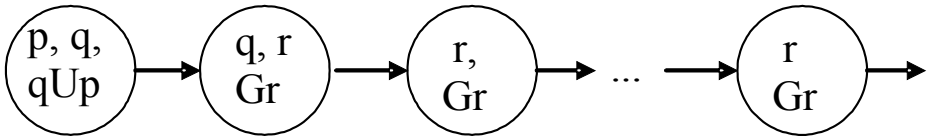
Последовательность "*миров*" в LTL можно толковать как бесконечную последовательность состояний дискретной системы, а отношение достижимости – как дискретные неделимые переходы системы:



Атомарные предикаты - базисные свойства процесса в состояниях:



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



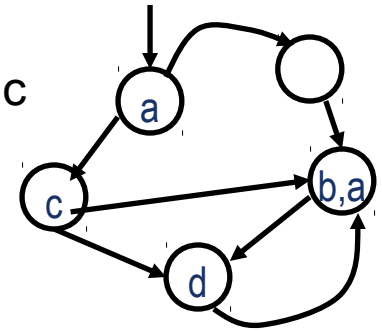
qUp – выполняется на вычислении

Gr – не выполняется на вычислении

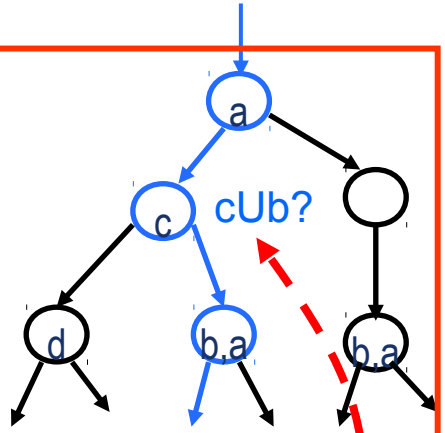
Линейное и ветвящееся время

Как конечным образом задать бесконечные вычисления - последовательности состояний

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*"

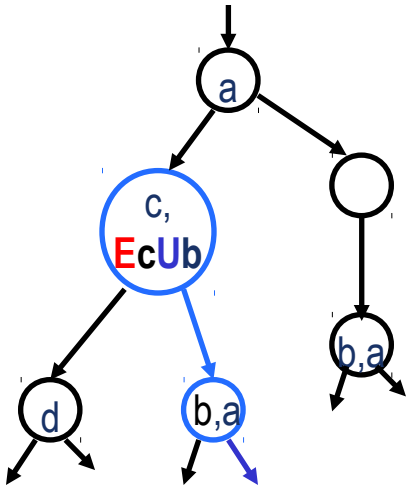
E φ ≡ "*существует такой путь из данного состояния, на котором* φ *истинна*"

A φ ≡ "*для всех путей из данного состояния формула пути* φ *истинна*"

Очевидно, **A** φ ≡ ¬**E** ¬φ

Логика ветвящегося времени – CTL*

Темпоральные логики ветвящегося времени
рассматривают возможные вычисления (пути на дереве) - траектории на развертке структуры Крипке



CTL* – Computational Tree Logic* - это одна из возможных логик ветвящегося времени

Грамматика:
- Формулы состояний $\alpha ::= p \mid \neg\alpha \mid \alpha \vee \alpha \mid E\,\varphi$
- Формулы путей $\varphi ::= \alpha \mid \varphi U \varphi \mid X\varphi$

формула α состояния s является формулой пути σ, если это состояние s является начальным состоянием пути σ

Базис CTL* = {¬, ∨, U, X, E}

LTL и CTL – подклассы CTL*

Формулы CTL*:

E(¬p & **X A FG** q)

E(a & **AX**(b**U**c)]

A (a **U** ¬ (**F** b))

Формулы LTL:

AG(p ⇒**F** q)

A (¬a∨ **G**b & (a**U** ¬c))

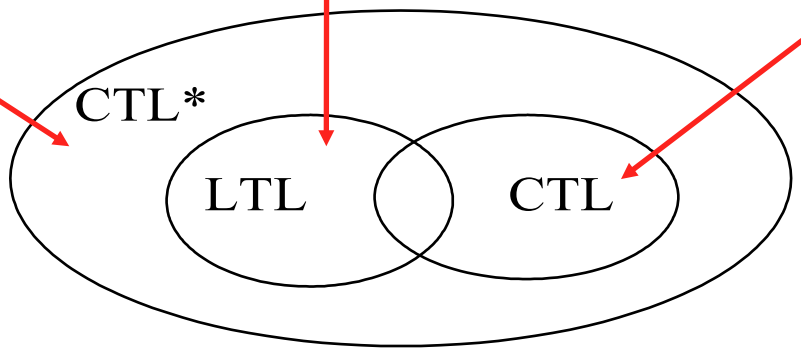
A (a **U** ¬b)

Формулы CTL:

AG(p&¬**EF**(q⇒r))

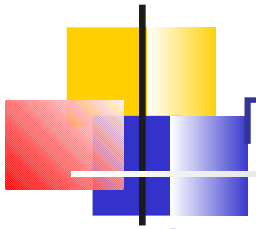
EF(a & **E**(a**U** ¬c))

A (a **U** ¬ b)



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

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

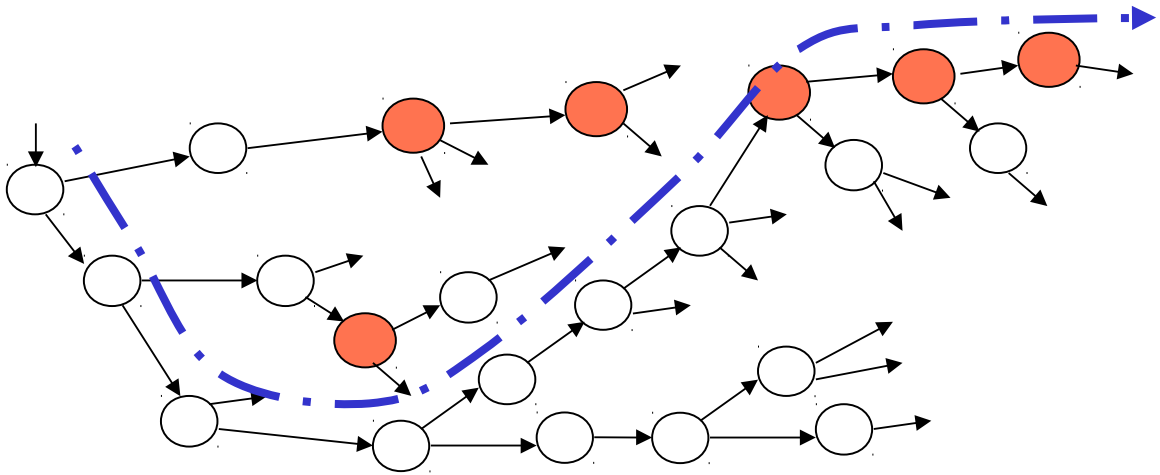


Примеры спецификации требований в логике CTL

- AGAF Restart
  - из любого состояния при любом функционировании системы обязательно вернемся в состояние рестарта
- ¬EF(int >0.01)
  - не существует такого режима работы, при котором интенсивность облучения пациента превысит 0.01 радиан в сек
- AG (req ⇒ 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 активным до выполнения условия r

Пример свойства, выраженного формулой логики CTL

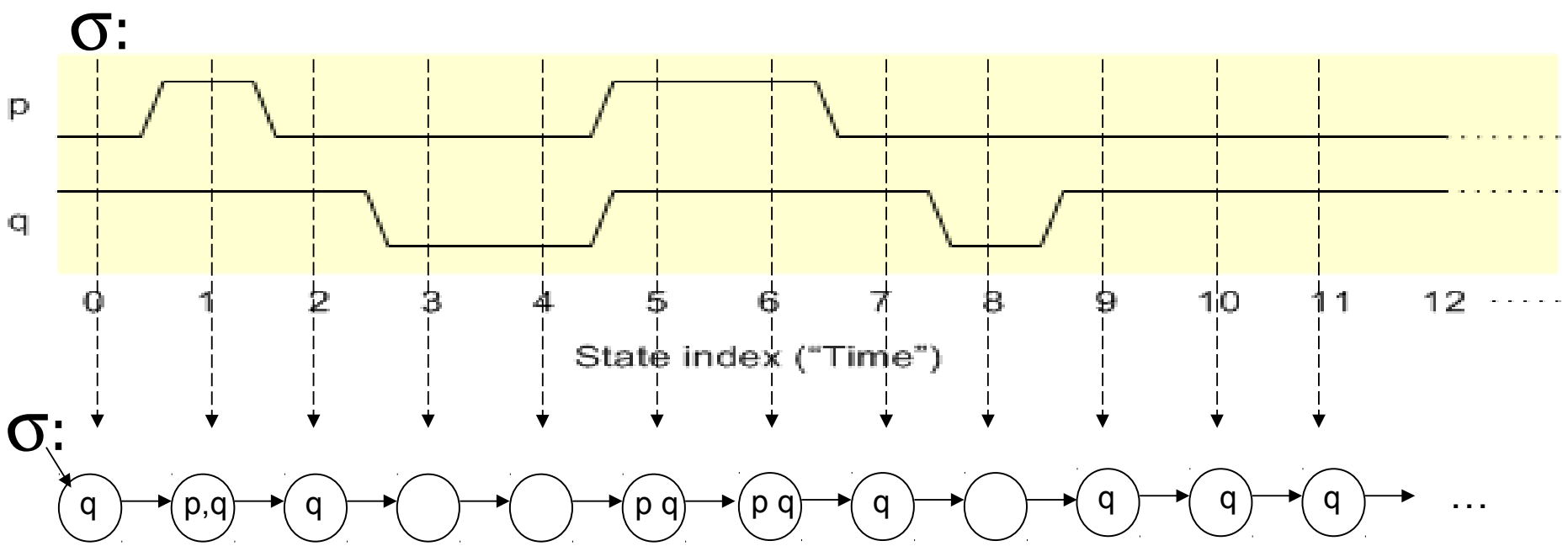
- *Любой грешник всегда имеет шанс вернуться на путь истинной веры*



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’

Спецификация свойств поведения дискретных схем



$\sigma_0 \models \neg p$

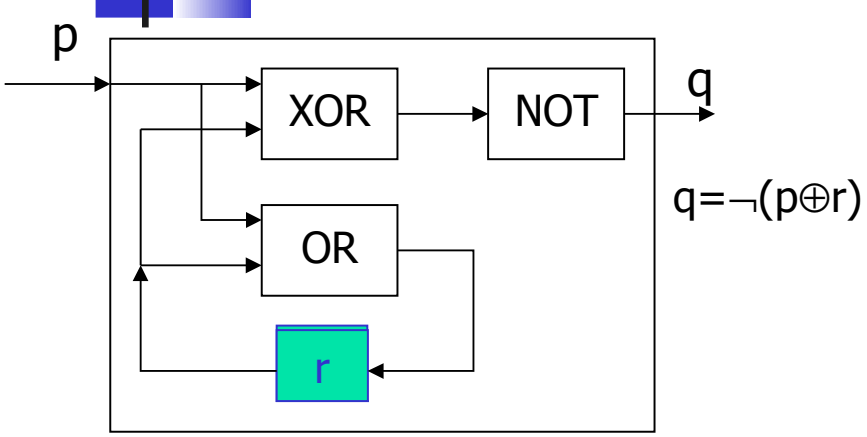
$\sigma_0 \models \mathsf{F}\ \neg q$

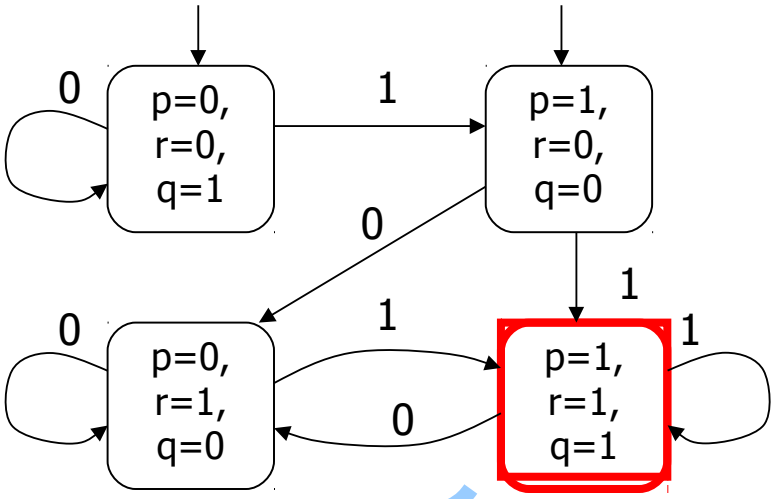
$\sigma_0 \models \mathsf{G}(p \rightarrow q)$

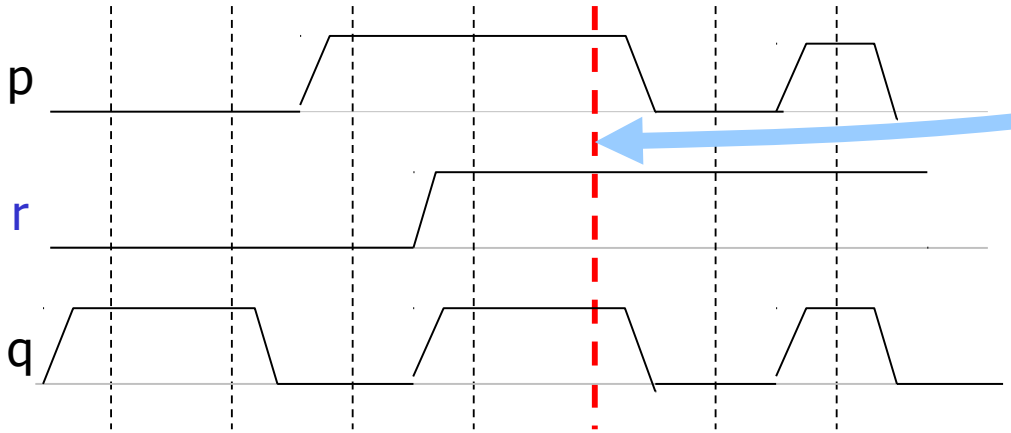
$\sigma_0 \models q\ \mathsf{U}\ p$

$\sigma_0 \models \mathsf{FG}(\neg p \wedge q)$

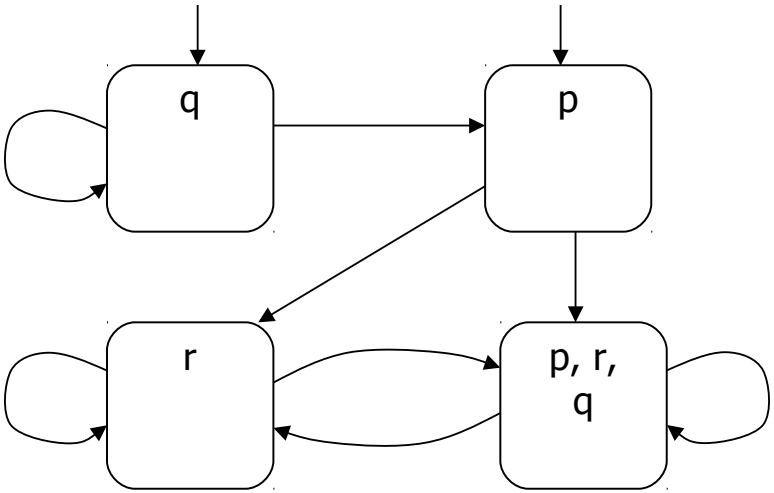
Структура Крипке синхронных схем с памятью







Структура Крипке:



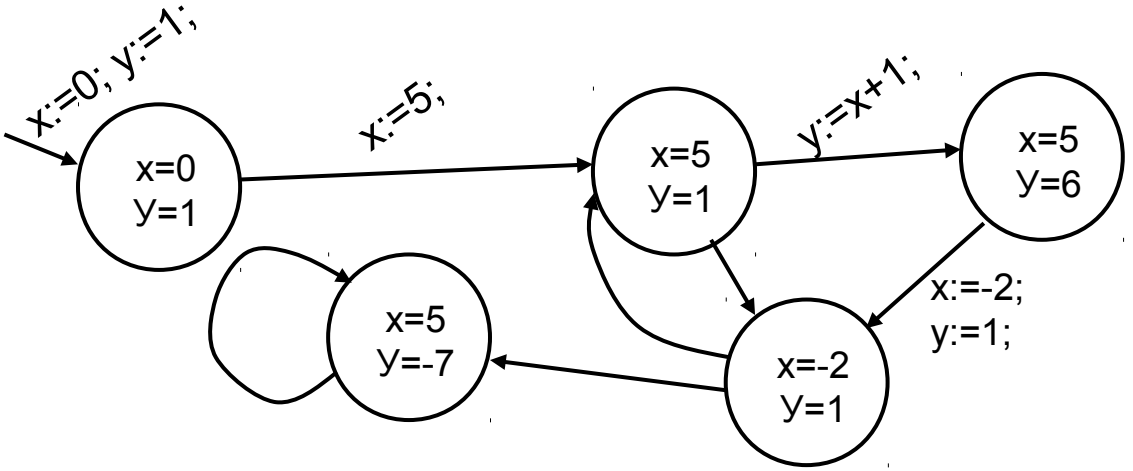
- Переходы выполняются в каждом такте
- Задержка: логическая схема - 0, память – 1
- Два начальных состояния

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
```

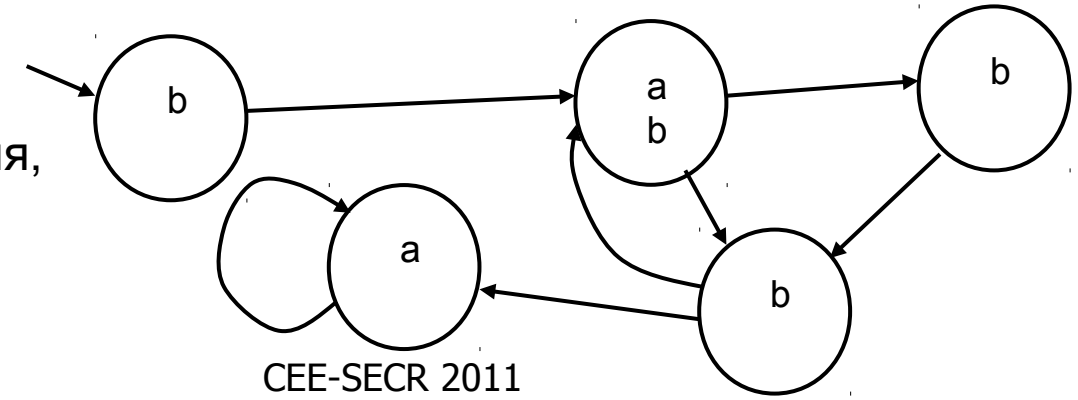
Состояние программы – вектор значений ее переменных И метки (pc)

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

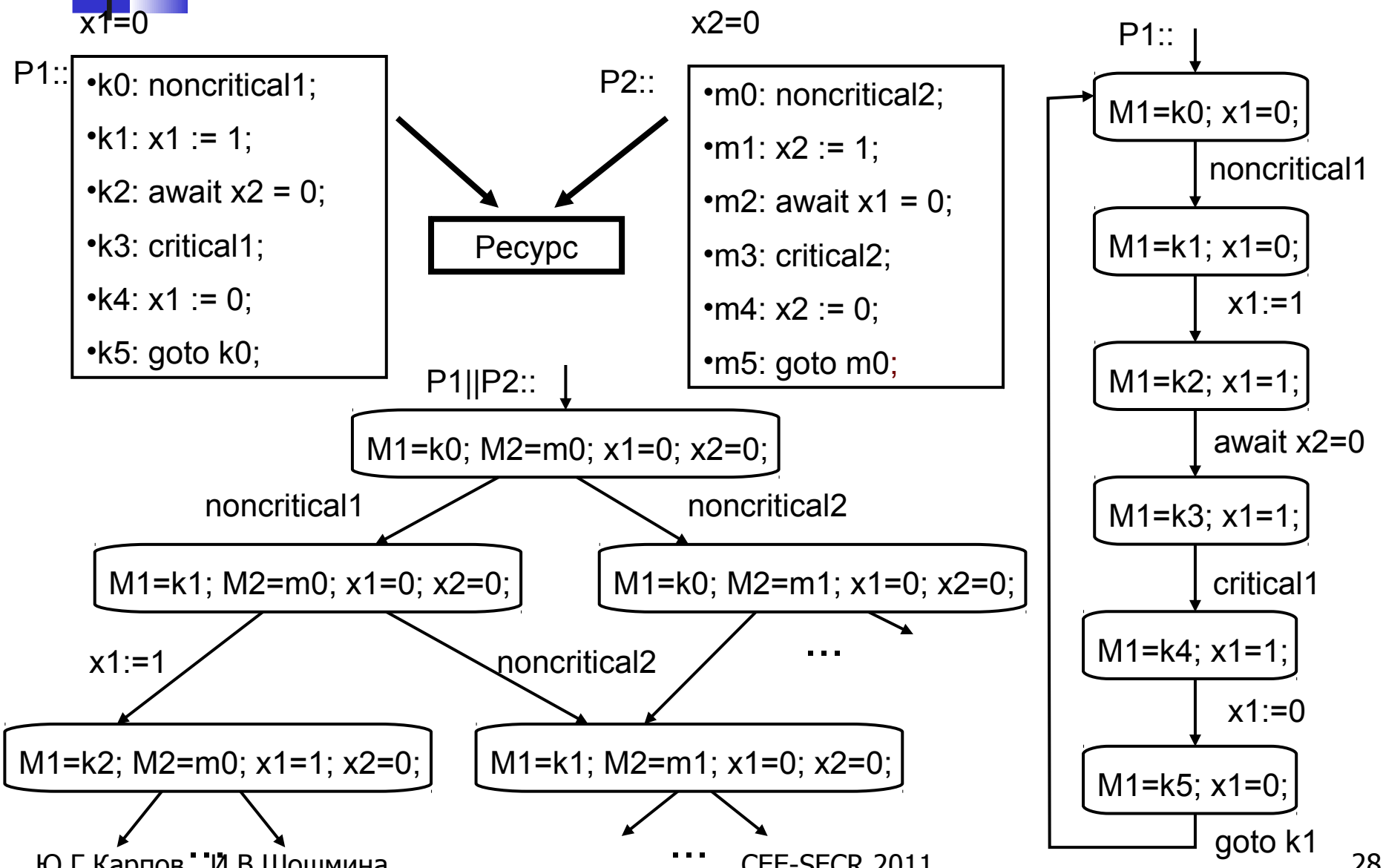


Пусть атомарные утверждения, *ИНТЕРЕСУЮЩИЕ НАС:*

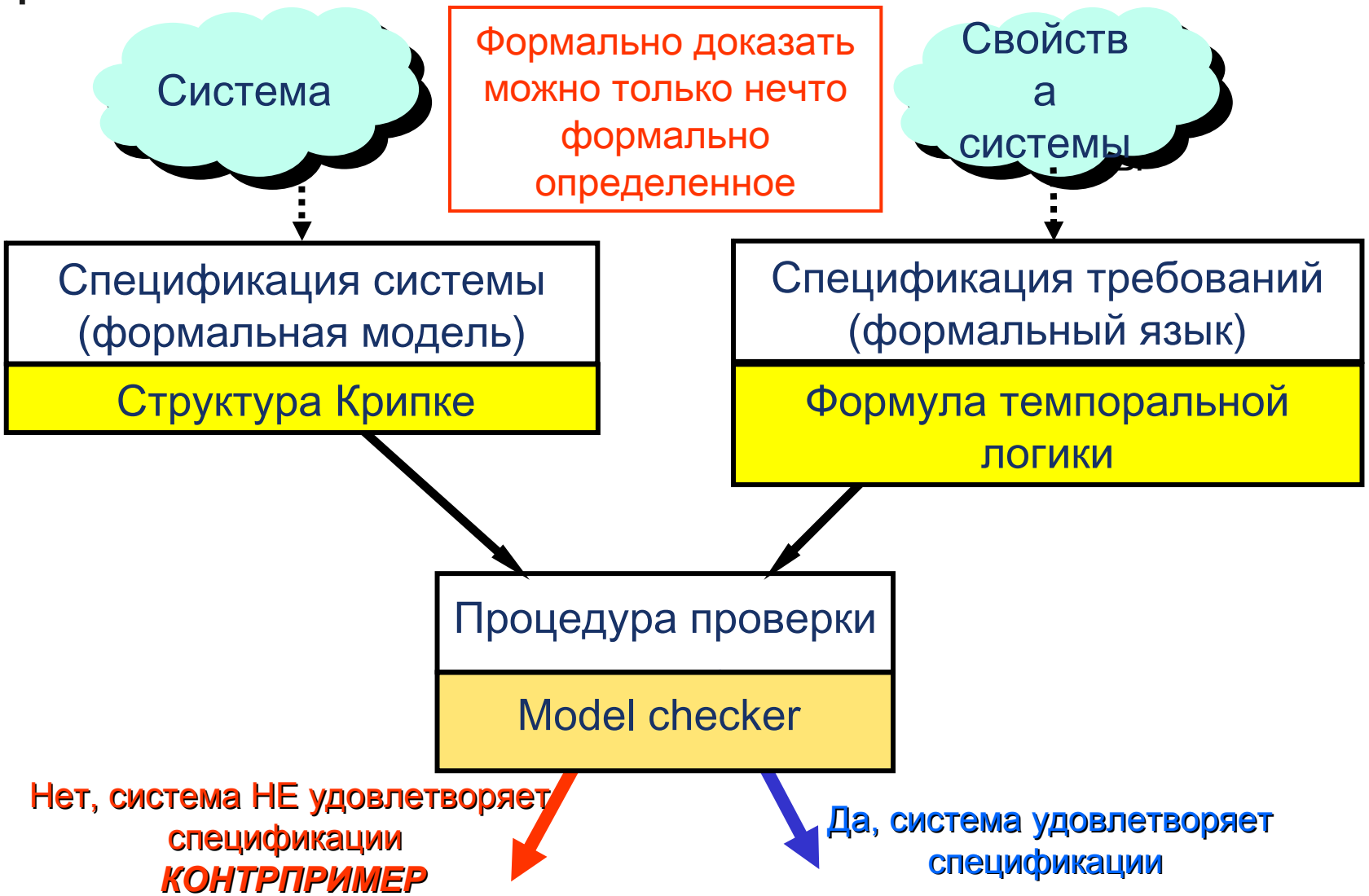
a=x>y; b=|x+y|<3



Структура Крипке - модель параллельной программы



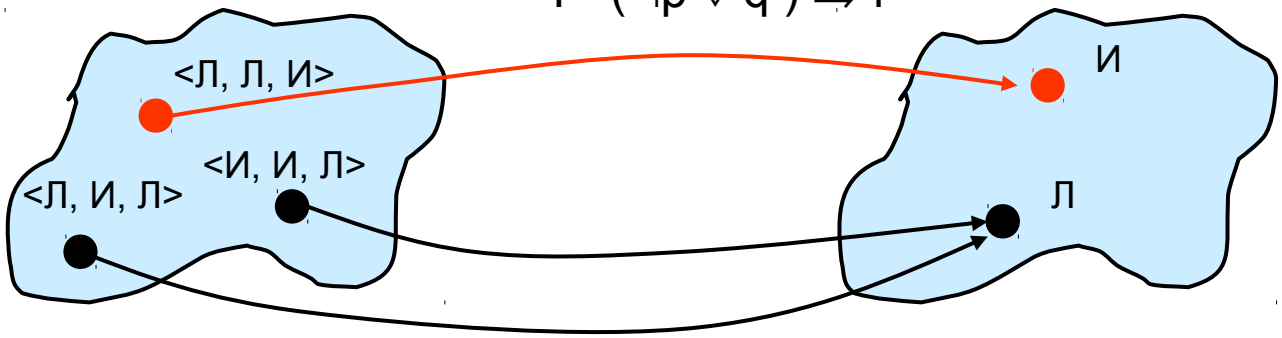
Общая схема верификации Model checking



Model checking – проверка того, является ли К моделью Ф

Логика высказываний

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

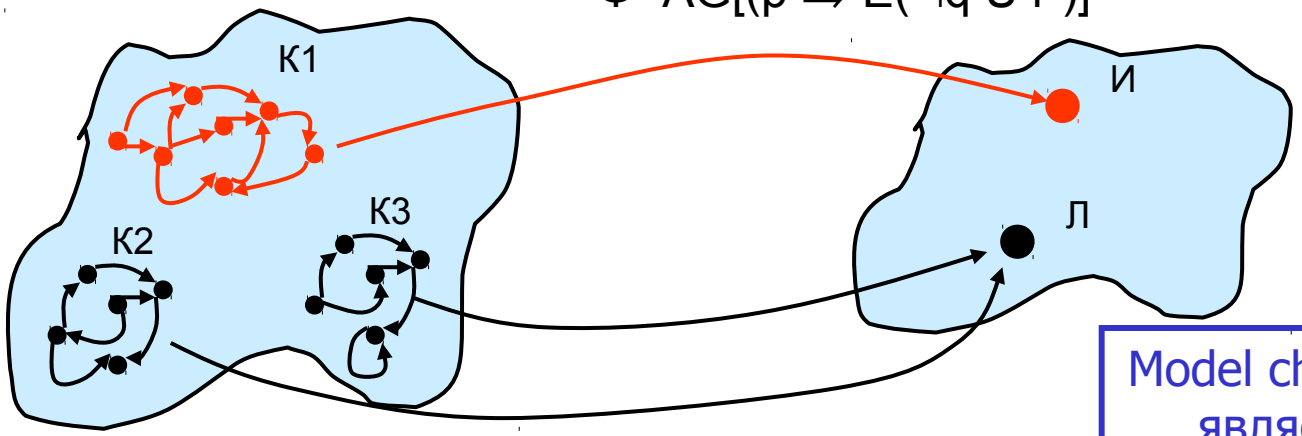


Интерпретации PL – наборы значений переменных <p, q, r>

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

Темпоральная логика

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



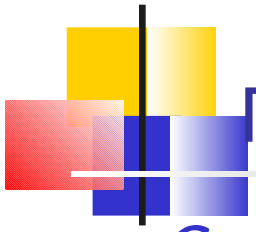
Интерпретации TL – структуры Крипке, в каждом состоянии которых свой набор значений переменных <p, q, r>

Model checking – проверка того, является ли К моделью Ф

Интерпретация К1 - модель формулы Ф

Материализация абстрактных понятий

- Формализмы, на которых основан метод Model Checking:
  - темпоральные логики
  - линейное и ветвящееся время
  - структуры Крипке
  - недетерминированные автоматы Бюхи
  - бесконечные множества бесконечных цепочек (ω-языки)
  - синхронная и асинхронная композиция автоматов
  - неподвижные точки операторов на решетках предикатов ...
- Все эти абстракции, которые и представить себе нельзя, привели к разработке мощной техники верификации, которая стала частью промышленной технологии разработки критического ПО и аппаратуры
- Зрелость техники верификации – АВТОМАТИЗАЦИЯ этапов, разработаны "*Push Button*" ИНСТРУМЕНТЫ для верификации
- Сложность очень мала: CTL: O(|M| * |Φ|), LTL: O(|M| * $2^{|\Phi|}$)
- Model checking - "реабилитация формальных методов"



Примеры использования подхода

С помощью Model checking были найдены ошибки во многих критических программных системах

- 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

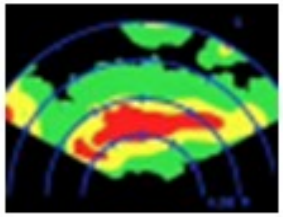
Rockwell Collins

Who Are We?

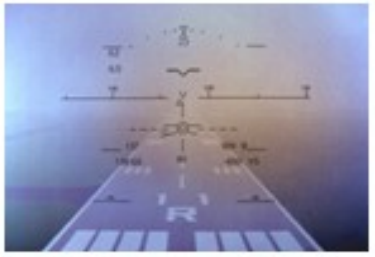
ADVANCED COMPUTING SYSTEMS

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

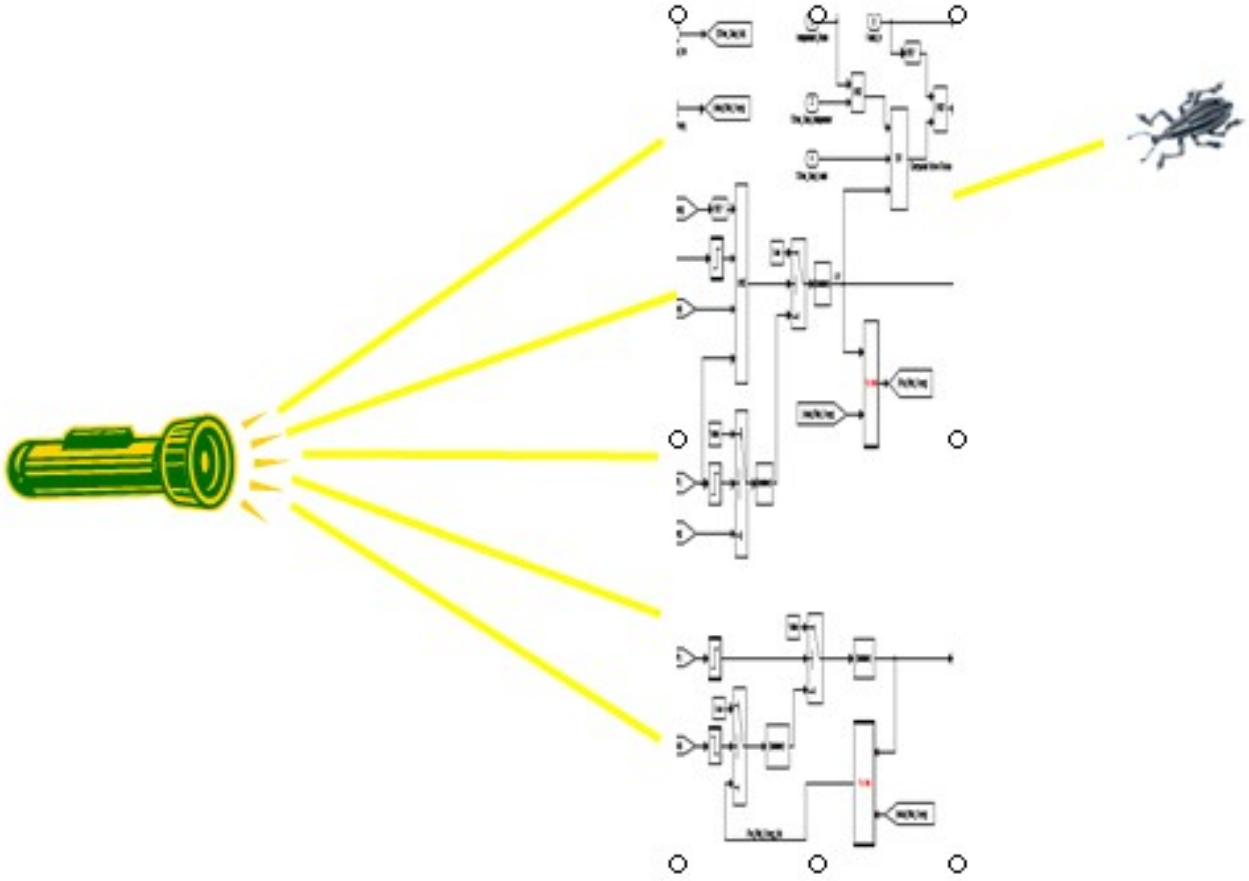
What Are Model Checkers?

- 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)

Advantage of Model Checking

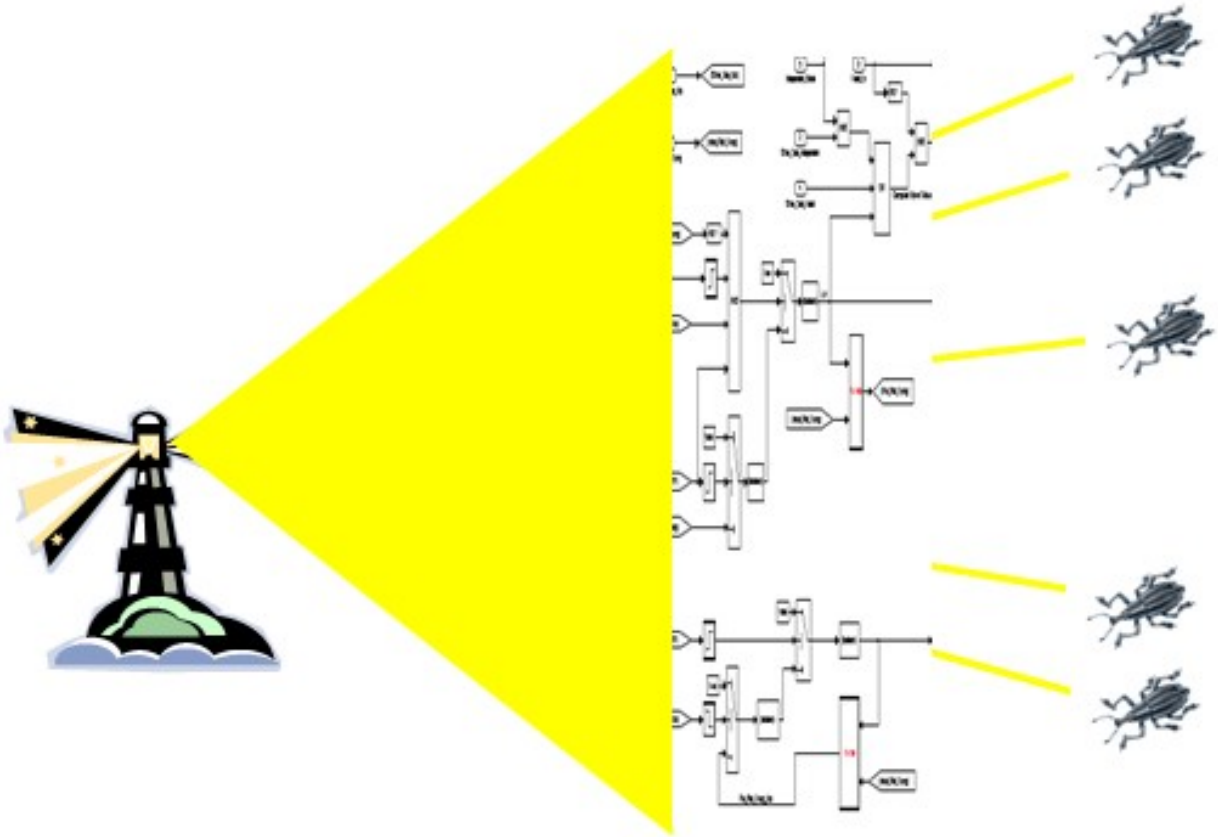
Testing Checks Only the Values We Select

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



Advantage of Model Checking

Model Checker Tries Every Possible Input and State!



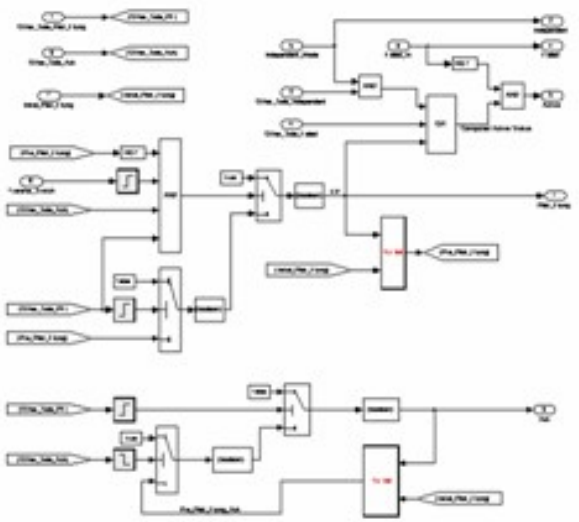
Example - ADGS-2100 Adaptive Display & Guidance System

ADVANCED COMPUTING SYSTEMS

883 Subsystems

9,772 Simulink Blocks

2.9 x 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

Summary of Errors Found

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**

Conclusions

- **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**

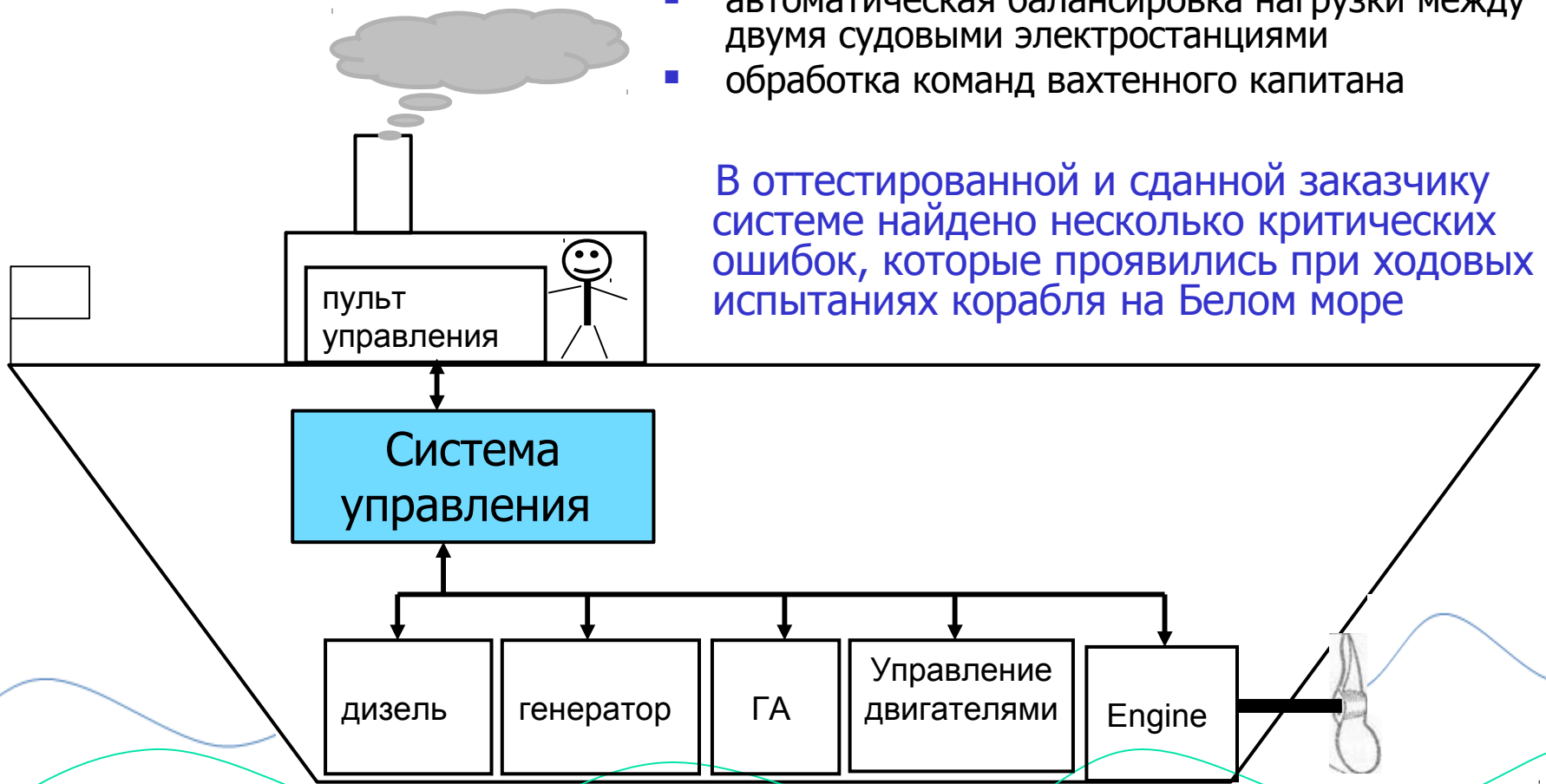
Пример: верификация системы управления электродвижением судна

Проект научной группы Петербургского Политеха

Функции системы управления:

- контроль состояния судовых электростанций и обработка критических ситуаций
- автоматическая балансировка нагрузки между двумя судовыми электростанциями
- обработка команд вахтенного капитана

В оттестированной и сданной заказчику системе найдено несколько критических ошибок, которые проявились при ходовых испытаниях корабля на Белом море



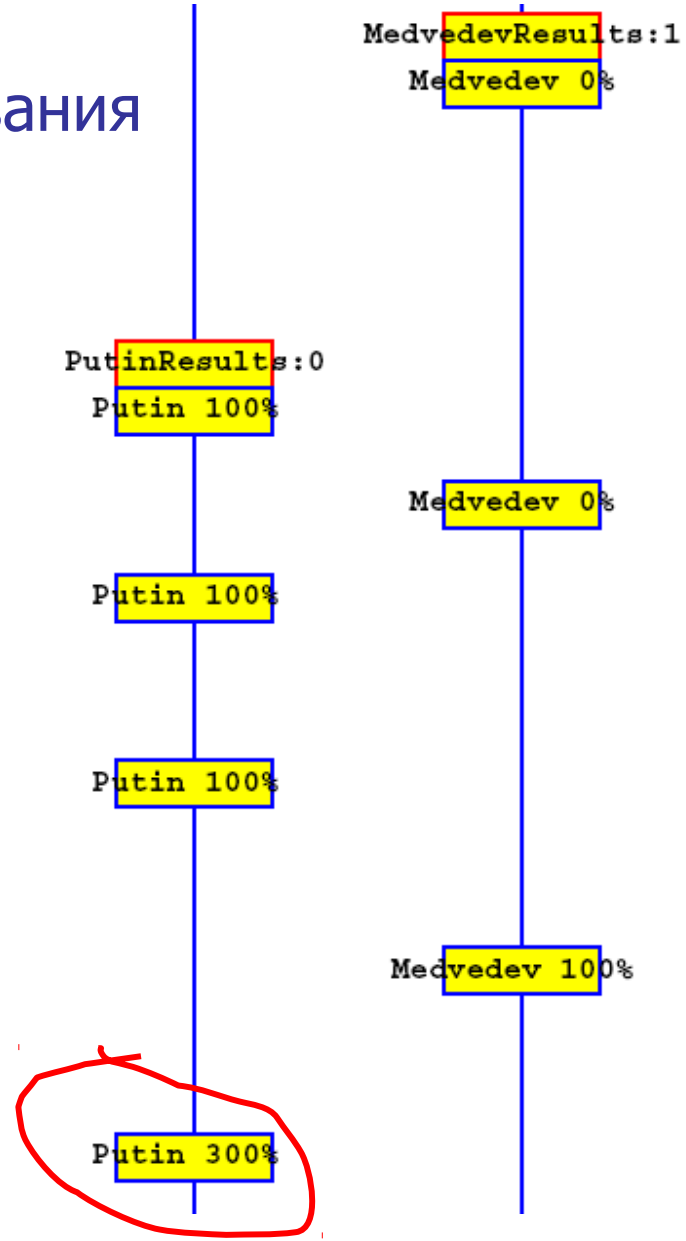
Верификация системы голосования

- Верификация на системе Spin программной системы, которая (предположительно) обеспечивает голосование на “Эхе Москвы”, обнаруживает ошибку

Проверка свойства:

“*За Путина не проголосуют 300%*”

Контрпример:

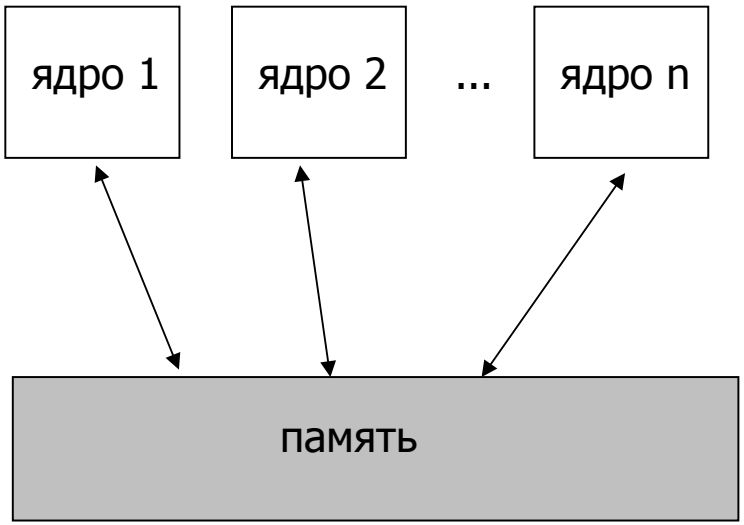


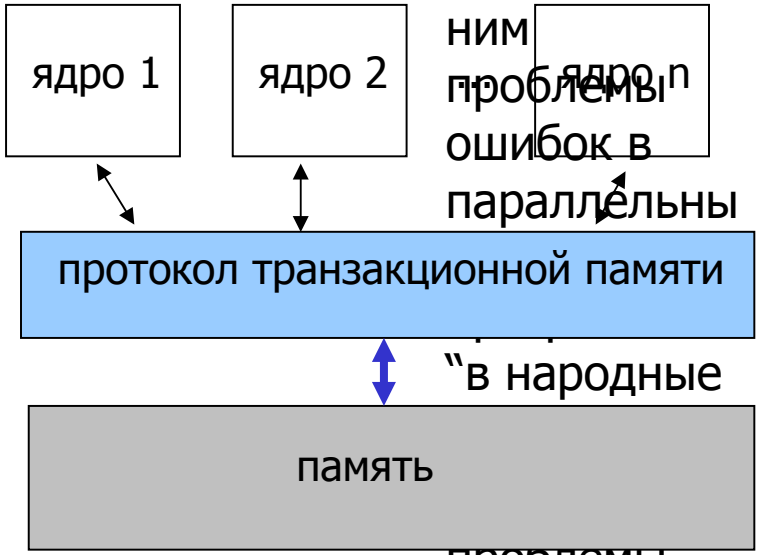
Success Story.
Транзакционная память в многоядерных процессорах

-

Многоядерные процессоры вывели параллельное программирование и связанные с ним проблемы ошибок в параллельны “в народные проблемы ошибок

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. предложили оконный алгоритм транзакционной памяти вместе с аналитическим доказательством его корректности

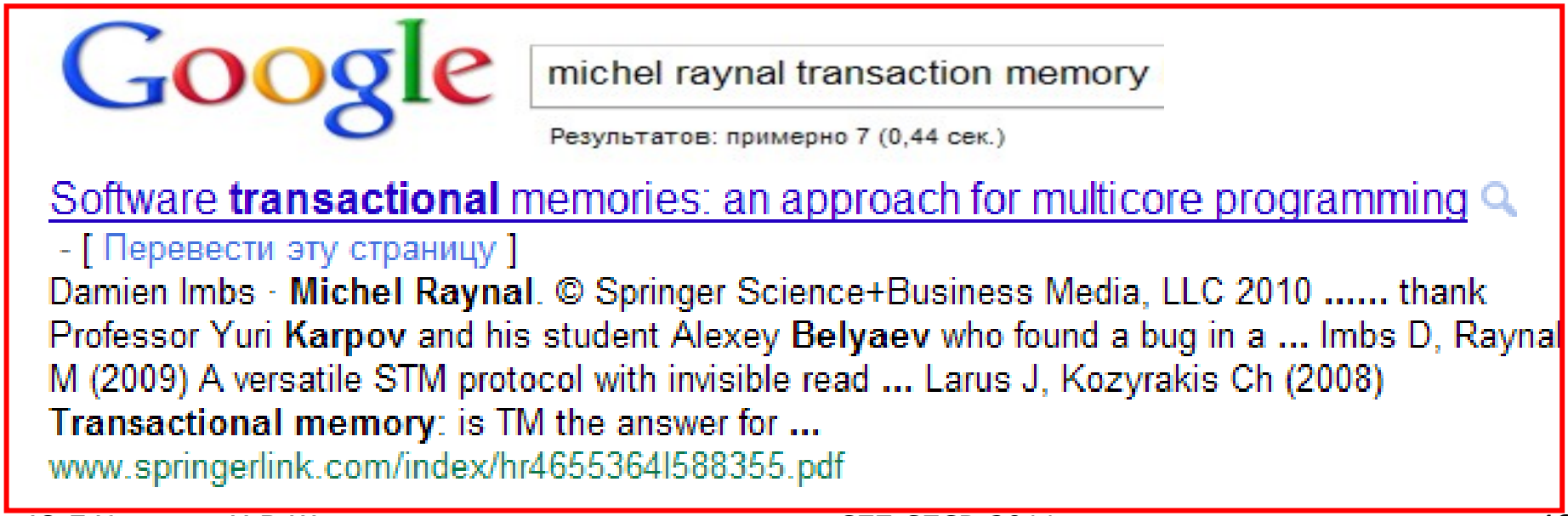
Michel Raynal - a professor of computer science since 1981. At IRISA (CNRS-INRIA-University joint computing research laboratory located in Rennes), he founded a research group on Distributed Algorithms in 1983.

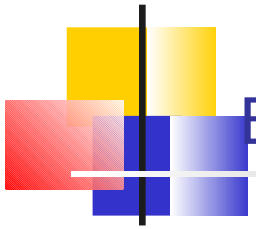
His research interests include distributed algorithms, distributed computing systems and dependability. His main interest lies in the fundamental principles that underlie the design and the construction of distributed computing systems. He has been Principal Investigator of a number of research grants 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

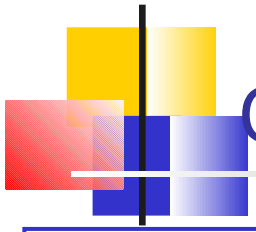
- Imbs D.; Raynal M. *Software Transactional Memories: an Approach for Multicore Programming*. LNCS, 5698, 2009
- Эта работа была дана студенту 4 курса Петербургского Политеха Алексею Беляеву для проверки протокола в рамках его НИРС
- Студент - описал требования к протоколу формулой LTL,
  - использовал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





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 верификации 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

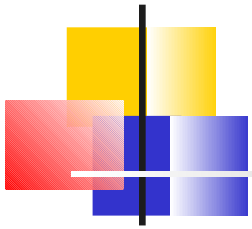
“Формальные методы должны стать частью образования каждого исследователя в области информатики и каждого разработчика ПО так же, как другие ветви прикладной математики являются необходимой частью образования в других инженерных областях.

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

Обеспечение качества разработанной технической системы является важнейшей составляющей каждой инженерной специальности

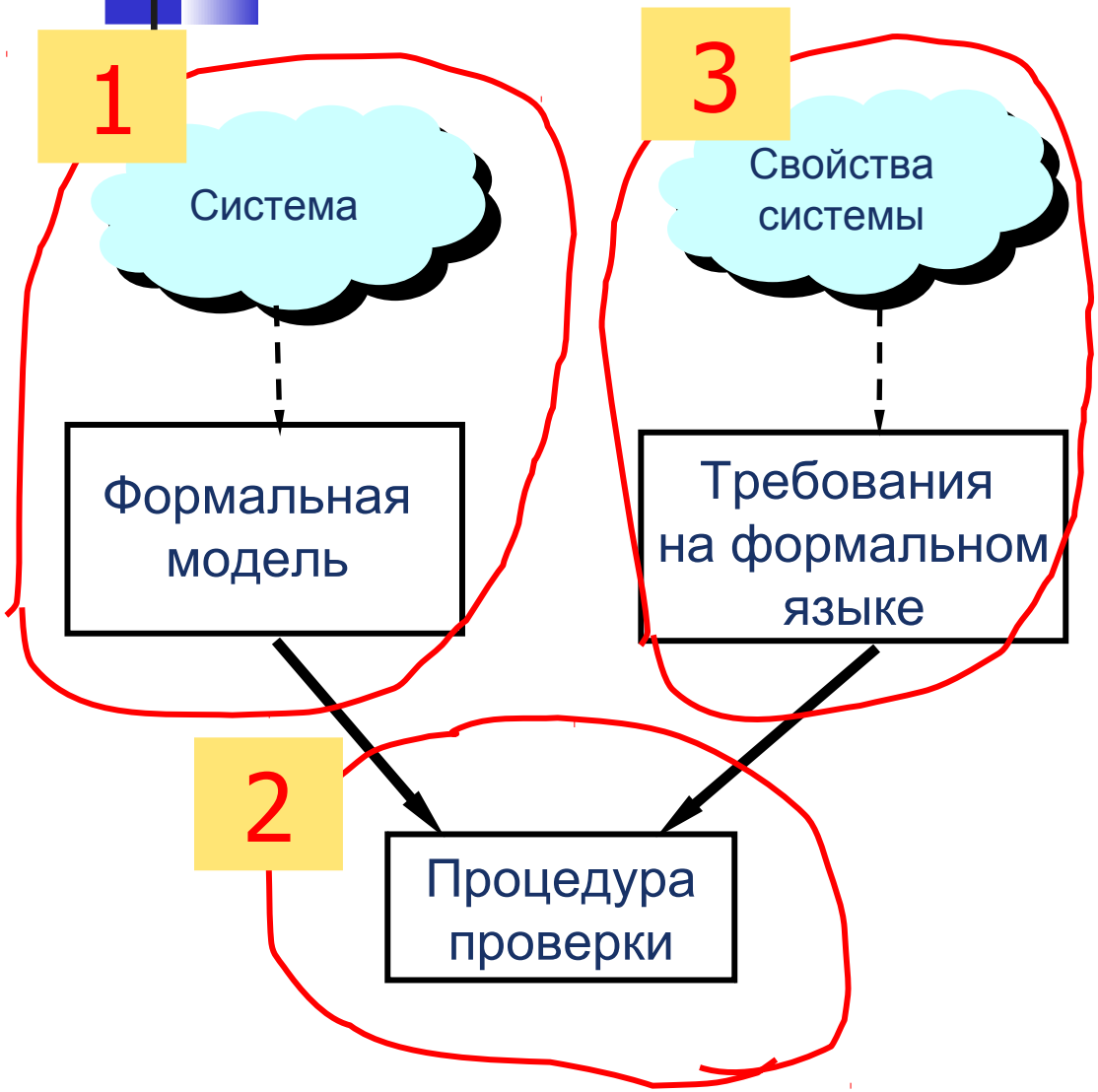
Конструкторы самолетов изучают аэродинамику, строители – сопротивление материалов, машиностроители изучают теорию механизмов и машин. Каждая инженерная специальность имеет свою область прикладной математики, на которой основаны теории, позволяющие гарантировать качество инженерных разработок в этих областях

В области разработки ПО необходимые разделы прикладной математики – это формальные методы, на которых основывается верификация: формальная логика, теория автоматов, теория алгоритмов



Проблемы верификации

Проблемы при верификации программных систем

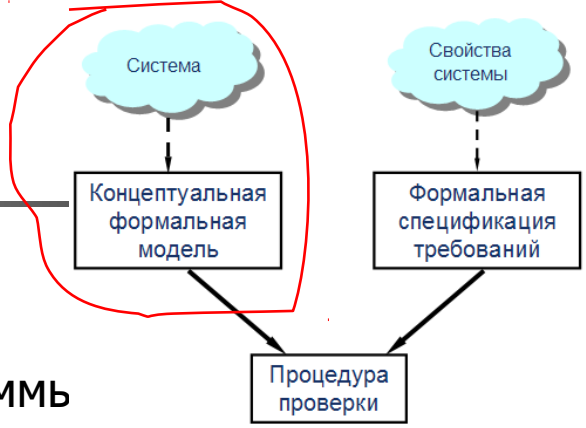


1. Модель из кода или код из модели?

2. “State explosion problem”?

3. Какие требования проверять?

Модель из кода или код из модели?

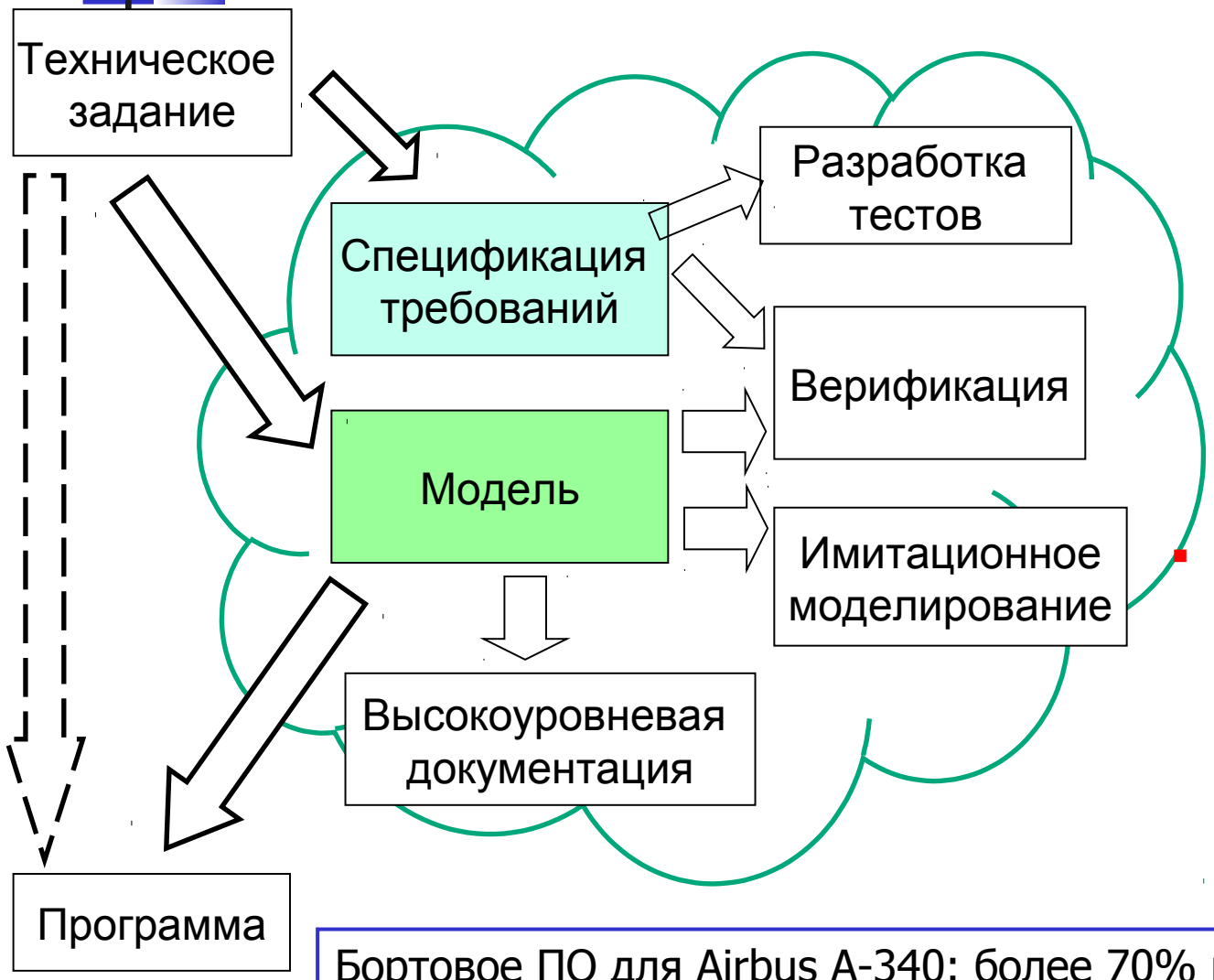


Высказывания классиков

- Абсурдно ставить проблему доказательства программь 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)

Разумно ли СНАЧАЛА строить самолет, а ПОТОМ проводить анализ, чтобы увидеть, полетит ли он?

Проектирование ПО на основе моделей (Model Based 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

Бортовое ПО для Airbus A-340: более 70% кода сгенерировано

"State explosion problem"

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

Но этого совершенно недостаточно!

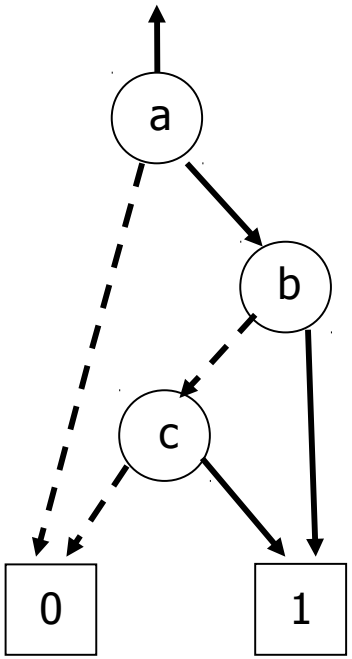
Методы борьбы с проблемой "State explosion":

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

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

- Эффективная форма представления булевых функций

$F = ab \vee a\neg bc$



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

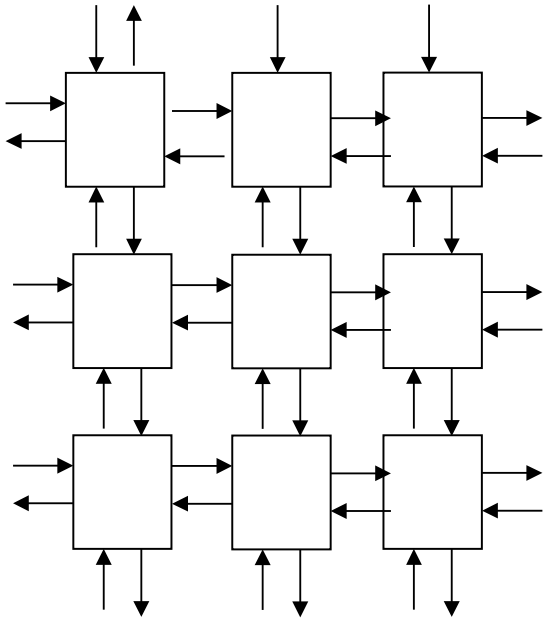
Применения:

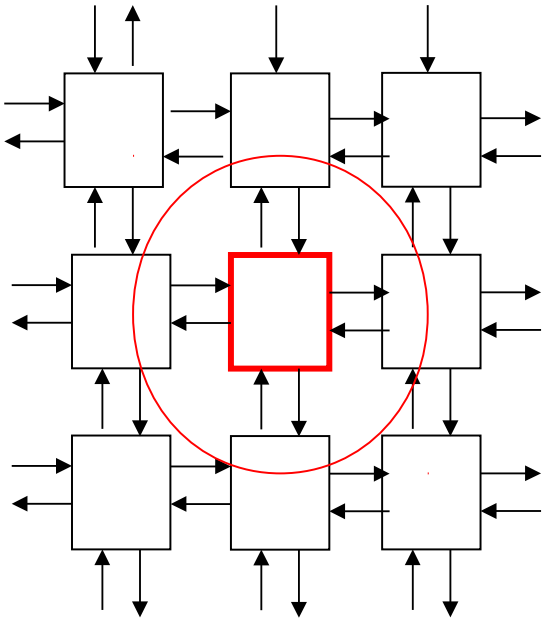
- везде, где можно использовать булевы функции
- революция в работе с конечными структурами данных

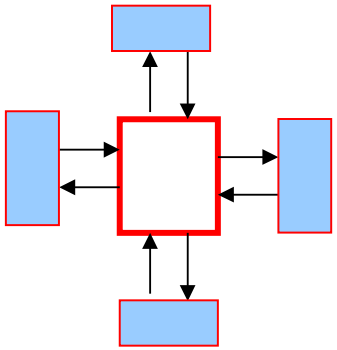
Использование BDD в алгоритмах верификации позволило увеличить сложность верифицируемых систем (число состояний структуры Крипке) в миллиарды миллиардов миллиардов раз – с 10^6 до 10^{100} и даже до 10^{300}

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

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





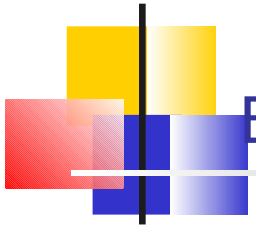


Какие требования проверять?

Понятие “полное” множество требований формально определить нельзя

- Законы Айзека Азимова для роботов
  - Первый Закон: *Робот не может причинить вред человеку или своим бездействием допустить, чтобы человеку был причинен вред*
  - Второй Закон: ...
  - Третий Закон: ...
- Что такое *вред человеку*? Смерть- вред? Эвтаназия (легкая смерть)?
  - - в Швейцарии, ... – законодательно разрешена
  - - в России – преступление, квалифицируется как умышленное убийство

- Формальная спецификация – это контракт между программистом и заказчиком, который обеспечивает их обоих общим точным, недвусмысленным пониманием цели разработки и ее эволюции
- Методы формальной спецификации: B-спецификация [Abrial, 1996], VDM [Jones 1990], ASM [Borger, 2003]

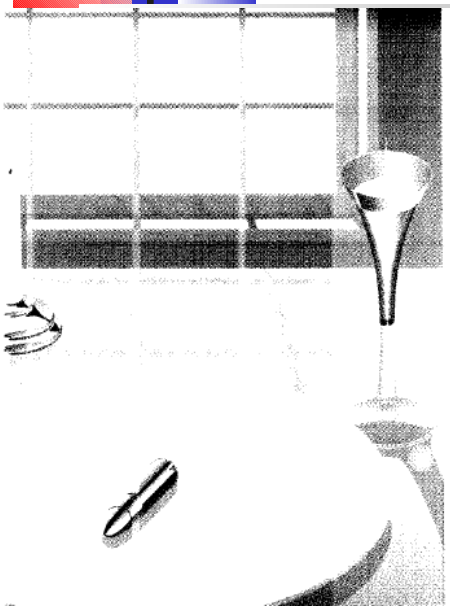


Выводы

- Наивен вопрос о том, можно ли выполнить полное доказательство правильности реального ПО
  Понятие “полной правильности” неформальное, его формализовать нельзя
- Можно ли ожидать, что для бортовых систем масштаба AirBus A330 (2 M строк) или Boeing 770 (4 M строк) можно формально доказать выполнимость критических свойств всего ПО? Вопрос остается открытым
- Для отдельных подсистем эта задача сегодня вполне разрешима
- Формальные модели, на которых должны основываться спецификация, разработка и верификация ПО, уже входят в стандарты
- DO-178C: допускает возможность использования
  (в отличие от DO-178B)
  - формальных методов,
  - разработку, основанную на моделях (Model-Based Development)
  - ОО технологии

Biting the Silver Bullet

David Harel, IEEE Computer, 1992



Biting the Silver Bullet

Toward a Brighter Future for System Development

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

- Frederic Brooks (1986 г., "No Silver Bullet") писал 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, визуальные языки, ... Весьма пессимистический взгляд
- David Harel: хотя единственного средства ("серебряной пули") нет, но интегральное использование новых идей, в частности, моделирования, формальных методов, визуального программирования и генерации кода может внести существенный вклад в разработку сложных reactive systems

Возрастающее использование формальных методов для верификации ПО на ранних стадиях разработки

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

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

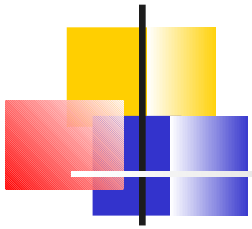
Д. Харел (1992):

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

Сегодняшние успехи формальной верификации показывают:

"БУДУЩЕЕ НАЧИНАЕТСЯ СЕГОДНЯ"

Верификация достигла уровня зрелости, хотя все еще требует больших усилий в ее применении



Спасибо за внимание