

Верификация параллельных программных и аппаратных систем



Курс лекций

Шошмина Ирина Владимировна

Карпов Юрий Глебович



План курса

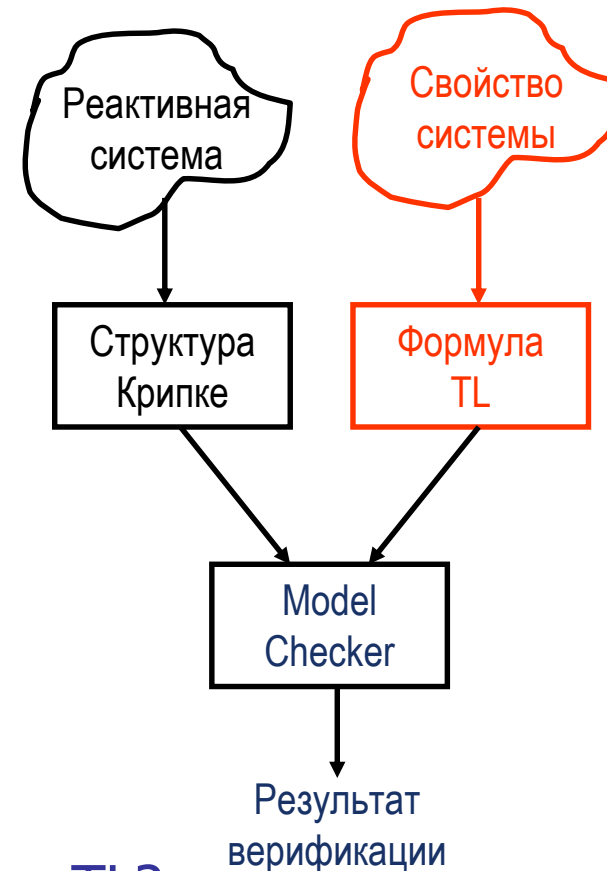
1. Введение
2. Метод Флойда-Хоара доказательства корректности программ 1
3. Метод Флойда-Хоара доказательства корректности программ 2
4. Темпоральные логики
5. Алгоритм model checking для проверки формул CTL
6. BDD и их применение
7. Символьная проверка моделей
8. Автоматный подход к проверке выполнения формул LTL
9. Система верификации Spin и язык Promela. Примеры верификации
10. Структура Крипке как модель реагирующих систем
11. Темпоральные свойства систем
12. Применения метода model checking
13. Количественный анализ дискретных систем при их верификации
14. Верификация систем реального времени
15. Современное состояние верификации

Спецификация и проверка свойств

Существующие верификаторы позволяют проверить для данной структуры Крипке любое свойство (требование), заданное в виде формулы темпоральной логики

ЕСЛИ ХВАТИТ РЕСУРСОВ

- Сколько свойств проверять?
- Какие свойства проверять?
Являются ли они “существенными”
- Как выразить ‘ключевые’, функциональные требования к конкретной системе формулами TL?



**Эти вопросы нетривиальны.
Не все имеют удовлетворительное решение**



Специфицировать требования на языке TL непросто

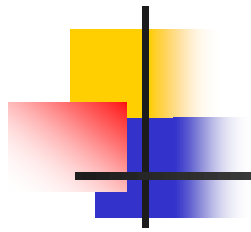
- Технология верификации методом Model checking автоматизированна (“**push button technology**”), однако разработчик должен сам формулировать требования к проверяемой программе на языке LTL или CTL. Для каждой системы нужно придумывать свои требования
- Запись требований на языке TL не всегда ясна, написание TL формул требует от разработчика знания стандартных правил, идиом, ...

Temporal logics (CTL, LTL, ...)

- powerful, expressive
- but sometimes non-intuitive, even for experts

Gerard Holzmann
(создатель SPIN, Bell Labs.)

- Понимание того, как используется темпоральная логика для выражения требований к дискретным системам приобретается при решении задач



Примеры спецификации требований на LTL

Пример спецификации требования (LTL)

Как выразить свойство P:

"запрос req в конце концов приведет к получению ack"

■ $req \rightarrow ack$

неправильно: Выполнимость формулы LTL проверяется только для ПЕРВОГО состояния вычисления. Поэтому $req \rightarrow ack$ означает: *"в начальном состоянии если req истинно, то истинно и ack"*

■ $Greq \rightarrow ack$.

неправильно: понимается как $(Greq) \rightarrow ack$, *"если во всех состояниях выполняется req, то в начальном выполняется ack"*

■ $G(req \rightarrow ack)$

неправильно: *"в любом состоянии если выполняется req, то в этом же состоянии должно выполняться и ack"*

Спецификация причинной зависимости (2)

Как выразить свойство P:

"запрос req в конце концов приведет к получению ack"

- $G(req \rightarrow Fack)$

не совсем правильно: "в любом состоянии если выполняется *req*, то когда-нибудь в будущем (включая настоящее) должно выполняться и *ack*". Эта формула получила свой значок " $\sim>$ ": $\varphi \sim> \psi \equiv G(\varphi \rightarrow F\psi)$.

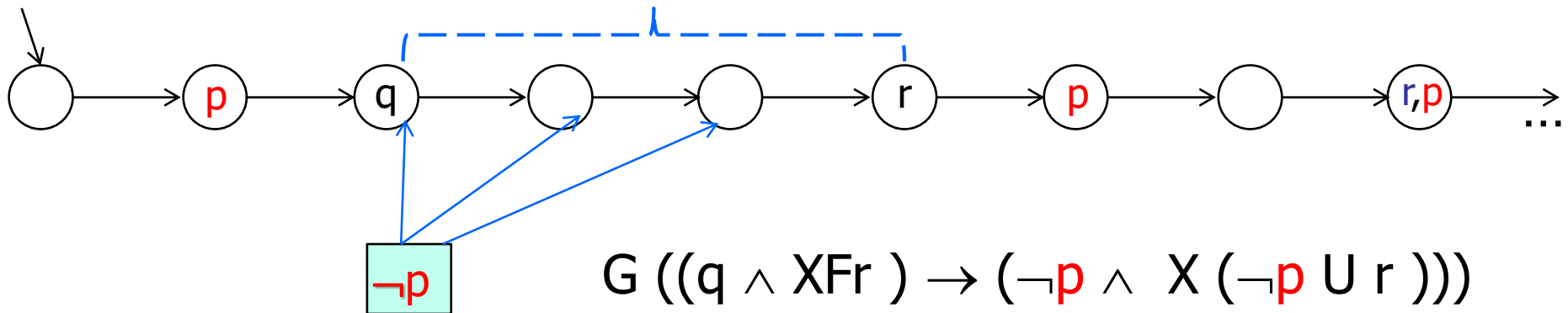
Но она будет истинна и в случаях, когда в том же состоянии, в котором истинно *req*, будет истинно и *ack*, что не является

- $G(req \rightarrow XFack)$

более точно отражает причинно-следственную связь между *req* и *ack*, но выполняется и в случае, если *req* вообще не наступает!

Об однозначности формализации: требование к поведению дискретных систем в LTL

"Событие p не должно выполняться между событиями q и r "



Требование формулируется относительно НАЧАЛЬНОГО состояния

Формулировка требования на естественном языке допускает несколько интерпретаций:

p не должна выполняться от очередного q

- до ближайшего r , или
- до всех будущих r ?
- включая r ?

Формальное определение на языке LTL **однозначно** определяет:

здесь определено, что '**после наступления q до ближайшего r** ...

Спецификация систем: спецификация работы светофора



- $G(\text{green} \cup \text{yellow} \wedge \text{yellow} \cup \text{red} \wedge \text{red} \cup \text{yellow} \wedge \text{yellow} \cup \text{green})$

Спецификация некорректна:

(она требует, чтобы все время были все цвета – и зеленый, и желтый, и красный)

- $G(\text{green} \rightarrow X(\text{green} \cup (X(\text{yellow} \cup (X(\text{red} \cup (X(\text{red} \cup (X(\text{yellow} \cup (X(\text{yellow} \cup \text{green}))))))))))$

И этой спецификации недостаточно

Необходимо добавить: может быть активен ровно один цвет



Примеры проверяемых свойств (LTL)

- $GF \text{ DeviceEnabled}$

событие *DeviceEnabled* выполняется неопределенно часто

- $G[(p \rightarrow X(\neg q \cup r))]$

после того, как p выполнялся, q не выполнится, пока не станет активным r

- $G[p \rightarrow Xq \ \& \ XXq]$

после того, как сигнал p стал активным, со следующего такта сигнал q будет активным по крайней мере два такта

- $G(\text{updateA}_x \wedge F \text{readB}_x \rightarrow \neg \text{readB}_x \cup \text{flushA}_x)$

между моментом, когда процесс A изменяет значение переменной x (updateA_x), и моментом, когда B читает значение этой же переменной (readB_x), x должно быть вытолкнуто из кэша процессом A (flushA_x)

- $G(\text{req} \rightarrow X(\text{req} \cup \text{ack}))$

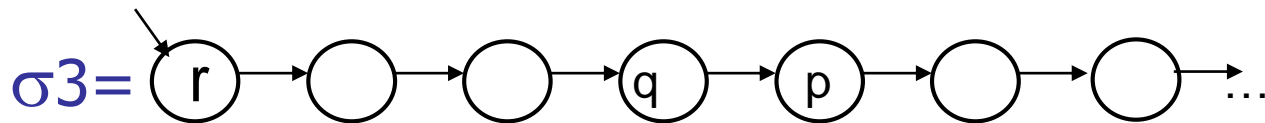
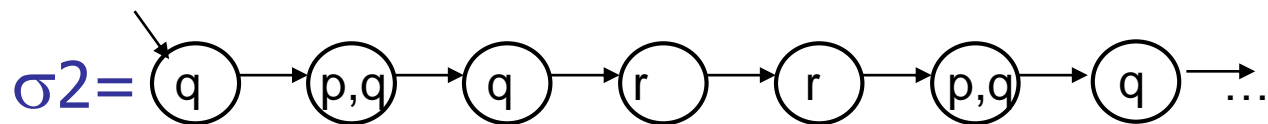
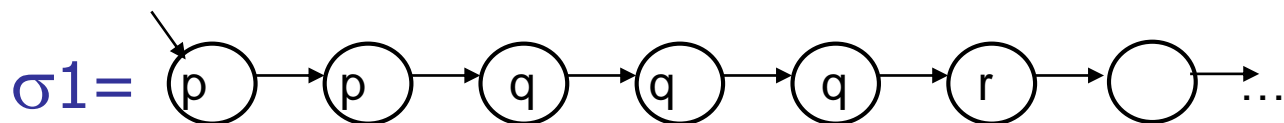
запрос не будет снят, пока на него не придет подтверждение

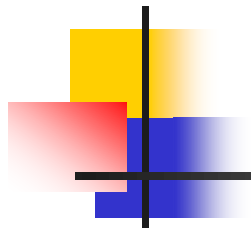
- $G[q \rightarrow X[(\neg p \cup r) \vee G \neg r]]$

между q и r событие p не выполняется

Пример экзаменационной задачи. Какие вычисления удовлетворяют данной спецификации?

- Спецификация: $p \cup (q \cup r)$



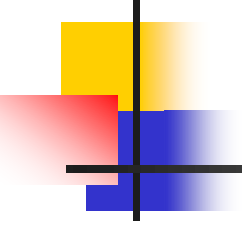


Примеры спецификации требований на CTL

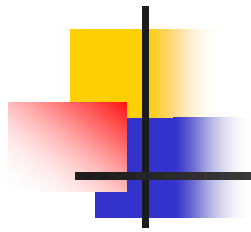


Примеры проверяемых свойств (CTL)

- $\text{at_start} \wedge \varphi \Rightarrow \text{AG}(\text{at_finish} \Rightarrow \psi)$
 - частичная корректность: если в начале программы (at_start) выполняется предусловие φ , то на любом вычислении (AG) если программа завершается (at_finish), то выполняется постусловие ψ
- **AGAF Restart**
 - при любом функционировании системы (на любом пути) из любого состояния системы всегда обязательно вернемся в состояние рестарта
- **AG EF Restart**
 - при любом функционировании системы (на любом пути) из любого состояния системы существует путь, по которому можно перейти в состояние рестарта
- $\text{E}[p \text{ U } \text{A}[q \text{ U } r]]$
 - существует путь, на котором p выполняется до тех пор, пока событие q станет выполняться до выполнения r на всех путях



ВЫВОД: спецификация требований
формулами темпоральной логики
непроста



Темпоральные операторы прошлого и LTL



Операторы прошлого и LTL

- LTL является стандартным общепризнанным языком выражения требований к дискретным событийным системам.
- В некоторых случаях для формулировки требований значительно проще **использовать темпоральные операторы прошлого**: формулы являются более короткими и более интуитивно понятными.
- Стандартный известный пример – спецификация причинной зависимости

“любое событие тревоги вызвано наступившим ранее отказом”

$G(\textit{alarm} \rightarrow P \textit{fault}),$

где P – оператор Past, *“случилось когда-то раньше”*.

Операторы прошлого не увеличивают мощности LTL,
т.е. любую формулу LTL^P (т.е. LTL с операторами прошлого) можно выразить формулой LTL только с операторами будущего.



Операторы прошлого. Формальная семантика

P – обратное F (F^{-1}),
 X^{-1} обратное X,
S – обратное U (Since)

$\sigma = S_0 S_1 S_2 \dots$; $\sigma_i \models \varphi$ означает: на вычислении σ_i истинно φ

$\sigma_i \models X^{-1} \phi \quad \underline{\text{iff}} \quad \sigma_{i-1} \models \phi \text{ и } i > 0$

$\sigma_i \models \phi \text{ S } \varphi \quad \underline{\text{iff}} \quad (\exists j: 0 \leq j \leq i) \sigma_j \models \varphi \text{ и } (\forall k: j < k \leq i) \sigma_k \models \phi$

Выводимые операторы P φ , H φ

P $\varphi = \text{True S } \varphi$

$\sigma_i \models P\varphi \quad \underline{\text{iff}} \quad (\exists k: 0 \leq k \leq i) \sigma_k \models \varphi$

H $\varphi = \neg (P \neg \varphi)$



Операторы прошлого

- Некоторые формулы LTL^P легко непосредственно переписать формулами LTL , например:

$$G(\textit{alarm} \rightarrow F^{-1} \textit{fault})$$

можно записать так:

$$G\neg\textit{alarm} \vee \neg\textit{alarm} \cup \textit{fault},$$

"либо сигнал отказа \textit{alarm} вообще никогда не случился на вычислении, либо его не было до тех пор, пока не выполнялся первый раз сигнал отказа \textit{fault} "

- Существует метод проверки модели формул LTL^P на основе метода проверки LTL
- Для этого используются **ИСТОРИЧЕСКИЕ ПЕРЕМЕННЫЕ** переменные

Исторические переменные: X^{-1}

Пример: “сигнал ‘alarm’ установится после того, как ‘crash’ появился в предыдущий момент времени”: $G(\text{alarm} \rightarrow X^{-1} \text{crash})$

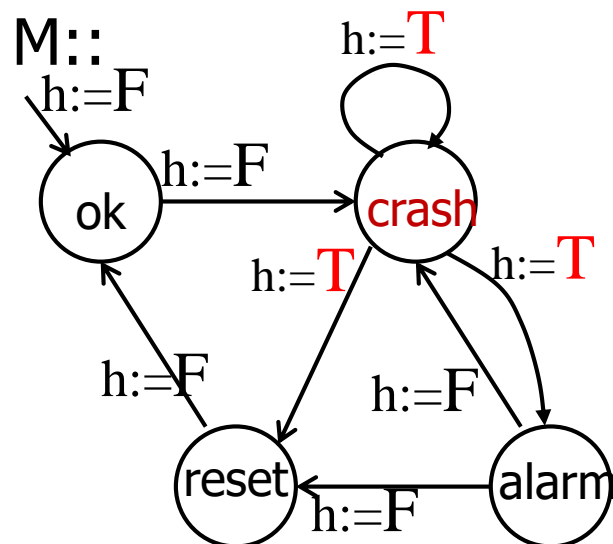
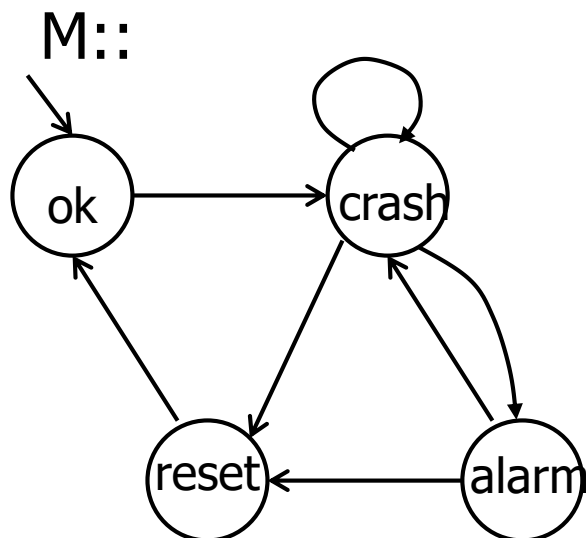
Ведем историческую переменную: $h \equiv X^{-1} \text{crash}$

Переформулируем требование $G(\text{alarm} \rightarrow h)$

На заданной структуре Крипке обозначим

$h = \text{True}$ на выходных ребрах из тех состояний, в которых выполняется *crash*

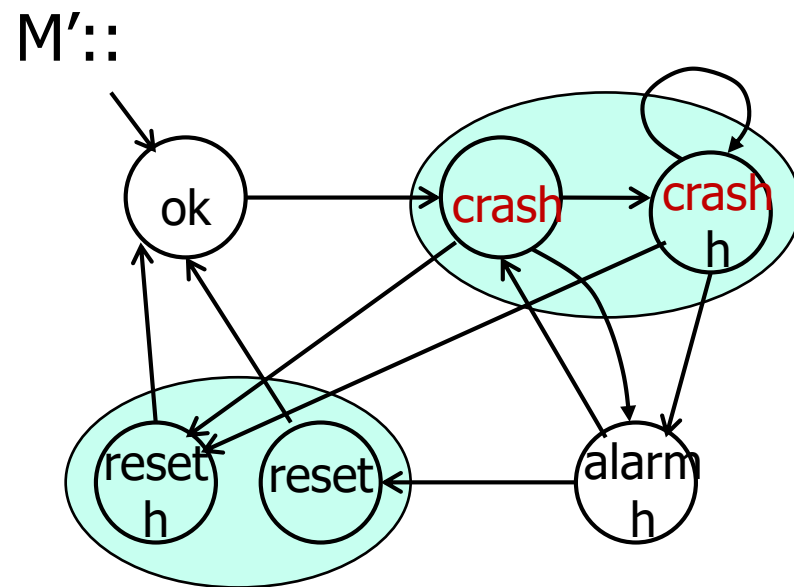
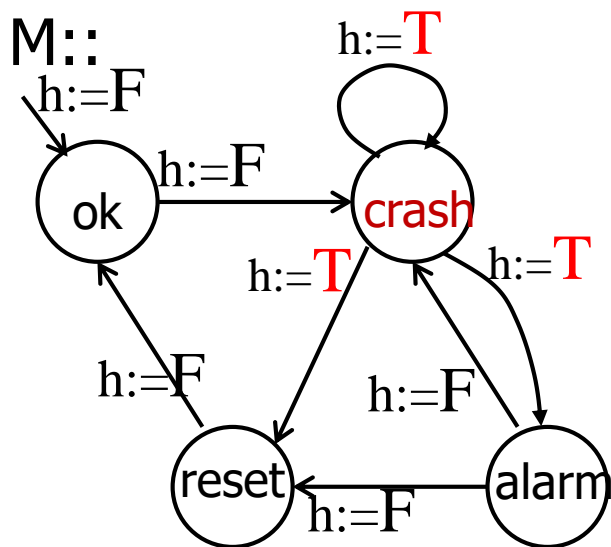
$h = \text{False}$ на выходных ребрах из тех состояний, в которых НЕ выполняется *crash*



Исторические переменные: X^{-1}

Дублируем состояния с разными h

(В структуре Крипке не могут быть состояния помечены одновременно и h , и $\neg h$)

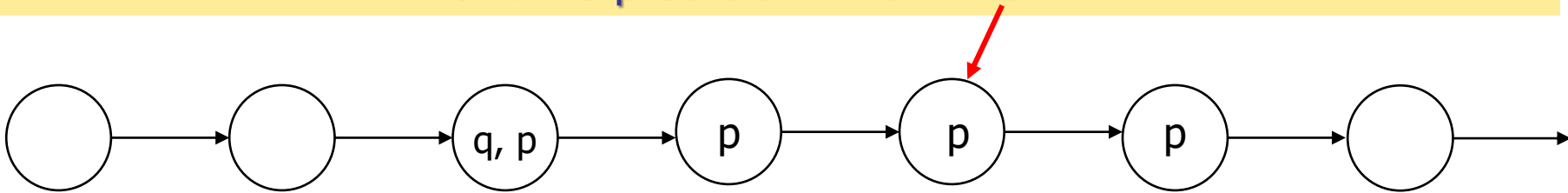


На развертке M' $h \equiv \text{True}$ в s , если в s выполняется оператор X^{-1} *crash*.

$G(\text{alarm} \rightarrow h)$ – достижимость!

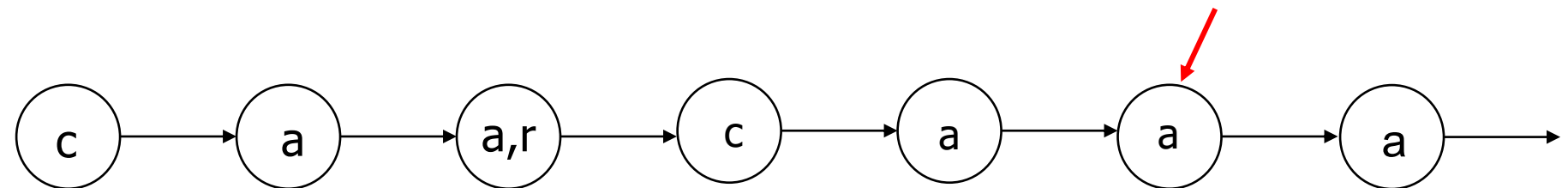
Исторические переменные (2) p Since q

p Since q – когда-то q стало истинным, и с тех пор (Since) до текущего момента p остается истинным



Пример: Если сигнал тревоги '*alarm*' установлен, то ранее была авария (сигнал '*crash*'), после которой НЕ БЫЛО сигнала '*reset*':

$G[alarm \rightarrow (\neg reset) S crash]$



По-другому: сигнал тревоги '*alarm*' еще не был обработан сигналом '*reset*' после последнего сигнала '*crash*' об аварии

Исторические переменные (2) p Since q

Пример: Если есть сигнал 'alarm', то был сигнал аварии 'crash', который НЕ был обработан (т.е. сброшен сигналом 'reset'):

$$G(alarm \Rightarrow (\neg reset) S crash)$$

Вводим историческую переменную $h \equiv (\neg reset) S crash$.

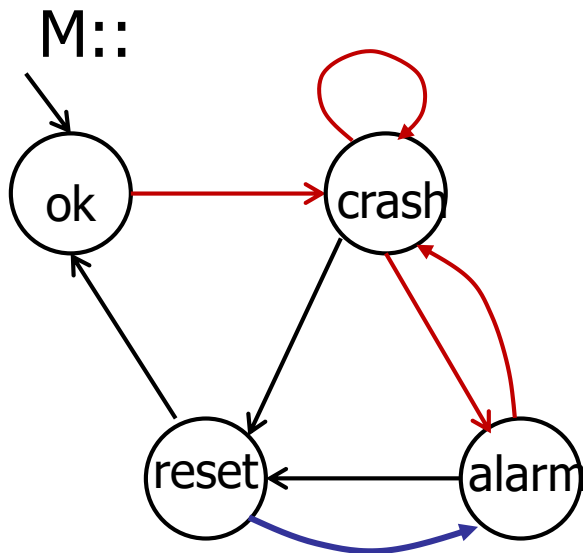
Меняем разметку структуры Крипке

На начальной стрелке $h:=F$ (раньше аварии не было)

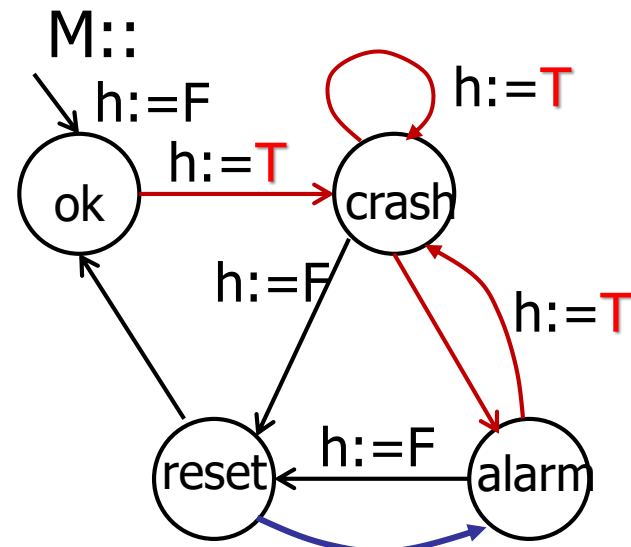
Далее, $h:=T$ на переходах, ведущих в состояние *crash*; (crash произошел)

$h:=F$ на переходах, ведущих в состояние $\neg(\neg reset)$, если оно без *crash*;

h не изменяется на переходах, ведущих в состояние $\neg reset$



Ю.Г.Карпов

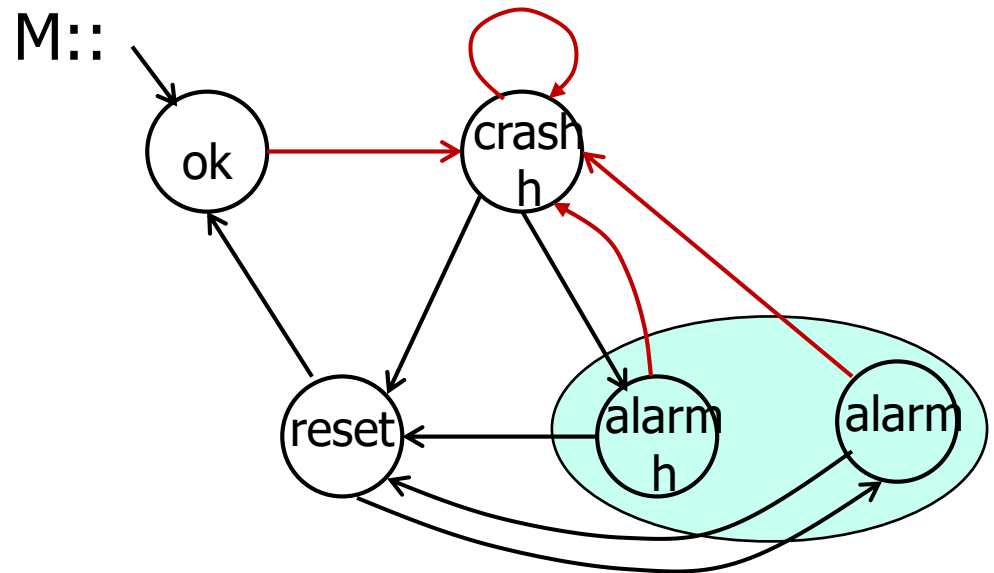
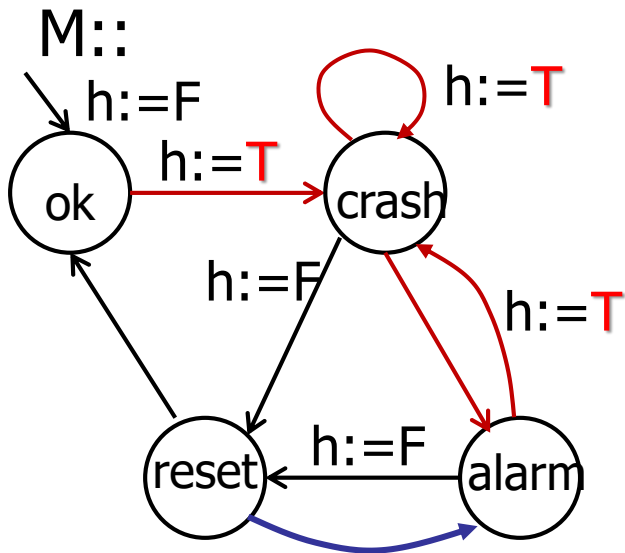


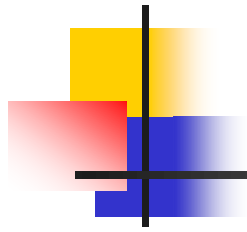
Верификация

Исторические переменные (2) ρ Since q

Дублируем состояния в структуре Крипке

LTL формула: $\mathbf{G}(alarm \Rightarrow h)$





Применение шаблонов спецификации требований



Шаблоны для спецификации специфических требований к программам

Dwyer M.B., Avrunin G.S., Corbett J.C. Property Specification Patterns for Finite-state Verification // Proceedings of the 2nd Workshop on Formal Methods in Software Practice. – 1998

- Предлагается система шаблонов, которая помогает практическому программисту точно специфицировать требования
- работа доступна в Интернете
- Система шаблонов состоит из **Типа** и **Области действия** требований
- Эта работа дает спецификацию на LTL и CTL каждого типа требования для каждой области действия.

Спецификация формулами LTL не очень проста

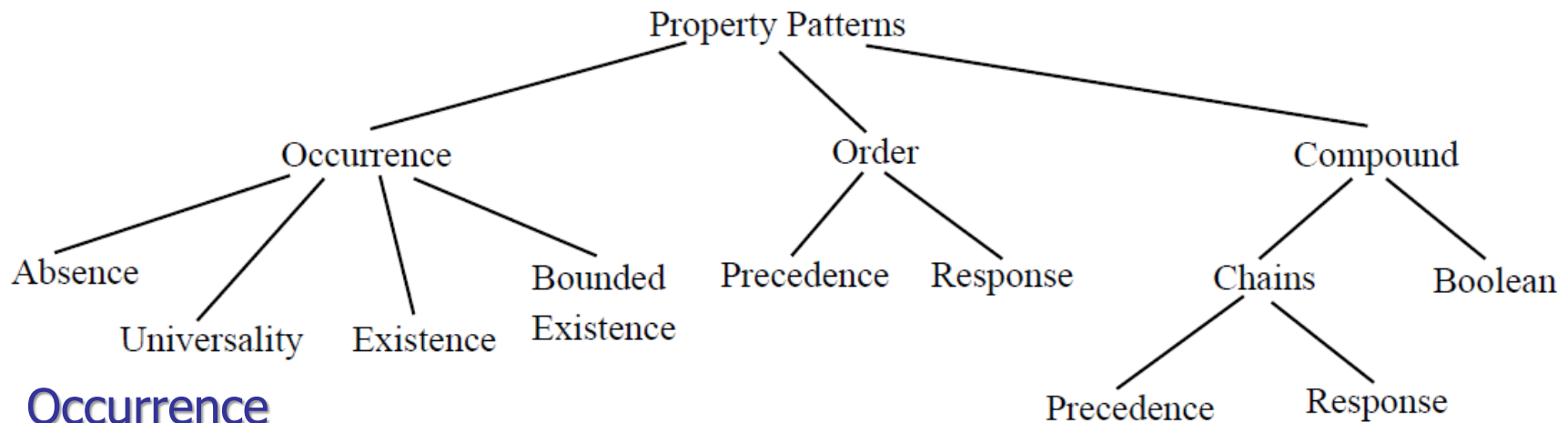
- Пример требования из (*)
- **Требование:** *"Между моментом, когда лифт вызван на этаж, и моментом, когда лифт открыл дверь на этом этаже, лифт мог прибыть на этот этаж самое большее два раза"*
- Спецификация на LTL:

$$\begin{aligned} G \big((call \wedge F (at_floor \wedge open)) \rightarrow (\neg at_floor \bigcup \\ (at_floor \wedge open \vee (at_floor \wedge \neg open) \bigcup \\ (open \vee \neg at_floor \bigcup \\ (at_floor \wedge open \vee (at_floor \wedge \neg open) \bigcup \\ (open \vee (\neg at_floor \bigcup (at_floor \wedge open)))))) \big) \big) \end{aligned}$$

Такие формулы не только трудны для понимания и написания, их трудно построить корректно из спецификации требования на естественном языке без "некоторого опыта"()*

* Dwyer M.B., Avrunin G.S., Corbett J.C. Property Specification Patterns for Finite-state Verification // Proc. 2nd Workshop on Formal Methods in Soft. Practice. – 1998

Структурирование шаблонов – типы требований



■ Occurrence

- **Absence** Событие не встречается
- **Universality** Событие встречается
- **Existence** Событие должно наступить когда-нибудь
- **Bounded Existence** Событие должно наступить k раз (at least k; at most k)

■ Ordering

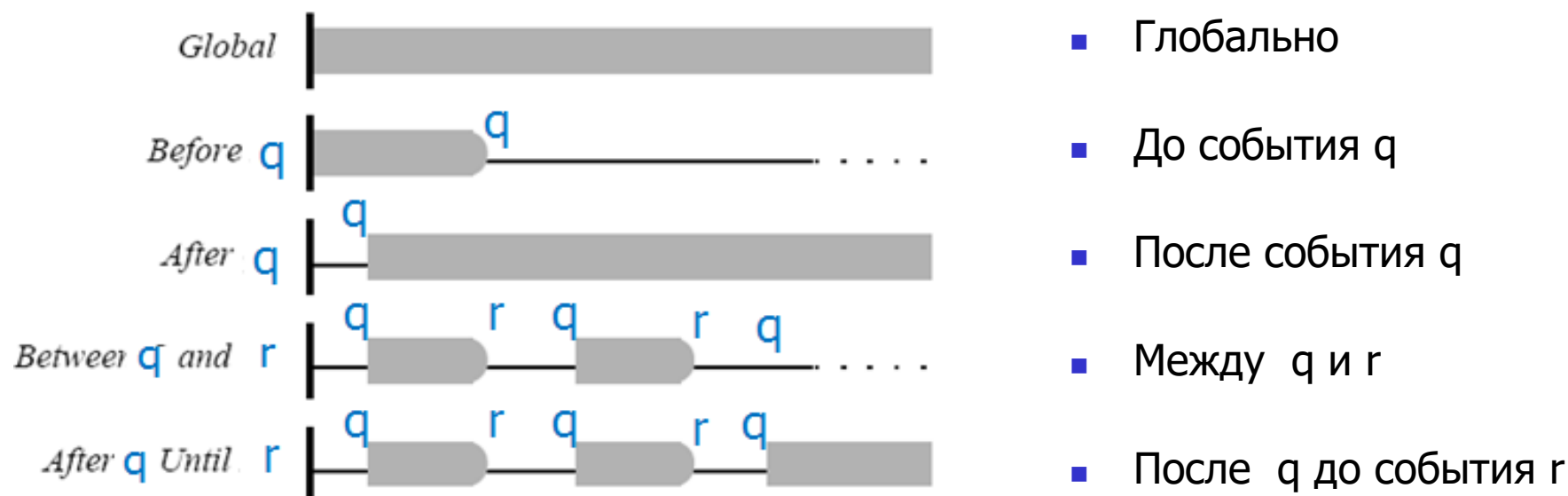
- **Precedence** Событие p всегда предшествует событию q
- **Response** Событие p всегда следует за событием q (причинная связь)

■ Compound

- **Chain Precedence** Одна цепочка событий всегда предшествует другой
- **Chain Response** Одна цепочка событий всегда следует за другой

Области действия выполнения свойства p в шаблоне

Свойство p должно выполняться:

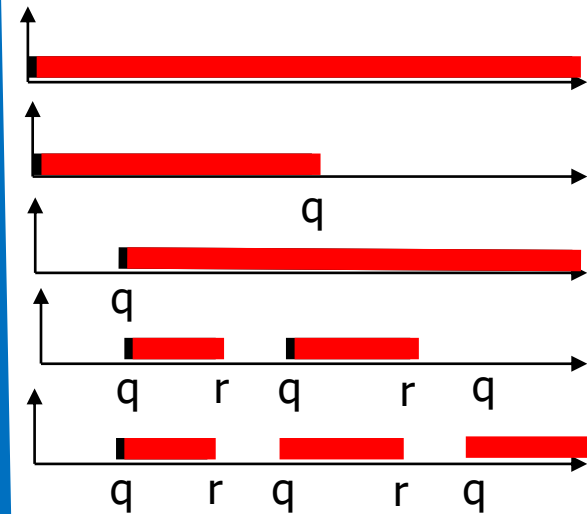


- В работе представлены формулы для всех типов шаблонов и всех областей действия как на LTL, так и на CTL

Шаблон "отсутствие": событие p не выполняется (Absence)

LTL

- Глобально $G \neg p$
- До q $Fq \Rightarrow \neg p \text{ U } q$
- После q $G (q \Rightarrow G \neg p)$
- Между q и r $G ((q \wedge XFr) \Rightarrow (\neg p \wedge X(\neg p \text{ U } r)))$
- После q до r $G(q \Rightarrow (\neg p \wedge X(\neg p \text{ U } (r \vee G(\neg r \wedge \neg p))))))$



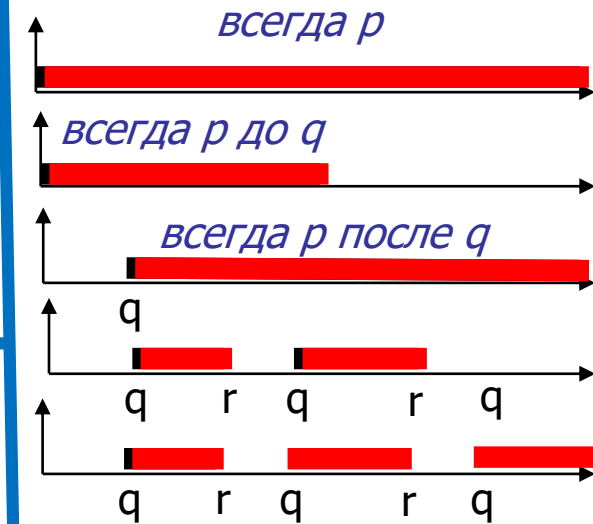
CTL

- Глобально $AG \neg p$
- До q $A(\neg p \text{ U } (q \vee AG \neg q))$
- После q $AG (q \Rightarrow AG \neg p)$
- Между q и r $AG (q \Rightarrow A(\neg p \text{ U } (r \vee AG \neg r)))$
- После q до r $AG (q \Rightarrow \neg E (\neg r \text{ U } (p \wedge \neg r)))$

Требование всеобщности выполнения события p (Universality)

LTL

- Глобально $G p$
- До q $Fq \Rightarrow p U q$
- После q $G(q \Rightarrow Gp)$
- Между q и r $G((q \wedge XFr) \Rightarrow (p \wedge X(p U r)))$
- После q до r $G(q \Rightarrow (p \wedge X(p U (r \vee G(\neg r \wedge p)))))$



CTL

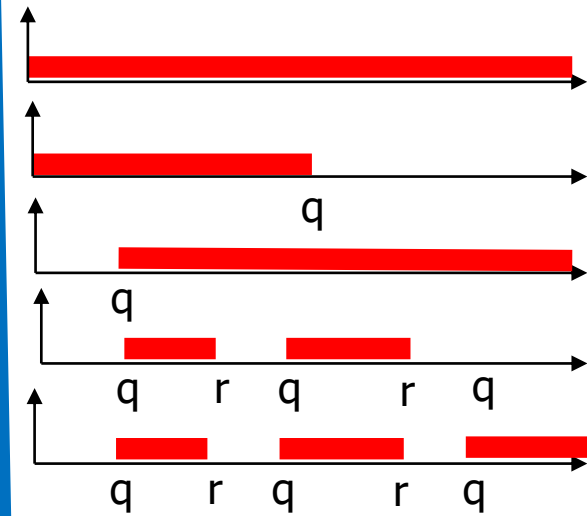
- Глобально $AG p$
- До q $A[(p \vee AG \neg q) W q]$
- После q $AG(q \Rightarrow AGp)$
- Между q и r $AG(q \wedge \neg r \Rightarrow A(p \vee AG \neg r) W r)$
- После q до r $AG(q \wedge \neg r \Rightarrow A(p W r))$

Шаблон "существование": событие p должно выполниться, по крайней мере, один раз (Existence)

LTL

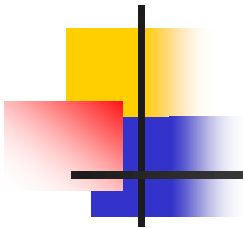
- Глобально Fp
- До q $\neg q \ W \ (p \wedge \neg q)$
- После q $G \neg q \vee \neg q \ U \ (q \wedge XF \ p)$
- Между q и r $G (q \wedge \neg r \Rightarrow (\neg r \ W \ (p \wedge \neg r)))$
- После q до r $G (q \wedge \neg r \Rightarrow (\neg r \ U \ (p \wedge \neg r)))$

Выполнение p:



CTL

- Глобально AFp
- До q $A(\neg q \ W \ (p \wedge \neg q))$
- После q $A(\neg q \ W \ (q \wedge AF \ p))$
- Между q и r $AG (q \wedge \neg r \Rightarrow A (\neg r \ W \ (p \wedge \neg r)))$
- После q до r $AG (q \wedge \neg r \Rightarrow A (\neg r \ U \ (p \wedge \neg r)))$



Формулировка **ОБЩИХ** требований к программным системам логического управления, отражающих их **ПРИРОДУ** как систем **ПАРАЛЛЕЛЬНЫХ** **ВЗАИМОДЕЙСТВУЮЩИХ ПРОЦЕССОВ**

Важность выделения **классов свойств** в том, что при проектировании системы следует проверять кроме специфичных, еще и общие свойства, позволяющие проверить отсутствие **типичных некорректностей параллельных систем**



Классы свойств реагирующих систем

- Достижимость (*reachability*) – конкретное состояние может быть достигнуто
- Безопасность (*safety*) - *нечто плохое никогда не произойдет*
- Свобода от дедлоков (*deadlock freeness*) – проверка блокировок
- Живость, живучесть (*liveness*) - *нечто хорошее обязательно произойдет*
- Справедливость (*fairness*) – дополнительные ограничения

Эти классы построены **не из назначения систем!**

Они отражают требования, которые являются обязательными для параллельно функционирующих взаимодействующих процессов произвольной природы и назначения и отражают сложность МС

Достижимость (*reachability*)

Некоторая ситуация может быть достигнута при функционировании

Свойство достижимости – это свойство, указанное в некотором состоянии, его выражает формула $EF\phi$, где ϕ не содержит темпоральных операторов

- R1: параметр N станет отрицательным
- R2: система войдет в критическую секцию

- R1: $EF N < 0$
- R2: $EF @crit_section$



Безопасность (*safety*)

Nothing bad happens

Достижимость
Безопасность
Дедлоки
Живость
Справедливость

При некоторых условиях некоторая ситуация никогда не может быть достигнута (гарантия того, что *нечто плохое никогда не произойдет*)

Свойство безопасности выражается формулой: $AG \neg \phi$

Безопасность с дополнительным условием:

$A(\neg \phi \ W \ \psi)$

событие ψ произойдет, только после того, как не произошло ϕ

Weak Until: $\phi W \psi = \phi U \psi \vee G \neg \psi$ (ψ может вообще не произойти)

Пример. “*Два процесса никогда не войдут в свои критические секции одновременно*”:

$AG \neg (@crint1 \wedge @crint2)$

ИНВАРИАНТ – это частный случай свойства безопасности - выполнение некоторого условия в каждом достижимом состоянии.

Свойства безопасности всегда проверяемы на **конечных** траекториях (поскольку нам просто нужно перебрать все состояния)



Свобода от блокировок (дедлоков)

Достижимость
Безопасность
Дедлоки
Живость
Справедливость

Взаимная блокировка процессов (deadlock) – состояние в структуре Крипке, из которого нет выхода.

Какое бы состояние ни было достигнуто существует какой-нибудь непосредственный преемник этого состояния

Если состояния дедлока опишем явно, то **отсутствие** дедлока можем свести к проверке БЕЗОПАСНОСТИ (safety)

Нечто хорошее когда-нибудь случится (проверка свойства траекторий)
(*нечто хорошее обязательно произойдет в будущем*)

Живость характеризует прогресс в нужном направлении

1 Любой запрос в конце концов будет обслужен

$AG(req \rightarrow AFresp)$ в CTL, $G(req \rightarrow Fresp)$ в LTL

2 Если долго стараться, то в конце концов получится

3 Если лифт вызван, он обязательно, рано или поздно, придет

4 Светофор, в конце концов, переключится на зеленый

5 Солнце будет выглядывать (бесконечно) часто: **GF солнце (LTL)**

6 Система всегда может вернуться в свое начальное состояние

$AG EF @init$ в CTL (в LTL формулы для выражения этого нет)



Ни свойства живости, ни свойства безопасности в LTL

Достижимость
Безопасность
Дедлоки
Живость
Справедливость

Свойства живости и свойства безопасности выделены, так как **алгоритм проверки модели** для этих свойств может быть разным.

Свойства живости и свойства безопасности отличаются по контрпримерам, их опровергающим

Свойства безопасности: у любого контрпримера существует конечный префикс, после которого какие состояния не добавляй, ситуация не улучшится. Иначе любое слово с этим конечным префиксом будет нарушать свойство безопасности.

Свойства живости: не существует никакого конечного префикса, который мог бы его опровергнуть. Его опровергают бесконечные контрпримеры.

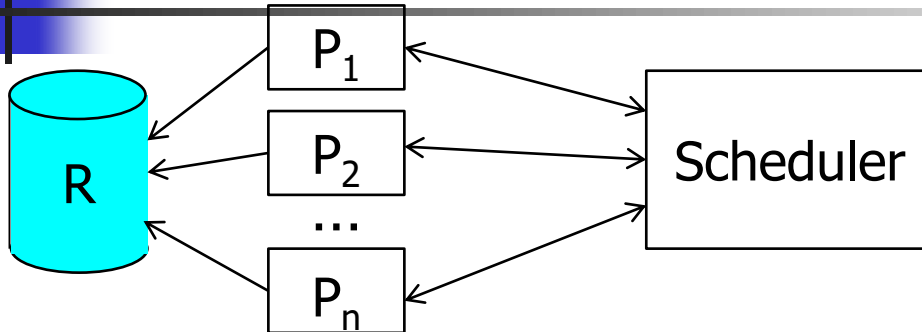
Свойства, не являющиеся ни свойством живости, ни свойством безопасности: демонстрируют характеристики как свойств живости, так и безопасности. Например, **$a \cup b$** .

Для свойств безопасности существует более эффективный алгоритм проверки модели. Алгоритм для свойств живости проверяются все типы свойств.

Поэтому свойства ни живости-ни безопасности на практике часто называют свойствами живости

Условия справедливости (fairness)

Достижимость
Безопасность
Дедлоки
Живость
Справедливость



На практике справедливость
может обеспечить
ПЛАНИРОВЩИК

■ Безусловная справедливость

GF *succeed* *бесконечно часто будет произведен ход*

■ Сильная справедливость (strong fairness) – запрос периодичен:

GF *attempt* \Rightarrow GF *succeed*

"Если ход может быть произведен ПЕРИОДИЧЕСКИ, то он производится неопределенно часто"

■ Слабые формы справедливости (weak) – запрос постоянен:

FG *attempt* \Rightarrow GF *succeed*

*"Если процесс готов выполнить ход **ПОСТОЯННО**, то он будет выполнен неопределенно часто"*
"Если будем у окошка стоять бесконечно долго, то получим неопределенно много справок"

G *attempt* \Rightarrow F *succeed*

*"Если ресурс запрашивается **ПОСТОЯННО**, то он когда-нибудь будет выделен"* или:

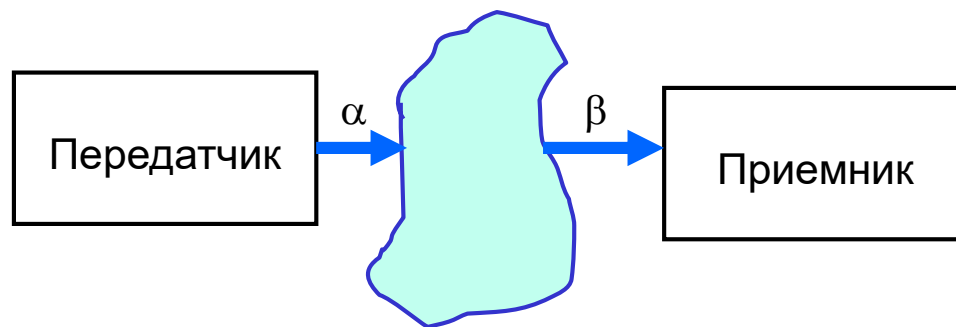
"Если будем у окошка стоять бесконечно долго, то справку получим"

Справедливость (fairness): ограничения модели реалистичными траекториями

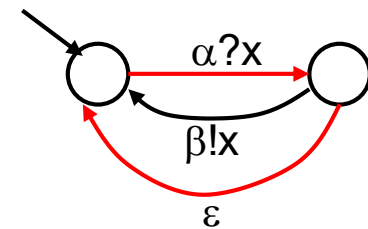
Достижимость
Безопасность
Дедлоки
Живость
Справедливость

В модели могут появиться такие вычисления, которые не случаются в реальной системе

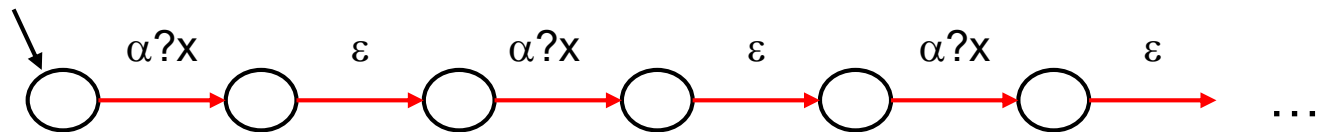
Пример: В модели канала с потерями мы абстрагируемся от конечного значения вероятности потерь, поэтому в МОДЕЛИ может появиться вычисление, на котором потери сообщения в канале бесконечны



Модель канала с потерями



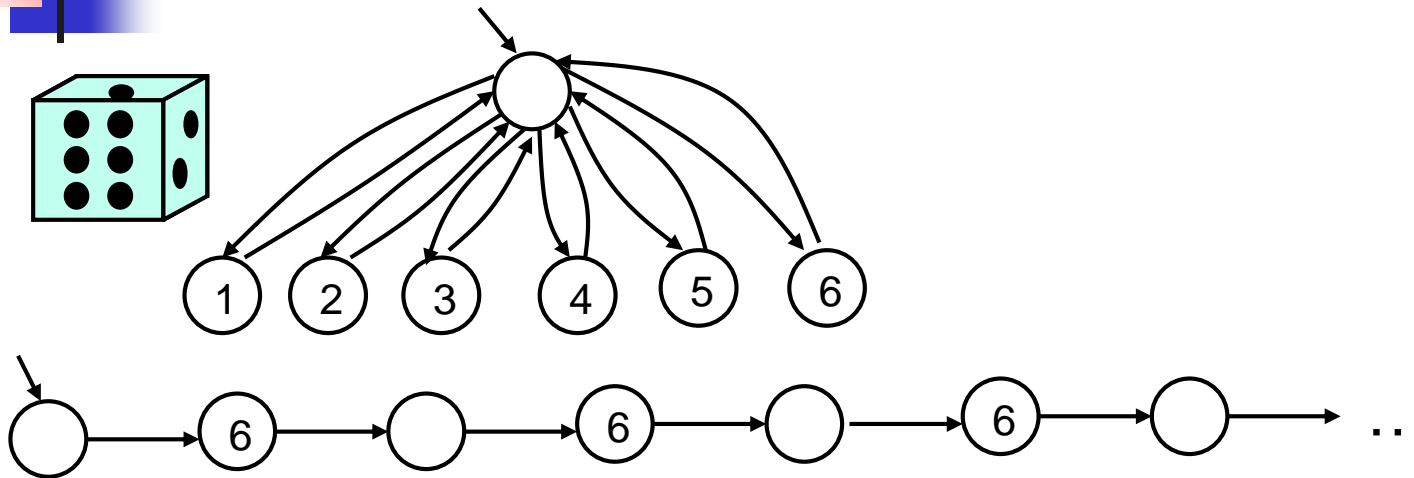
Такое вычисление
в модели возможно



В модели (в структуре Крипке) не можем выразить свойства: цепочка, состоящая из произвольного конечного числа потерь, возможна, а состоящая ТОЛЬКО из потерь, невозможна (на практике она имеет нулевую вероятность). Но проверку свойств на модели мы ДОЛЖНЫ ограничить только реалистичными траекториями. КАК?

Модель бросания игральной кости

Достижимость
Безопасность
Дедлоки
Живость
Справедливость



В этой модели возможны нереалистичные вычисления, например, такие, на которых некоторые грани никогда не выпадают. На реальных вычислениях каждая грань выпадает неопределенно часто (мы абстрагируемся от значений вероятности выпадения граней).

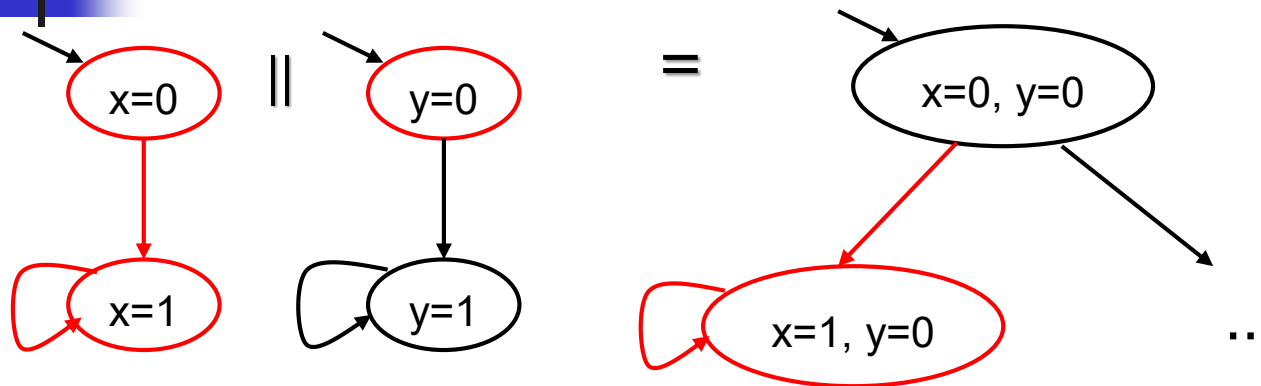
НЕ УЧИТЫВАЮТСЯ ВЕРОЯТНОСТИ ВЫПАДЕНИЯ ГРАНЕЙ

Нужно ограничить анализ модели только реалистичными траекториями. КАК?

Нереалистичные траектории в подобных случаях тоже называются несправедливыми, хотя к “справедливости” это не имеет никакого отношения

Несправедливые вычисления в модели интерливинга

Достижимость
Безопасность
Дедлоки
Живость
Справедливость

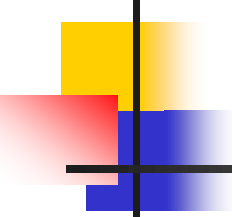


Второй процесс может сделать ход,
но ему не дают права хода

Модель чередования функционирования **нескольких** асинхронных параллельных процессов (interleaving): в каждом глобальном состоянии системы возможен шаг ЛЮБОГО готового процесса.

Несправедливым вычислением является такое, в котором один из процессов не предоставляется право сделать ни одного шага

КАК БЫТЬ?



Несправедливые вычисления при верификации мешают проверке некоторых свойств

- Как ограничить проверку свойств системы только справедливыми траекториями?



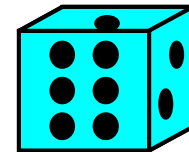
Учет свойств справедливости в LTL

- Нужно использовать:

для условия справедливости формулу:

<условие справедливости> \Rightarrow <проверяемое свойство>

если условие выполняется, то свойство будет выполнено



Остаемся в рамках той же модели, но вводим ограничение:
те вычисления, которые являются реалистичными, назовем FAIR
“Справедливыми”, и попытаемся при анализе рассматривать только их

Например, для системы, в которой есть процесс бросания игровой кости. Проверяемое свойство с учетом требования справедливости:

$GF1 \wedge GF2 \wedge GF3 \wedge GF4 \wedge GF5 \wedge GF6 \Rightarrow \langle \text{проверяемое свойство} \rangle$

каждая грань выпадает неопределенно часто –

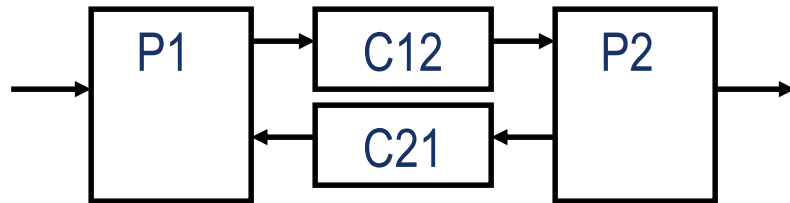
ИМЕННО ЭТО выполняется в реальности

(хотя в реальности выполняется более сильное требование:
вероятности встречаемости каждой грани на любом вычислении равны!)

Это **ДОПОЛНИТЕЛЬНОЕ** требование. Только если на вычислении выполняется это требование, мы будем вычисление анализировать на предмет выполнения свойств верификации.

Введение условия (требования) справедливости дает возможность отбросить
НЕРЕАЛИСТИЧНЫЕ траектории, являющиеся результатом использования
упрощенной модели, представляющей реальный процесс

Пример формулировки свойств: АВ протокол



Функциональное требование: Цепочка сообщений на выходе составляет начальный отрезок посланных

Дополнительные требования:

Safety: Любое полученное на выходе сообщение было послано передатчиком

Liveness: Любое посланное сообщение будет в конце концов получено

$G(\text{emitted } m \Rightarrow F \text{ received } m)$

Это свойство на модели системы не выполняется, если модель допускает произвольное количество потерь в канале. Обычно для проверки живучести требуется ограничение справедливости. Здесь –

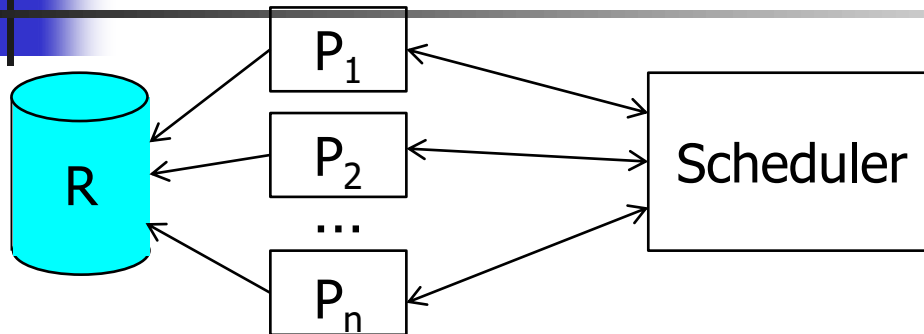
предположение об отсутствии постоянных потерь:

Fairness: $GF\text{--}loss \Rightarrow G(\text{emitted } m \Rightarrow F \text{ received } m)$

Рассматриваем свойства только на траекториях, на которых выполнено $GF\text{--}loss$

Условия справедливости (fairness)

Достижимость
Безопасность
Дедлоки
Живость
Справедливость



Справедливость может обеспечить ПЛАНИРОВЩИК

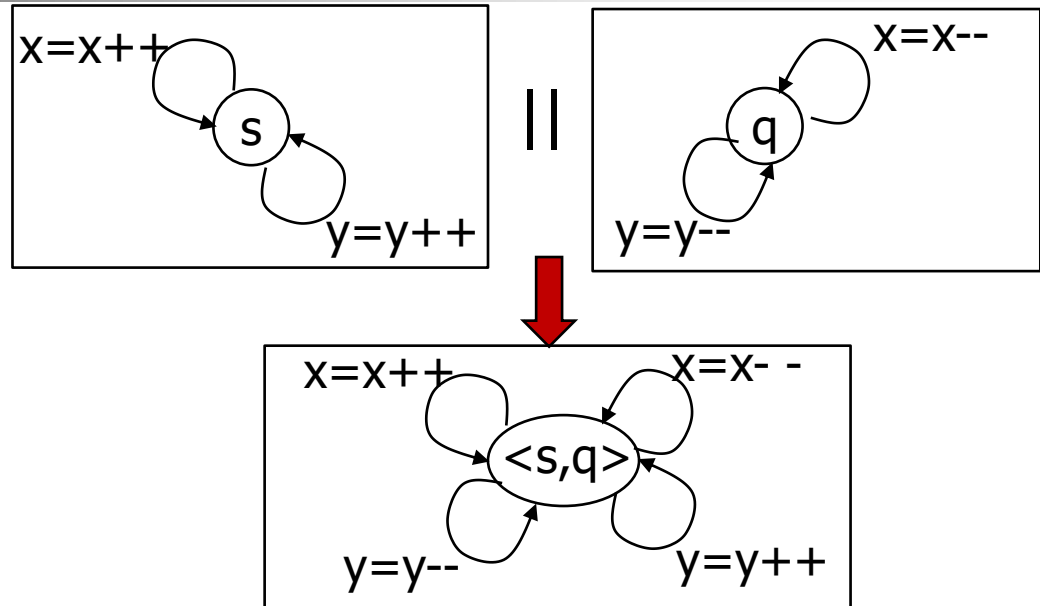
- Часто условие справедливости затрагивает **ВСЕ состояния всех процессов**.
- **Слабая справедливости (weak):**
 $FG \text{ attempt} \Rightarrow GF \text{ succeed}$ (формулу надо писать для всех состояний!)
*"Если переход из состояния возможен бесконечно долго (**постоянно**), то переход будет выделен неопределенно часто"*
- Очевидно, что одну формулу в этом случае записать нельзя.

Поддержка слабой справедливости в Spin'e

```
byte x=0, y=0;

active proctype A()
{
    do
        :: (1) -> x++;
        :: (1) -> y++;
    od
}

active proctype B()
{
    do
        :: (1) -> x--;
        :: (1) -> y--;
    od
}
```



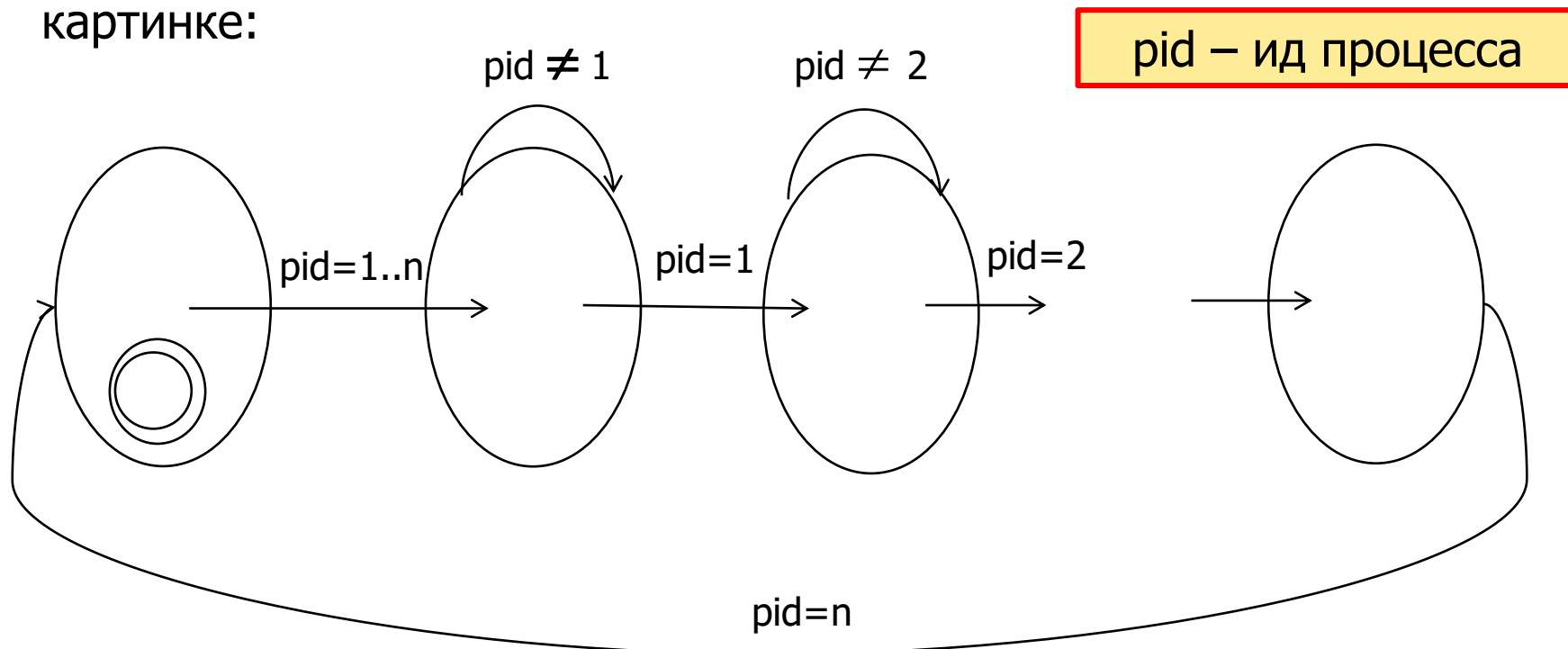
Spin поддерживает один из вариантов слабой справедливости процессов: если процесс содержит оператор, который **постоянно готов к выполнению**, то этот оператор рано или поздно выполнится

- При установленном ключе `-f` Spin отбросит те траектории, в которых присутствуют **только** операторы `x++`, `y++`, `x--`, `y--`.

Spin будет анализировать только те траектории, в которых есть операторы и инкремента, и декремента. Чтобы убрать другие некорректные траектории, например, работающие только с `x` или только с `y`, нужны специальные условия

Гипотезы справедливости: слабая справедливость в SPIN (алгоритм Боснацки)

- Строится $(n+1)$ копий параллельной композиции процессов
- Только в нулевой копии сохраняются допускающие состояния, соответствующие $\langle \text{проверяемому свойству} \rangle$
- Далее в копиях перенаправляются дуги по приведенной ниже картинке:



- Свойство проверяется уже на такой системе
- Это увеличивает сложность проверки в $(n+1)$ раз

Упражнение – как Spin поддерживает требование справедливости

- Проверьте выполнимость в Spin'e:

```
...
active proctype P ( ) {
  do
  :: (x<3)  -> x++
  :: (x==3) -> L0: x=0
  :: (x>0)  -> L1: y=x
  od
}

active proctype Q ( ) { ... }
...
```

Свойства:

$\varphi 1. [] \leftrightarrow P@L0$

$\varphi 2. \leftrightarrow P@L1$

$\varphi 3. \leftrightarrow [] (x==3) \rightarrow [] \leftrightarrow P@L0$

$\varphi 4. [] \leftrightarrow x>0 \rightarrow [] \leftrightarrow P@L1$

$\varphi 1$ не выполняется: как только $x>0$, оператор 3 всегда может быть выбран

$\varphi 2$ не выполняется: бесконечно могут выполняться 1 и 2 операторы

$\varphi 2$ выполняется: бесконечно могут выполняться 1 и 2 операторы

$\varphi 4$ не выполняется: 1-й оператор может выполняться при $x>0$

Для системы процессов выполняется только $\varphi 3$



Условия справедливости и CTL



Как ввести требование справедливости в CTL?

Как при проверке модели в CTL отбросить те пути, которых не может быть в реальных вычислениях?

В CTL нельзя записать формулы справедливости:
 $\Phi = A (GF\varphi \Rightarrow \text{“проверяемое свойство”})$

Условия справедливости выразить в CTL невозможно

Для проверки свойств с условием справедливости разработана fair CTL



Справедливая CTL

Явно вводится свойство “справедливости” путей, и проверка проводится только на справедливых путях

Зададим конечный список F подмножеств состояний структуры Крипке вида: $\{F_i \mid i=1, \dots, k\}$, $F_i \subseteq S$; (a set of fairness conditions)

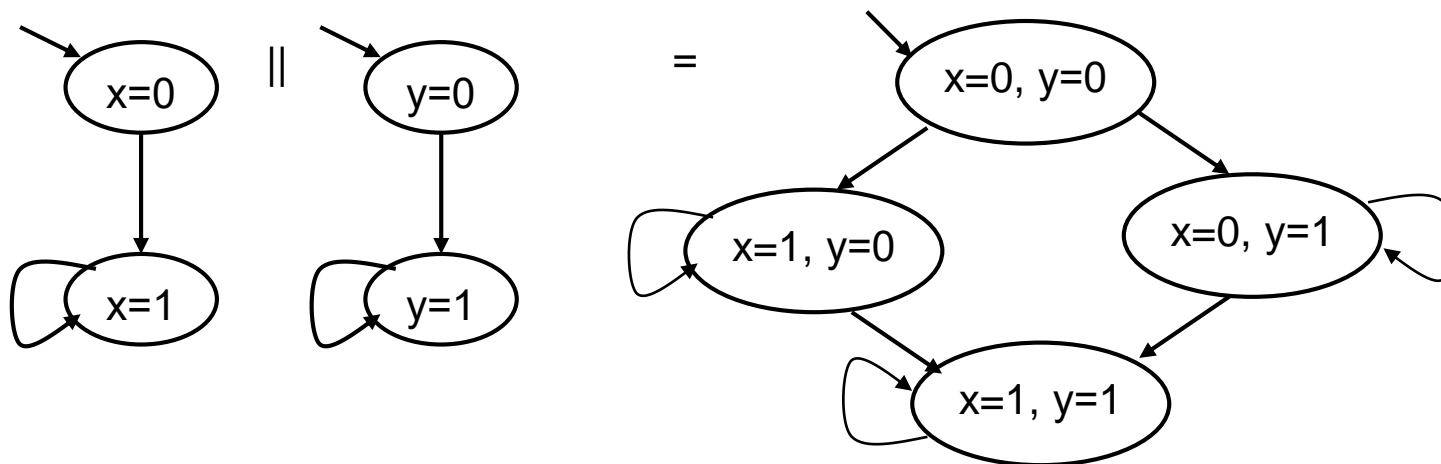
Будем считать, что “справедливое” вычисление π - это такое вычисление, которое посещает каждое из множеств состояний F_i бесконечное число раз (на справедливом пути хотя бы одно состояние из каждого F_i должно встречаться бесконечно много раз)

Условие *fairness* вычисления можно формализовать так:

Путь π *справедливый* iff $\forall i \in \{1, \dots, k\} \quad \text{inf}(\pi) \cap F_i \neq \emptyset$

$\text{inf}(\pi)$ – множество состояний, которые встречаются бесконечное число раз на π

Пример с параллельными процессами - CTL



На вычислении, в котором выполняется операция только одного процесса, система застревает в состоянии $\langle x=1, y=0 \rangle$ или $\langle x=0, y=1 \rangle$

Условие справедливости в этом примере формулируется как два множества состояний $\{F1, F2\}$:

$F1 = \{\text{состояния, на которых удовлетворяется } x=1\}$, и

$F2 = \{\text{состояния, на которых удовлетворяется } y=1\}$

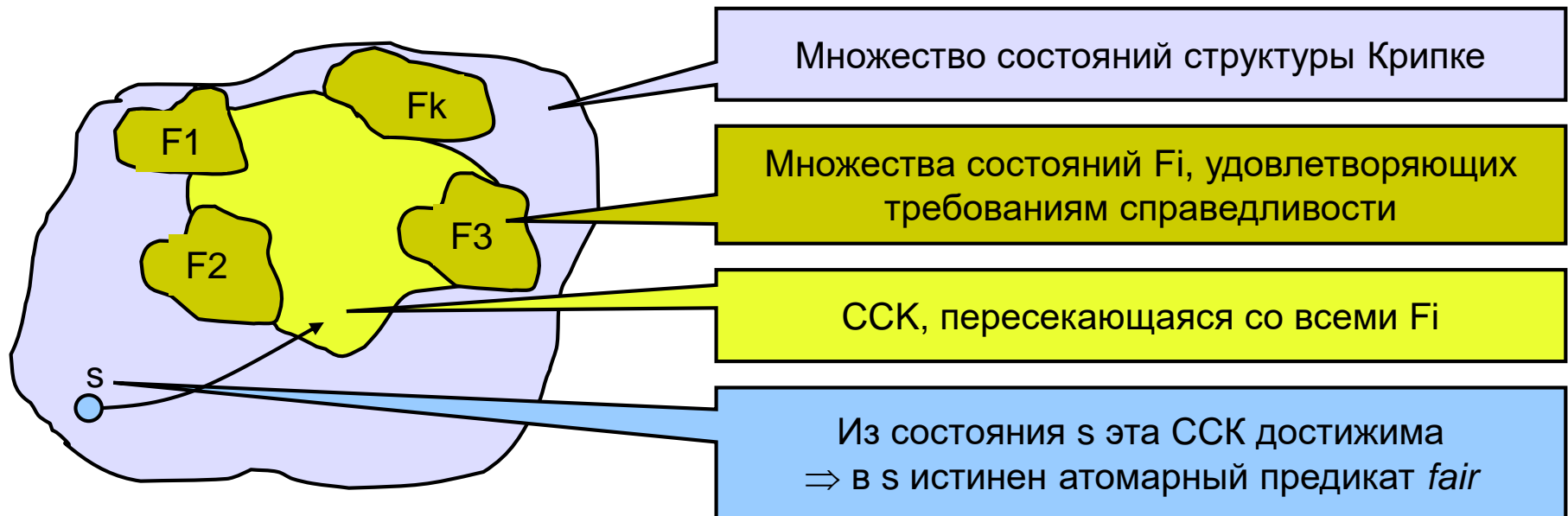
Состояния из множеств $F1$ и $F2$ должны неопределенно часто встретиться в каждой траектории модели, на которой мы хотим проверять СВОЙСТВА СИСТЕМЫ

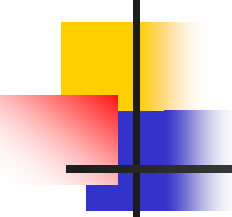
Формализация требований справедливости

Введем ДОПОЛНИТЕЛЬНЫЙ атомарный предикат *fair* – ему удовлетворяют все состояния, из которых *существует* вычисление, проходящее через каждое из множеств F_1, F_2, \dots, F_k бесконечное число раз

В состоянии s новый атомарный предикат *fair* истинен *тогда*, когда в структуре Крипке существует достижимая из s ССК, которая пересекается со всеми множествами F_1, F_2, \dots, F_k (фактически, выполняется $s \models_{\text{fair}} EG \text{ true}$)

Вводится справедливая структура Крипке: $M_F = (S, S_0, AP, R, L, F)$, структура, в которой достижимы ССК, проходящие через F





Проверка свойств CTL с требованием справедливости (базис EX, EU, EG)

Верификацию с условием справедливости можно выполнить так:

1. Находим все состояния s_{fair} , в которых удовлетворяется предикат *fair*
2. По уже обработанным подформулам формулы ϕ вычисляем $s \models_{\text{fair}} \phi$ так:

$$s \models_{\text{fair}} p \quad \equiv \quad s \models (\text{fair} \wedge p)$$

$$s \models_{\text{fair}} EX \phi \quad \equiv \quad s \models EX (\text{fair} \wedge \phi)$$

$$s \models_{\text{fair}} E (\phi \cup \psi) \quad \equiv \quad s \models (E (\phi \cup (\text{fair} \wedge \psi)))$$

$$s \models_{\text{fair}} EG\psi$$

$s \models_{\text{fair}} EX \phi$ - из s существует путь, такой, что в его следующем состоянии выполняется $\text{fair} \wedge \phi$

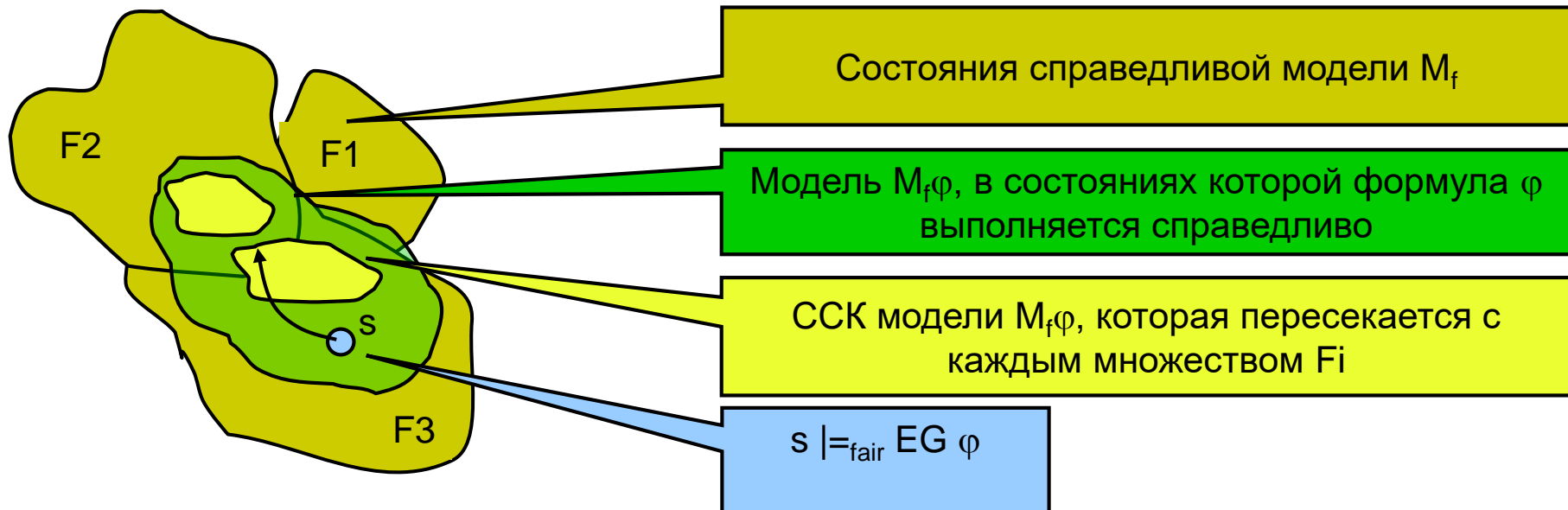
$s \models_{\text{fair}} E (\phi \cup \psi)$ - из s существует путь, помеченный ϕ в состоянии, помеченное $\text{fair} \wedge \psi$

Вычисление $s \models_{\text{fair}} EG\psi$ требует чуть большего внимания

$$s \models_{\text{fair}} EG \varphi$$

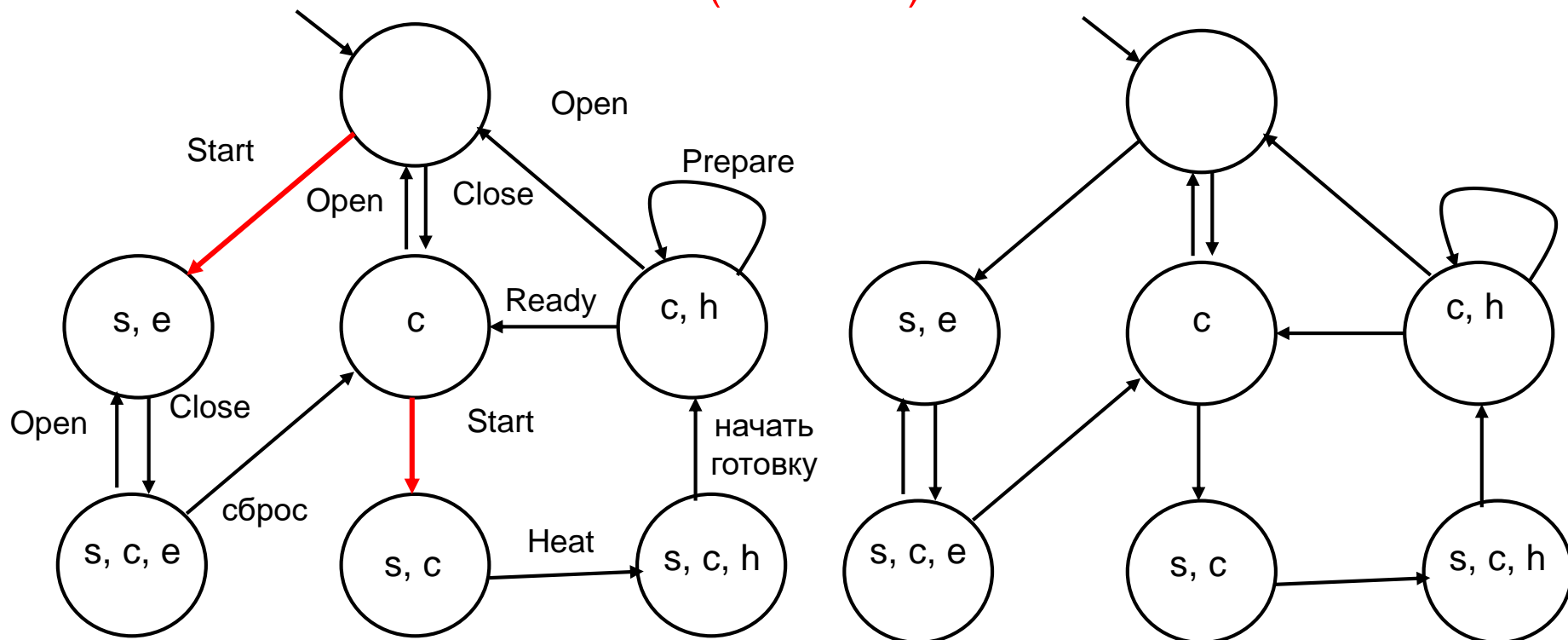
Проверка подформулы $EG\varphi$ с требованием справедливости

1. Ограничиваем структуру Крипке до модели M_f , в которой нет состояний, не относящихся к множествам F_1, \dots, F_k (строим “справедливую” модель)
2. Ограничиваем *справедливую* модель M_f такой $M_{f\varphi}$, во всех состояниях которой формула φ выполняется
3. Находим $\{C_1, C_2, \dots, C_m\}$ – множество таких ССК, которые пересекаются с КАЖДЫМ ограничением справедливости F_i .
4. Формула $EG\varphi$ *справедливо* выполняется в состоянии s *тогда*, когда из s достижима одна из сильно связанных компонент C_i



Пример: Микроволновая печь

$$\Phi = AG (s \Rightarrow AF h)$$



Атомарные предикаты:

Start – s

Close – c

Heat - h

Error - e

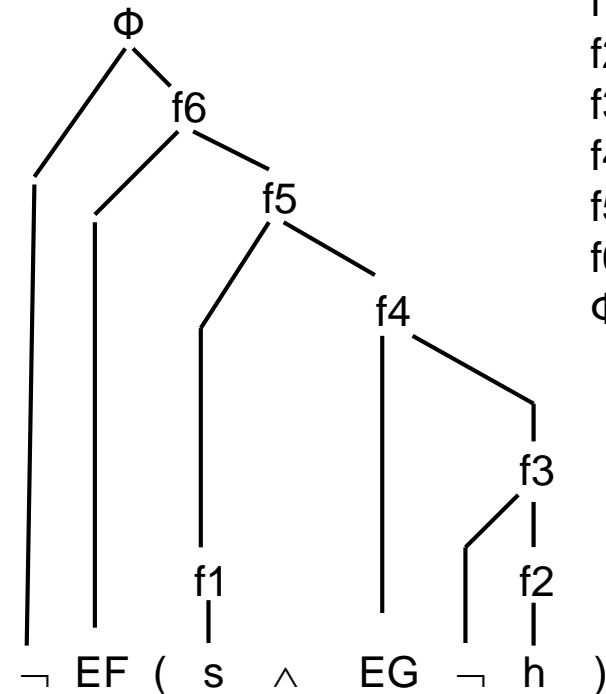
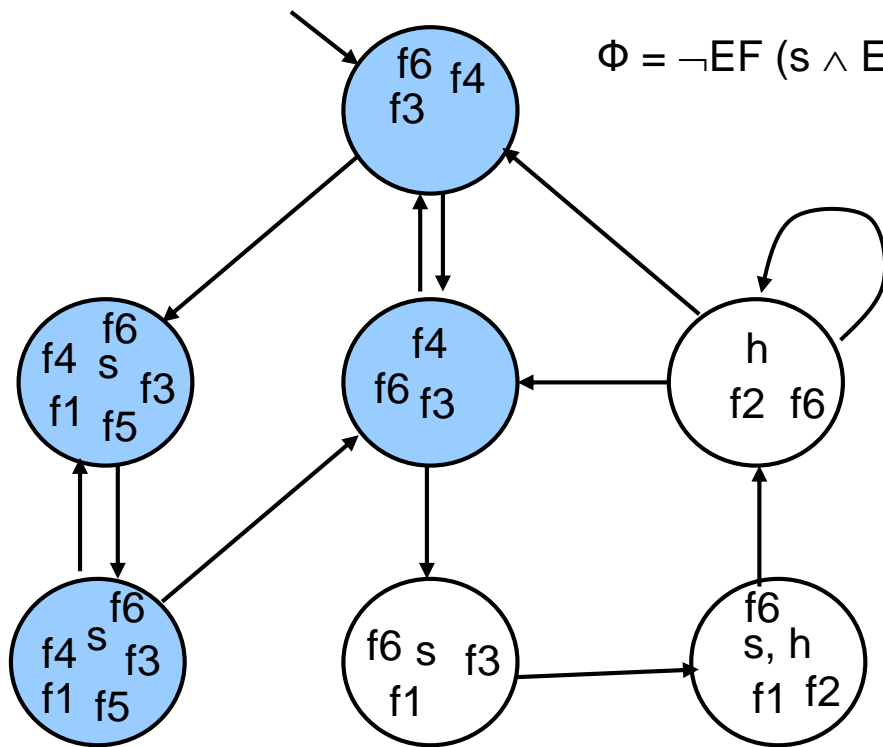
Хотим проверить: всегда если печь запустили (нажали Start), то она нагреется:

$$\Phi = AG (Start \Rightarrow AF Heat) = AG (s \Rightarrow AF h)$$

$$\Phi = \neg EF (s \wedge EG \neg h)$$

Построение подформул формулы Φ

Φ не выполняется



$f1 = s$
 $f2 = h$
 $f3 = \neg f2$
 $f4 = EG f3$
 $f5 = f1 \wedge f4$
 $f6 = EF f5$
 $\Phi = \neg f6$

Подструктура Крипке, удовлетворяющая $f3$, выделена. Ее ССК выделена голубым
 эта ССК достижима из всех состояний, поэтому $f6$ выполняется во всех состояниях
 Φ не выполняется ни в одном состоянии

Анализ с учетом справедливости

Печь не нагреется, если пользователь будет, например, открывать-закрывать дверцу, после запуска печи

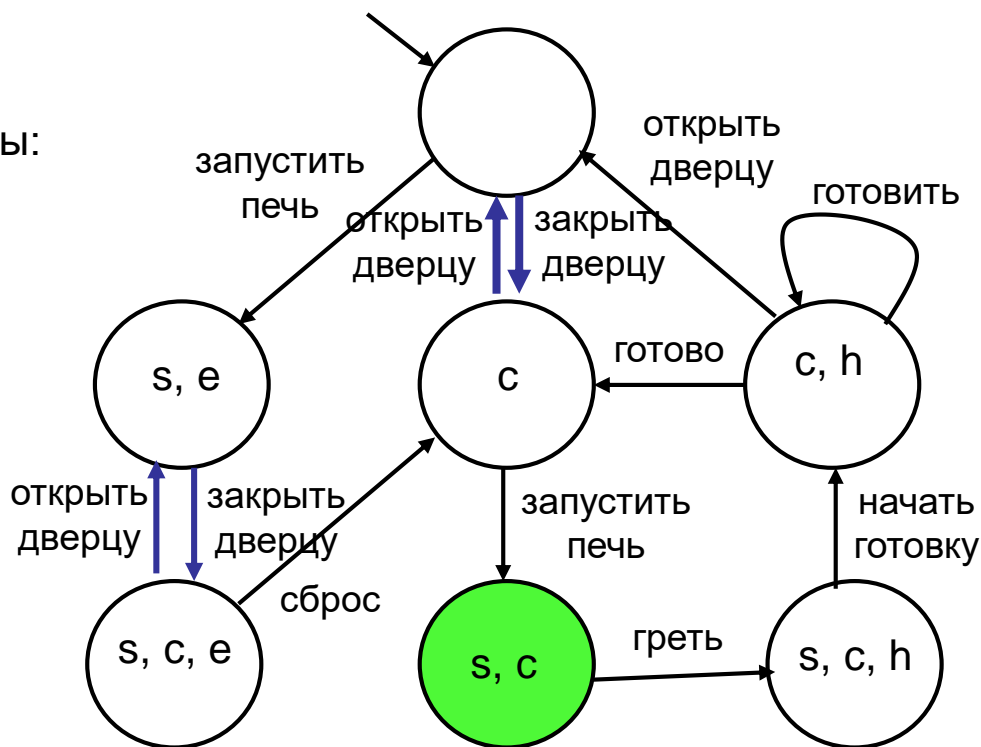
Атомные предикаты:

Start – s

Close – c

Heat – h

Error – e



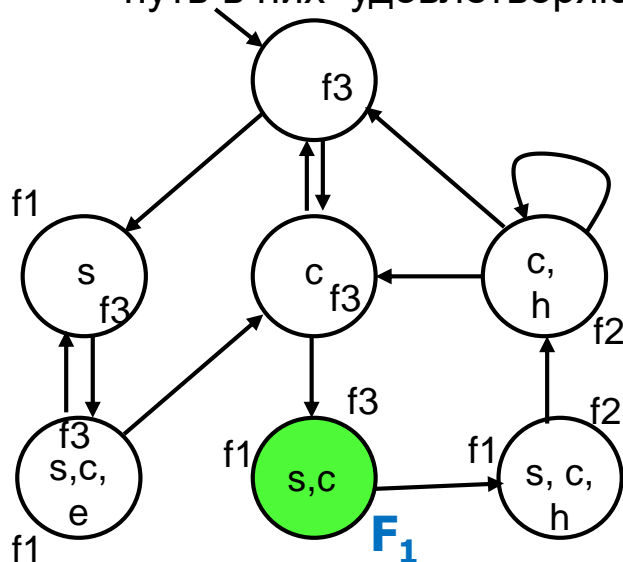
Ограничение справедливости:

Если пользователь правильно работает с печью, то вычисление обязательно проходит через $F1 = \{Start \wedge Close \wedge \neg Error\}$

Вычисление справедливой $\Phi = \neg EF (s \wedge EG \neg h)$ (т.е. при условии прохождения через F_1)

1. Находим все состояния, в которых удовлетворяется предикат *fair*. Это состояния, из которых существует вычисление, проходящее бесконечное число раз через состояние F_1 (*у нас – все*)
2. По подформулам ϕ вычисляем $s_{\text{fair}}(\phi)$ так: (*у нас – не изменяются*)

$s_{\text{fair}}(\phi) \equiv s \models (\text{fair} \wedge \phi)$	$f1 = s$
$s_{\text{fair}}(EX\phi) \equiv s \models (EX (\text{fair} \wedge \phi))$	$f2 = h$
$s_{\text{fair}}(E(\psi \cup \eta)) \equiv s \models (E(\psi \cup (\text{fair} \wedge \eta)))$	$f3 = \neg f2$
	$f4 = EG f3$
	$f5 = f1 \wedge f4$
	$f6 = EF f5$
	$\Phi = \neg f6$
3. Вычисление $s \models_{\text{fair}} EG f$ состоит в том, что выделяются ССК, помеченные f , которые пересекаются с каждым $\{F_i\}$ и эти ССК и путь в них удовлетворяют $\text{fair} EG f$



В состоянии s предикат *fair* истинен *тогда*, когда на структуре Крипке существует такая ССК, достижимая из s , которая пересекается со всеми множествами F_1, F_2, \dots, F_k . У нас одно F_i с одним состоянием, из всех состояний оно достигается, поэтому *fair* истинен во всех состояниях

$f4$ – не выполняется нигде – ССК пересечения F_1 с *fair*
 fair – пустое множество
 $f5=f6$ - пустые множества
 Φ – истинна везде



Заключение

Выбор множества проверяемых формул осуществляется человеком – верификатором, никто не может сказать, насколько ПОЛНО осуществлена проверка – все ли нужные свойства системы проверены.

Введение шаблонов помогает формализации требований к системе.

Определение классов свойств позволяет не пропустить при верификации некоторые важные типы свойств, общих для всех параллельных процессов.

Классы свойств ПАРАЛЛЕЛЬНЫХ процессов

Достижимость (reachability):

некоторое конкретное состояние будет достигнуто

Безопасность (safety):

нечто плохое не произойдет

Живость (liveness) :

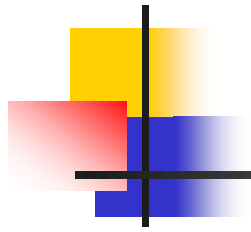
нечто хорошее выполнится

Свобода от дедлоков (deadlock freeness, non blocking)

процесс не будет заблокирован другим процессом

Справедливость (fairness):

ограничение нереальных или нежелательных вычислений при анализе



Спасибо за внимание