Pragmatic model checking: from theory to implementations

Niels Lohmann

Universität Rostock
Traditio et Innovatio

1

How does a model checker work?

2

How to build an effective model checker?

3

How to use a model checker pragmatically?

1

How does a model checker work?

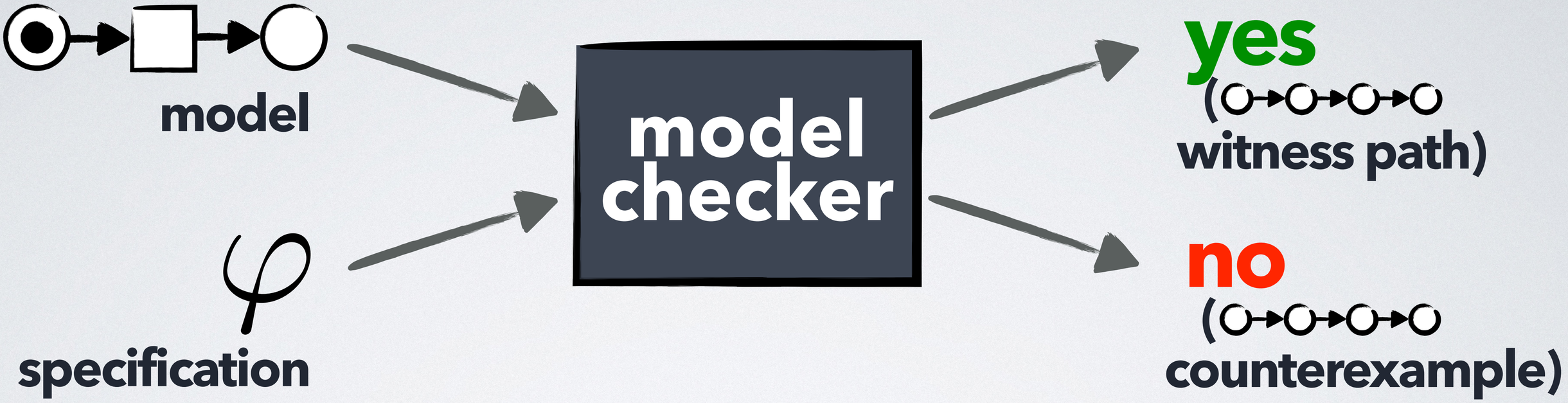
2

How to build an effective model checker?

3

How to use a model checker pragmatically?

Model checking in a nutshell



Model checking in a nutshell

model

φ

specification

model checker

yes

(witness path)

no

(counterexample)

memory overflow

State explosion

business process* with 66 parallel branches

*** modeled by IBM customers using the IBM Websphere business modeler**

State explosion

business process* with 66 parallel branches

state space:
$2^{66} \approx$ **$2.37 \cdot 10^{19}$ markings**

*** modeled by IBM customers using the IBM Websphere business modeler**

State explosion

business process* with 66 parallel branches

state space:
$2^{66} \approx$ **$2.37 \cdot 10^{19}$ markings**

memory for state space:
608 exabytes** (66 bit per marking)

*** modeled by IBM customers using the IBM Websphere business modeler**

**** mega, giga, peta, tera, exa...**

State explosion

business process* with 66 parallel branches

state space:
$2^{66} \approx$ **$2.37 \cdot 10^{19}$ markings**

memory for state space:
608 exabytes** (66 bit per marking)

time for state space generation
7799 years (10 CPU cycles/marking at 3 GHz)

*** modeled by IBM customers using the IBM Websphere business modeler**

**** mega, giga, peta, tera, exa...**

State explosion

business process* with 66 parallel branches

state space:
$2^{66} \approx$ **$2.37 \cdot 10^{19}$ markings**

memory for state space:
608 exabytes** (66 bit per marking)

time for state space generation
7799 years (10 CPU cycles/marking at 3 GHz)

energy consumption for state space generation
2,9 megatons of TNT (at 50 watts)

*** modeled by IBM customers using the IBM Websphere business modeler**

**** mega, giga, peta, tera, exa...**

State explosion

business process* with 66 parallel branches

state space:
$2^{66} \approx$ **$2.37 \cdot 10^{19}$ markings**

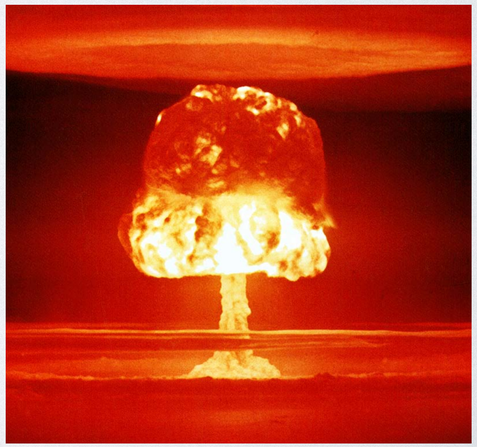
memory for state space:
608 exabytes** (66 bit per marking)

time for state space generation
7799 years (10 CPU cycles/marking at 3 GHz)

energy consumption for state space generation
2,9 megatons of TNT (at 50 watts)

*** modeled by IBM customers using the IBM Websphere business modeler**

**** mega, giga, peta, tera, exa...**



The core algorithm

```
   markings = []
   search(m0,φ)

   def search(m,φ):
1     check(m,φ)
2     markings.add(m)
3     for t in enabled(m):
4        m' = fire(m,t)
5        if not m' in markings:
6           search(m',φ)
```

- **search** is a simple depth first search (+ SCC detection)
- **check** depends on the nature of the property and may terminate search
- **enabled** and **fire** implement the Petri net firing rule

Reduction techniques

apply **theoretic** results to

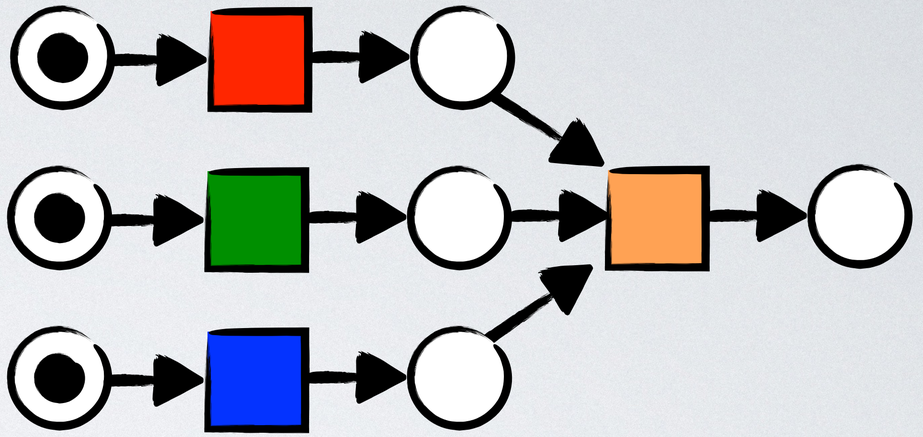
1. store fewer markings
2. fire fewer transitions / generate fewer markings

} while **preserving the property**

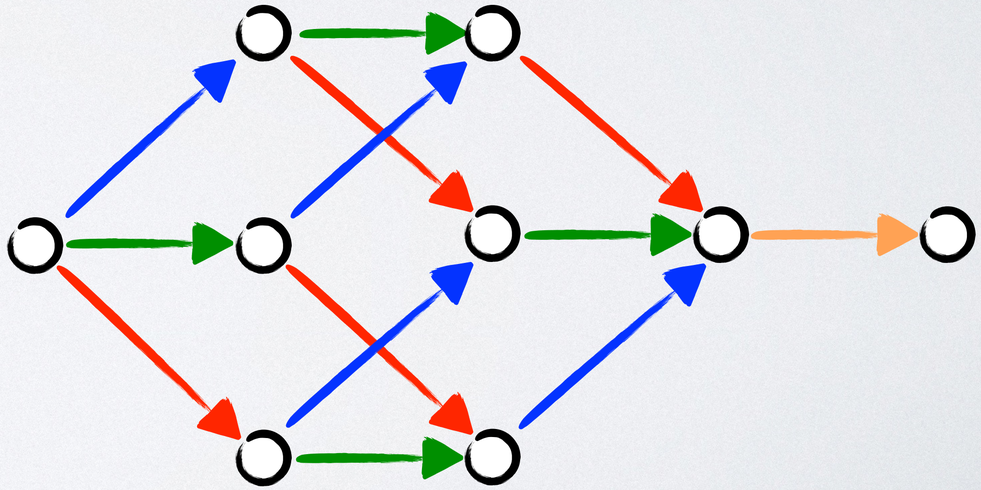
3. store markings more efficiently

1. Partial order reduction

observation: concurrent transitions can be fired in **any order**, yielding the same final state

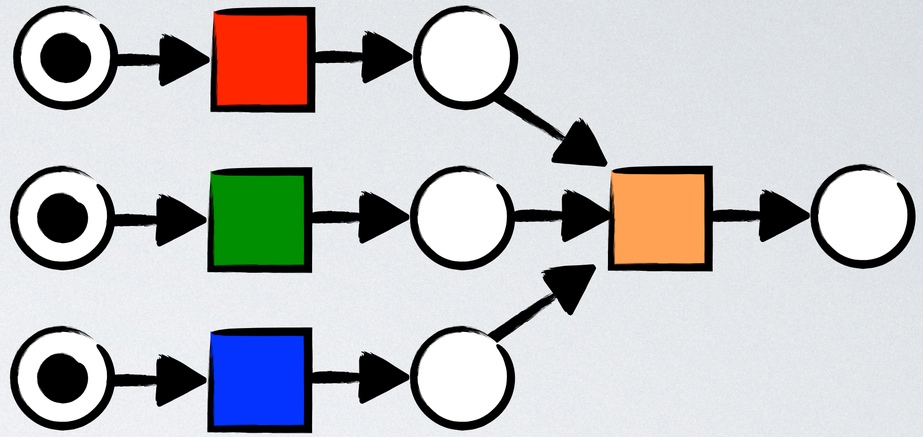


idea: fix one ordering by **postponing** the firing of some transitions



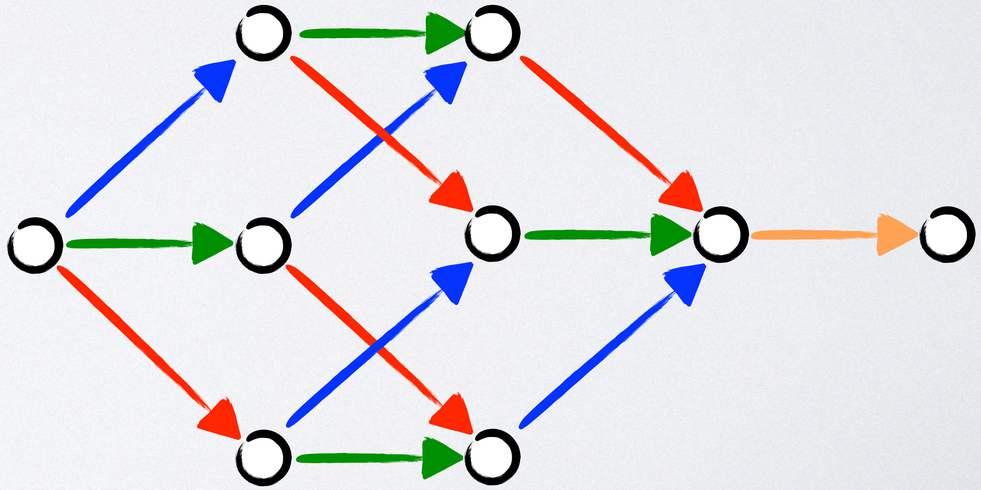
implementation: search on Petri net structure

1. Partial order reduction



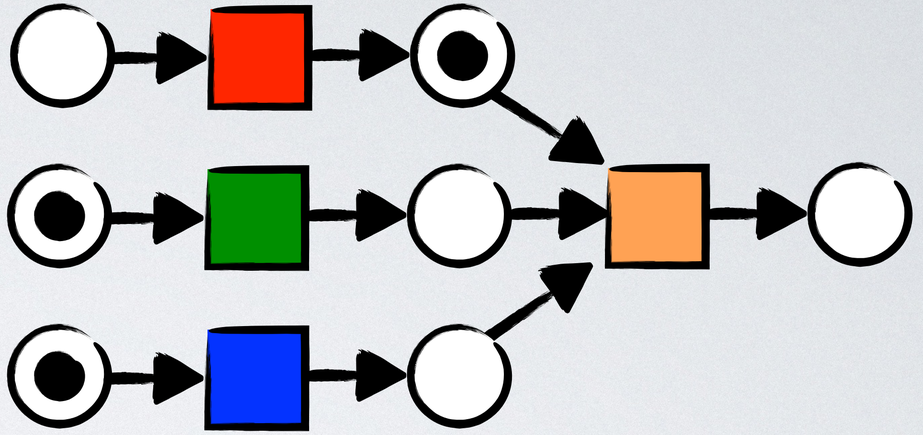
algorithm (sketch):

- choose one activated transition
- until fixed point is reached: add all conflicting transitions



only fire these transitions

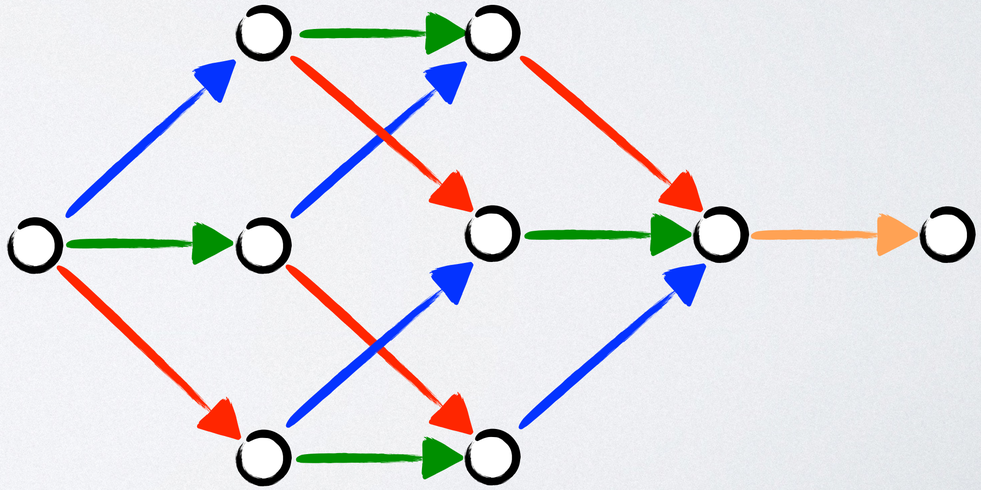
1. Partial order reduction



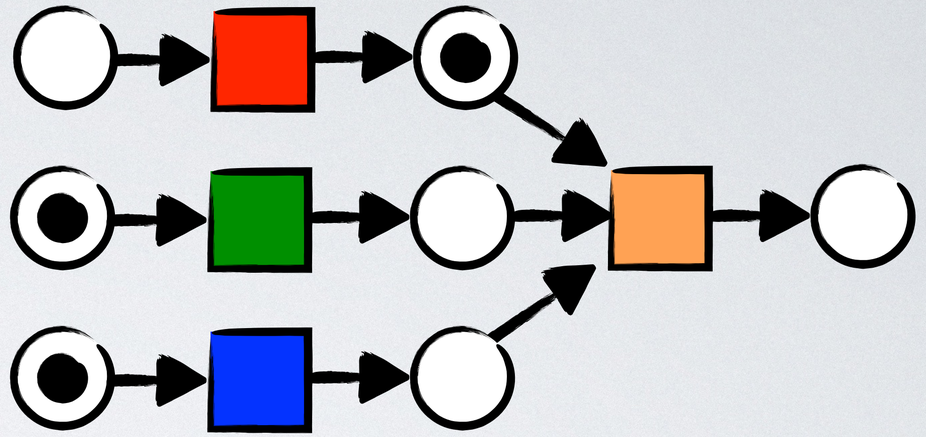
algorithm (sketch):

- choose one activated transition
- until fixed point is reached: add all conflicting transitions

only fire these transitions



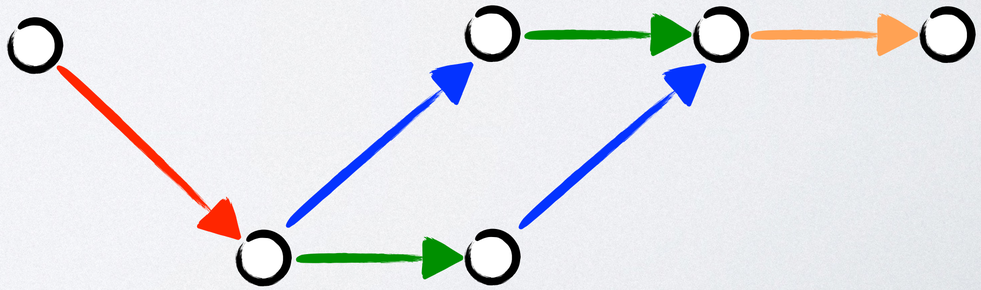
1. Partial order reduction



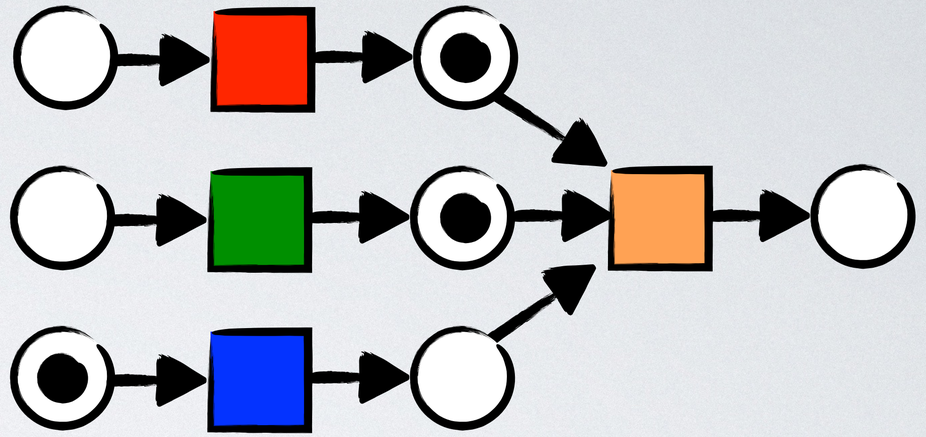
algorithm (sketch):

- choose one activated transition
- until fixed point is reached: add all conflicting transitions

only fire these transitions



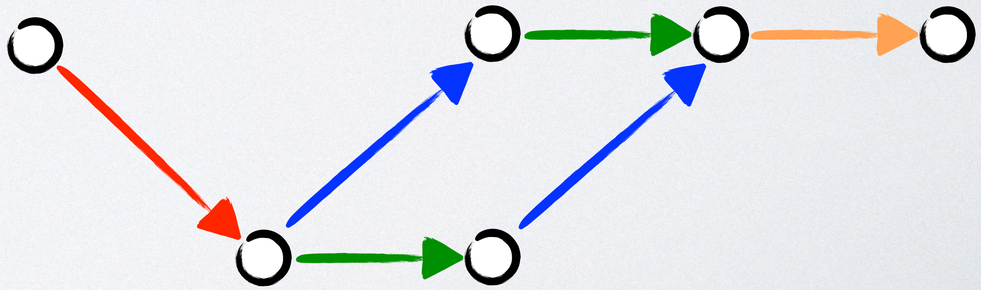
1. Partial order reduction



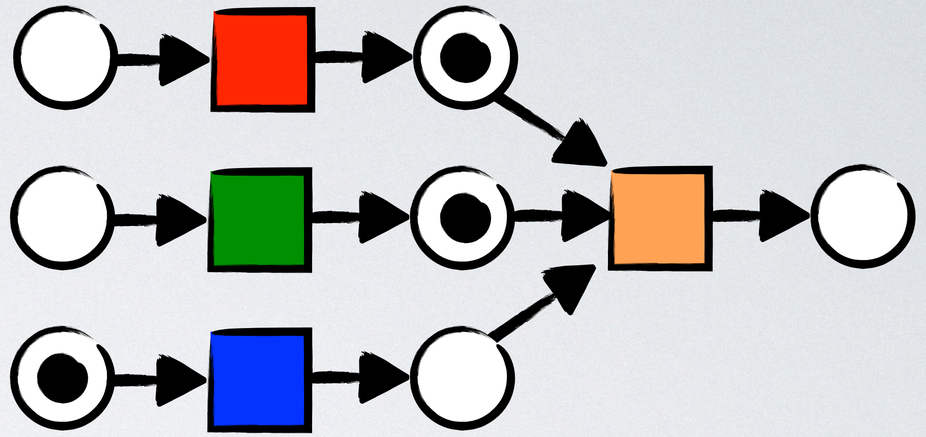
algorithm (sketch):

- choose one activated transition
- until fixed point is reached: add all conflicting transitions

only fire these transitions



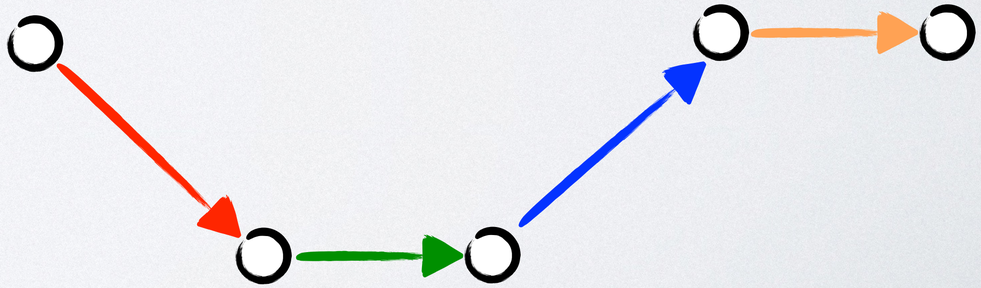
1. Partial order reduction



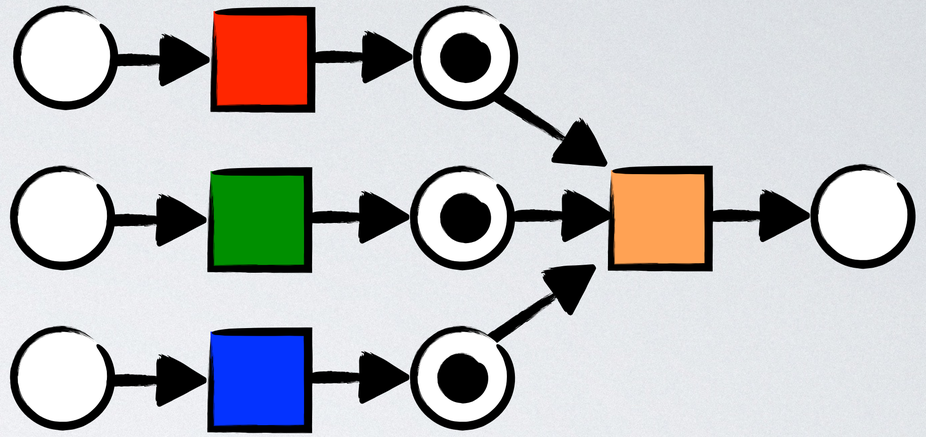
algorithm (sketch):

- choose one activated transition
- until fixed point is reached: add all conflicting transitions

only fire these transitions



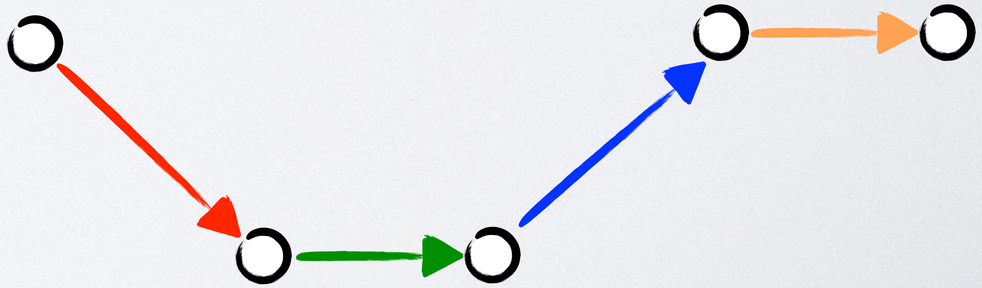
1. Partial order reduction



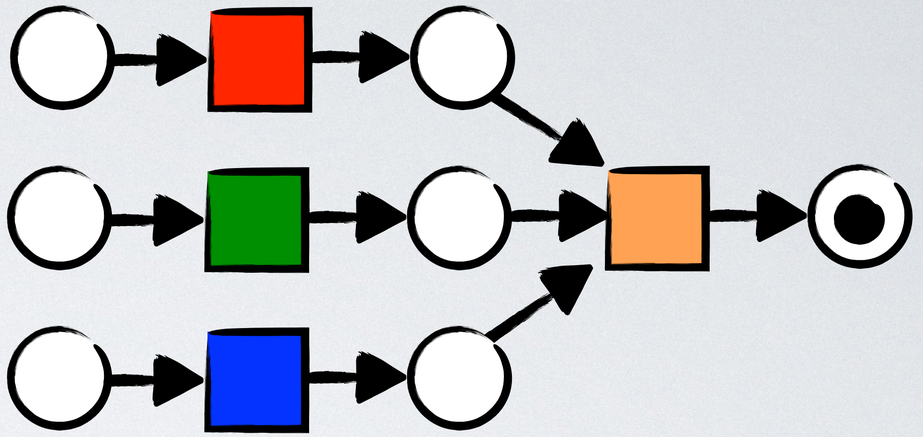
algorithm (sketch):

- choose one activated transition
- until fixed point is reached: add all conflicting transitions

only fire these transitions



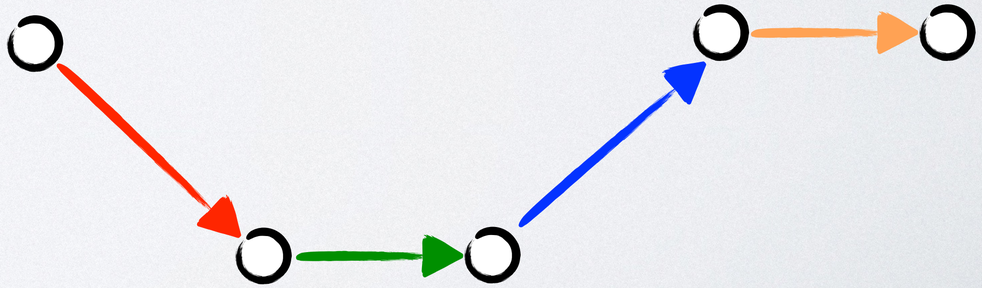
1. Partial order reduction



algorithm (sketch):

- choose one activated transition
- until fixed point is reached: add all conflicting transitions

only fire these transitions

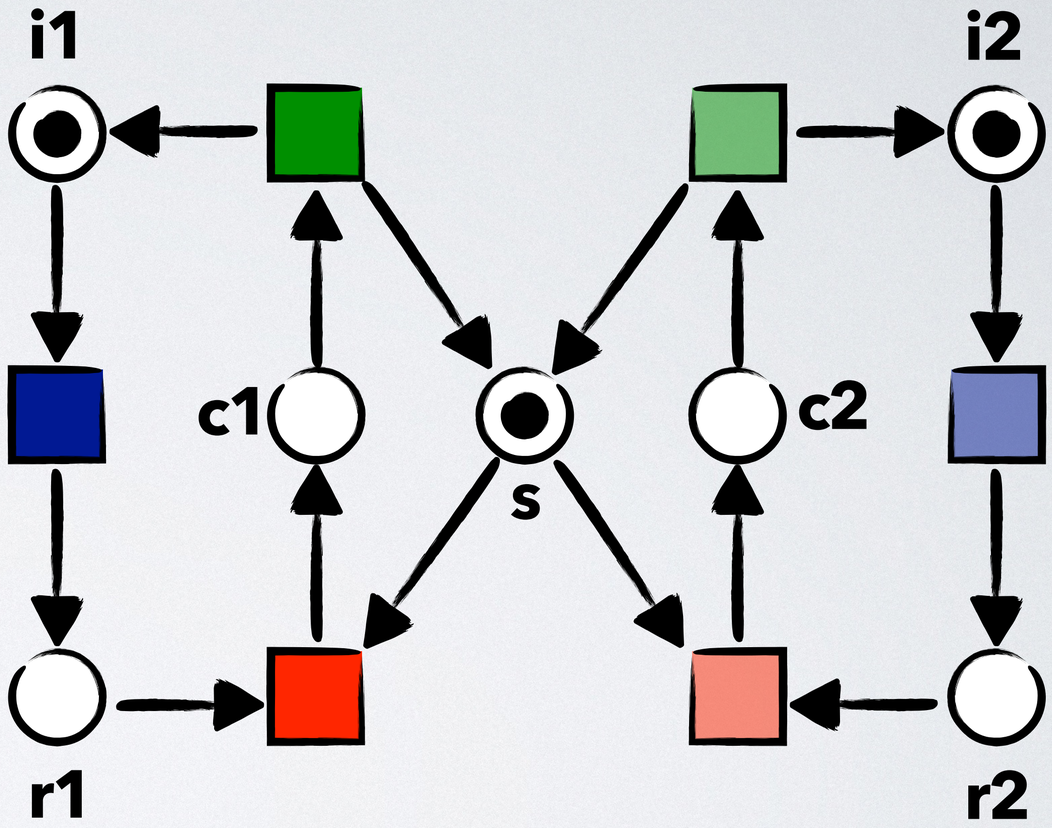


2. Symmetry reduction

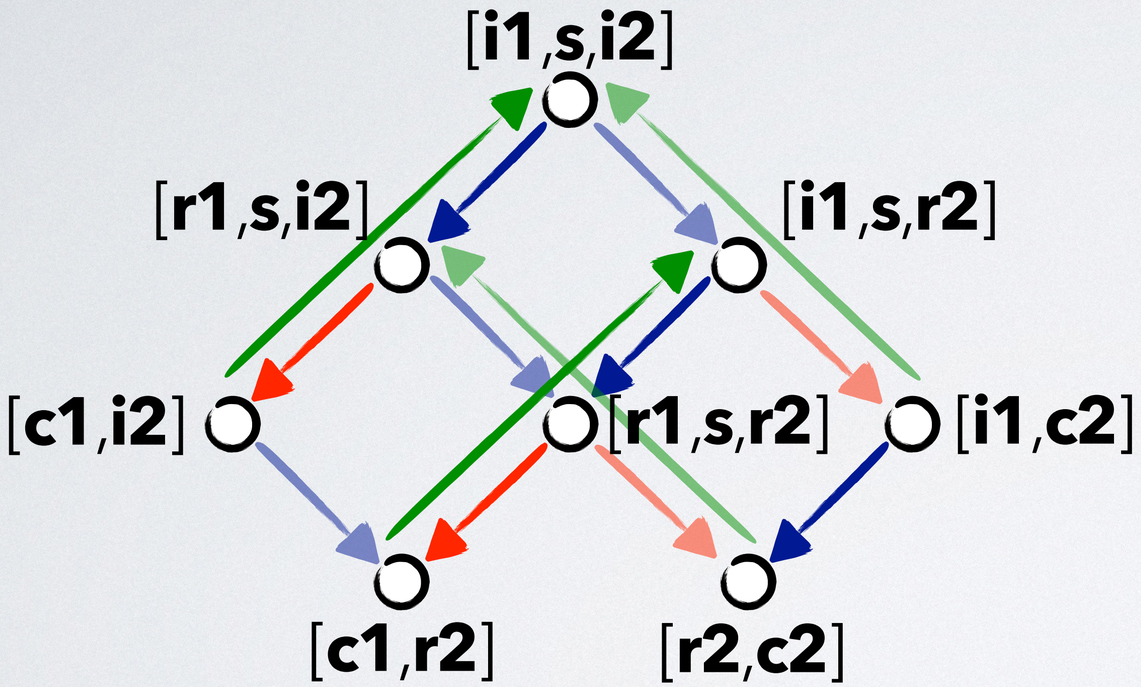
observation: symmetric models (e.g. due to copies of components) have **symmetric behavior**

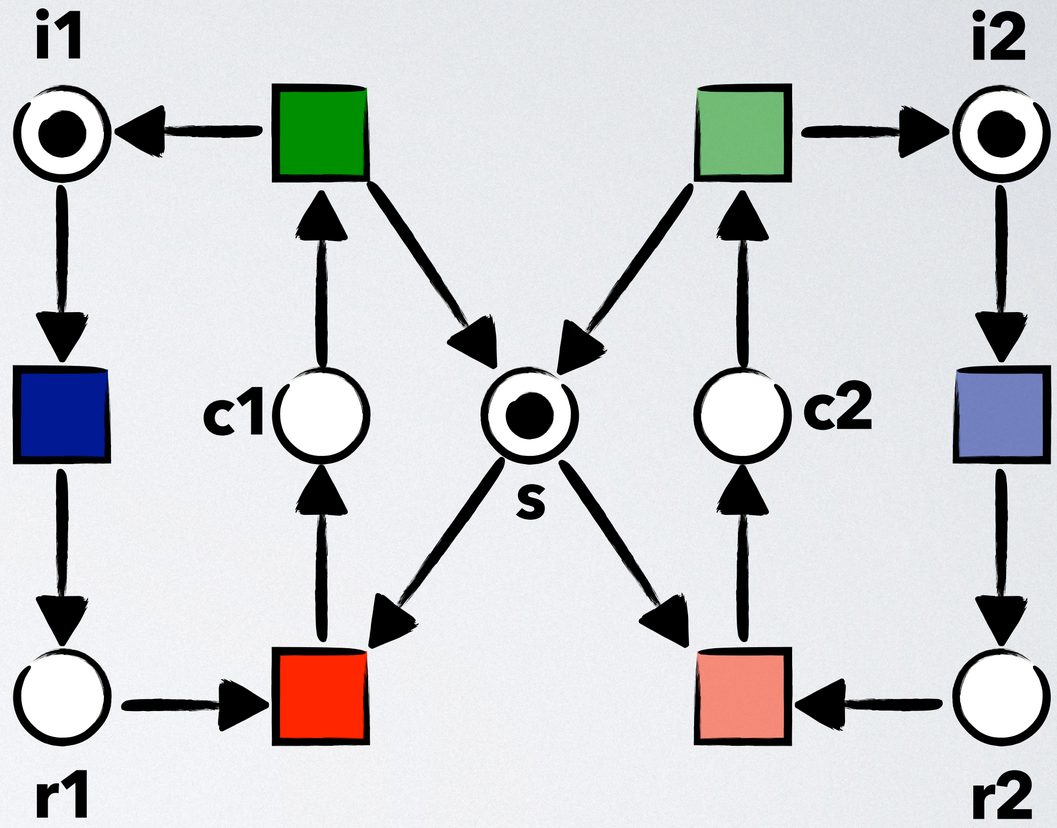
idea: do not store markings if a symmetric copy is already stored

implementation: Petri net graph automorphisms

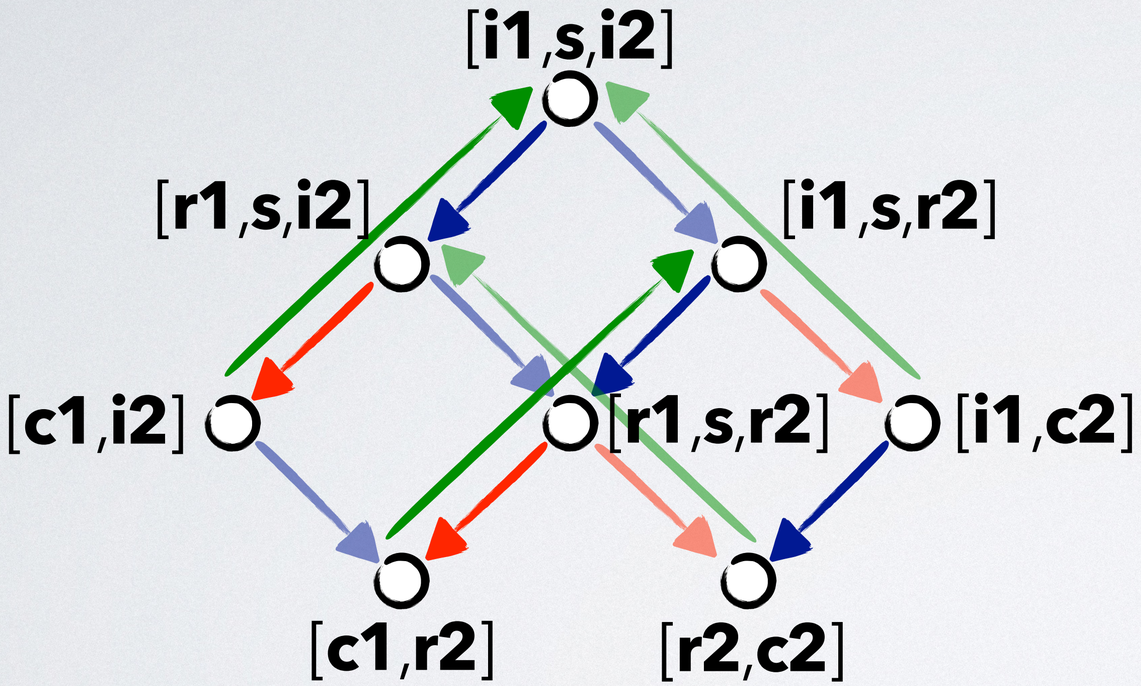


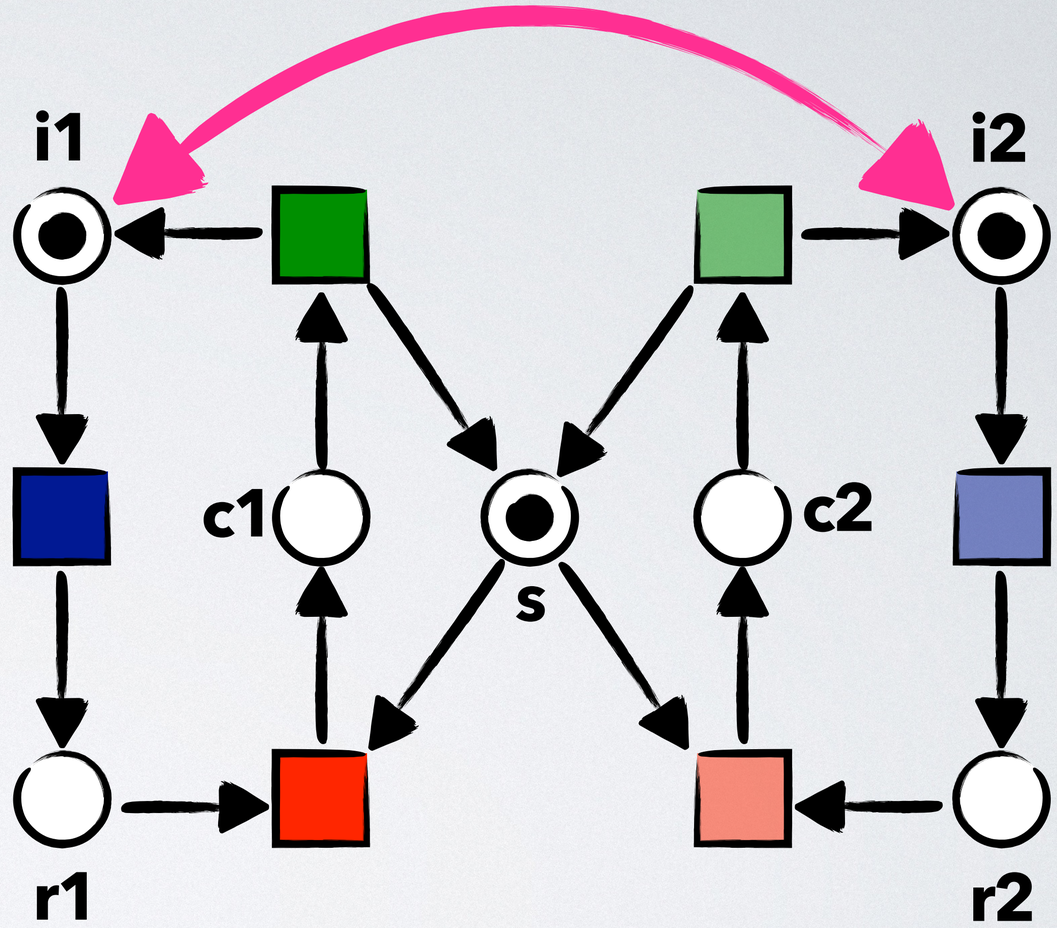
2. Symmetry reduction





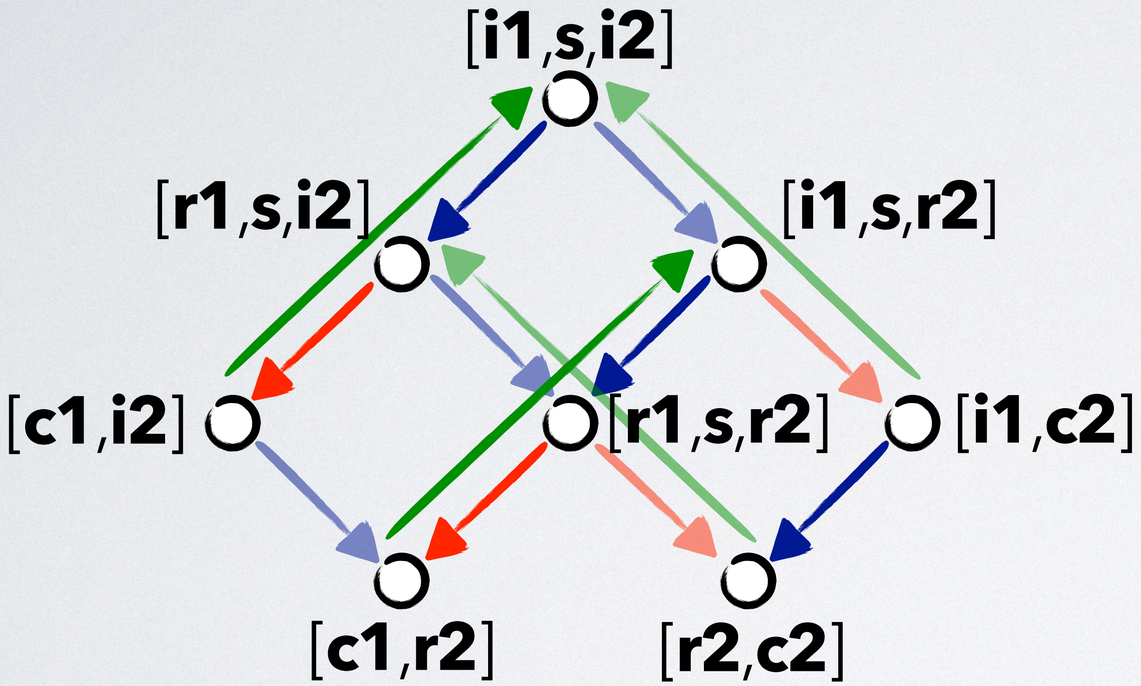
2. Symmetry reduction

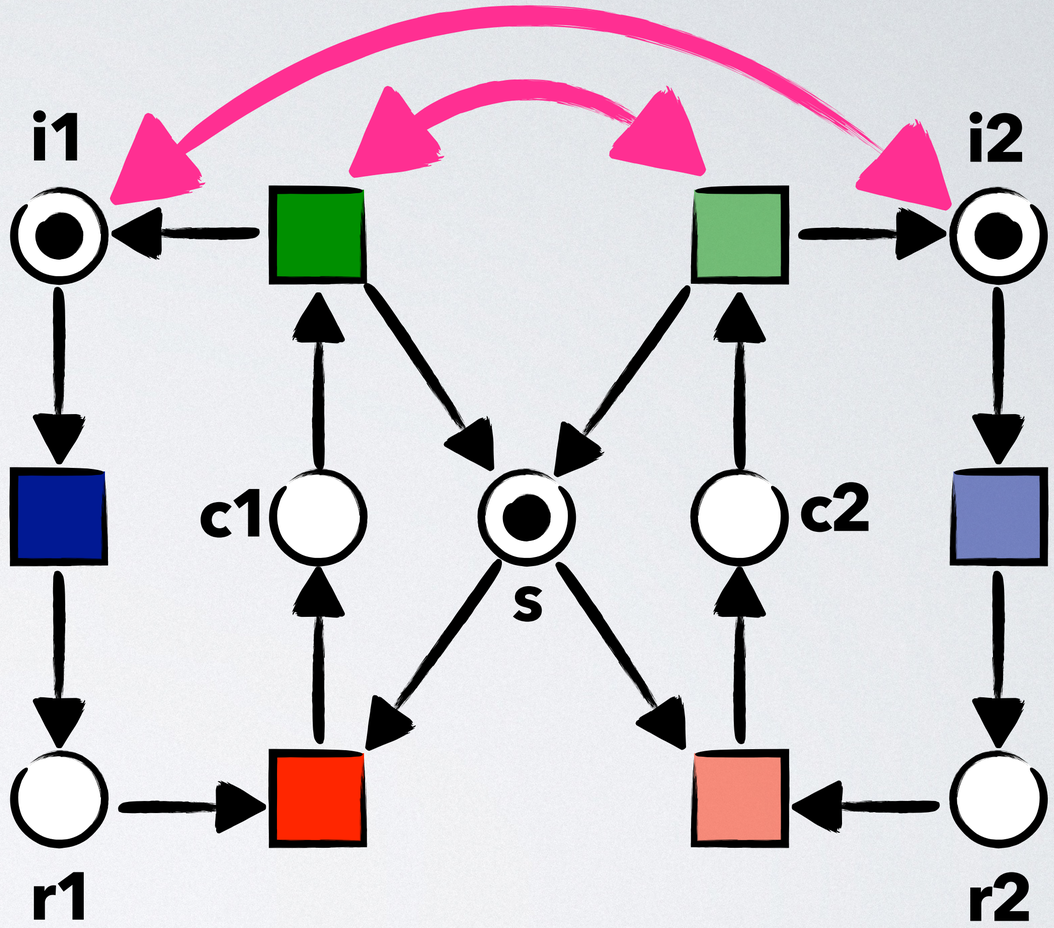




graph automorphisms

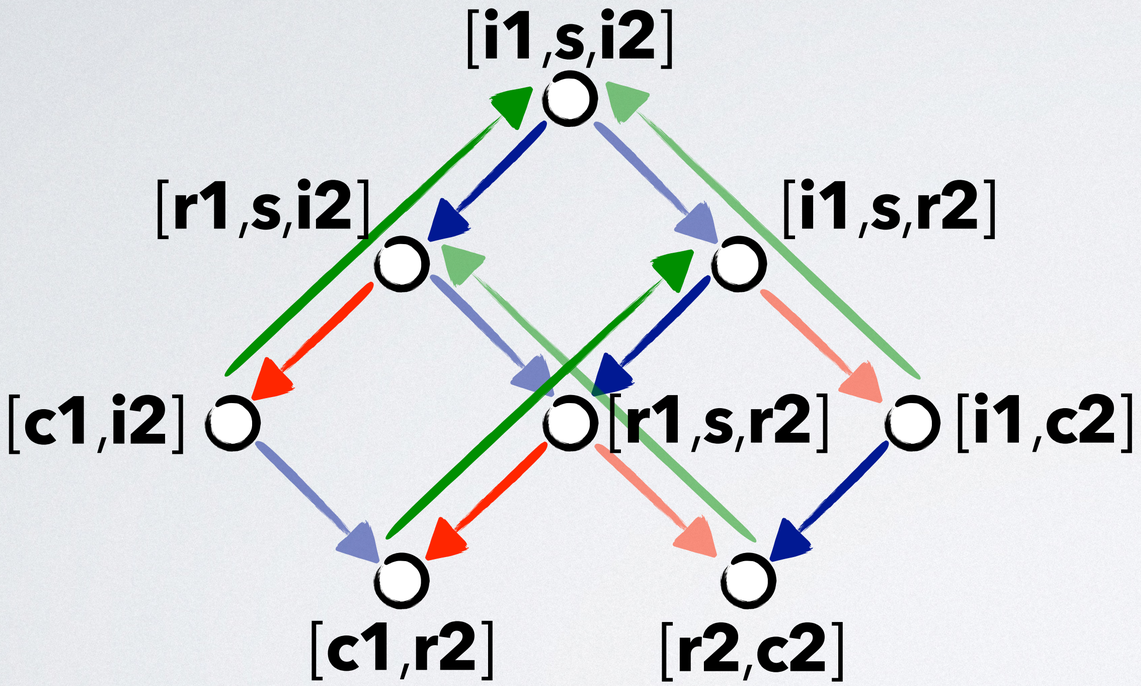
2. Symmetry reduction

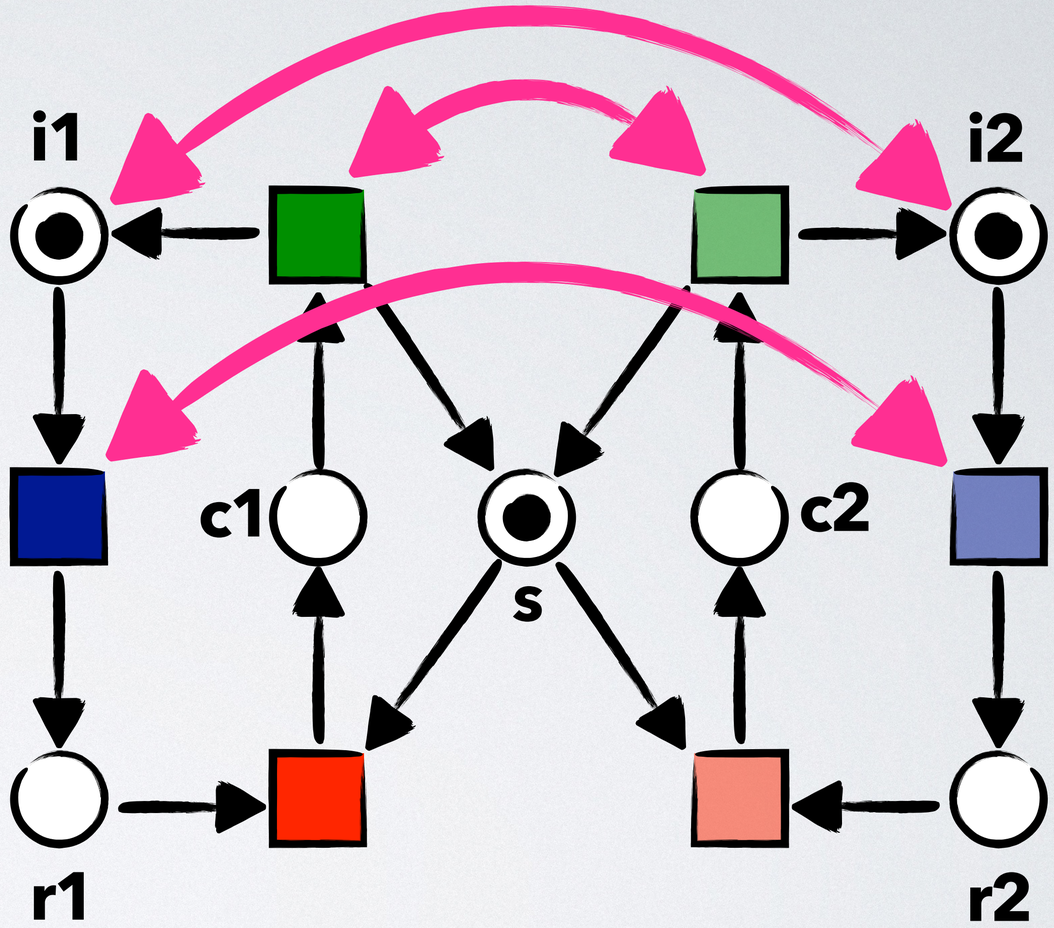




graph automorphisms

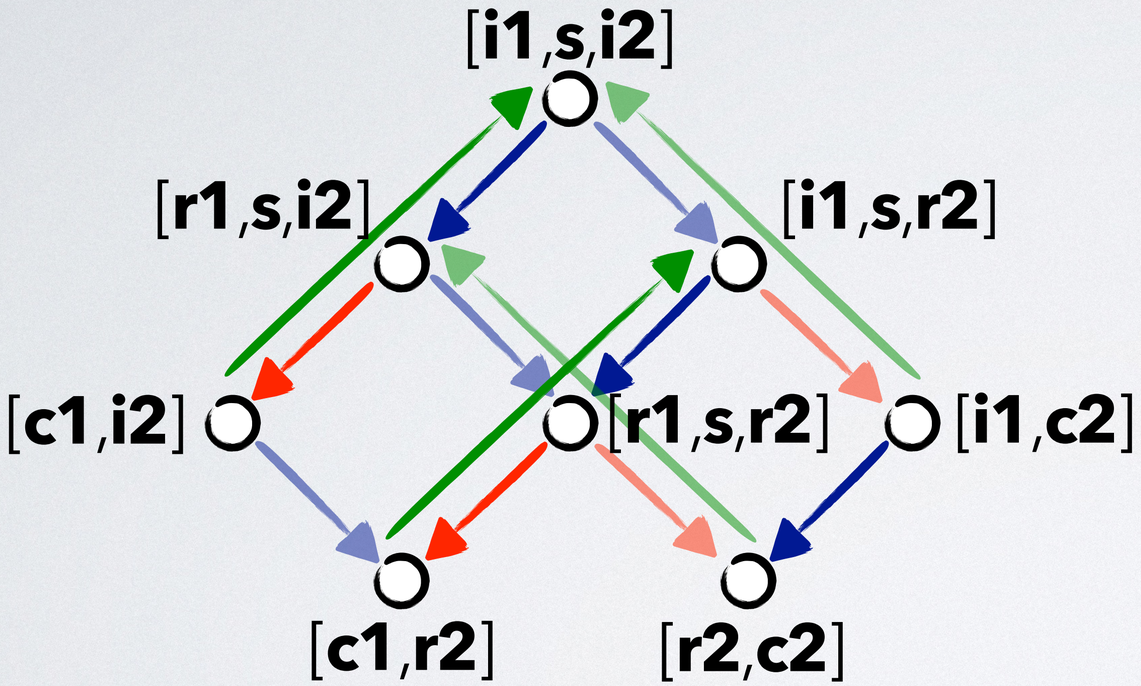
2. Symmetry reduction

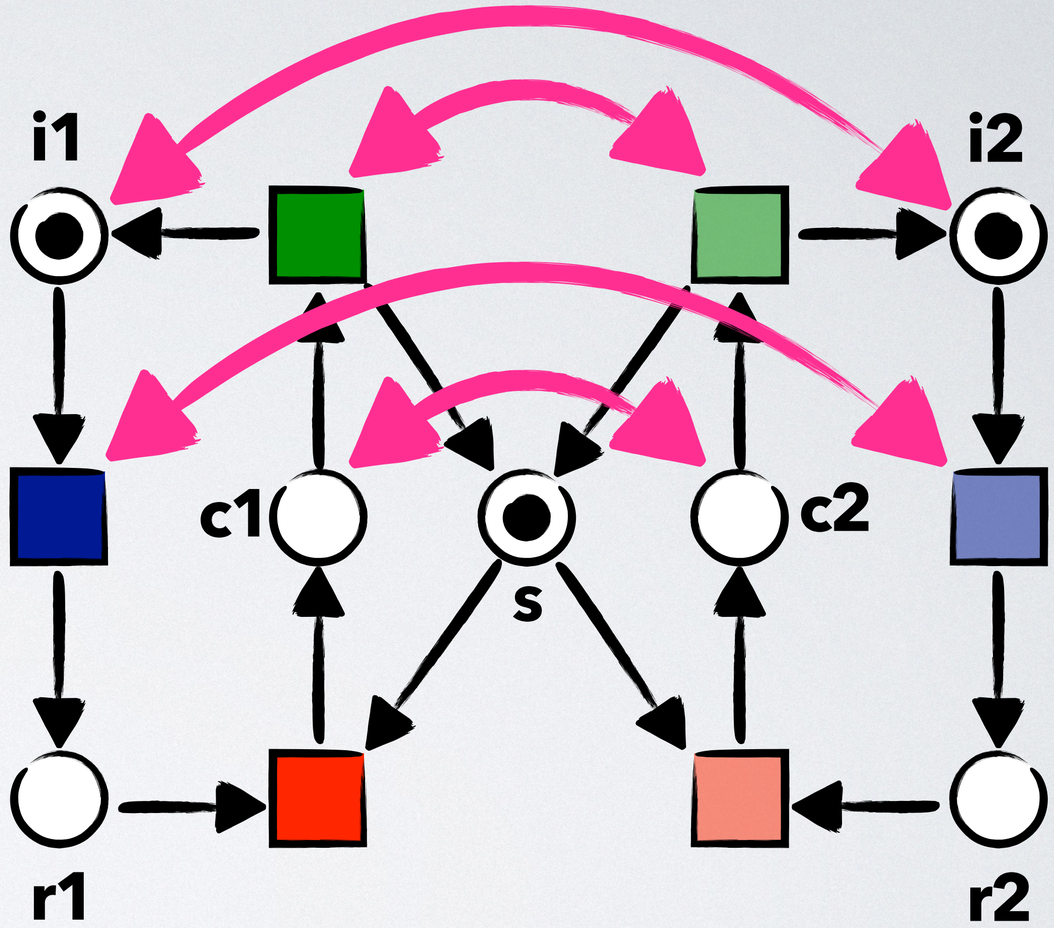




graph automorphisms

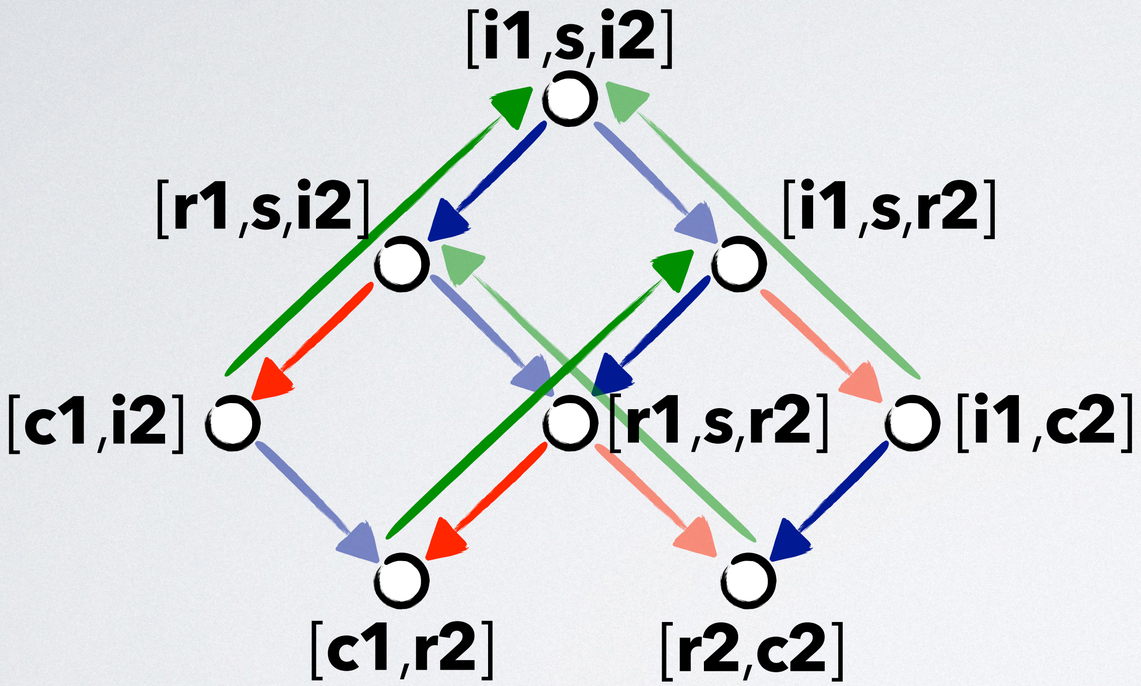
2. Symmetry reduction

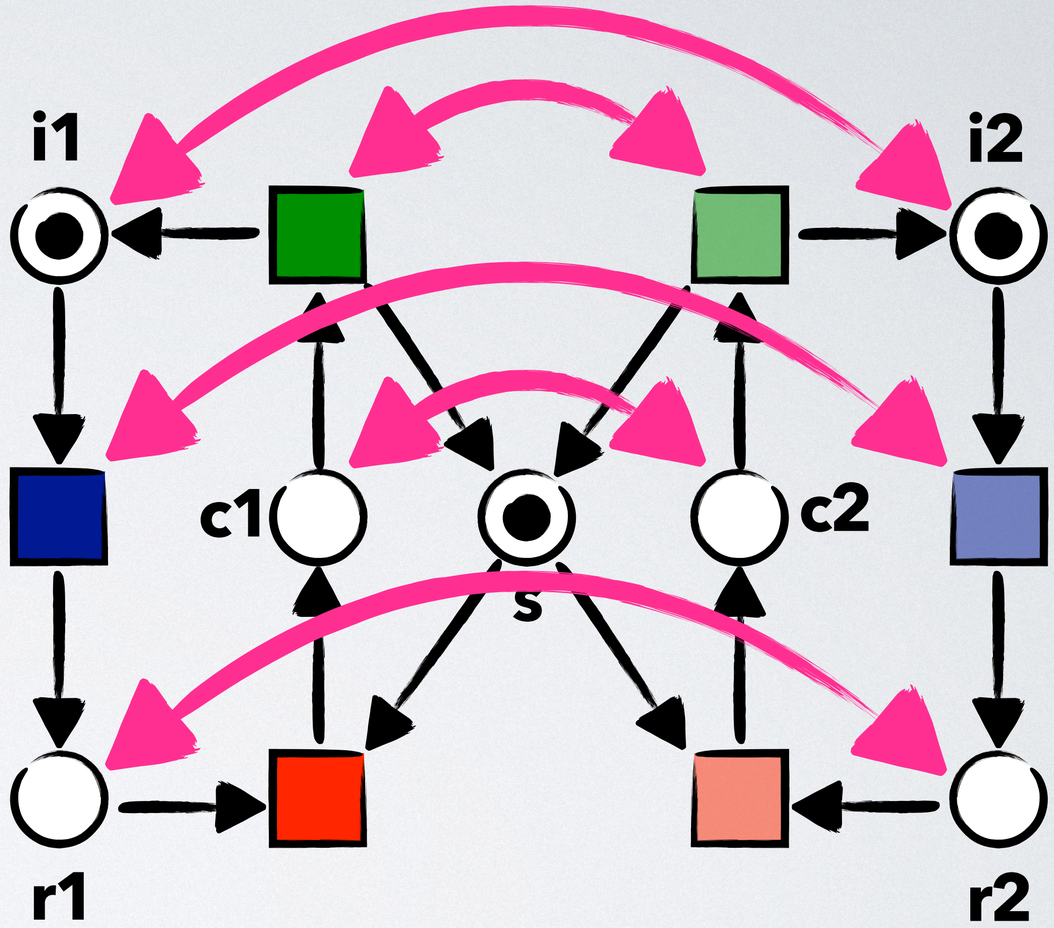




graph automorphisms

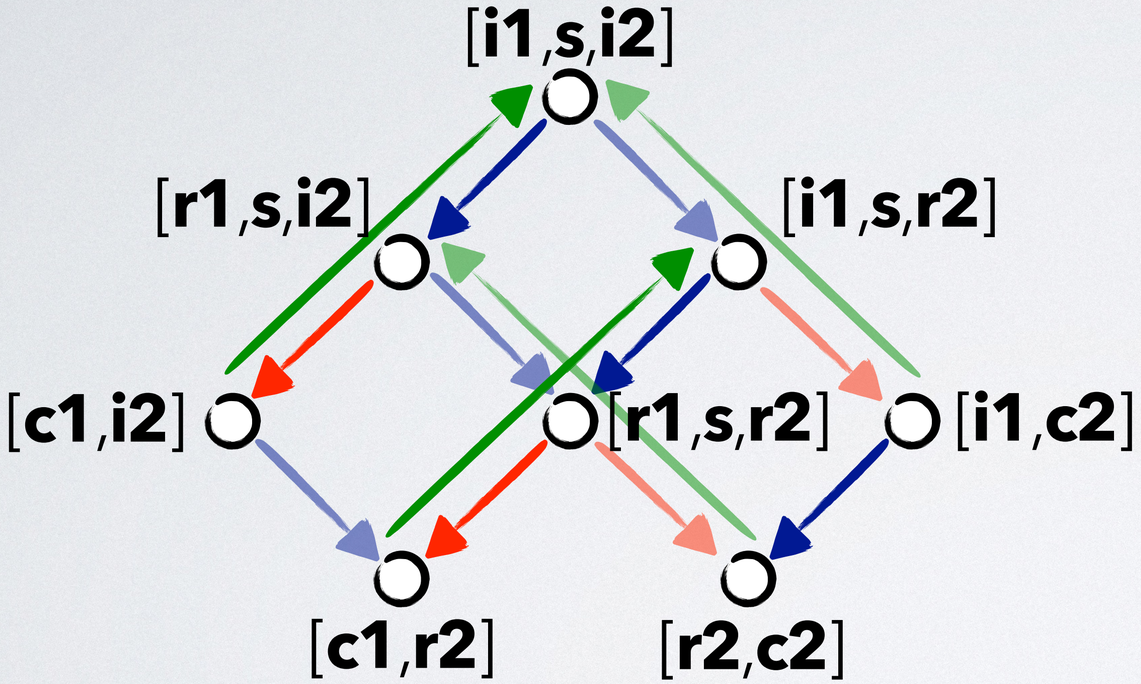
2. Symmetry reduction

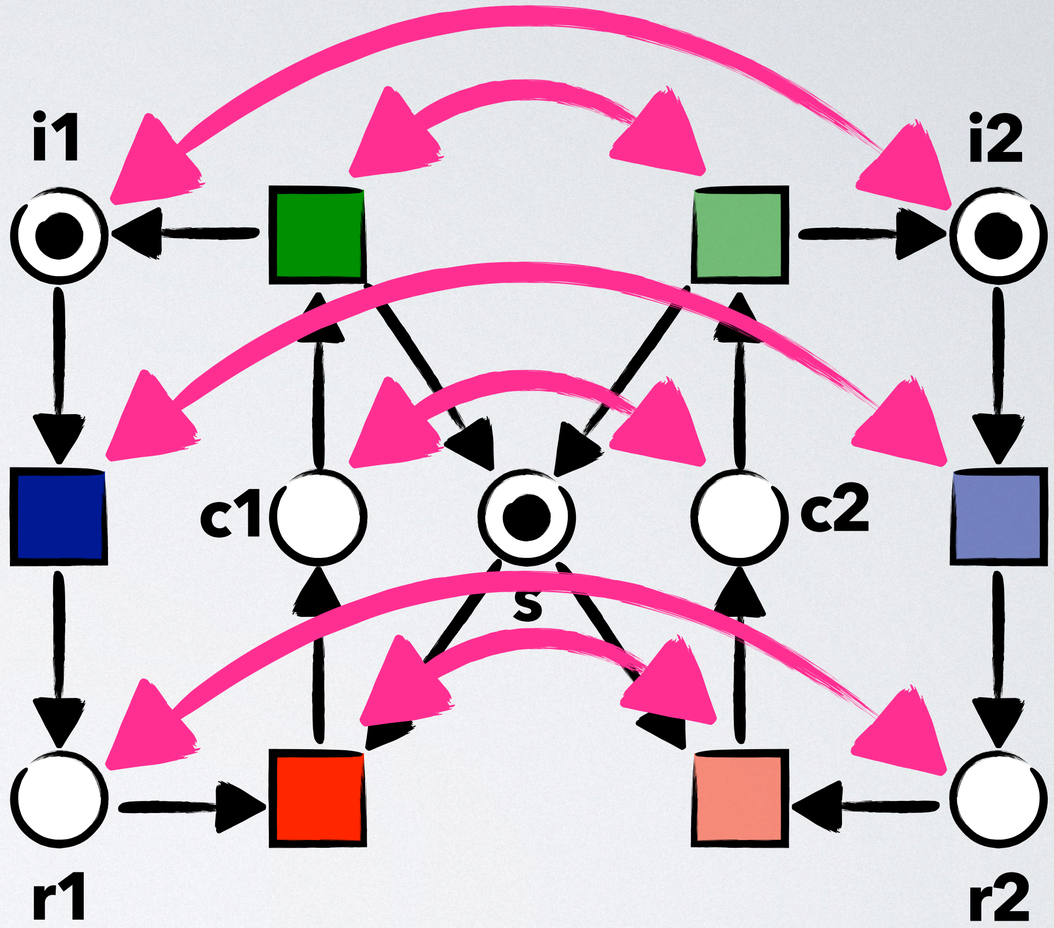




graph automorphisms

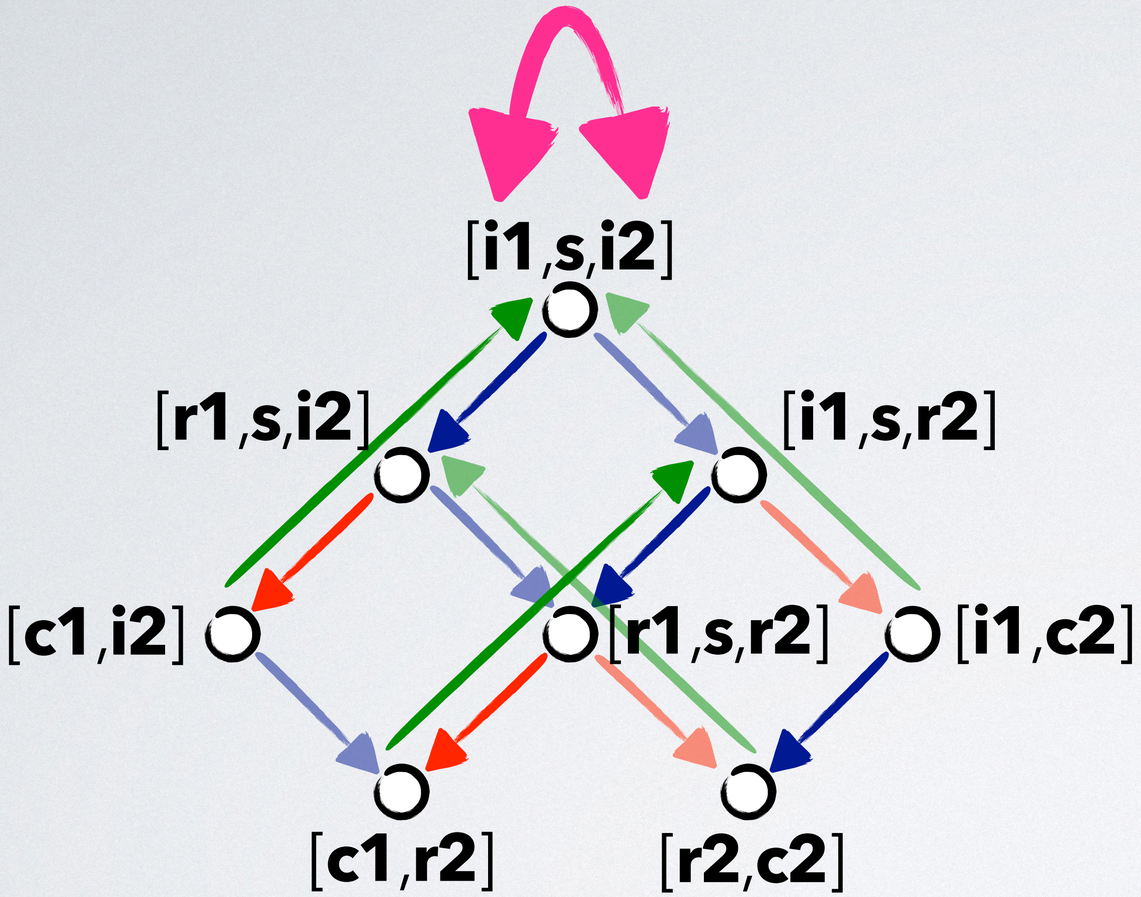
2. Symmetry reduction



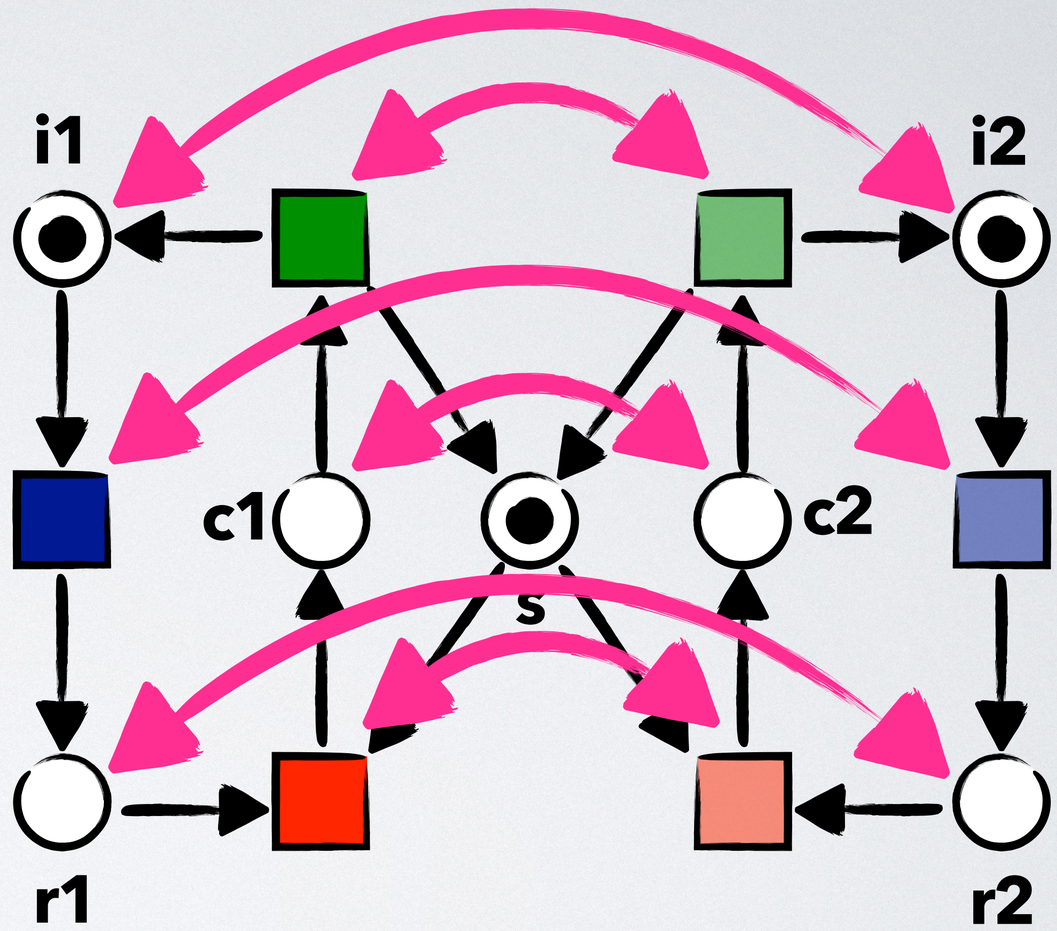


graph automorphisms

2. Symmetry reduction

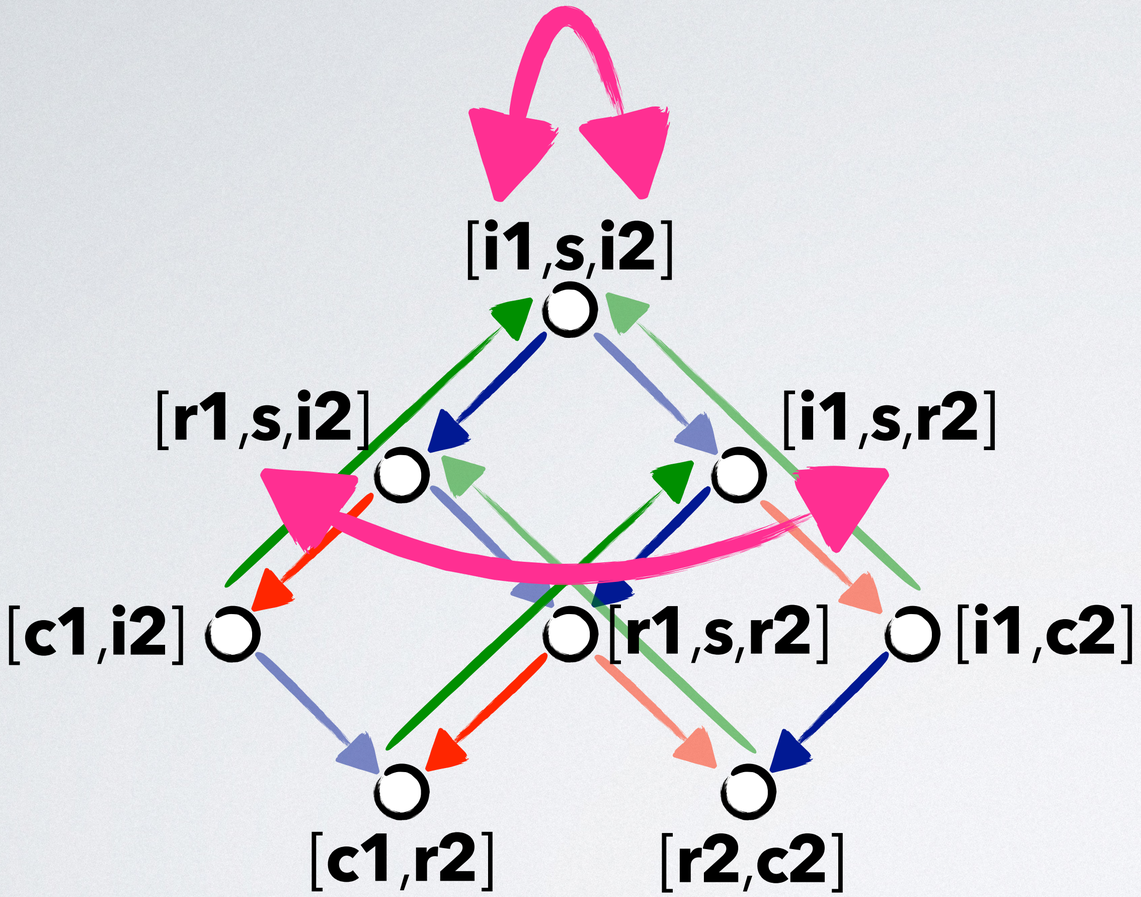


symmetric markings

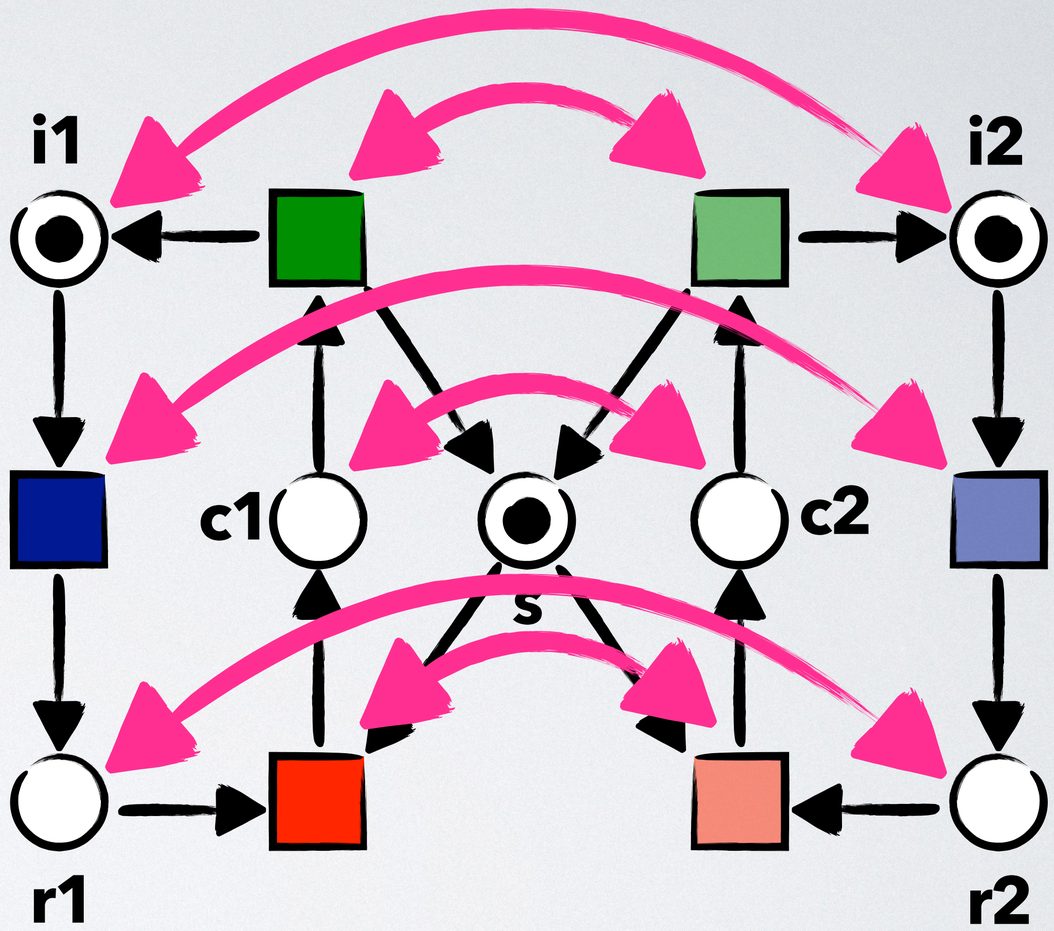


graph automorphisms

2. Symmetry reduction

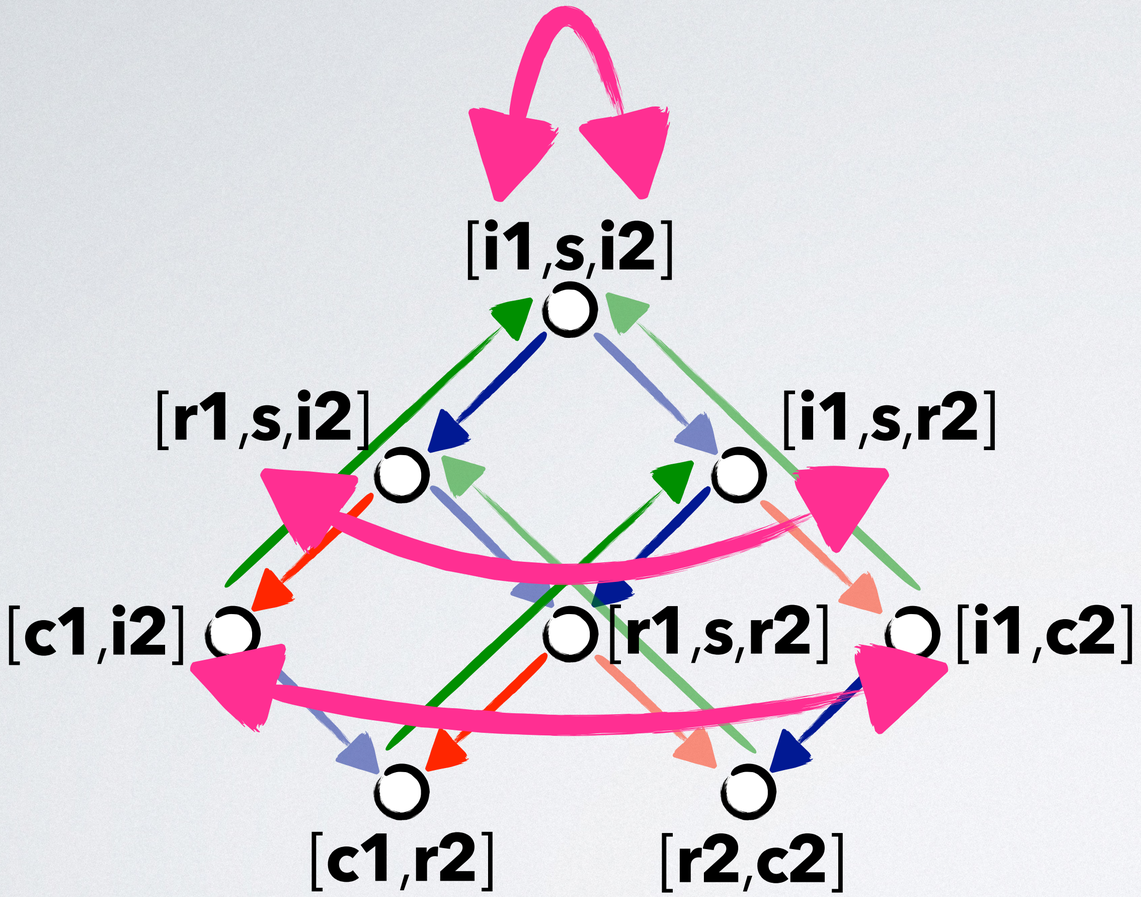


symmetric markings

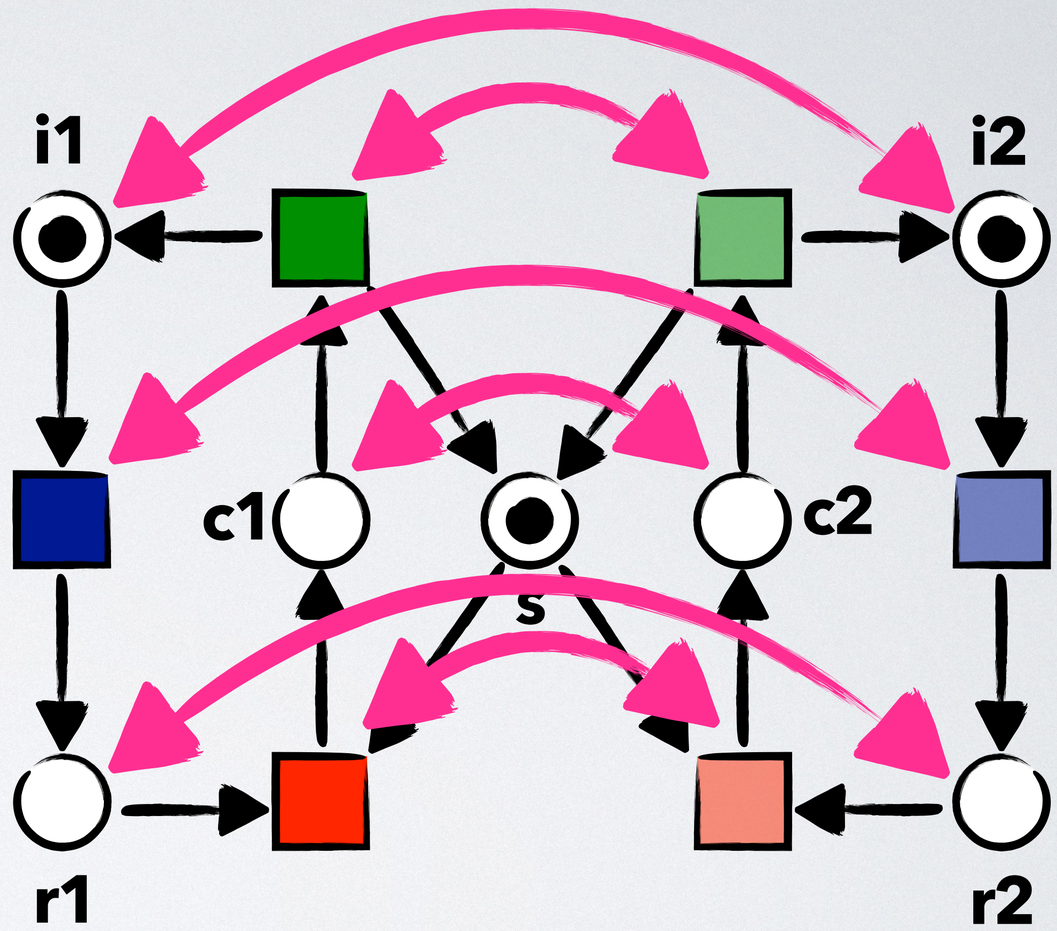


graph automorphisms

2. Symmetry reduction

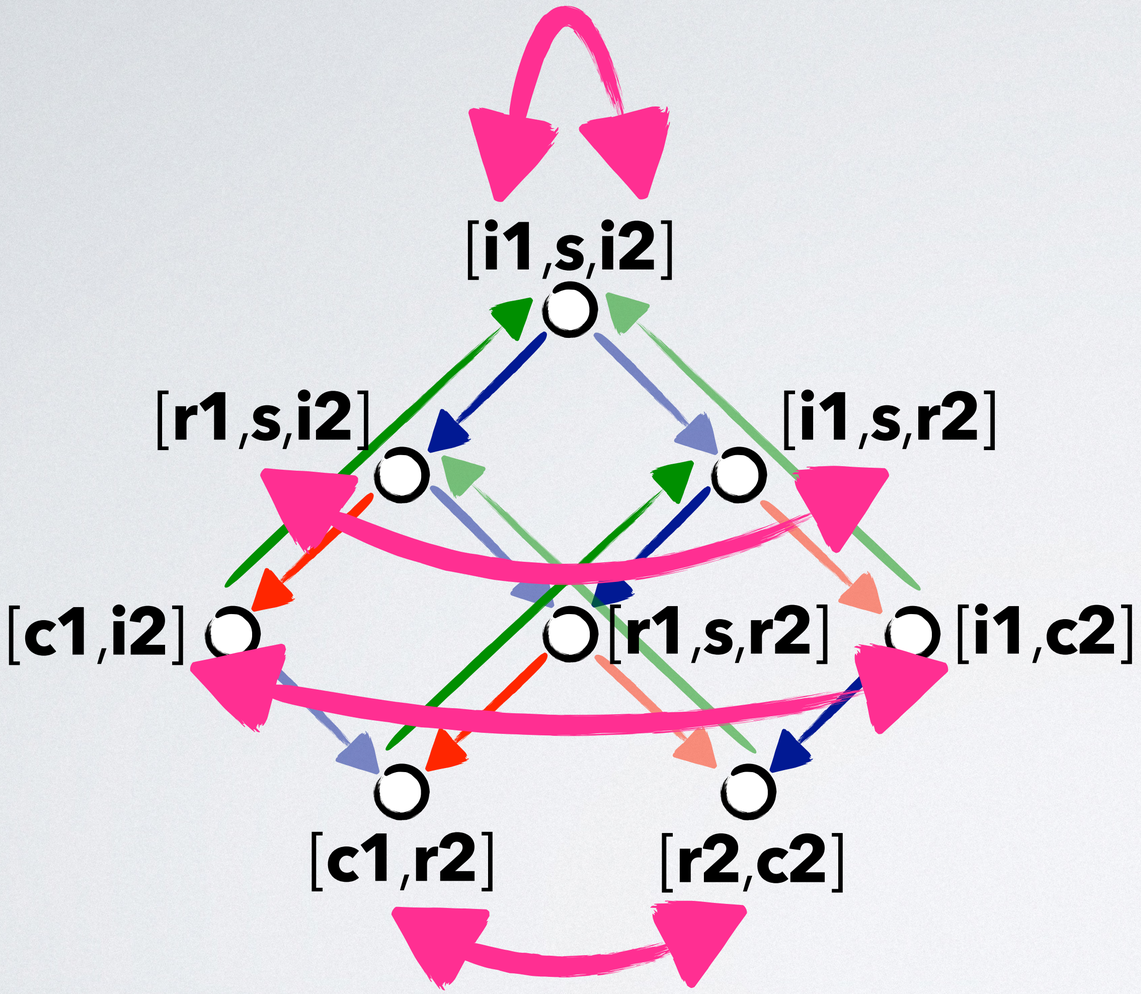


symmetric markings

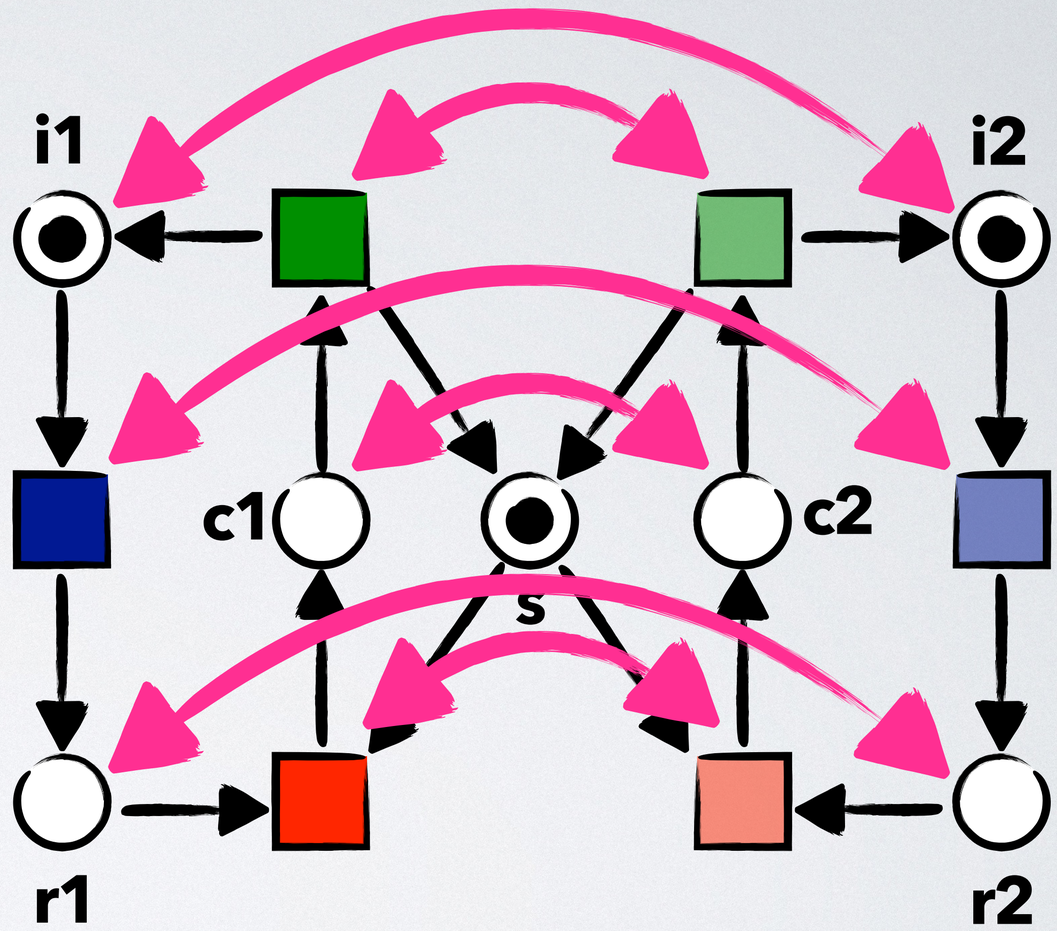


graph automorphisms

2. Symmetry reduction

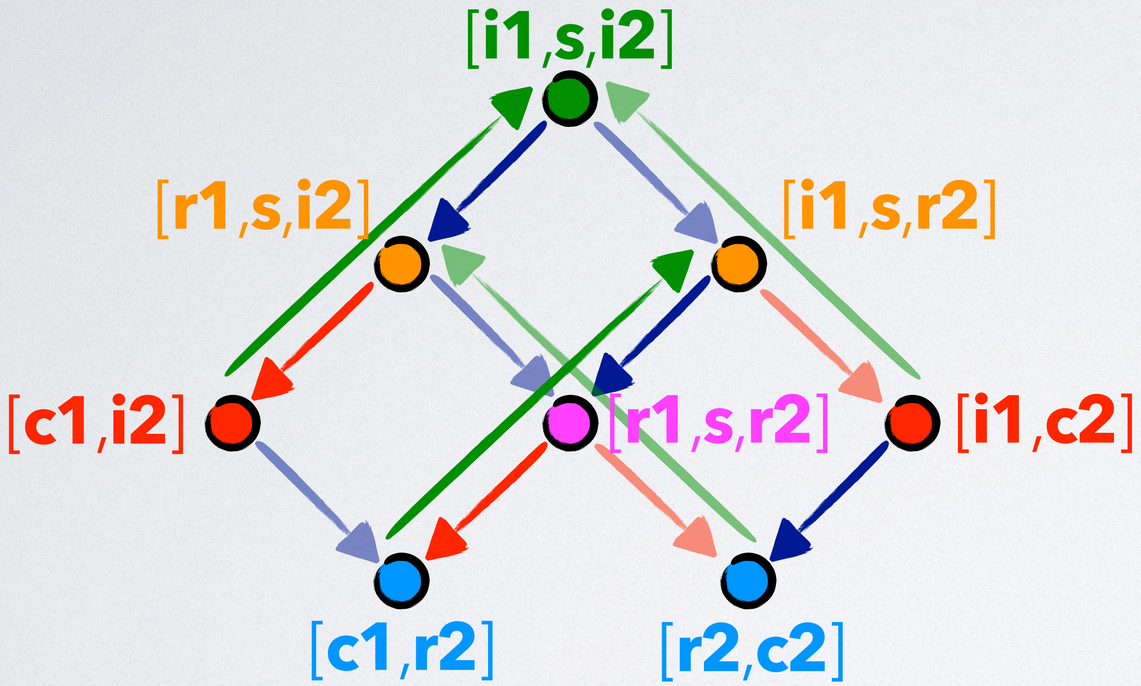


symmetric markings

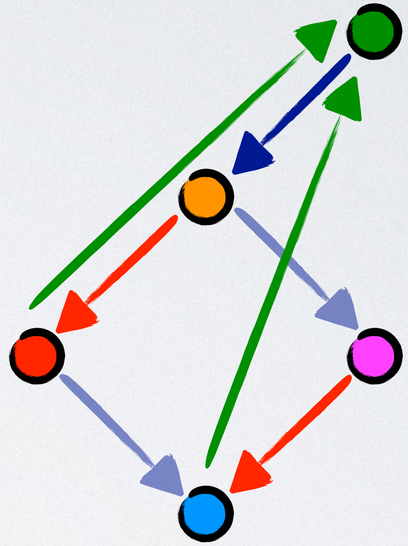


graph automorphisms

2. Symmetry reduction



complete state space



reduced state space

3. State compression

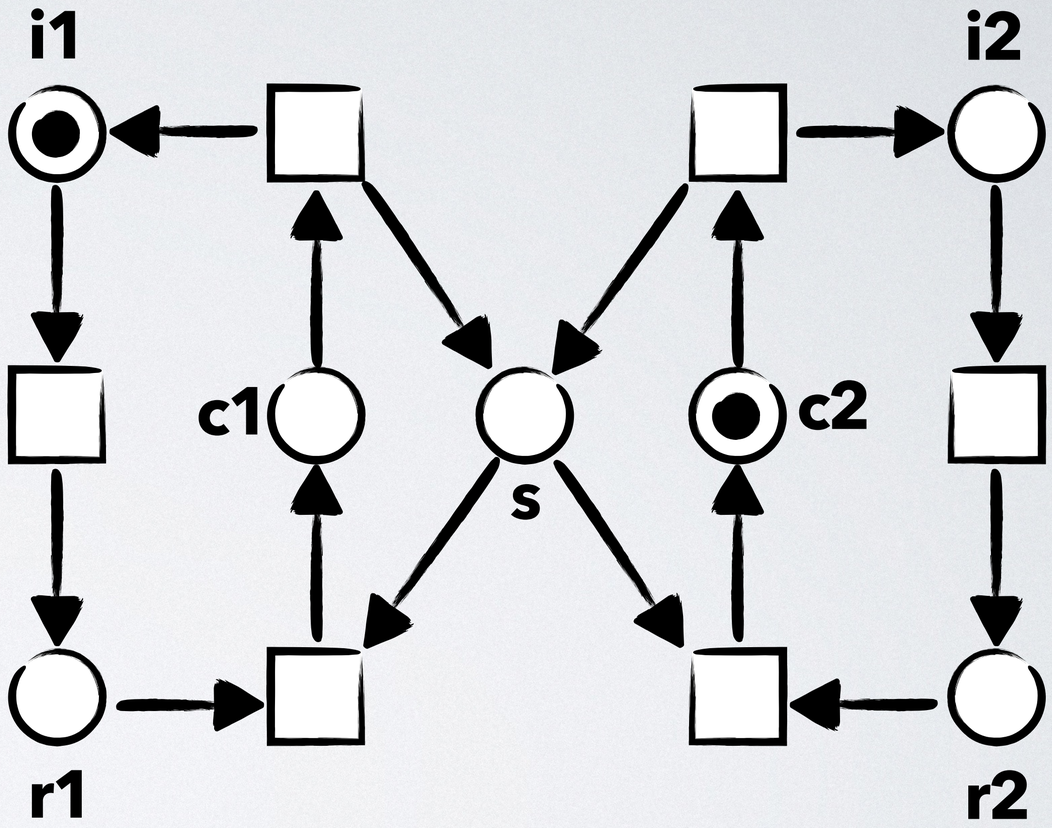
i1 r1 c1 s i2 r2 c2

m = [1,0,0,0,0,0,1]

observation: the value of some places **depend** on the value of others

idea: do not store markings of "**implicit places**"

implementation: Petri net place invariants

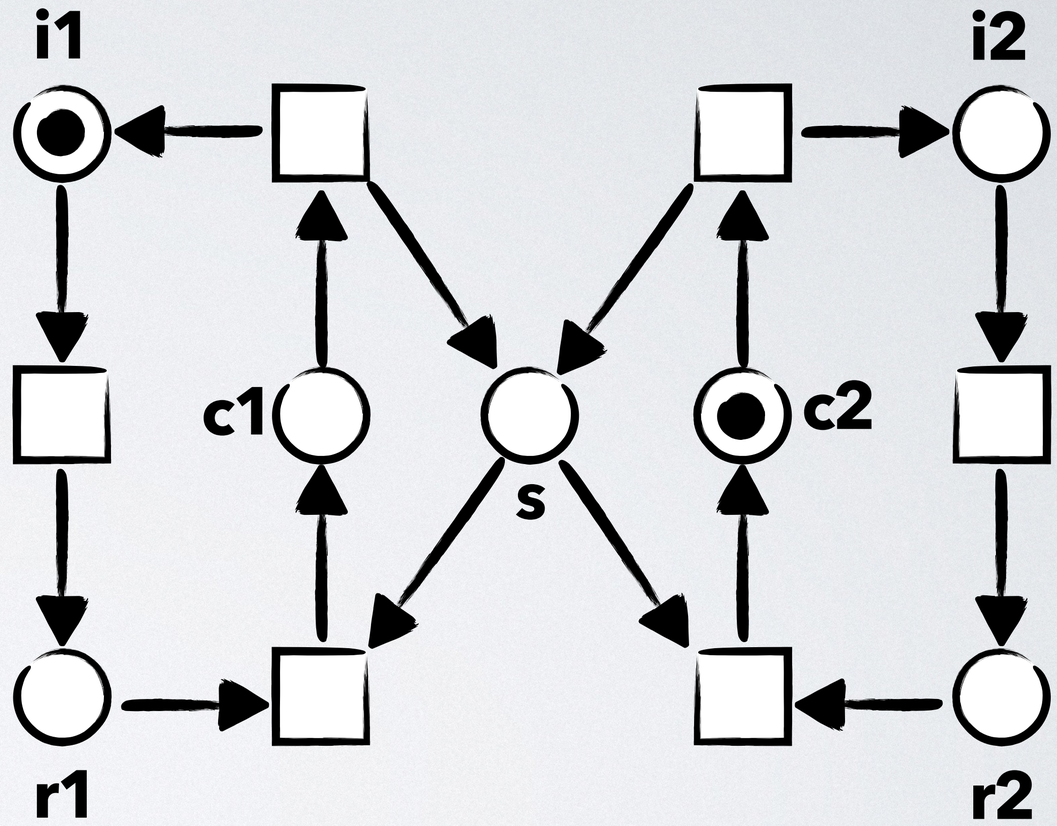


3. State compression

place invariants: for all reachable markings m:

m(**c1**) + m(**c2**) + m(**s**) = 1
m(**i1**) + m(**c1**) + m(**r1**) = 1
m(**i2**) + m(**c2**) + m(**r2**) = 1

i1 r1 c1 s i2 r2 c2
m = [1,0,0,0,0,0,1]



3. State compression

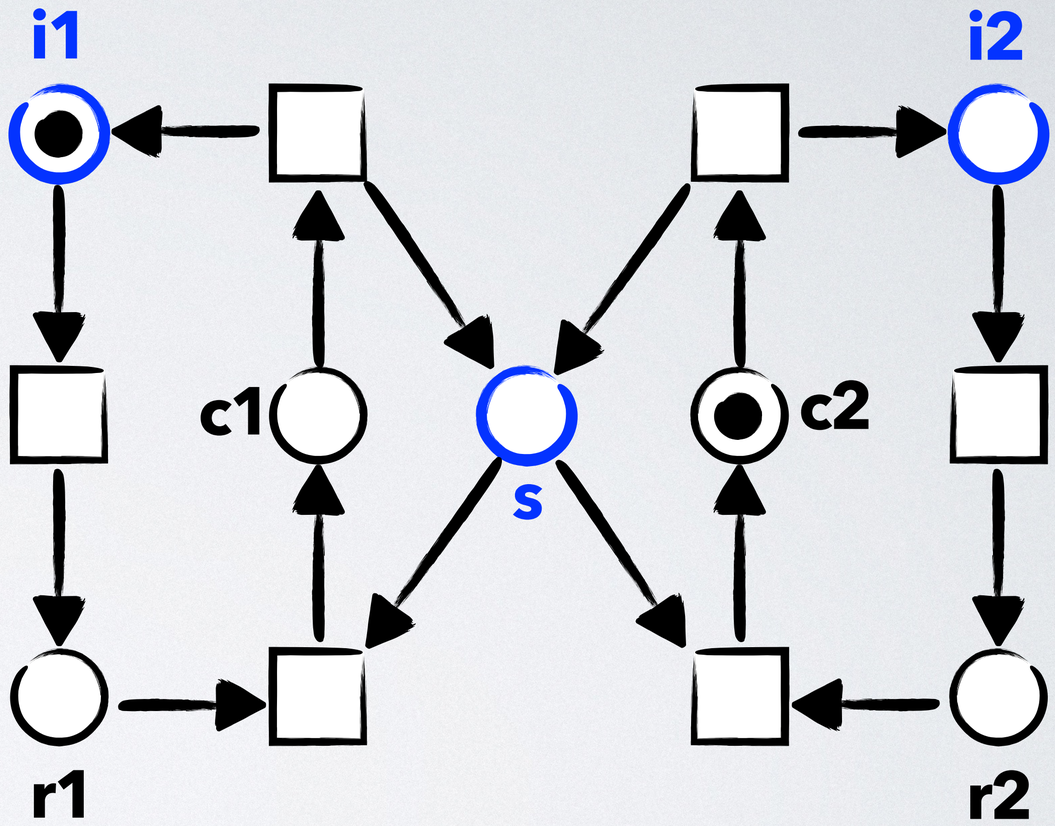
place invariants: for all reachable markings m:

m(**c1**) + m(**c2**) + m(**s**) = 1
m(**i1**) + m(**c1**) + m(**r1**) = 1
m(**i2**) + m(**c2**) + m(**r2**) = 1

therefore:

m(**s**) = 1 − m(**c1**) − m(**c2**)
m(**i1**) = 1 − m(**c1**) − m(**r1**)
m(**i2**) = 1 − m(**c2**) − m(**r2**)

i1 **r1** **c1** **s** **i2** **r2** **c2**
m = [~~1~~,0,0,~~0~~,~~0~~,0,1]



i1, **i2**, and **s** are **implicit**

Reduction techniques: implementation

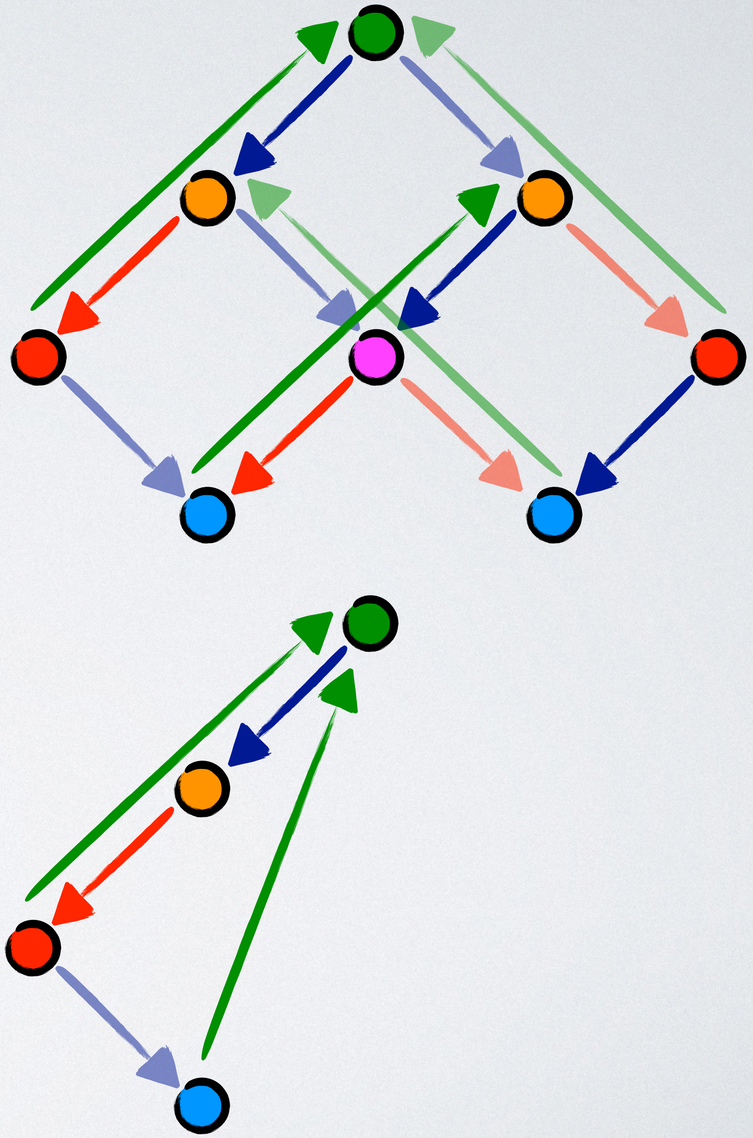
```
   markings = []
   c = compressor()
   s = symmetries()
   search(m0,φ)

   def search(m,φ):
1     check(m,φ)
2     markings.add(c.compress(m))
3     for t in selection(enabled(m)):
4       m' = fire(m,t)
5       if not s.symm(m') in markings:
6         search(m',φ)
```

- **compressor** and **symmetries** preprocess the net
- **compress** creates shorter marking vectors
- **selection** chooses some enabled transitions
- **symm** checks if symmetric marking is already stored

Reduction techniques: summary

- can be combined
- a lot of them work on the Petri net structure
- preprocessing pays off
- some are Petri net exclusive
- decades of research



1 How does a model checker work?

2 How to build an effective model checker?

3 How to use a model checker pragmatically?

Key: Pragmatism

good news

model checking
is decidable

bad news

at a devastating
complexity

- in the end: we only optimize a small depth-first search…
  … that is run billions of times
- we need to understand every aspect
- pragmatism is key: we want a result at all costs

The programming language: C++

manual memory management

type system

low-level optimization

portability

preprocessor

threading

optional object orientation

Used anti-patterns

- heavy use of **preprocessor** (conditional compilation, compile-time decisions, architecture-dependent constants)
- **god objects**
- a lot of **global** variables
- no standard libraries/ generic algorithms (STL, Boost)
- remember: performance, not reusability

```
unsigned int depth_first() {
#ifdef DEPTH_FIRST
    ostream* graphstream = NULL;
    unsigned int i;
    State* NewState;
#ifdef CYCLE
    bool IsCyclic;
    unsigned int silentpath; // nr of consecuti
    Transition** fl;
#endif
    // init initial marking and hash table
    isbounded = 1;
#ifndef CYCLE
    // organize output file for -m/-M/-g/-G par
    if (Globals::gmflg) {
        graphstream = new ofstream(Globals::gra
        if (graphstream->fail()) {
            abort(4, "cannot open graph output
            _cfilename_(Globals::graphfile));
        }
    }
    if (Globals::GMflg) {
        graphstream = &std::cout;
    }
#else
    silentpath = 0;
#endif
#ifndef DISTRIBUTE
#if defined(SYMMETRY) && SYMMINTEGRATION==1
    Trace = new SearchTrace [Globals::Places[0]
#endif
    // initialize hash table
    for (i = 0; i < HASHSIZE; ++i) {
#ifdef BITHASH
        BitHashTable[i] = 0;
#else
        binHashTable[i] = NULL;
#endif
    }
#endif
#ifdef WITHFORMULA
```

Data structures

frontend (parser)

syntactic sugar

class hierarchies

generated code/
standard libraries

focus on simplicity

backend (verification)

canonic representation

flat C-style arrays

simple and user-specified
data types

focus on performance

A remark on complexity

- While solving an EXPSPACE-complete problems, **don’t be afraid of complexity!**
- Preprocessing and optimizations can have dramatic impact.
- NP-complete (SAT-solving) or NP-hard (integer linear programming) problems can be “feasible”.

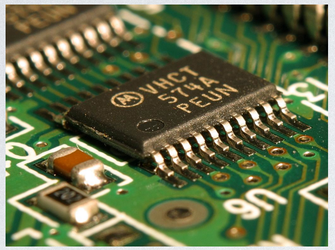
Formalism

compilers

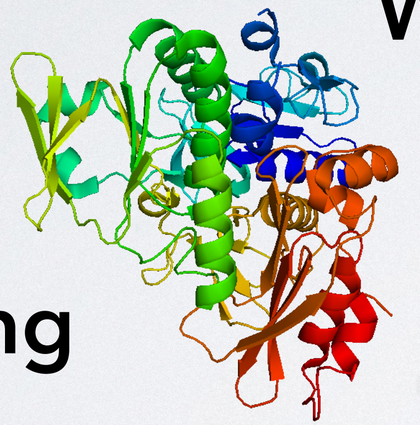
modeling formalisms

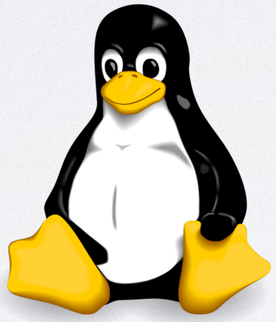
WebSphere software

BPMN™



WS-BPEL





AI planning

- varying feature set
- domain dependent
- moving target, short lifespan

verification formalism

low-level Petri nets

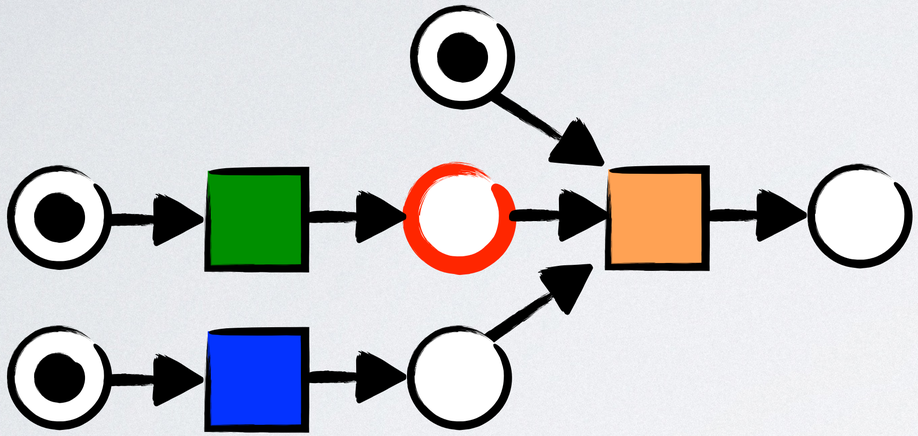
- no structural restrictions
- domain independent
- sound, mature theory

Petri net model checking

- exploit domain knowledge to optimize core functions
  - firing transitions
  - storing markings
- know lifecycle of concepts
- often contradicts object orientation

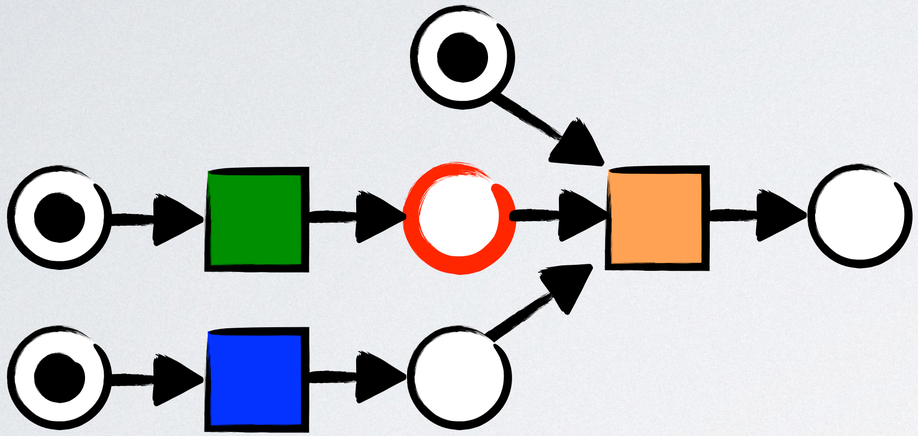
Checking enabledness

orange transition
is disabled



Checking enabledness

orange transition
is disabled



scapegoat
place

Checking enabledness

orange transition
is disabled

blue
transition
fires

scapegoat
place

Checking enabledness

orange transition
is disabled

blue
transition
fires

scapegoat
place

scapegoat place
still unmarked

Checking enabledness

orange transition
is disabled

blue
transition
fires

scapegoat
place

scapegoat place
still unmarked ⇒ orange transition
is still disabled

Storing markings

markings	bit vectors	bin tree
[0,1,3,3,1]	00 01 11 11 01	00 01 11 11 01
[0,1,3,2,0]	00 01 11 10 00	00 01 11 10 00
[0,1,3,2,1]	00 01 11 10 01	00 01 11 10 01

Storing markings

markings	bit vectors	bin tree
[0,1,3,3,1]	00 01 11 11 01	00 01 11 11 01
[0,1,3,2,0]	00 01 11 10 00	00 01 11 10 00
[0,1,3,2,1]	00 01 11 10 01	00 01 11 10 01

Storing markings

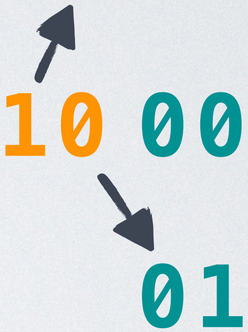
markings	bit vectors	bin tree
[0,1,3,3,1]	00 01 11 11 01	1 01
[0,1,3,2,0]	00 01 11 10 00	00 01 11 10 00
[0,1,3,2,1]	00 01 11 10 01	00 01 11 10 01

Storing markings

markings	bit vectors	bin tree
[0,1,3,3,1]	00 01 11 11 01	1 01
[0,1,3,2,0]	00 01 11 10 00	00 01 11 10 00
[0,1,3,2,1]	00 01 11 10 01	00 01 11 10 01

Storing markings

markings	bit vectors	bin tree
[0,1,3,3,1]	00 01 11 11 01	1 01
[0,1,3,2,0]	00 01 11 10 00	00 01 11 10 00
[0,1,3,2,1]	00 01 11 10 01	01



Know what you need

- sometimes, only a special case is needed:
  - Tarjan algorithm where only TSCCs are needed
  - linear algebra (sparse matrix, only carrier is needed)
- no library can offer this

Optimizations

- profiling, low-level optimizations (caching)
- know your limits: (in LoLA: malloc)

Academic software design

hardly any tenure programmers

seldom seen as important

there is no (paying) customer

no agreed feature set

definitely no coding professionals

university cannot teach experience

coding is never top priority

your thesis is!

very fast scientific progress

moving targets

maintenance is not enforced

once the paper is out, nobody cares

hard to collect/ keep knowledge

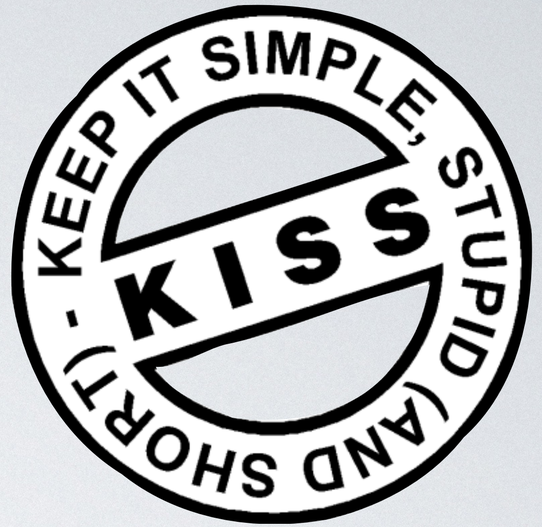
people leave frequently and for good

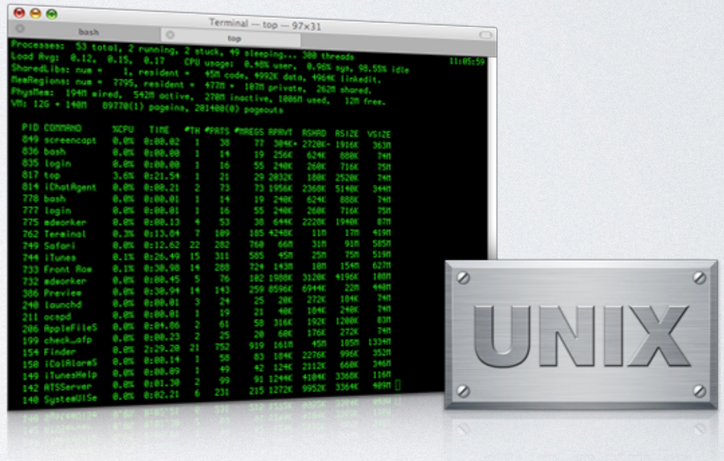
frequently changing staff

2-5 year contracts

Lessons learned

- prototyping and check on real data
- KISS; few dependencies
- split large tools to smaller “brain-sized” units
- test coverage to avoid the fear of breaking everything
- goal orientation:
  no UI, integration via streams





1 How does a model checker work?

2 How to build an effective model checker?

3 **How to use a model checker pragmatically?**

Verification questions: the don’ts

Don’t ask for global states.

Usually, only a **few aspects** (marking of a few places) are relevant.

Don’t ask two things at once.

If the properties affect different parts of the model, ask **separately**.

Don’t use the X operator.

In **distributed systems**, it makes no sense to ask for “the” next state.

Don’t order unordered things.

Usually, **all components** should be correct, not just component #1.

Verification questions: the dos

Ask simple questions.

Sometimes, you don't need **temporal logics** at all.

Make a verification model.

Manipulating the original model may help to ask simpler questions.

Use domain knowledge.

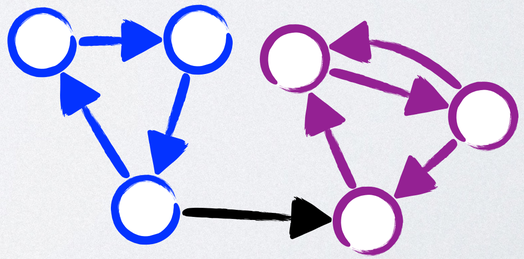
Exploit **implicit assumptions** about the model and the property.

Simple properties

complex	simple	reason
EF φ	reachability of φ	trivial check function
AG φ	¬**EF** ¬φ	again reachability
AGEF φ	check for φ in all TSCC	reachability + TSCC detection

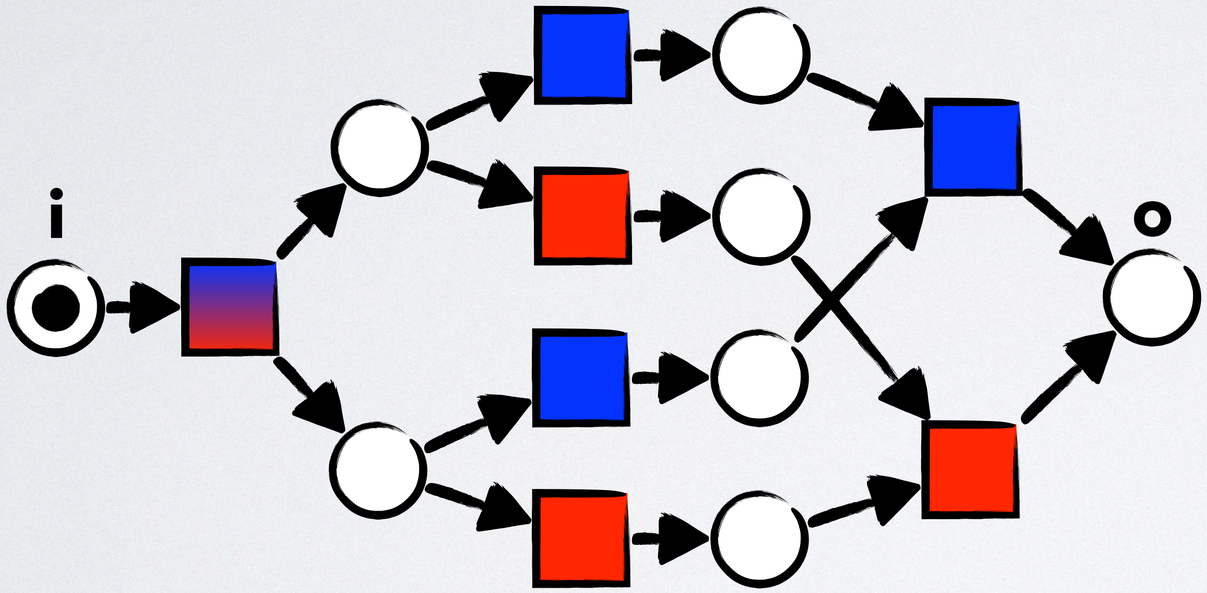
lesson: (hardly) no need
for complex CTL* formulae

compilers will help to find best match



Checking relaxed soundness

Definition: A workflow net is ***relaxed sound*** iff for all transitions exists a terminating firing sequence.



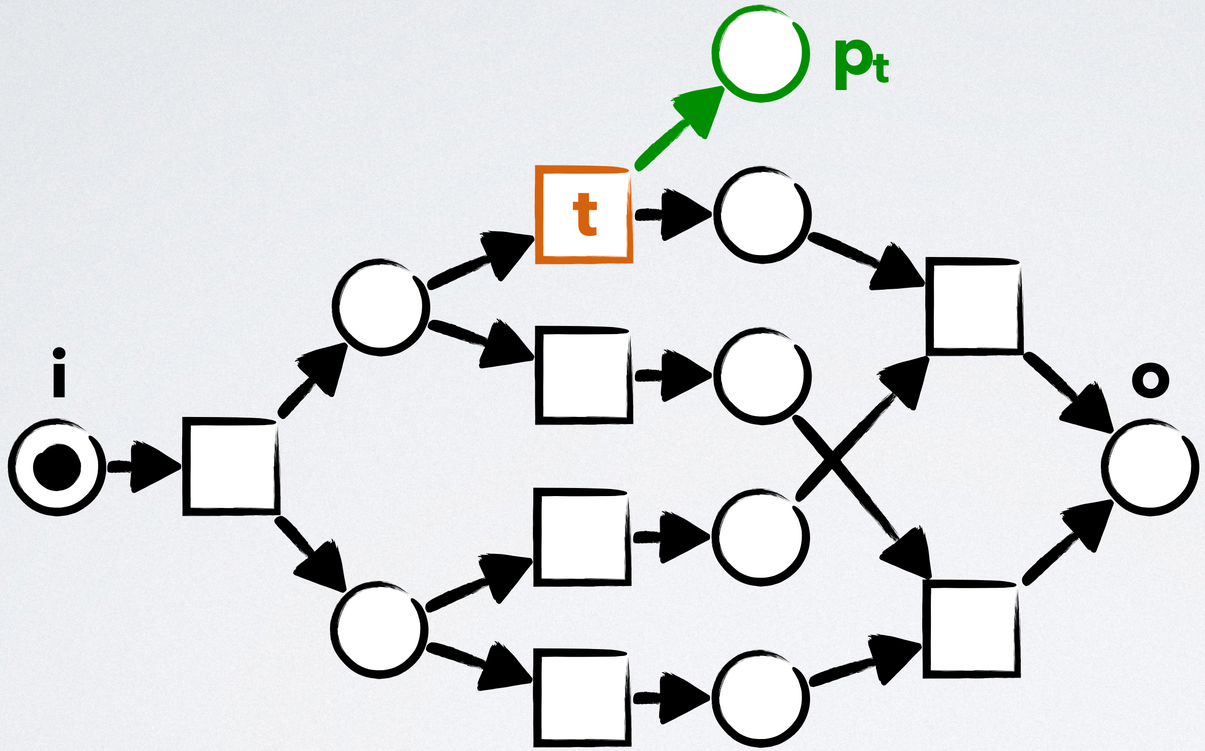
net is relaxed sound

if choices are not synchronized, the net deadlocks

but every transition can fire in a terminating sequence

Net manipulation: algorithm

1. for all transitions **t**: create a net with test place p_t:



2. check for **EF (p_t > 0 ∧ o = 1)**

Net manipulation: summary

- check 8 nets instead of 1

+ checking reachability is simpler than extended CTL

+ each state space is smaller than the original

+ one counterexample for each failure

+ parallelizable

Checking soundness

Definition: A workflow net is ***sound*** iff (1) the final marking [o] is always reachable (2) [o] is the only marking with tokens on place o, and (3) no transition is dead.

+ **domain knowledge**: nets are free-choice

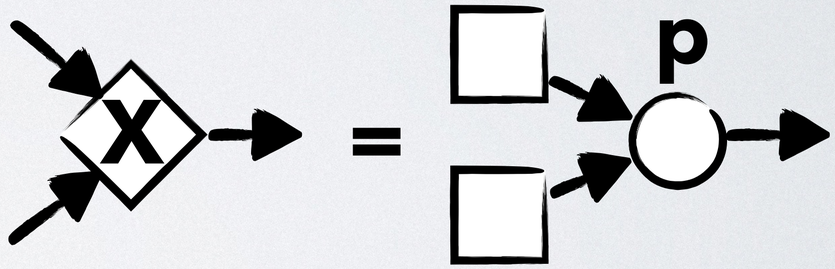
Definition: A free-choice workflow net is ***sound*** iff
(1) AGEF [o] holds,
(2) the net is 1-safe, and
(3) no transition is dead.

Checking soundness

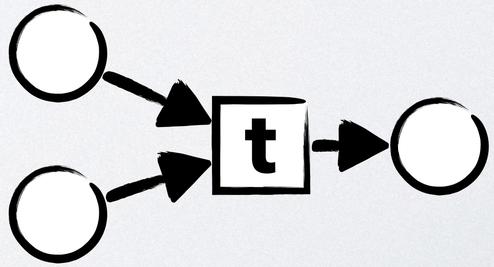
Definition: A free-choice workflow net is ***sound*** iff (1) AGEF [o] holds, (2) the net is 1-safe, and (3) no transition is dead.

(1) check if the marking [o] is reachable in all TSCCs.

(2) for every join-place p, check if m with m(p)>1 is reachable.



(3) for every transition t, check if •t is reachable



Pragmatic use: summary

- help the model checker help you
- reformulate your question
- many small state spaces are better than one large

1

How does a model checker work?

2

How to build an effective model checker?

3

How to use a model checker pragmatically?

Conclusions

Take home points

- model checking = theory + practice + **pragmatism**
- **academic software design** is a discipline on its own
- asking the **right question(s)** is crucial

Pragmatic model checking: from theory to implementations

Niels Lohmann

Universität Rostock
Traditio et Innovatio

Copyrights

Atombombentest Romeo, public domain United States Department of Energy
http://commons.wikimedia.org/wiki/File:Castle_Romeo.jpg

Raspberry Pi, CC-BY-SA Jwrodgers
http://commons.wikimedia.org/wiki/File:RaspberryPi.jpg

apoptosis inducing factor, GPL
http://en.wikipedia.org/wiki/File:Apoptosis_inducing_factor.png

Tux, by lewing@isc.tamu.edu
http://en.wikipedia.org/wiki/File:Tux.png

Integrated Circuit, public domain
http://commons.wikimedia.org/wiki/File:Chip.jpg

Websphere software logo, public domain
http://commons.wikimedia.org/wiki/File:Websphere_logo.png

BPMN logo, copyright Object Management Group (OMG)