Abstraction and Refinement

State-space explosion

In practice, model-checking encounters the problem of state-space explosion:

   due to data: var with n bits $\rightarrow 2^n$ states

   due to concurrency: n parallel components with $n!$ different orderings

Countermeasures:

   Compression: efficient representations (e.g. BDDs)

   Reduction: find a simpler, equivalent problem

   Abstraction: identify and ignore "unimportant" information

Example 1 (loop)

Consider the following program with three numeric variables x, y, z.

```
ℓ₁:   y = x+1;

ℓ₂:   z = 0;

ℓ₃:   while (z < 100) z = z+1;

ℓ₄:   if (y < x) error;
```

Question: Is the error location reachable?

Example 2 (Sorting)

Another program with three numeric variables x, y, z.

$\ell_1:$ `if x > y then  swap x,y  else  skip;`

$\ell_2:$ `if y > z then  swap y,z  else  skip;`

$\ell_3:$ `if x > y then  swap x,y  else  skip;`

$\ell_4:$ `skip`

Assumption: initially, x, y, z are all different

Question: Are x, y, z sorted in ascending order when reaching ℓ_4?

Example 3 (Device driver)

C code for Windows device driver

Operations on a semaphor: lock, release

Lock and release must be used alternatingly

Abstraction

Idea: throw away (abstract from) "unimportant" information

Handling *infinite* state spaces
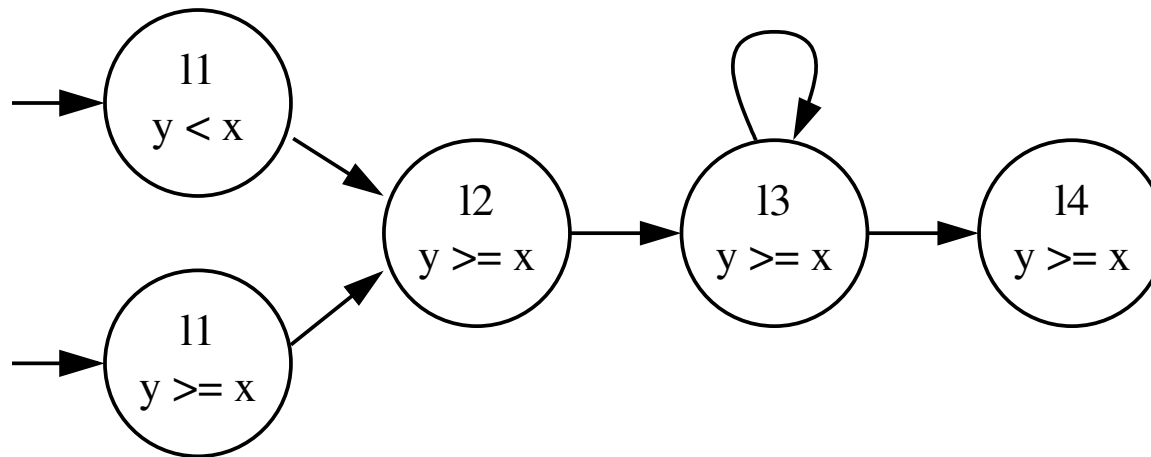
Reduce (large) finite problems to smaller ones

Alternative point of view: merge "equivalent" states

Example 1

Omit concrete values of x, y, z; retain only the following information: program counter, predicate $y < x$
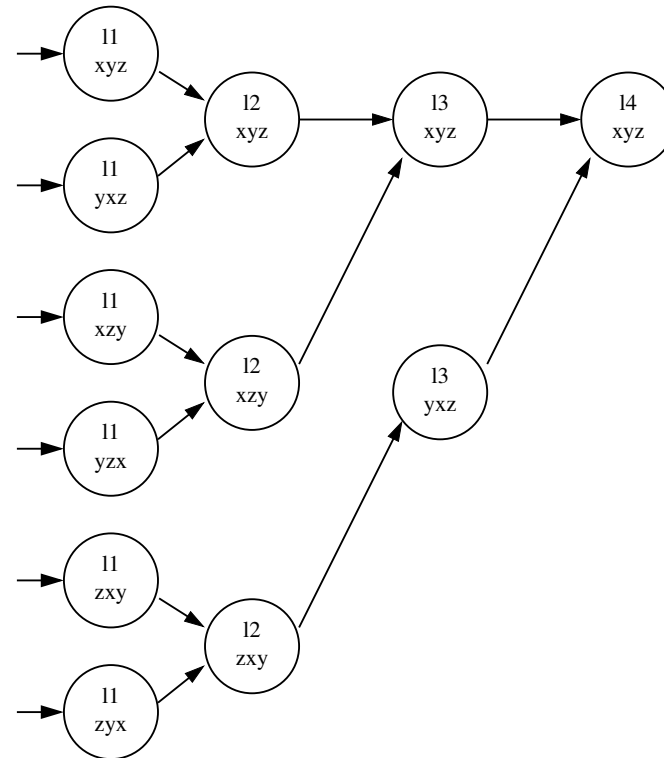
Resulting (abstract) Kripke structure:

Result: ℓ_4 is reachable only with $y \geq x$; the error will not happen.

Example 2

Omit concrete values of `x`, `y`, `z`; retain only program counter and permutation of x, y, z

Result: ℓ_4 is reachable only with *xyz*; no error.

Questions: What is the logical relation between the original programs and their abstract versions? What do the abstract versions really say about the original programs?

In Example 1, the error is unreachable in both the original and the abstract version.

However, in Example 1, the original structure terminates but the abstract version does not.

Which conditions must hold for the abstract structure in order to draw meaningful conclusions about the original structure?

Simulation

Let $\mathcal{K}_1 = (S, \rightarrow_1, s_0, AP, \nu)$ and $\mathcal{K}_2 = (T, \rightarrow_2, t_0, AP, \mu)$ be two Kripke structures (S, T are possibly *infinite*), and let $H \subseteq S \times T$ be a relation.

H is called a simulation from $\mathcal{K}_1$ to $\mathcal{K}_2$ iff
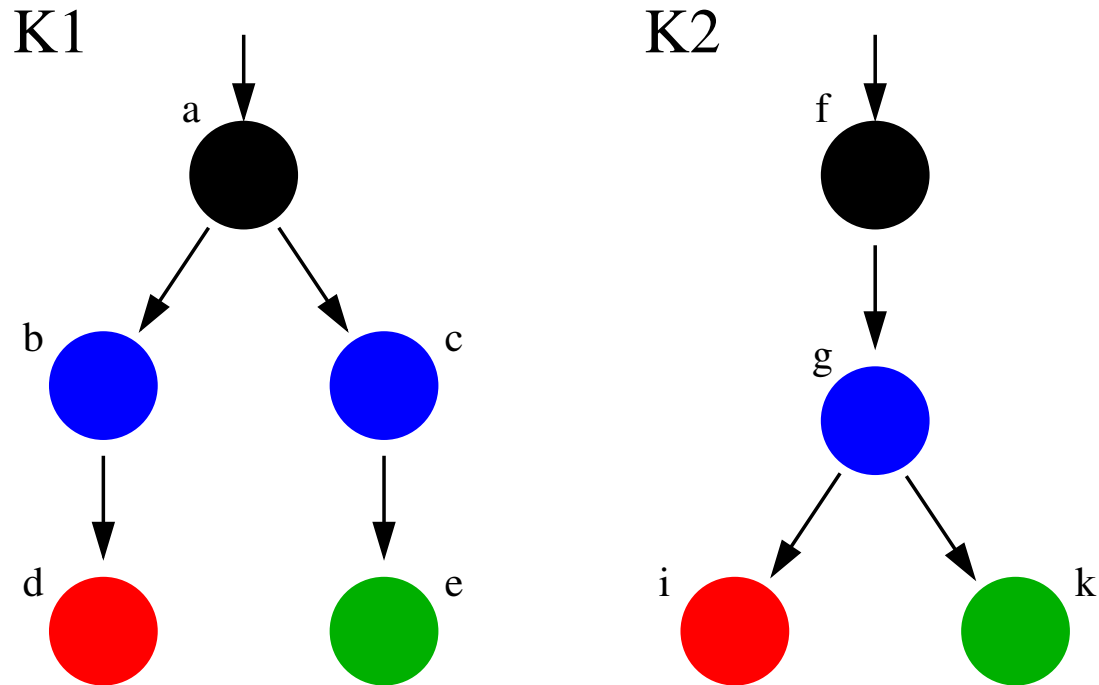
(i) $(s_0, t_0) \in H$;

(ii) for all $(s, t) \in H$ we have: $\nu(s) = \mu(t)$;

(iii) if $(s, t) \in H$ and $s \rightarrow_1 s'$, then there exists t' such that $t \rightarrow_2 t'$ and $(s', t') \in H$.

We say: $\mathcal{K}_2$ simulates $\mathcal{K}_1$ (written $\mathcal{K}_1 \leq \mathcal{K}_2$) if such a simulation H exists.

Intuition: $\mathcal{K}_2$ can do anything that is possible in $\mathcal{K}_1$.

$\mathcal{K}_2$ simulates $\mathcal{K}_1$ (with $H = \{(a,f),(b,g),(c,g),(d,i),(e,k)\}$).

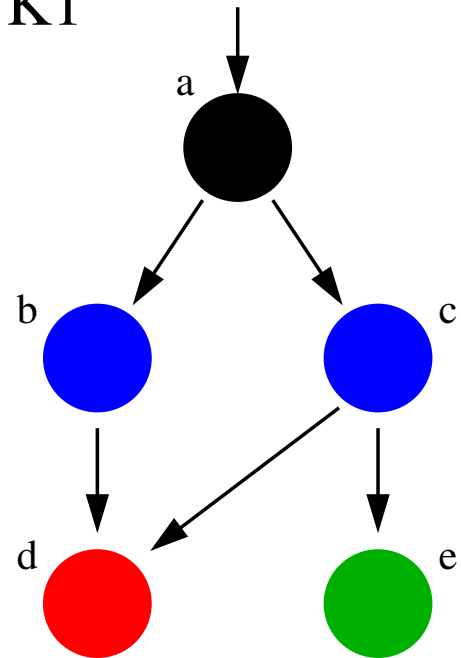However,: $\mathcal{K}_1$ does *not* simulate $\mathcal{K}_2$!

Bisimulation

A relation H is called a bisimulation between $\mathcal{K}_1$ and $\mathcal{K}_2$ iff H is a simulation from $\mathcal{K}_1$ to $\mathcal{K}_2$ and $\{\,(t, s) \mid (s, t) \in H\,\}$ is a simulation from $\mathcal{K}_2$ to $\mathcal{K}_1$.

We say: $\mathcal{K}_1$ and $\mathcal{K}_2$ are bisimilar (written $\mathcal{K}_1 \equiv \mathcal{K}_2$) iff such a relation H exists.
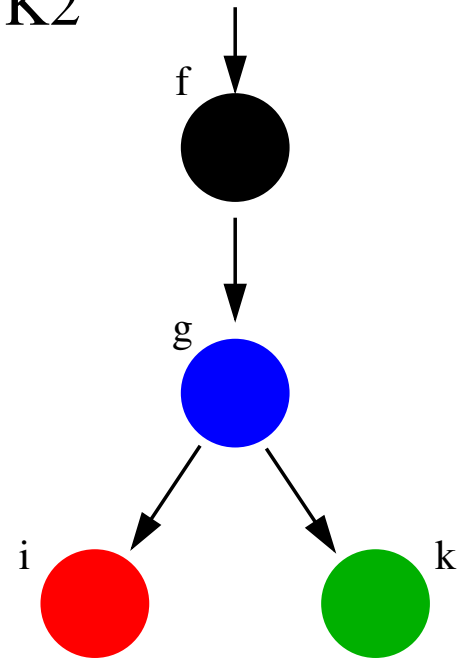
Careful: In general, $\mathcal{K}_1 \le \mathcal{K}_2$ and $\mathcal{K}_2 \le \mathcal{K}_1$ do *not* imply $\mathcal{K}_1 \equiv \mathcal{K}_2$!

(Bi-)Simulation and model checking

Let $\mathcal{K}_1 \leq \mathcal{K}_2$ and ϕ an LTL formula. Then:

$\mathcal{K}_2 \models \phi$ implies $\mathcal{K}_1 \models \phi$ (for universal model checking).

The other direction is not guaranteed!

Existential abstraction

Let $\mathcal{K} = (S, \rightarrow, r, AP, \nu)$ be a Kripke structure (concrete structure).

Let $\approx$ be an equivalence relation on S such that for all $s \approx t$ we have $\nu(s) = \nu(t)$ (we say: $\approx$ respects ν).

Let $[s] := \{\, t \mid s \approx t \,\}$ denote the equivalence class of s; $[S]$ denotes the set of all equivalence classes.

The abstraction of S w.r.t. $\approx$ denotes the structure $\mathcal{K}' = ([S], \rightarrow', [r], AP, \nu')$, where
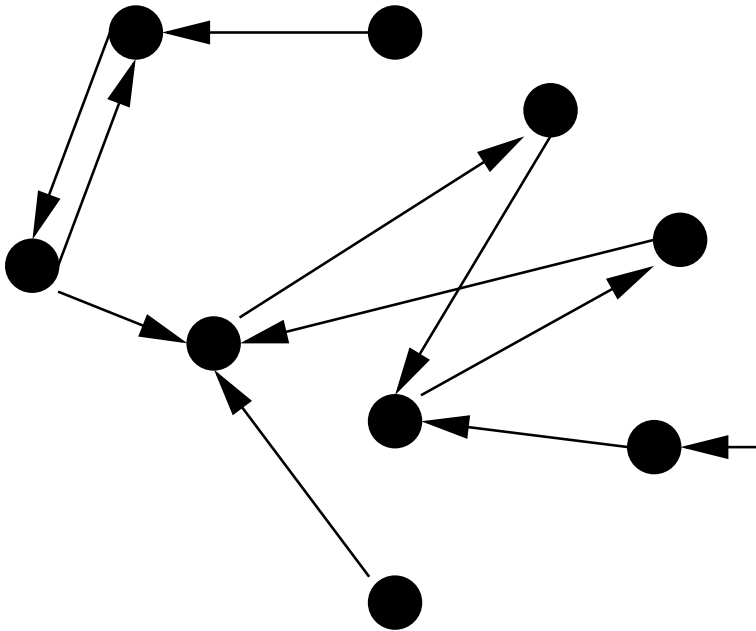
$[s] \rightarrow' [t]$ for all $s \rightarrow t$;

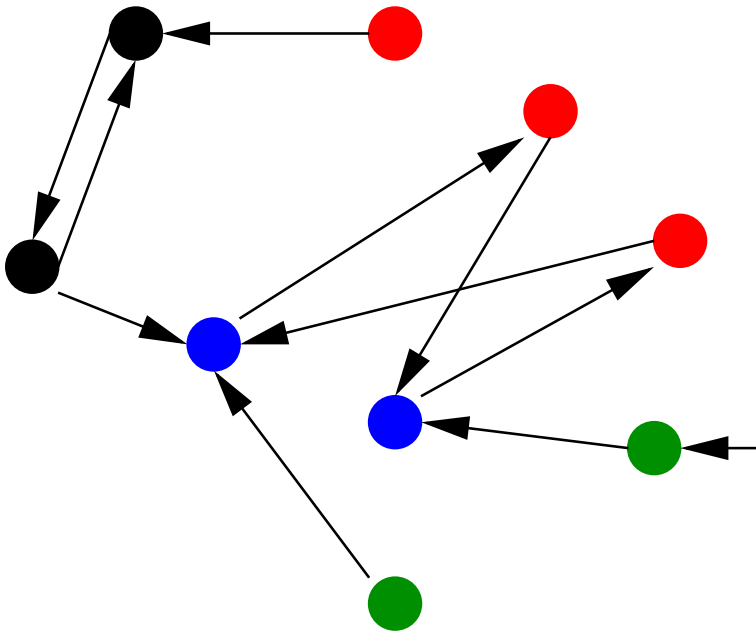$\nu'([s]) = \nu(s)$ (this is well-defined!).
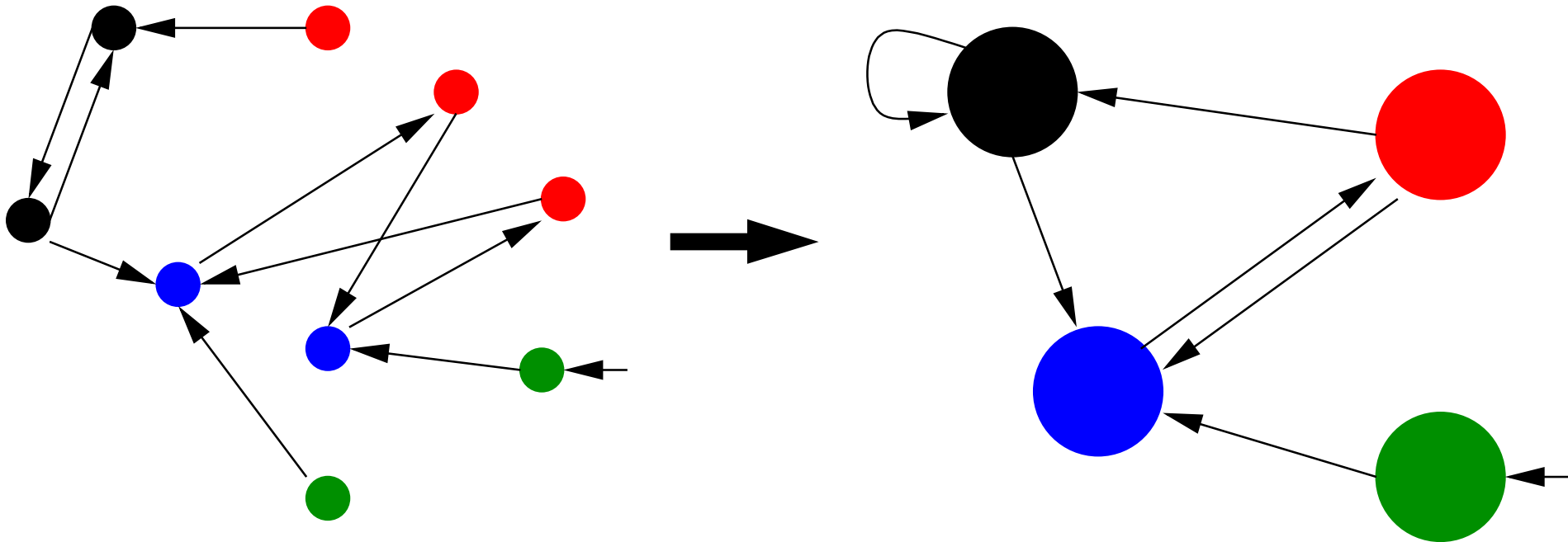
Example

Consider the Kripke structure below:

States partitioned into equivalence classes:
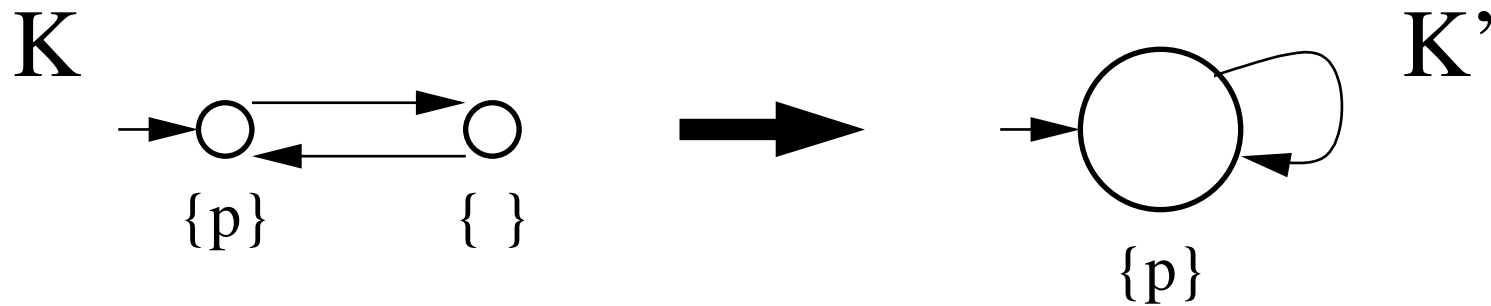
Abstract structure obtained by quotienting:

Let $\mathcal{K}'$ be a structure obtained by abstraction from $\mathcal{K}$.

Then $\mathcal{K} \leq \mathcal{K}'$ holds (simulation relation: $\{\,(s, [s]) \mid s \in S\,\}$)

Thus, if $\mathcal{K}'$ satisfies some LTL formula, so does $\mathcal{K}$.

What happens if $\approx$ does *not* respect ν?

$$K \qquad\qquad\qquad K'$$

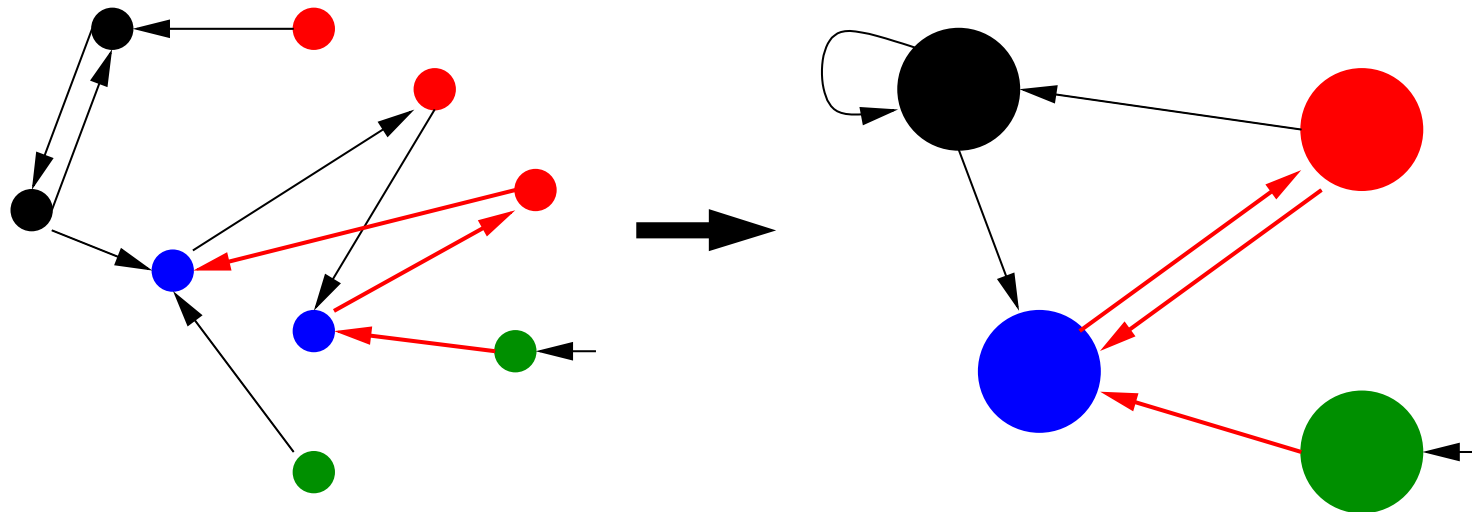$$\{p\} \qquad \{\} \qquad\qquad \{p\}$$

Then $\mathcal{K} \not\preceq \mathcal{K}'$ does not hold.

Example: The abstraction satisfies $\mathbf{G}\,p$, the concrete system does not.

Abstraction gives rise to additional paths in the system:
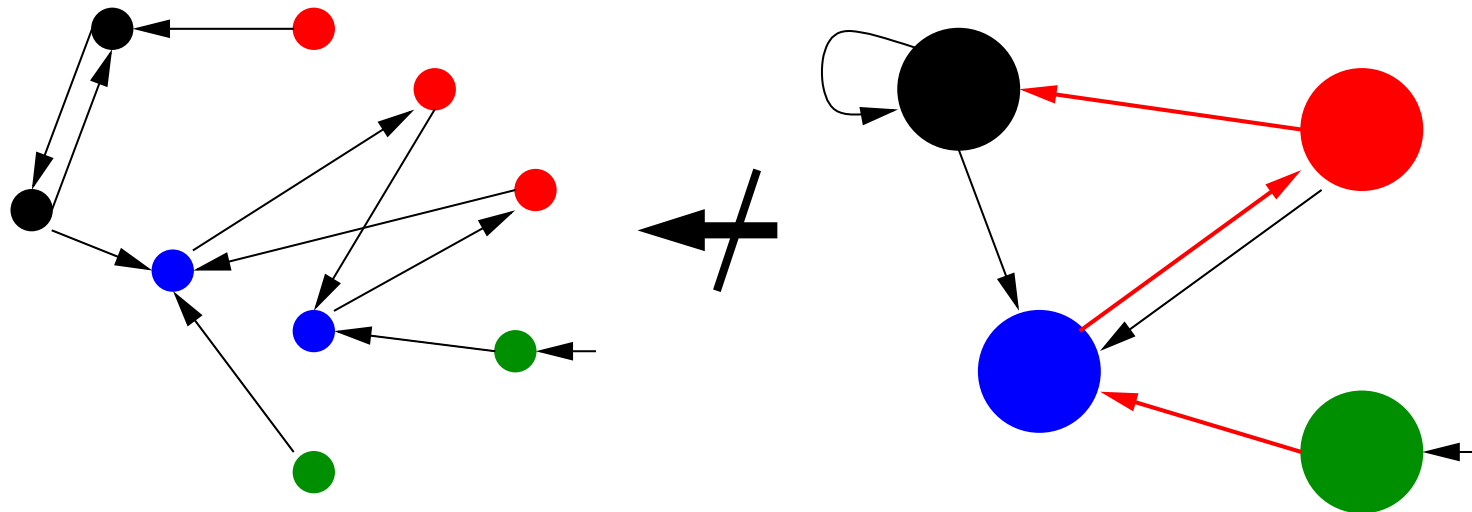
Every concrete run has got a corresponding run in the abstraction . . .

Abstraction gives rise to additional paths in the system:

. . . but not every abstract run has got a corresponding run in the concrete system.

Suppose that $\mathcal{K}' \not\models \phi$, where ρ is a counterexample.

Check whether there is a run in $\mathcal{K}$ that "corresponds" to ρ.

If yes, then $\mathcal{K} \not\models \phi$.

If no, then we can use ρ to refine the abstraction; i.e. we remove some equivalences from the relation H, introducing additional distinct states in $\mathcal{K}'$ so that ρ disappears.

The refinement can be repeated until a definite answer for $\mathcal{K} \models \phi$ (positive or negative) can be determined. This technique is called counterexample-guided abstraction refinement (CEGAR) [Clarke et al, 1994].

The abstraction-refinement cycle

Input: $\mathcal{K}, \phi$
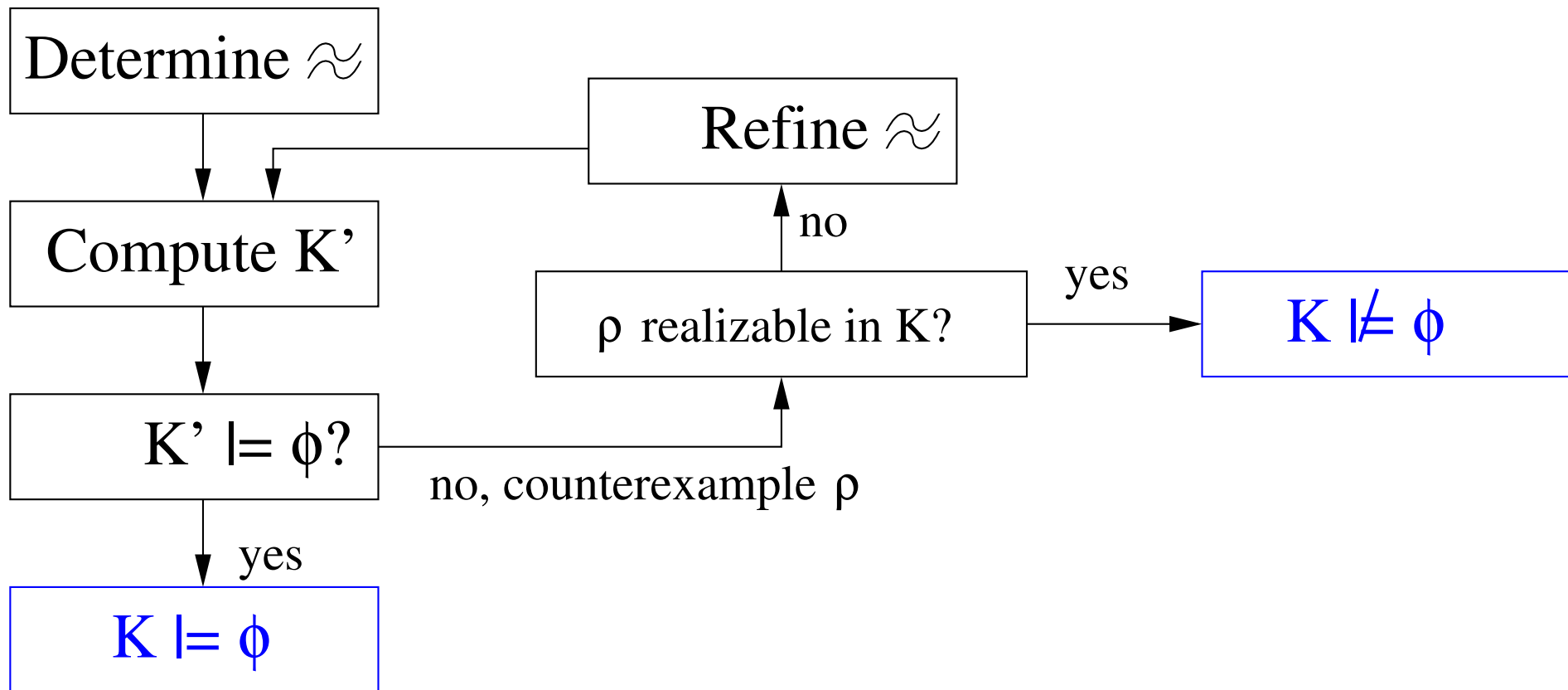
Simulation of ρ

Problem: Given a counterexample ρ, is there a run corresponding to ρ in $\mathcal{K}$?
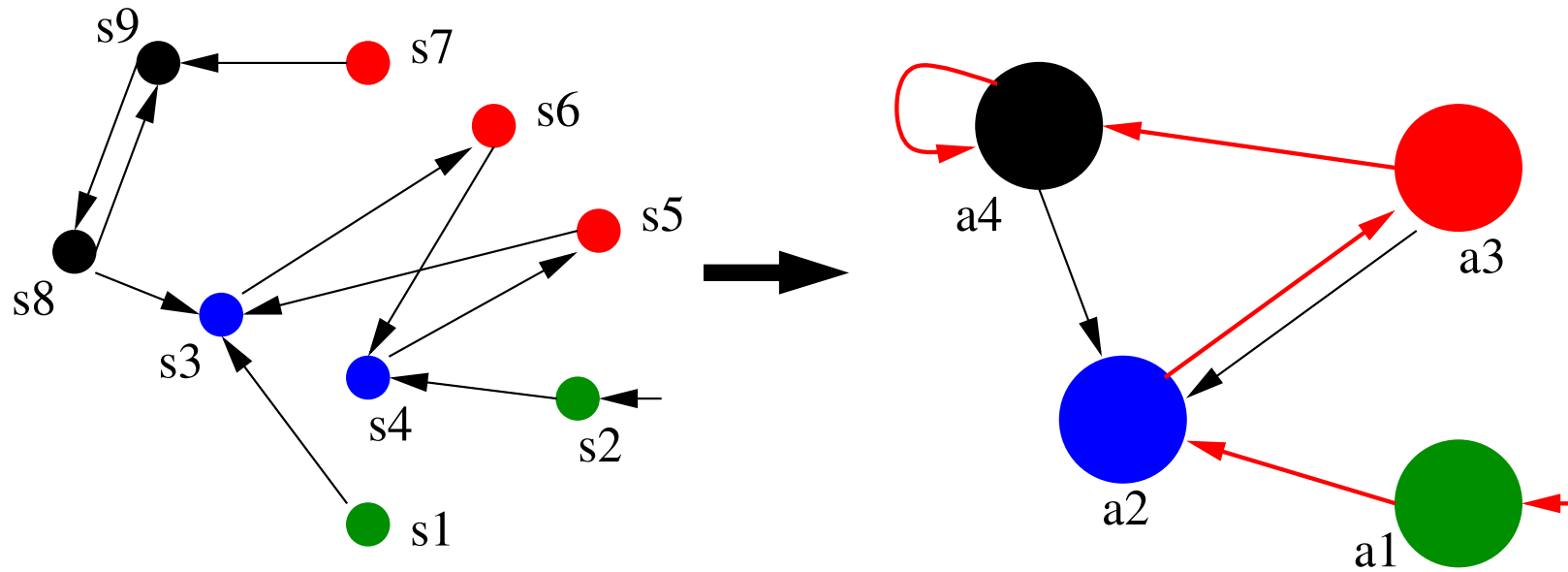
Solution: "Simulate" ρ on $\mathcal{K}$.

Remark: Any counterexample ρ can be partitioned into a finite stem and a finite loop, i.e. $\rho = w_S w_L^\omega$ for suitable w_S, w_L.

Case distinction: The simulation may fail in the stem or in the loop.

Example 1: G ¬*black*

Abstraction yields a counterexample with stem $a_1 a_2 a_3 a_4$ and loop a_4.

Simulating the stem

Let $w_S = b_0 \cdots b_k$.

Start with $S_0 = \{r\}$. (We have $b_0 = [r]$.)

For $i = 1, \ldots, k$, compute $S_i = \{ t \mid t \in b_i \wedge \exists s \in S_{i-1} : s \to t \}$.

If $S_k \neq \emptyset$, then there is a concrete correspondence for w_S.

If $S_k = \emptyset$: Find the smallest index ℓ with $S_\ell = \emptyset$: The refinement should distinguish the states in $S_{\ell-1}$ and those $b_{\ell-1}$-states that have immediate successors in b_ℓ.

Example: $w_S = a_1 a_2 a_3 a_4$

$$S_0 = \{s_2\}, \qquad S_1 = \{s_4\}, \qquad S_2 = \{s_5\}, \qquad S_3 = \emptyset.$$
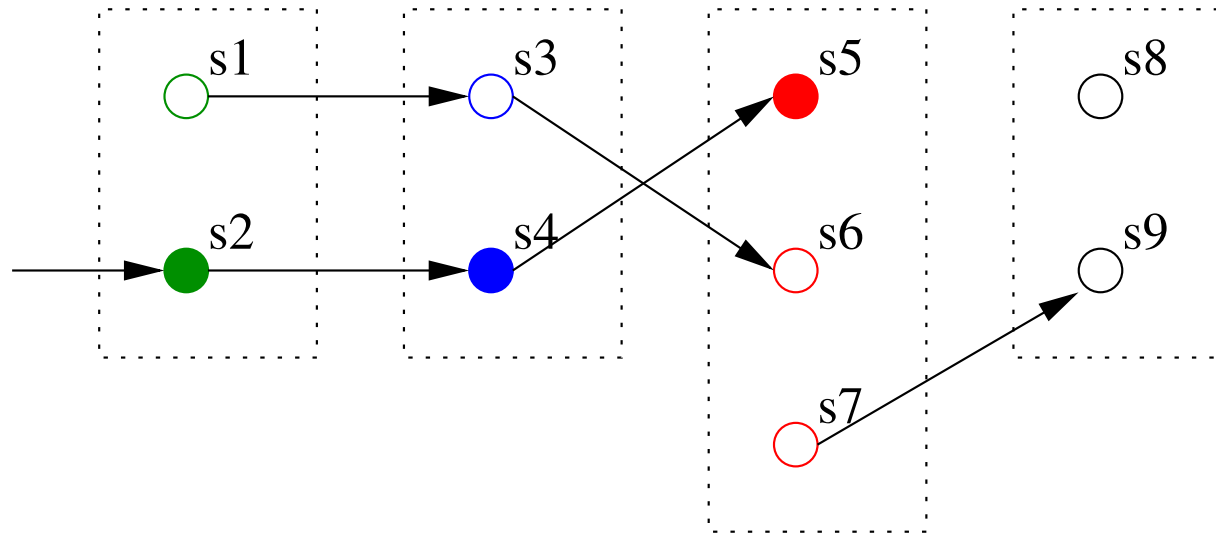
In the next refinement, s_5 and s_7 must be distinguished.

Possible new equivalence classes: $\{s_5, s_6\}, \{s_7\}$ or $\{s_5\}, \{s_6, s_7\}$.

Next try: **G** ¬*black* with refinement
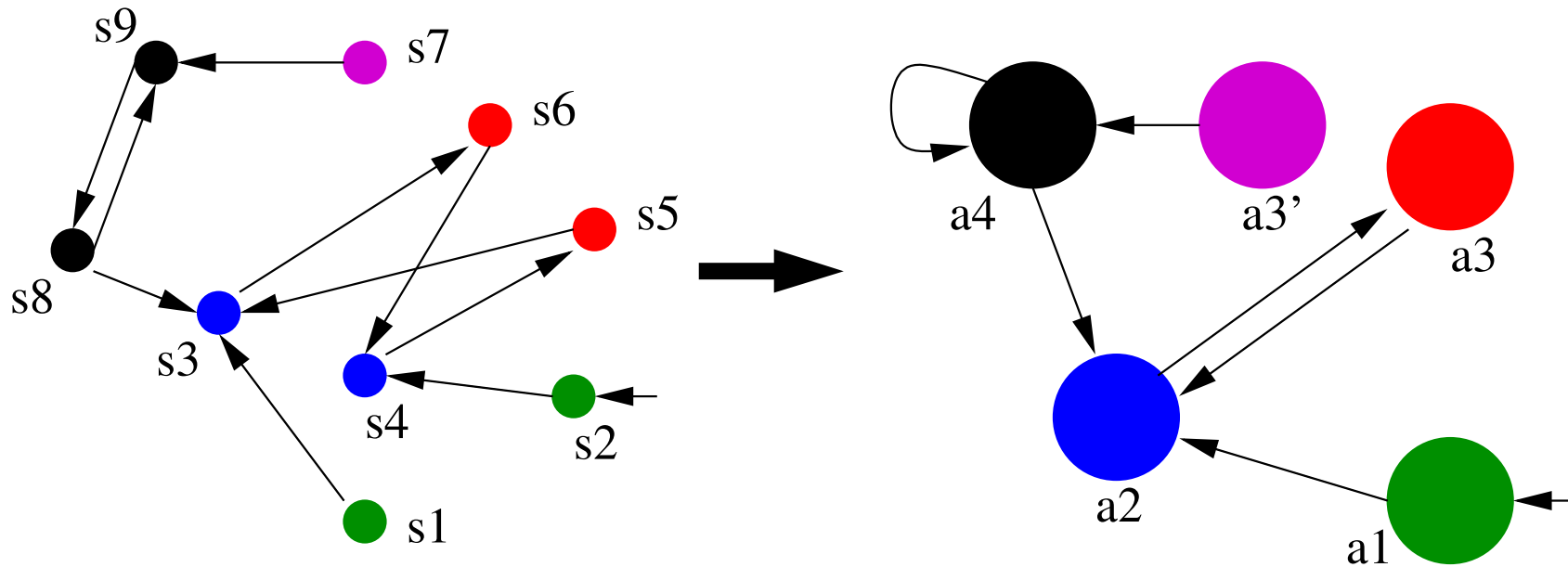

The new abstraction does not yield any counterexample; therefore, **G** ¬*black* also holds in the concrete system.
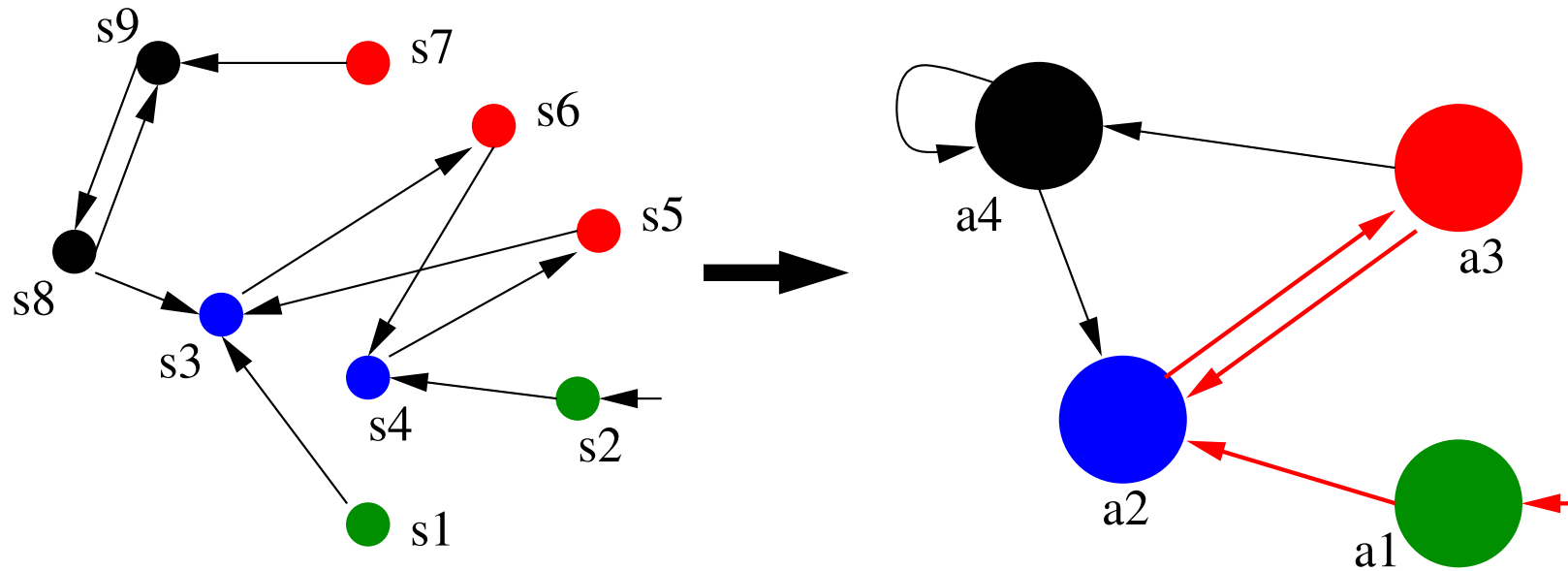
The abstraction yields a counterexample with stem $a_1 a_2$ and loop $a_3 a_2$.

Simulating a loop

Assume $w_S = b_0 \cdots b_k, \quad w_L = c_1 \cdots c_\ell$

w_S is simulated as before, however w_L may have to be simulated multiple times.

Let m be the size of the smallest equivalence class in w_L:

$$m = \min_{i=1,\ldots,\ell} |c_i|$$

Then we simulate the path $w_S w_L^{m+1}$; doing so, either the simulation will fail, or we will discover a real counterexample.

Refinement: same as before.

Example: $w_S = a_1 a_2$, $w_L = a_3 a_2$, $m = 2$
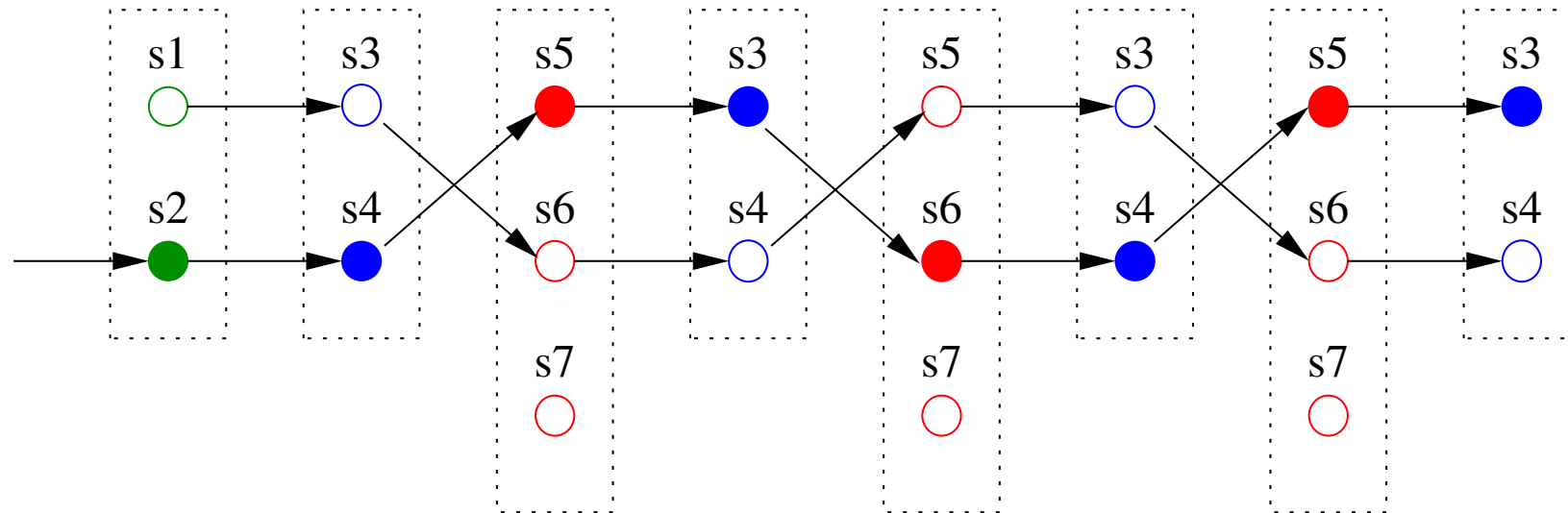
The simulation succeeds because there is a loop around s_4.

Thus, there is a real counterexample, so $\mathcal{K} \not\models \phi$.

CEGAR: first discussed in 1994

Since about 2000: software model checkers using CEGAR

SLAM / SDV (Microsoft Research)

Blast (Standford / EPFL)

Magic (CMU)

"canonical application:" device drivers (many lines of code but few data dependencies)

Example: Device driver verification

(example taken from Ball, Rajamani 2000)

Input: C program (device driver)

Safety property given by a finite automaton,
e.g. "acquire/release" operations must be in proper order:
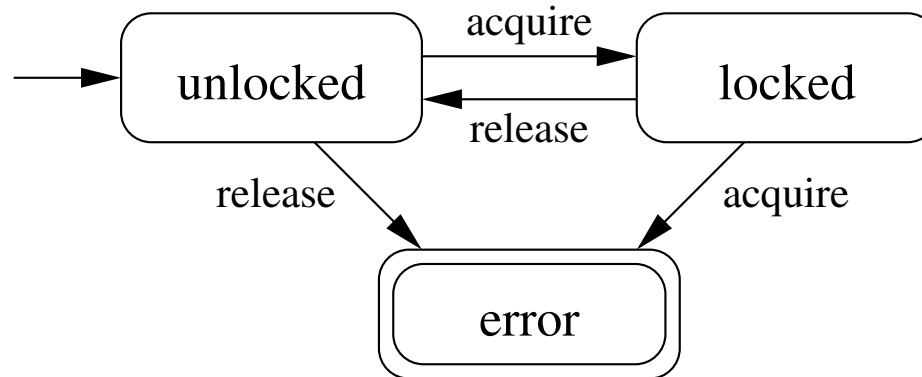
More complicated property: up to 30 states

Preprocessing: embed the automaton into the C program using assertions

Abstraction in Boolean Programs

Set of predicates *P* (boolean expressions, e.g. `x==y`)

equivalence classes distinguish lines of code and fulfilled predicates

Implementation: Translate the C program into a "boolean program", where all variables are of type Boolean.

BDD-based model checker, tests for reachability of failed assertions and returns an execution (of the boolean program) that reaches the failed assertion.

Generating the predicates

Initial predicates: State of the property automaton.

Every instruction of the C program translated into an assignment of the predicate variables.

Compute the weakest precondition for validity or invalidity of each predicate after the instruction.

Find out which predicates imply the weakest precondition.

Example: predicate $p := (x = y)$, instruction $x = x + 1$

weakest precondition: $x + 1 = y$

p implies $\neg(x + 1 = y)$.

Neither p nor $\neg p$ imply $x + 1 = y$.

$\Rightarrow$ translate the statement into:

```
if p then p:=false else p:=unknown
```

Use of theorem provers to find the right preconditions (theories for arithmetics, pointers etc required!)

Not always decidable!
(The theorem prover may reply "I don't know".)

Many calls to the theorem prover, generating the abstraction becomes the bottleneck.

Checking a counterexample

A counterexample is a sequence of instructions, e.g.:

```
x = c;

d = c+1;

y = d;

if (y != x+1) error;
```

Obviously, this sequence cannot lead to the error (it is a "spurious" counterexample).

Translate the instructions into a formula $F = F_1 \wedge F_2 \wedge F_3 \wedge F_4$:

$$F_1 = x_1 = c_0 \wedge y_1 = y_0 \wedge d_1 = d_0 \wedge c_1 = c_0$$

$$F_2 = x_2 = x_1 \wedge y_2 = y_1 \wedge d_2 = c_1 + 1 \wedge c_2 = c_1$$

$$F_3 = x_3 = x_2 \wedge y_3 = d_2 \wedge d_3 = d_2 \wedge c_3 = c_2$$

$$F_4 = y_3 \neq x_3 + 1$$

Observation: The example is spurious iff F is unsatisfiable.

How do we gain new predicates from the formula?

Interpolants

Let F, G be two formulae s.t. $F \wedge G$ is unsatisfiable.

J is called a (Craig) interpolant between F and G iff

    (i) F implies J;

    (ii) $J \wedge G$ is unsatisfiable;

    (iii) J contains only atomic propositions that also appear in F and in G.

Let Z be the variables occurring in F but not in G.
Then $\exists Z : F$ is a Craig interpolant between F and G.

Let Z be the variables occurring in G but not in F.
Then $\forall Z : \neg G$ is a Craig interpolant between F and G.

Application of interpolation

Compute interpolants between F_1 and $F_2 \wedge F_3 \wedge F_4$,
between $F_1 \wedge F_2$ and $F_3 \wedge F_4$ etc.

In the example (using existential quantification):

$$x_1 = c_1$$

$$d_2 = x_2 + 1$$

$$y_3 = x_3 + 1$$

Add those interpolants as predicates (for the respective line of code):

Optimization: Reduce the sequence to those instructions that actually influence the error condition ("slicing").

Let J be an interpolant between (z.B.) F_1 and $F_2 \wedge F_3 \wedge F_4$.

J depends only on variables with index 1, i.e. it is a predicate over the state after the first instruction.

J is implied by F_1, i.e. we kow that after executing the first statement J holds.

   $\Rightarrow$ In the new abstraction J will be true after the first instruction.

$J \wedge G$ is unsatisfiable, i.e. the states represented by J cannot reach the error state.

   $\Rightarrow$ The new abstraction excludes the previous error path.

Conclusion: Adding the predicate J will improve the abstraction.

Computing interpolants

(depends on the logic in which *F* and *G* are expressed!)

In general:

    the strongest postcondition of *F* w.r.t. *true* is an interpolant.

    the weakest precondition *G* w.r.t. *false* is an interpolant.

Let us consider the case of propositional logic.

Example

Consider the following sequence. The assume statement cannot be executed:

x:=true; y:=x;   assume($\neg y \wedge z$);

Translation into propositional-logic formulae:

$$F_1 = x_1 \wedge (y_1 \leftrightarrow y_0) \wedge (z_1 \leftrightarrow z_0)$$

$$F_2 = (x_2 \leftrightarrow x_1) \wedge (y_2 \leftrightarrow x_1) \wedge (z_2 \leftrightarrow z_1)$$

$$F_3 = (\neg y_2 \wedge z_2) \wedge (x_3 \leftrightarrow x_2) \wedge (y_3 \leftrightarrow y_2) \wedge (z_3 \leftrightarrow z_2)$$

Computation of the strongest interpolants:

between F_1 and $F_2 \wedge F_3$: $\quad J_1 := x_1$

between $F_1 \wedge F_2$ and F_3: $\quad J_2 := x_2 \wedge y_2$

Computation of the weakest interpolants:

between $F_1 \wedge F_2$ and F_3: $\quad K_2 := y_2 \vee \neg z_2$

between F_1 and $F_2 \wedge F_3$: $\quad K_1 := x_1 \vee \neg z_1$

Both J_1, J_2 and K_1, K_2 are Hoare annotations proving spuriousness of the sequence:

$\{\text{true}\} \quad x{:=}\text{true}; \quad \{x\} \quad y{:=}x; \quad \{x \wedge y\} \quad \text{assume}(\neg y \wedge z); \{\text{false}\}$

$\{\text{true}\} \ x{:=}\text{true}; \ \{x \vee \neg z\} \ y{:=}x; \ \{y \vee \neg z\} \ \text{assume}(\neg y \wedge z); \{\text{false}\}$