# jMoped

## A Checker for Java Programs

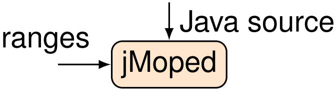
Dejvuth Suwimonteerabuth

Technische Universität München

Joint work with Felix Berger, Stefan Schwoon, and Javier Esparza

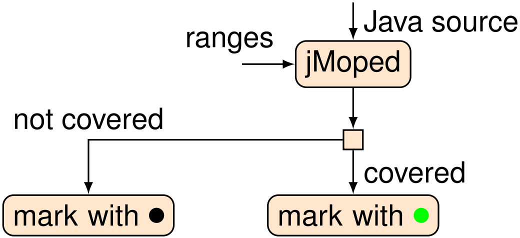
# jMoped: an Eclipse plug-in

jMoped performs a reachability analysis to check the reduced Java method.



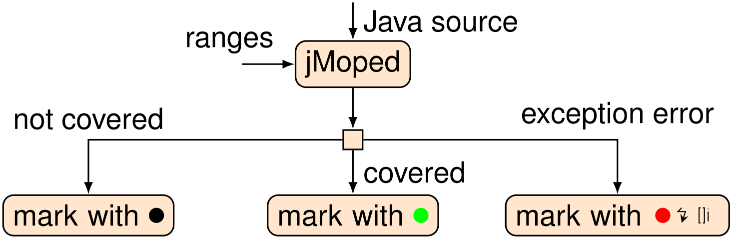
# jMoped: an Eclipse plug-in

Generates coverage information from model-checking results.



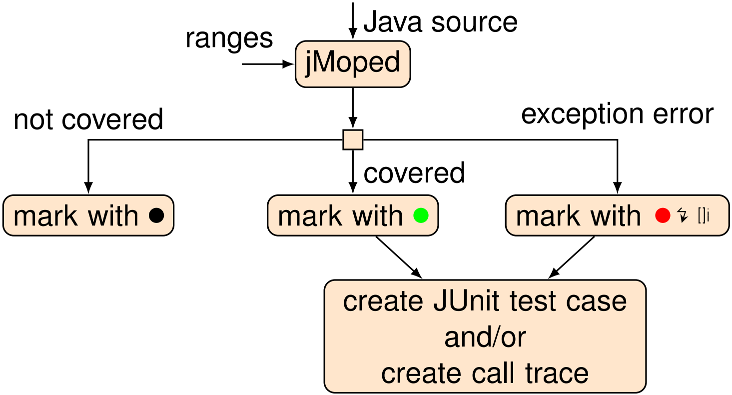
# jMoped: an Eclipse plug-in

Tests for common Java errors, i.e. assertion violations, null-pointer exceptions, array bound violations.



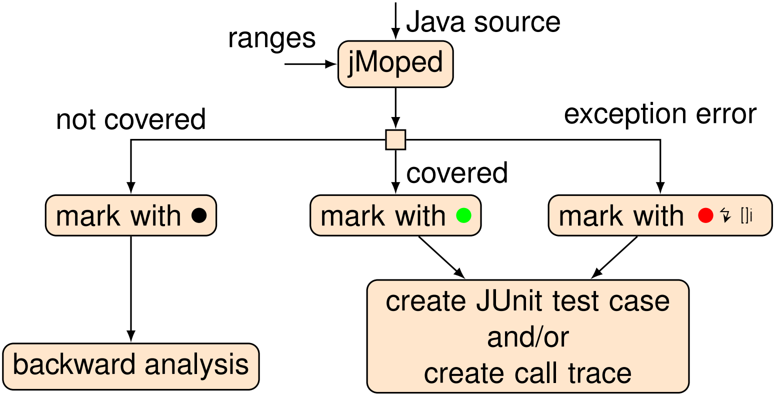
# jMoped: an Eclipse plug-in

Generates JUnit test cases or call traces with concrete inputs.

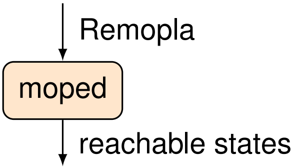


# jMoped: an Eclipse plug-in

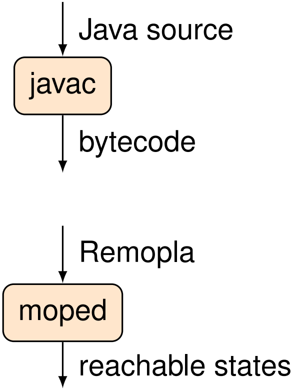
Searches backwards from uncovered instructions.



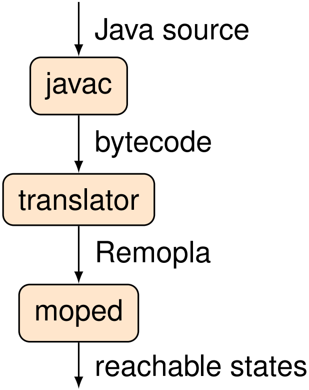
# Overview of the Architecture



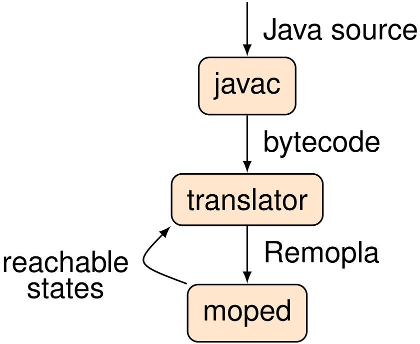
# Overview of the Architecture



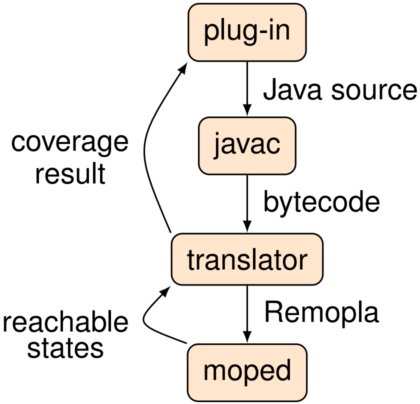
# Overview of the Architecture



# Overview of the Architecture



# Overview of the Architecture



# Supported Java Features

- Assignments
- Control statements
- Method calls and recursions
- Exceptions
- Strings (very limited)
- Multi-dimensional arrays
- Object-oriented programming
  - Inheritance
  - Abstraction
  - Polymorphism

# Limitations

- No float or negative values
- No multi-threading
- State space is rebuilt for each analysis

Performance limitations

- Analysis: jMoped is too slow for heaps bigger than 64 blocks
- Translation: A class in the Java library often calls many other classes $\longrightarrow$ very big program!

# Demonstrations

More demos . . .

# Conclusion

- Symbolic testing: uses a BDD-based model checker for testing a large set of inputs.
- Generates coverage information and find some common errors.
- User-friendly interface, model checker is hidden.
- Can be used as a complement to JUnit.
- Supports backward analysis.

http://www7.in.tum.de/tools/jmoped

# Part I: Rewriting Models of Boolean Programs

Javier Esparza

Technische Universität München

# Automatic verification using model-checking

Initiated in the early 80s in USA and France.

Exhaustive examination of the state space using (hopefully) clever techniques to avoid state explosion.

Very successful for hardware or “low-level” software:

- Applied to commercial microprocessors, telephone switches launching protocols, brake systems, or the dutch Delta works.
- Model-checking groups at all major hardware companies.
- ACM Software System Award 2001, Gödel Prize 2000, Kannellakis Awards 1998 and 2005.

# Software model-checking

Big research challenge of the 00s: extension to ‘high-level’ software.

Three main research questions:

- Integration of the tools in the software development process.
  - Users trust their hardware but may not trust their software: “post-mortem” verification, “backstage” verification tools . . .
- Automatic extraction of models from code.
- Algorithms for infinite-state systems.
  - Software systems are very often infinite-state.

# A “lazy” approach to software verification

Construct a sequence of increasingly faithful models that under- or overapproximate the code.

Underapproximations: 32-bit integer $\longrightarrow$ 2-bit integer, 500MB heap $\longrightarrow$ 10B heap.

# A “lazy” approach to software verification

Construct a sequence of increasingly faithful models that under- or overapproximate the code.

Underapproximations: 32-bit integer $\longrightarrow$ 2-bit integer, 500MB heap $\longrightarrow$ 10B heap.

Overapproximations using predicate abstraction:

# A "lazy" approach to software verification

Construct a sequence of increasingly faithful models that under- or overapproximate the code.

Underapproximations: 32-bit integer $\longrightarrow$ 2-bit integer, 500MB heap $\longrightarrow$ 10B heap.

Overapproximations using predicate abstraction:

> Define a set of predicates over the dataspace.
> Example: $x < y$ $x = 0$

# A "lazy" approach to software verification

Construct a sequence of increasingly faithful models that under- or overapproximate the code.

Underapproximations: 32-bit integer $\longrightarrow$ 2-bit integer, 500MB heap $\longrightarrow$ 10B heap.

Overapproximations using predicate abstraction:

Define a set of predicates over the dataspace.
Example: $x < y$ $x = 0$

Associate to each predicate a boolean variable.
Example: $x < y \mapsto a$ $x = 0 \mapsto b$

# A "lazy" approach to software verification

Construct a sequence of increasingly faithful models that under- or overapproximate the code.

Underapproximations: 32-bit integer $\rightarrow$ 2-bit integer, 500MB heap $\rightarrow$ 10B heap.

Overapproximations using predicate abstraction:

Define a set of predicates over the dataspace.
Example: $x < y$ $x = 0$

Associate to each predicate a boolean variable.
Example: $x < y \mapsto a$ $x = 0 \mapsto b$

Overapproximate by a program over these variables.

Example: $x := y$ is overapproximated by

$a := \text{false};$

**if** $(a$ **and** $b)$ **then** $b := \text{false}$

**else** $b := \text{true}$ **or** $\text{false}$

Both under- and overapproximations are boolean programs:

Same control-flow structure as code + possibly nondeterminism.

Only one datatype: booleans.

Conceptually could also take any enumerated type but booleans are the bridge to SAT and BDD technology.

# Rewriting models of boolean programs

Boolean programs are still pretty complicated objects:

- Procedures/methods and recursion.
- Concurrency and communication (threads, cobegin-coend sections).
- Object-orientation.

Must be “compiled” into simpler and formal models.

# Rewriting models of boolean programs

Boolean programs are still pretty complicated objects:

- Procedures/methods and recursion.
- Concurrency and communication (threads, cobegin-coend sections).
- Object-orientation.

Must be “compiled” into simpler and formal models.

Use rewriting to model boolean programs. In a nutshell:

- Model program states as terms.
- Model program instructions as term-rewriting rules.
- Model program executions as sequences of rewriting steps.

# Fundamental analysis problem: Reachability

But reachability between two states not enough for verification purposes

Safety properties often characterized by an infinite set of dangerous states.

Set of initial states also possibly infinite.

Generalized reachability problem: Given two (possibly infinite) sets $I$ and $D$ of initial and dangerous states, respectively, decide if some state of $D$ is reachable from some state of $I$.

Challenge: Find a finite ("symbolic") representation of the (possibly infinite) set of states reachable or backward reachable from a given (possibly infinite) set of states.

- $pre^*(S)$ denotes the set of predecessors of $S$. (states backward reachable from states in $S$)
- $post^*(S)$ denotes the set of successors of $S$. (states forward reachable from states in $S$)

Strategies: Compute $pre^*(D)$ and check if $I \cap pre^*(D) = \emptyset$, or compute $post^*(I)$ and check if $post^*(I) \cap D = \emptyset$

# Program for the rest of Part I

Rewriting models for:

- Procedural sequential programs.
- Multithreaded while-programs.
- Multithreaded procedural programs.
- Procedural programs with cobegin-coend sections.

For each of those:

- Complexity of the reachability problem.
- Finite representations for symbolic reachability.

# A rewriting model of procedural sequential programs

State of a procedural boolean program: $(\, g, \ell, n, (\ell_1, n_1) \ldots (\ell_k, n_k) \,)$, where

- $g$ is a valuation of the global variables,
- $\ell$ is a valuation of local variables of the currently active procedure,
- $n$ is the current value of the program pointer,
- $l_i$ is a saved valuation of the local variables of the caller procedures, and
- $n_i$ is a return address.

Modelled as a string $g \, \langle \ell, n \rangle \, \langle \ell_1, n_1 \rangle \, \ldots \, \langle \ell_k, n_k \rangle$

Instructions modelled as string-rewriting rules, e.g. $t \, \langle t, m_0 \rangle \longrightarrow f \langle f \, t \, f, p_0 \rangle \, \langle t, m_1 \rangle$

Prefix-rewriting policy:

$$\frac{u \longrightarrow w}{u \, v \xrightarrow{\; r \;} w \, v}$$

# An example

**bool function** *foo*($\ell$)
$f_0$: **if** $\ell$ **then**
$f_1$: **return** $\mathrm{false}$
**else**
$f_2$: **return** $\mathrm{true}$
**fi**

**procedure** *main*()
**global** $b$
$m_0$: **while** $b$ **do**
$m_1$: $b := foo(b)$
**od**;
$m_2$: **return**

$$b\,\langle t, f_0\rangle \;\longrightarrow\; b\,\langle t, f_1\rangle$$
$$b\,\langle f, f_0\rangle \;\longrightarrow\; b\,\langle f, f_2\rangle$$
$$b\,\langle \ell, f_1\rangle \;\longrightarrow\; f$$
$$b\,\langle \ell, f_2\rangle \;\longrightarrow\; t$$

$$t\,m_0 \;\longrightarrow\; t\,m_1$$
$$f\,m_0 \;\longrightarrow\; f\,m_2$$
$$b\,m_1 \;\longrightarrow\; b\langle b, f_0\rangle\, m_0$$
$$b\,m_2 \;\longrightarrow\; \epsilon$$

($b$ and $\ell$ stand for both $t$ and $f$)

# Prefix string rewriting. From theory . . .

First studied by Büchi in 64 under the name regular canonical systems as a variant of semi-Thue systems.

Theorem: Given an effectively regular (possibly infinite) set $S$ of strings, the sets $pre^*(S)$ and $post^*(S)$ are also effectively regular.

# Prefix string rewriting. From theory . . .

First studied by Büchi in 64 under the name regular canonical systems as a variant of semi-Thue systems.

Theorem: Given an effectively regular (possibly infinite) set $S$ of strings, the sets $pre^*(S)$ and $post^*(S)$ are also effectively regular.

Rediscovered by Caucal in 92.

# Prefix string rewriting. From theory . . .

First studied by Büchi in 64 under the name regular canonical systems as a variant of semi-Thue systems.

Theorem: Given an effectively regular (possibly infinite) set $S$ of strings, the sets $pre^*(S)$ and $post^*(S)$ are also effectively regular.

Rediscovered by Caucal in 92.

Polynomial algorithms by Bouajjani, E., Maler and Finkel, Willems, Wolper in 97.

- Saturation algorithms: the automata for $pre^*(S)$ and $post^*(S)$ are essentially obtained by adding transitions to the automaton for $S$.
  (Algorithms for similar models by Alur, Etessami, Yannakakis, and Benedikt, Godefroid, Reps and . . . )

# . . . to applications

Efficient algorithms by E., Hansel, Rossmanith and Schwoon in 00.

Theorem (informal): Let $\Sigma$, $R$ be the alphabet and set of rules of a 2-normalized prefix-rewriting system system and let $A$ be a “small” NFA over $\Sigma$.

An NFA for $post^*(L(A))$ can be constructed in $O(|\Sigma||R|^2)$ time and space.

An NFA for $pre^*(L(A))$ can be constructed in $O(|\Sigma|^2|R|)$ time and $O(|\Sigma||R|)$ space.

# . . . to applications

Efficient algorithms by E., Hansel, Rossmanith and Schwoon in 00.

Theorem (informal): Let $\Sigma$, $R$ be the alphabet and set of rules of a 2-normalized prefix-rewriting system system and let $A$ be a “small” NFA over $\Sigma$.

An NFA for $post^*(L(A))$ can be constructed in $O(|\Sigma||R|^2)$ time and space.

An NFA for $pre^*(L(A))$ can be constructed in $O(|\Sigma|^2|R|)$ time and $O(|\Sigma||R|)$ space.

BDD-based algorithms by E. and Schwoon in 01.

# . . . to applications

Efficient algorithms by E., Hansel, Rossmanith and Schwoon in 00.

Theorem (informal): Let $\Sigma$, $R$ be the alphabet and set of rules of a 2-normalized prefix-rewriting system system and let $A$ be a "small" NFA over $\Sigma$.

An NFA for $post^*(L(A))$ can be constructed in $O(|\Sigma||R|^2)$ time and space.

An NFA for $pre^*(L(A))$ can be constructed in $O(|\Sigma|^2|R|)$ time and $O(|\Sigma||R|)$ space.

BDD-based algorithms by E. and Schwoon in 01.

MOPED model checker by Schwoon in 02.

# . . . to applications

Efficient algorithms by E., Hansel, Rossmanith and Schwoon in 00.

Theorem (informal): Let $\Sigma$, $R$ be the alphabet and set of rules of a 2-normalized prefix-rewriting system system and let $A$ be a "small" NFA over $\Sigma$.

An NFA for $post^*(L(A))$ can be constructed in $O(|\Sigma||R|^2)$ time and space.

An NFA for $pre^*(L(A))$ can be constructed in $O(|\Sigma|^2|R|)$ time and $O(|\Sigma||R|)$ space.

BDD-based algorithms by E. and Schwoon in 01.

MOPED model checker by Schwoon in 02.

MOPS checker by Chen and Wagner in 02.

# . . . to applications

Efficient algorithms by E., Hansel, Rossmanith and Schwoon in 00.

Theorem (informal): Let $\Sigma$, $R$ be the alphabet and set of rules of a 2-normalized prefix-rewriting system system and let $A$ be a "small" NFA over $\Sigma$.
An NFA for $post^*(L(A))$ can be constructed in $O(|\Sigma||R|^2)$ time and space.
An NFA for $pre^*(L(A))$ can be constructed in $O(|\Sigma|^2|R|)$ time and $O(|\Sigma||R|)$ space.

BDD-based algorithms by E. and Schwoon in 01.

MOPED model checker by Schwoon in 02.

MOPS checker by Chen and Wagner in 02.

"Model Checking an Entire Linux Distribution for Security Violations"
by Schwarz et al. at ACSAC 05.

# Büchi did it

Moshe Vardi:

Büchi automata, introduced by Büchi in the early 60s to solve problems in second-order number theory, have been translated, unlikely as it may seem, into effective algorithms for model checking tools.

# Büchi did it twice

Moshe Vardi:

Büchi automata, introduced by Büchi in the early 60s to solve problems in second-order number theory, have been translated, unlikely as it may seem, into effective algorithms for model checking tools.

Here:

Regular canonical systems, introduced by Büchi in the early 60s because he liked them, have been translated, unlikely as it may seem, into effective algorithms for software model checking tools.

# A rewriting model of multithreaded while-programs

Communication through global variables.

State determined by: $\{ g, (\ell_0, n_0), (\ell_1, n_1) \dots (\ell_k, n_k) \}$ where

- $g$ is a valuation of the global variables,
- $\ell_i$ is a valuation of the local variables of the $i$-th thread, and
- $n_i$ is the value of the program pointer of the $i$-th thread.

Modelled as a multiset

$$g \parallel \langle \ell_0, n_0 \rangle \parallel \langle \ell_1, n_1 \rangle \parallel \dots \parallel \langle \ell_k, n_k \rangle$$

Instructions modelled as multiset-rewriting rules, e.g.

$$t\, f \parallel m_0 \rightarrow f\, f \parallel m_1 \parallel \langle f, p_0 \rangle$$

Multiset rewriting, or rewriting modulo assoc. and comm. of $\parallel$.

# An example

**thread** $p()$

| | |
|---|---|
| $p_0$: | **if** ? **then** |
| $p_1$: | $b := \mathrm{true}$ |
| | **else** |
| $p_2$: | $b := \mathrm{false}$ |
| | **fi**; |
| $p_3$: | **end** |

$$b \parallel p_0 \rightarrow b \parallel p_1$$
$$b \parallel p_0 \rightarrow b \parallel p_2$$
$$b \parallel p_1 \rightarrow t \parallel p_3$$
$$b \parallel p_2 \rightarrow f \parallel p_3$$
$$b \parallel p_3 \rightarrow \epsilon$$

**thread** $main()$
**global** $b$

| | |
|---|---|
| $m_0$: | **while** $b$ **do** |
| $m_1$: | **fork** $p()$ |
| | **od**; |
| $m_2$: | **end** |

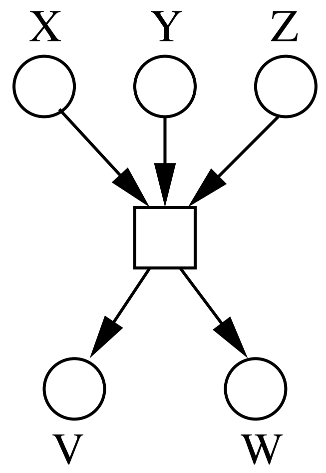
$$t \parallel m_0 \rightarrow t \parallel m_1$$
$$f \parallel m_0 \rightarrow f \parallel m_2$$
$$b \parallel m_1 \rightarrow b \parallel m_0 \parallel p_0$$
$$b \parallel m_2 \rightarrow \epsilon$$

# Multiset rewriting

Theorem [Mayr, Kosaraju, Lipton, 80s]: The reachability problem for multiset-rewriting is decidable but EXPSPACE-hard.

- Equivalent to the reachability problem for Petri nets.
- A place for each alphabet letter.
- A Petri net transition for each rewrite rule.

$$X \parallel Y \parallel Z \longrightarrow V \parallel W$$




Algorithms (not only proofs) quite complicated.

Negative results for $pre^*(\{s\})$ and $post^*(\{s\})$.

# Symbolic reachability for *pre*$^*$ and upward-closed sets

Upward-closed set: if some multiset $t$ belongs to the set, then $t \parallel t'$ also belongs to the set for every $t'$.

Finitely representable e.g. by the its of minimal elements.

Upward-closed sets capture properties that can be decided by inspecting a bounded number of threads (e.g. mutual exclusion).

Theorem [Abdulla et al. 96]: Given a multiset-rewriting system and an upward-closed set of states $S$, the set $pre^*(S)$ is upward-closed and effectively constructible.

- Very simple algorithm: compute $pre(S), pre^2(S), pre^3(S) \ldots$.

Extensions applied to multithreaded Java [Delzanno, Raskin, Van Begin 04].

# Monadic multiset-rewriting

Monadic rules $\equiv$ no global variables $\equiv$ no communication

# Monadic multiset-rewriting

Monadic rules $\equiv$ no global variables $\equiv$ no communication

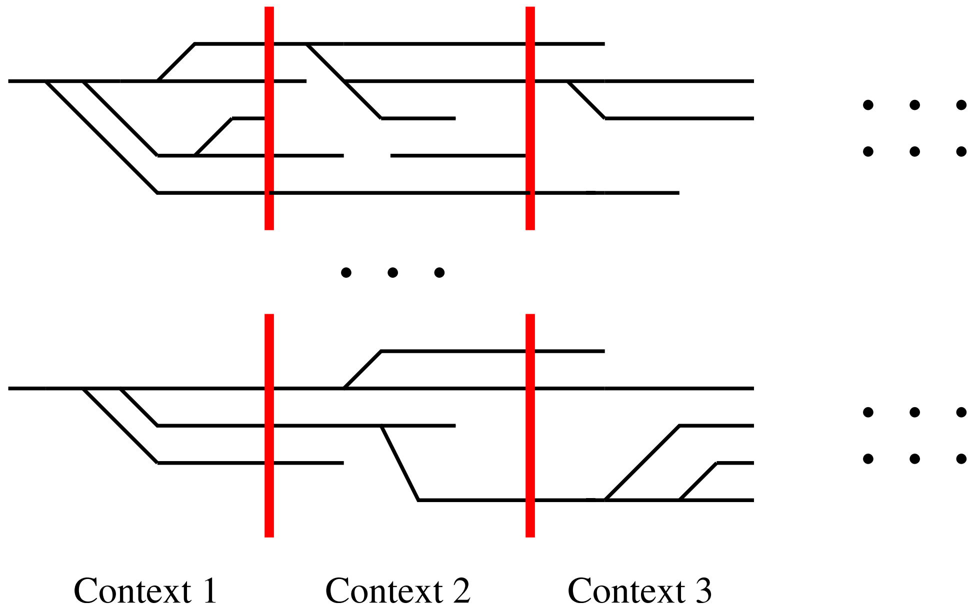
*. . . but what are threads that cannot communicate with each other good for?!!!*

# Monadic multiset-rewriting

Monadic rules $\equiv$ no global variables $\equiv$ no communication

*. . . but what are threads that cannot communicate with each other good for?!!!*

They are good for underapproximations [Qadeer and Rehof 05]



# Reachability

Theorem [Huyhn 85, E.95]: The reachability problem for monadic multiset-rewrite systems is NP-complete.

- Membership in NP not completely trivial.
- Hardness very easy, reduction from SAT:

  A thread for each variable $x_i$ that (a) nondeterministically chooses $l_i \in \{x_i, \overline{x}_i\}$ and (b) spawns a clause thread for each clause satisfied by $l_i$.

  The thread for a clause does nothing and terminates.

  Formula satisfiable iff there is state at which one thread per clause is active.

# Symbolic reachability for semi-linear sets

Semi-linear sets usually defined as subsets of $\mathbb{N}^n$.

- Finite union of linear sets.
- $\{r + \lambda_1 p_1 + \ldots + \lambda_n p_n \mid \lambda_1, \ldots, \lambda_n \in \mathbb{N}\}$.

Language interpretation: “commutative closure” of the regular languages.

Similar properties to regular languages: closure under boolean operations, decidable (but no longer polynomial) membership problem, etc.

Theorem [E.95]: Given a monadic multiset-rewriting system and a semi-linear set of states $S$, the sets $post^*(S)$ and $pre^*(S)$ are semi-linear and effectively constructible.

# Multithreaded procedural programs

Two-counter machines can be simulated by a program with two recursive threads communicating over two global (boolean) variables:

- Tops of the recursion stacks contains two copies of the machine's control point.
- Depths the two recursion stacks model the values of the counters.
- Calls and returns model increasing and decrementing the counters.
- One variable to ensure alternation of moves.
- One variable to keep the two copies of the control point "synchronized".

If communication takes place by rendezvous the two variables are no longer needed: programs without variables are still Turing powerful.

# Multithreaded procedural programs

Two-counter machines can be simulated by a program with two recursive threads communicating over two global (boolean) variables:

- Tops of the recursion stacks contains two copies of the machine's control point.
- Depths the two recursion stacks model the values of the counters.
- Calls and returns model increasing and decrementing the counters.
- One variable to ensure alternation of moves.
- One variable to keep the two copies of the control point "synchronized".

If communication takes place by rendezvous the two variables are no longer needed: programs without variables are still Turing powerful.

Communication-free case: [Bouajjani, Müller-Olm and Touili 05]

Communication through nested locks: [Kahlon and Gupta 06]

# A rewriting model for the communication-free case

State of a multithreaded procedural program without global variables:
multiset $\{s_1, s_2 \ldots, s_k\}$ of states of procedural programs, where

$$s_i = (\ell_{i0}, n_{i0})\,(\ell_{i1}, n_{i1}) \ldots (\ell_{im}, n_{im})$$

Modelled as a string $\# w_k \# w_{k-1} \# \ldots \# w_1$ where

$$w_i = \langle \ell_{i0}, n_{i0} \rangle \, \langle \ell_{i1}, n_{i1} \rangle \; \ldots \; \langle \ell_{im}, n_{im} \rangle$$

Instructions modelled as string-rewriting rules. A new thread is inserted to the left of its creator, e.g.

$$\# \, \langle b, m_1 \rangle \longrightarrow \# \, p_0 \, \# \, \langle f, m_3 \rangle$$

Threads "in the middle" of the string should also be able to "move": back to ordinary rewriting

$$\frac{u \longrightarrow w}{v_1 \, u \, v_2 \xrightarrow{r} v_1 \, w \, v_2}$$

# An example

**process** $p()$;

$p_0$: **if** (?) **then**

$p_1$: **call** $p()$

**else**

$p_2$: **skip**

**fi**;

$p_3$: **return**

$$\# p_0 \rightarrow \# p_1$$
$$\# p_0 \rightarrow \# p_2$$
$$\# p_1 \rightarrow \# p_0 \, p_3$$
$$\# p_2 \rightarrow \# p_3$$
$$\# p_3 \rightarrow \#$$

**process** $main()$

$m_0$: **if** (?) **then**

$m_1$: **fork** $p()$

**else**

$m_2$: **call** $main()$

**fi**;

$m_3$: **return**

$$\# m_0 \rightarrow \# m_1$$
$$\# m_0 \rightarrow \# m_2$$
$$\# m_1 \rightarrow \# p_0 \# m_3$$
$$\# m_2 \rightarrow \# m_0 \, m_3$$
$$\# m_3 \rightarrow \# \epsilon$$
$$\# \# \rightarrow \#$$

# Analysis

Theorem [BMOT05]: For every effectively regular set $S$ of states, the set $pre^*(S)$ is regular and a finite-state automaton recognizing it can be effectively constructed in polynomial time.

- Similar to $pre^*$ for monadic string-rewriting [Book and Otto 93].

Theorem [BMOT05]: For every effectively context-free set $S$ of states, the set $post^*(S)$ is context-free and a pushdown automaton recognizing it can be effectively constructed in polynomial time.

Counterexample to regularity: $P$ that spawns a copy of $Q$ and calls itself.

The number of threads is equal to the depth of the recursion.

Reachability set: $\{(\#q)^n \# p^{(n+1)} \mid n \geq 0\}$.

# Cobegin-coend sections

Difference with threads: implicit synchronization induced by the coend.

- “Threads have to wait for its siblings to terminate.”
- Corresponds to calling procedures in parallel.

Rewriting model only works well for the communication-free (monadic) case.

States modelled as terms with both $\|$ and $\cdot$ as infix operators e.g

$$(\langle t, p_1 \rangle \parallel q_2) \cdot \langle t\,f, m_1 \rangle$$

Rewriting modulo assoc. of $\cdot$ and assoc. and comm. of $\|$.

This model is called monadic process rewrite systems (monadic PRS) [Mayr 00].

# Analysis

Symbolic reachability with commutative hedge automata (CHA) [Lugiez 03].

Theorem [Bouajjani and Touili 05]: Given a monadic PRS, for every CHA-definable set of terms $T$, the sets $post^*(T)$ and $pre^*(T)$ are CHA-definable and effectively constructible.

Weaker approach: construct not the sets $post^*(T)$ or $pre^*(T)$ themselves, but representatives w.r.t. the equational theory.

Sufficient for control reachability problems.

Theorem [Lugiez and Schnoebelen 98, E. and Podelski 00]:
Let $R$ be a monadic PRS and let $A$ be a bottom-up tree automaton.
One can construct in $O(|R| \cdot |A|)$ time bottom-up tree automata recognizing a set of representatives of $post^*(L(A))$ and $pre^*(L(A))$.

# Conclusions

Rewriting concepts can be used to give elegant semantics to programming languages.

- String/multiset rewriting correspond to sequential/parallel computation.
- Monadic/non-monadic rewriting correspond to absence or presence of communication.
- Rewriting modulo useful for combining concurrency and procedures.

Symbolic reachability is the key problem to solve.

Comparison with process algebras:

- Process algebras have a notion of hiding or encapsulation.
- Rewriting much closer to automata theory $\longrightarrow$ algorithms.

# Verification of Infinite-state Systems

Javier Esparza

Software Reliability and Security Group

Institute for Formal Methods in Computer Science

University of Stuttgart

# Part II:

# Symbolic reachability for prefix rewriting

# Case study: Drawing skylines

```
static Random r = new Random();

static void m() {
  if (r.nextBoolean()) {
    s(); right(); if (r.nextBoolean()) m();
  }
  else { up(); m(); down(); }
}

static void s() {
    if (r.nextBoolean()) return;
    up(); m(); down();
}

public static void main() { s(); }
```

# Model

```
static void s() {
  s0: if (r.nextBoolean())
  s1: return;
  s2: up();
  s3: m();
  s4: down();
  s5:
}
```

**var** st:**stack** of $\{s_0, \ldots, s_5, \ldots\}$

$s_0 \longrightarrow s_1 \qquad s_0 \longrightarrow s_2$

$s_1 \longrightarrow \epsilon$

$s_2 \longrightarrow up_0\, s_3$

$s_3 \longrightarrow m_0\, s_4$

$s_4 \longrightarrow down_0\, s_5$

$s_5 \longrightarrow \epsilon$

# Symbolic reachability in prefix rewriting

Recall: program state $(g, \ell, n, (\ell_1, n_1) \dots (\ell_k, n_k))$ modelled as a word $g \langle \ell, n \rangle \langle \ell_1, n_1 \rangle \dots \langle \ell_k, n_k \rangle$.

Denote by $G$ the alphabet of valuations of globals.

Denote by $L$ the alphabet of pairs $\langle \ell, n \rangle$.

The set of possible programs states is given by $G\,L^*$

A subset of $GL^*$ words is regular if it can be recognized by a finite automaton.

Typically, the sets $I$ and $D$ of initial and dangerous program states are regular sets. (Even very simple ones, like $g\, l\, L^*$.)

Challenge: show that if $S \subseteq GL^*$ is (effectively) regular, then so are $pre^*(S)$ and $post^*(S)$.

This gives a procedure to check if $I \cap pre^*(D) = \emptyset$ or $post^*(I) \cap D = \emptyset$.

# Symbolic search

<u>Forward symbolic search</u>

Initialize $S := I$

Iterate $S := S \cup post(S)$ until fixpoint.

Backward search: replace $I$ by $D$, replace $post$ by $pre$.

Questions:

- Are $S \cup post(S)$ and $S \cup pre(S)$ regular for regular $S$ ?
- Does the search terminate ?

We answer these questions for backward search, the forward case is similar.

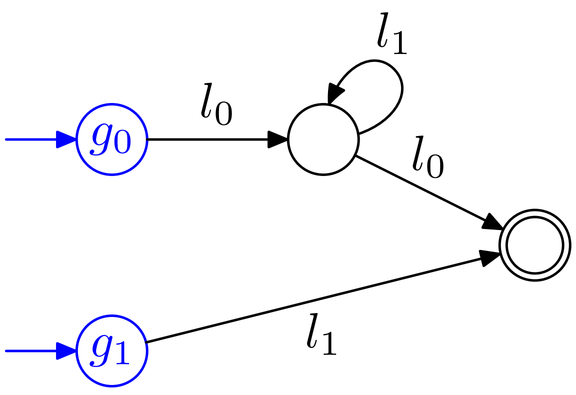
# If $S$ regular, then $S \cup pre(S)$ regular

We represent a regular set $S \subseteq GL^*$ by an NFA.

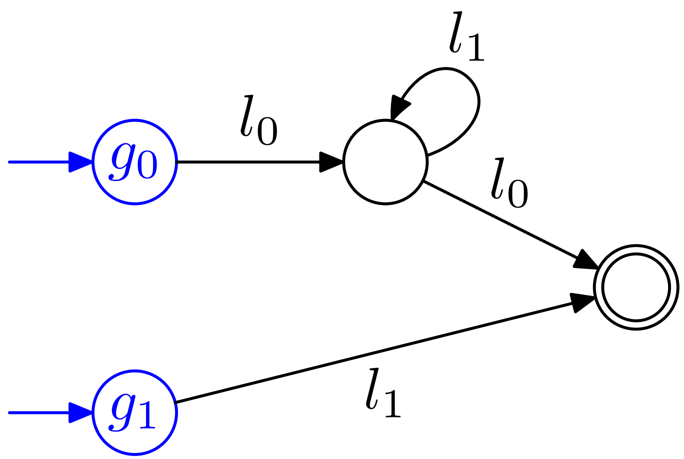
- $G$ as set of initial states, $L$ as alphabet.
- $gw$ recognized if $g \xrightarrow{w} q$ for some final state $q$.

Example: $G = \{g_0, g_1\}$ and $L = \{l_0, l_1\}$

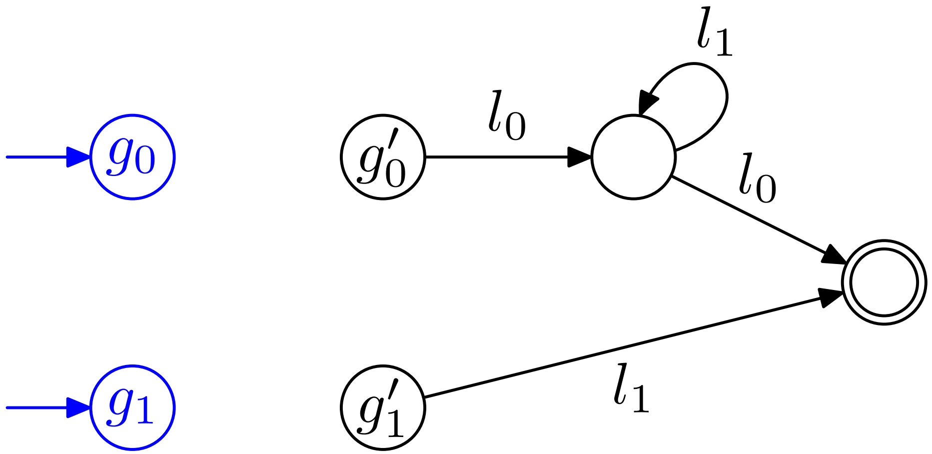
Automaton coding the set $g_0 l_1^* l_0 + l_1 l_1$ :



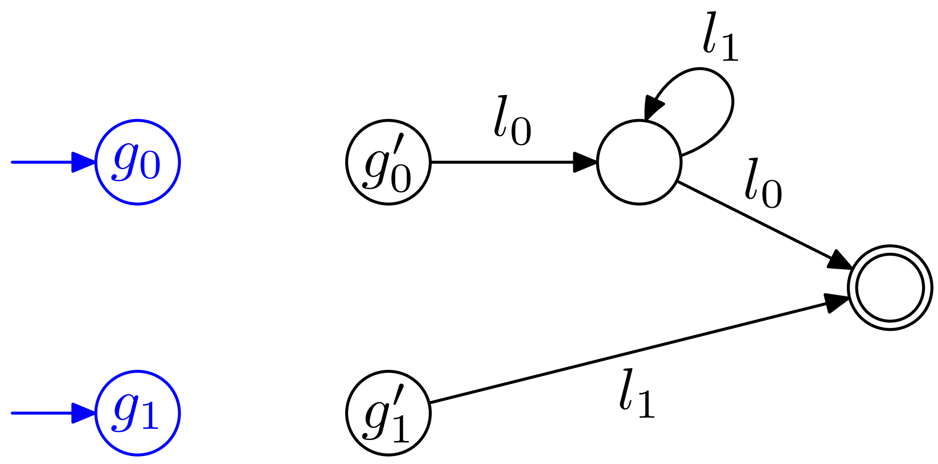
$$R = \{ \quad g_0\, l_0 \rightarrow g_0 \quad , \quad g_1\, l_1 \rightarrow g_0 \quad , \quad g_1\, l_1 \rightarrow g_1\, l_1\, l_0 \quad \}$$



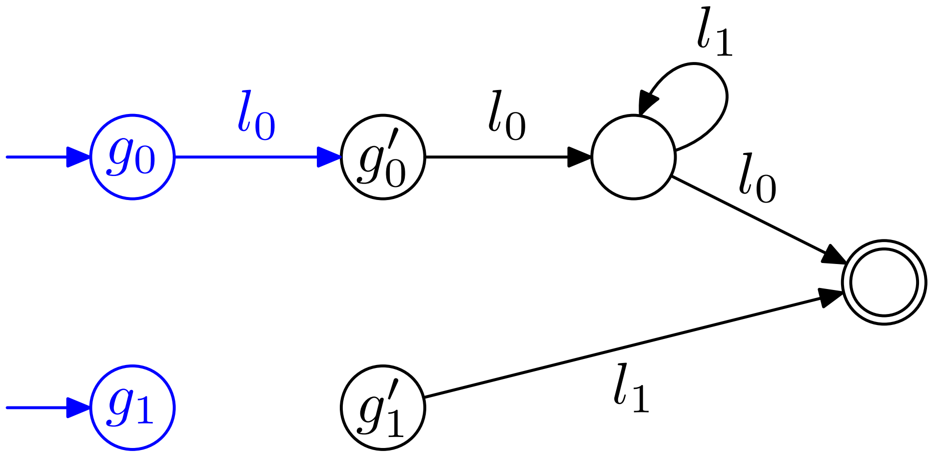
$$R = \{ \quad g_0\, l_0 \to g_0 \quad , \quad g_1\, l_1 \to g_0 \quad , \quad g_1\, l_1 \to g_1\, l_1\, l_0 \quad \}$$



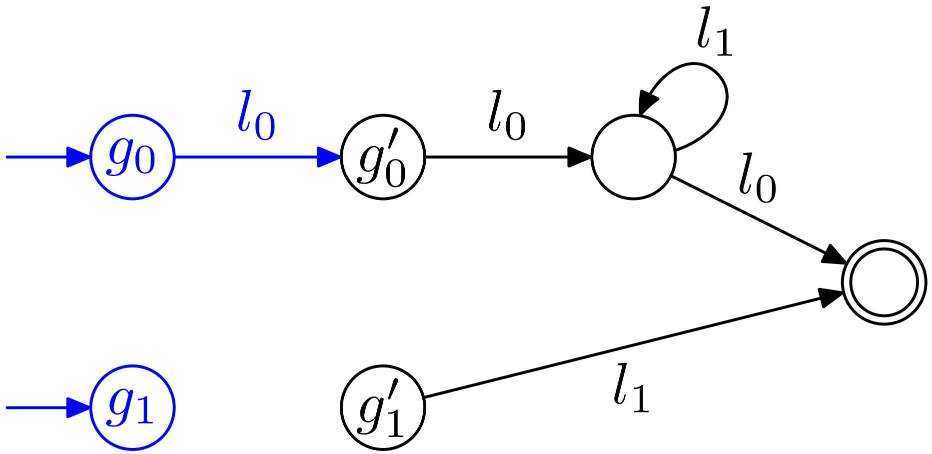
$g_0\, l_0 \rightarrow g_0$



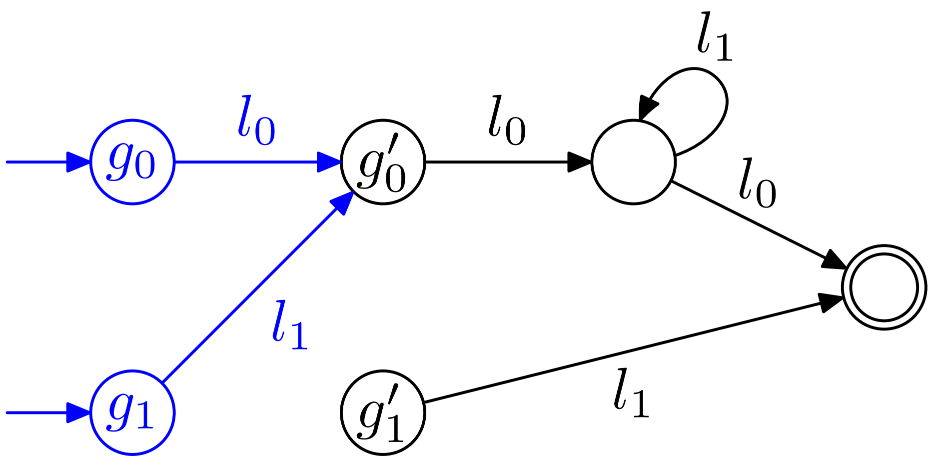
$g_0\, l_0 \rightarrow g_0$



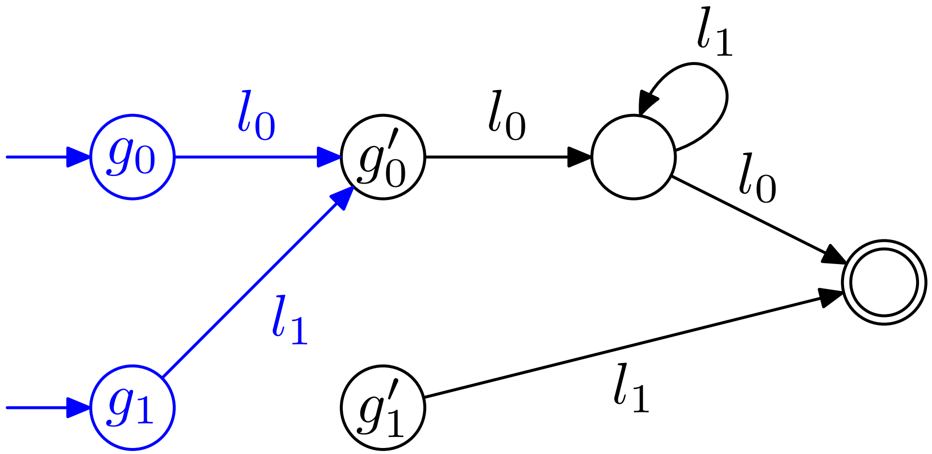
$g_1\, l_1 \rightarrow g_0$



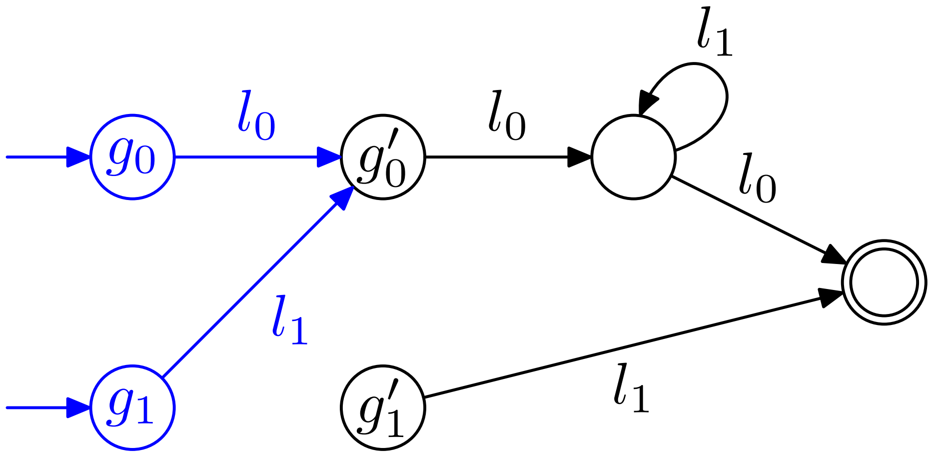
$g_1\, l_1 \to g_0$



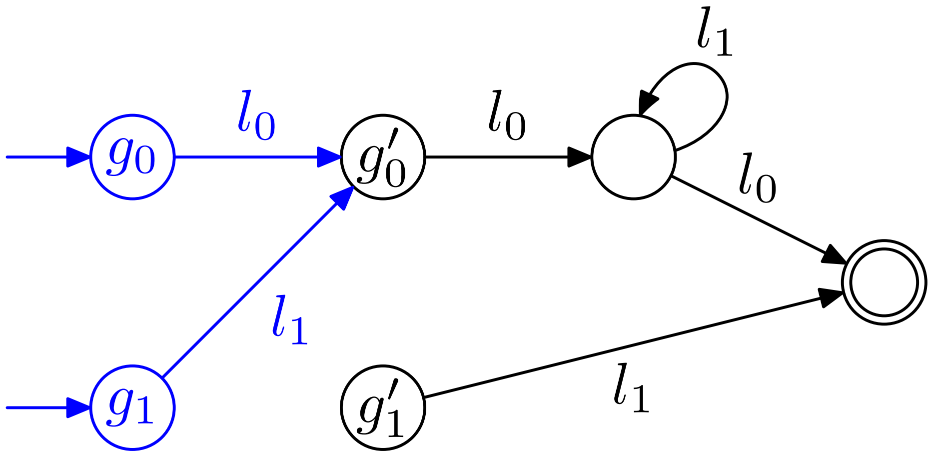
$g_1\, l_1 \rightarrow g_1\, l_1\, l_0$



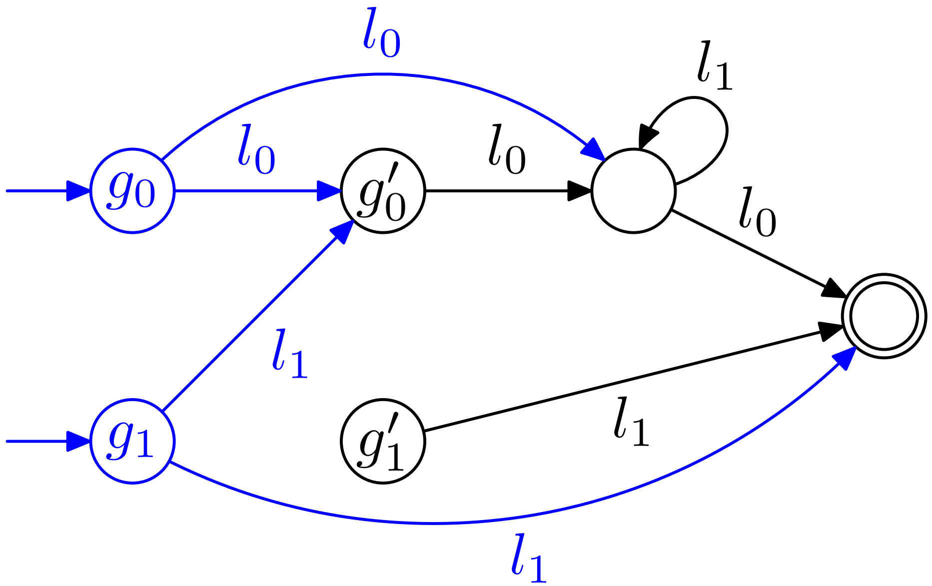
$$g_1\, l_1 \rightarrow g_1\, l_1\, l_0$$



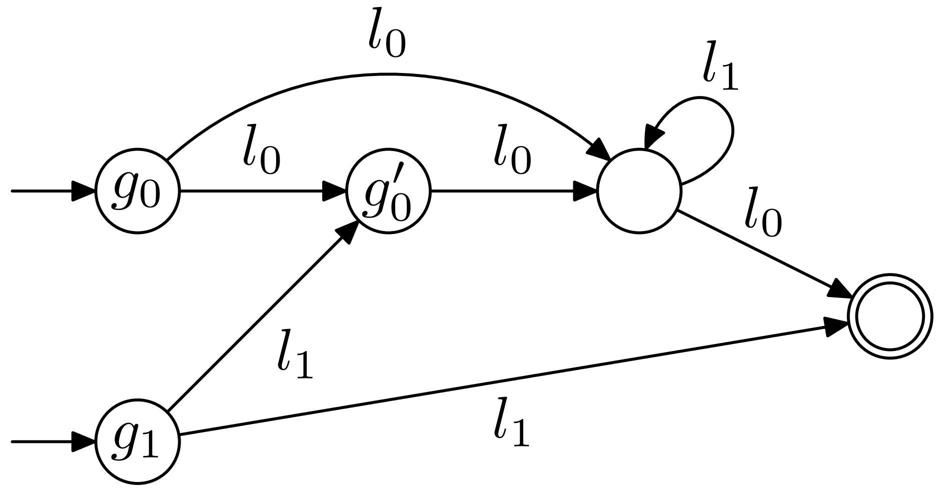
$$R = \{ \quad g_0\, l_0 \rightarrow g_0 \quad , \quad g_1\, l_1 \rightarrow g_0 \quad , \quad g_1\, l_1 \rightarrow g_1\, l_1\, l_0 \quad \}$$



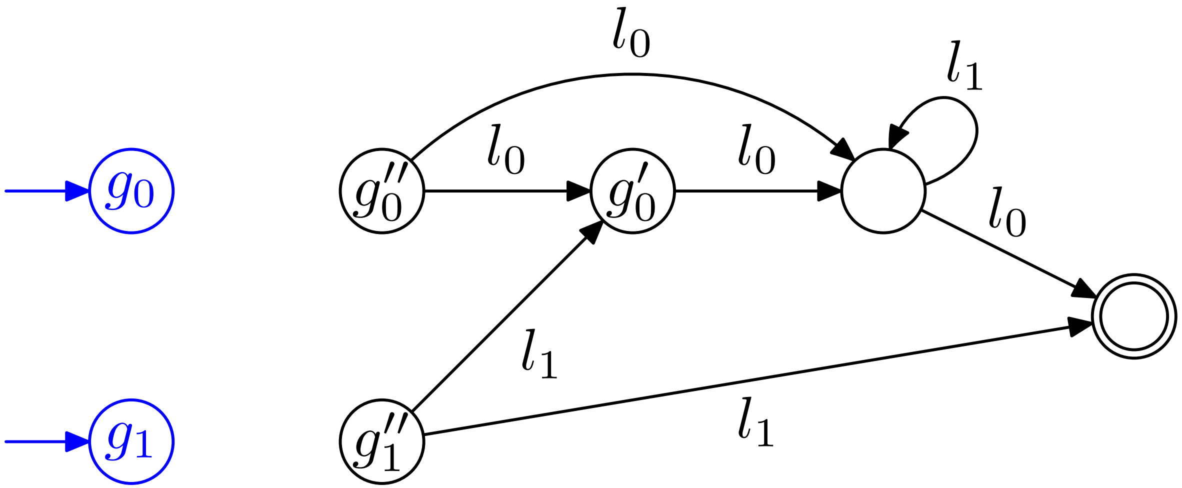
$$R = \{ \quad g_0\, l_0 \rightarrow g_0 \quad , \quad g_1\, l_1 \rightarrow g_0 \quad , \quad g_1\, l_1 \rightarrow g_1\, l_1\, l_0 \quad \}$$



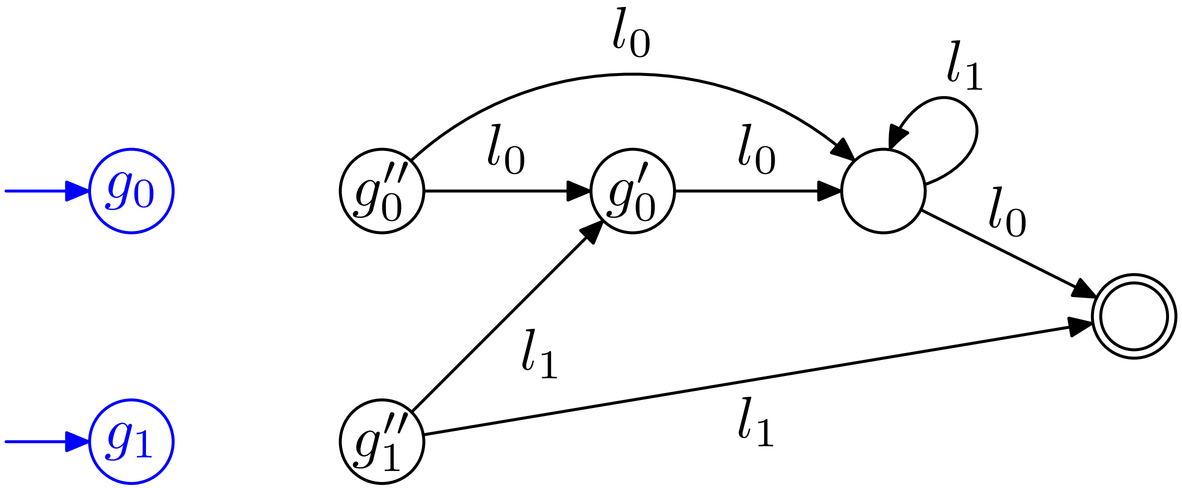
$$R = \{ \quad g_0\, l_0 \rightarrow g_0 \quad , \quad g_1\, l_1 \rightarrow g_0 \quad , \quad g_1\, l_1 \rightarrow g_1\, l_1\, l_0 \quad \}$$



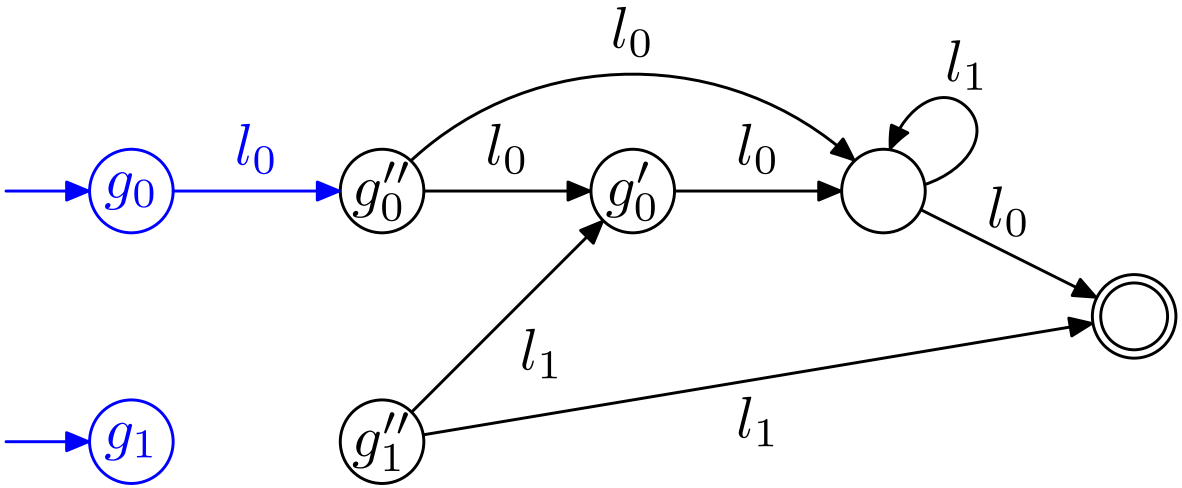
$$R = \{ \quad g_0\, l_0 \rightarrow g_0 \quad , \quad g_1\, l_1 \rightarrow g_0 \quad , \quad g_1\, l_1 \rightarrow g_1\, l_1\, l_0 \quad \}$$



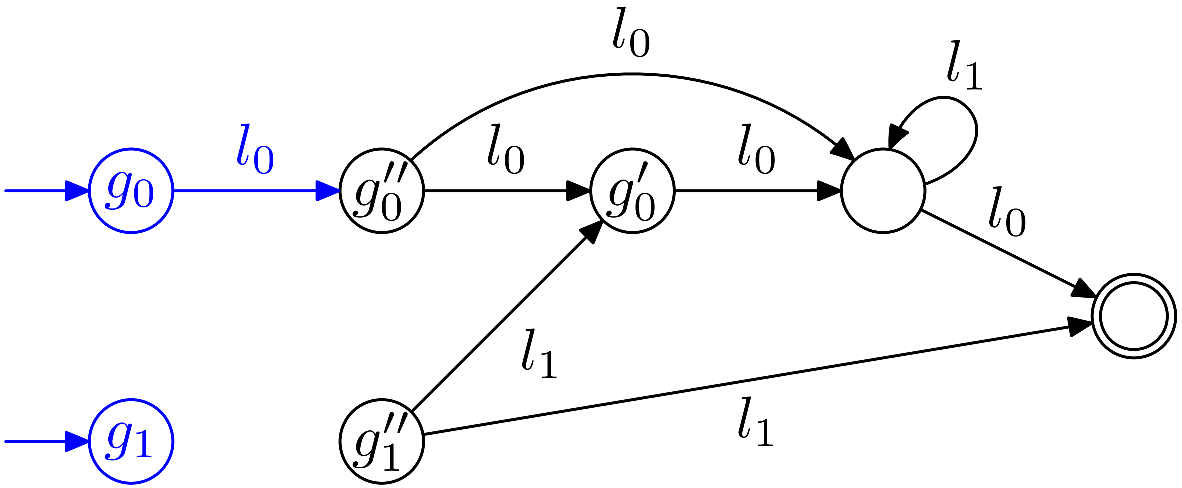
$g_0\, l_0 \rightarrow g_0$



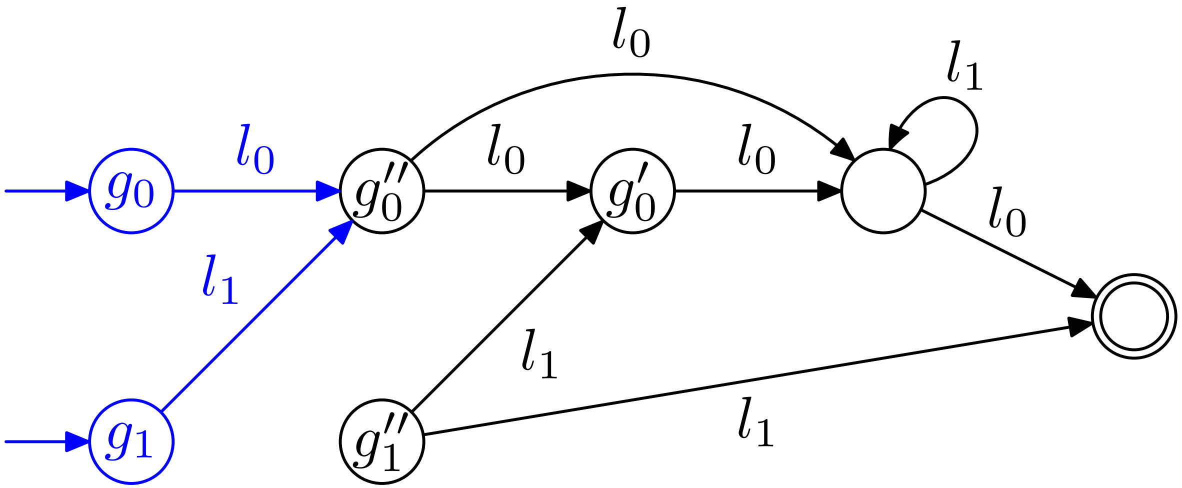
$g_0\, l_0 \rightarrow g_0$



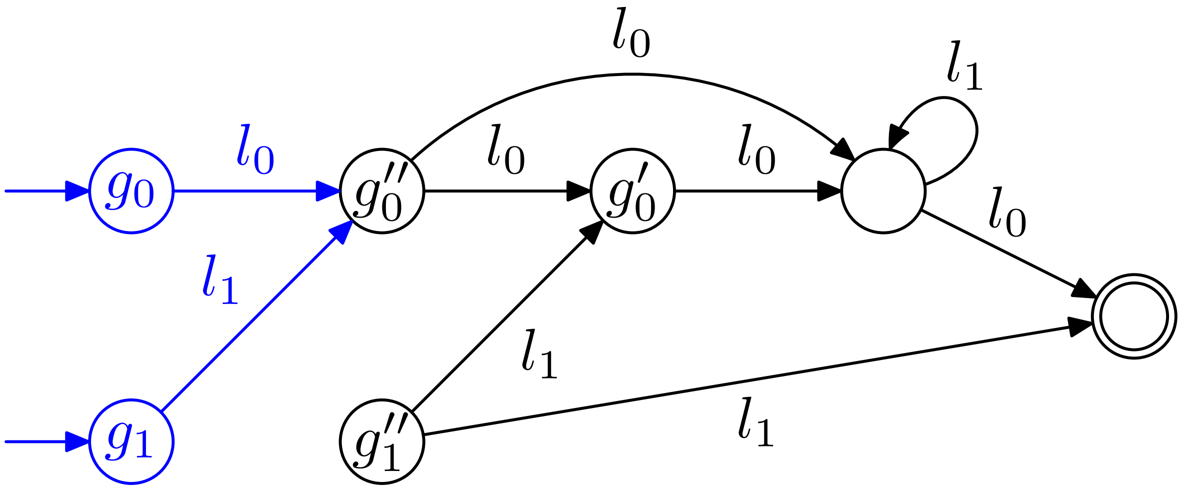
$g_1\, l_1 \rightarrow g_0$



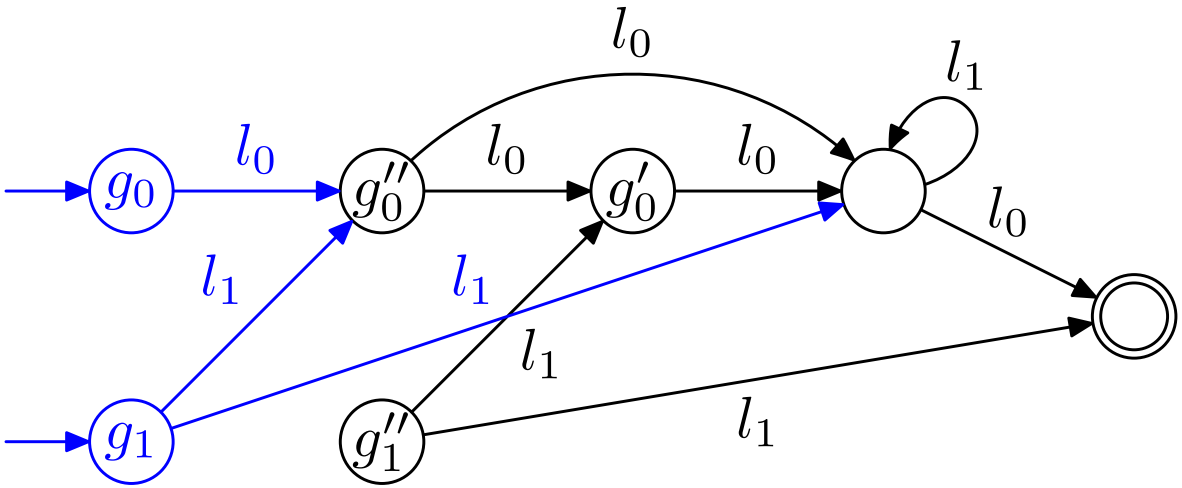
$g_1\, l_1 \rightarrow g_0$



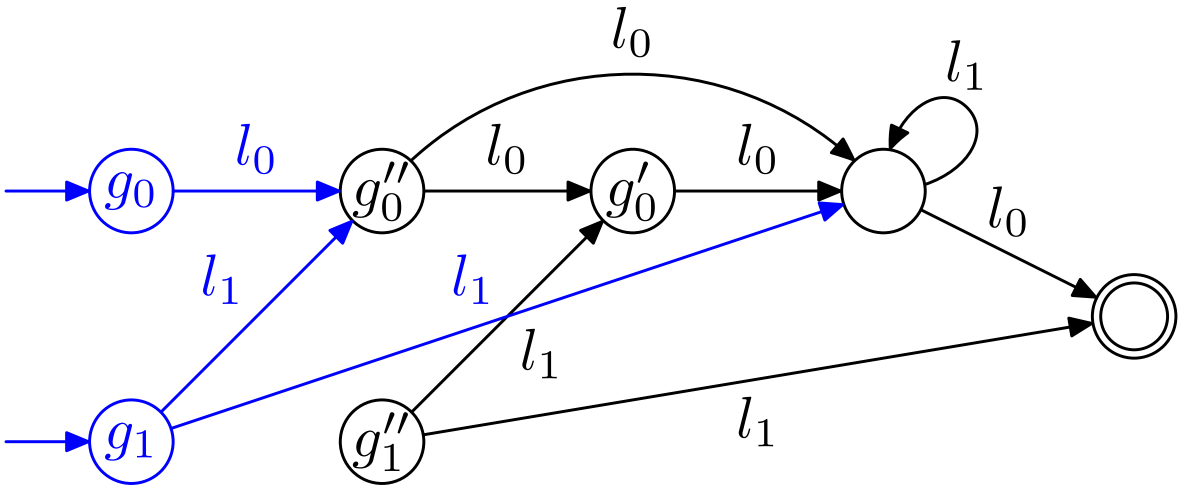
$$g_1\, l_1 \rightarrow g_1\, l_1\, l_0$$



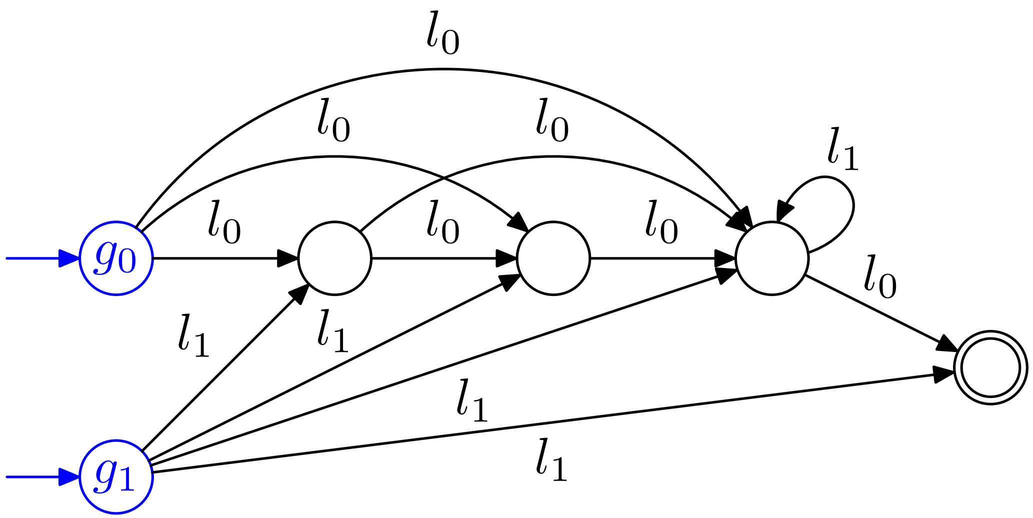
$$g_1 \, l_1 \rightarrow g_1 \, l_1 \, l_0$$



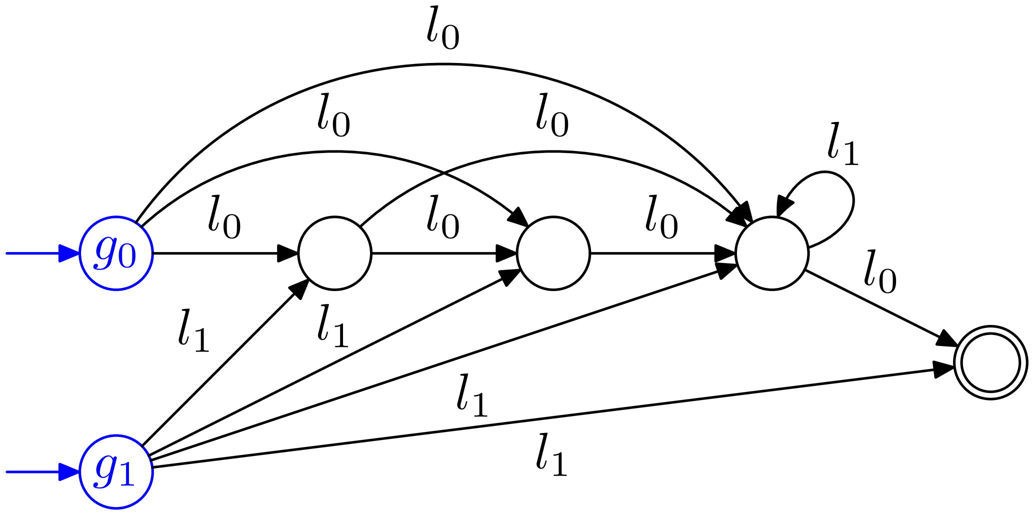
$$R = \{ \quad g_0\, l_0 \rightarrow g_0 \quad , \quad g_1\, l_1 \rightarrow g_0 \quad , \quad g_1\, l_1 \rightarrow g_1\, l_1\, l_0 \quad \}$$



$$R = \{ \quad g_0\, l_0 \to g_0 \quad , \quad g_1\, l_1 \to g_0 \quad , \quad g_1\, l_1 \to g_1\, l_1\, l_0 \quad \}$$



$$R = \{ \quad g_0\, l_0 \rightarrow g_0 \quad , \quad g_1\, l_1 \rightarrow g_0 \quad , \quad g_1\, l_1 \rightarrow g_1\, l_1\, l_0 \quad \}$$



# Termination fails

$G = \{g_0, g_1\}, L = \{l_0, l_1\}$

$R = \{ g_0 l_0 \rightarrow g_0 , g_1 l_1 \rightarrow g_0 , g_1 l_1 \rightarrow g_1 l_1 l_0 \}$

# Termination fails

$G = \{g_0, g_1\}, L = \{l_0, l_1\}$

$R = \{ g_0 l_0 \to g_0 \, , \, g_1 l_1 \to g_0 \, , \, g_1 l_1 \to g_1 l_1 l_0 \}$

$S_0 = D \qquad\qquad = g_0 l_0 l_1^* l_0 + g_1 l_1$

# Termination fails

$G = \{g_0, g_1\}, L = \{l_0, l_1\}$

$R = \{ g_0\, l_0 \rightarrow g_0\, ,\; g_1\, l_1 \rightarrow g_0\, ,\; g_1\, l_1 \rightarrow g_1\, l_1\, l_0 \}$

$$
\begin{aligned}
S_0 &= D && = g_0\, l_0\, l_1^*\, l_0 \;+\; g_1\, l_1 \\
S_1 &= S_0 \cup pre(S_0) && = g_0\, (l_0 + l_0^2)\, l_1^*\, l_0 + \\
& && \quad g_1\, l_1\, (\epsilon + l_0)\, l_1^*\, (\epsilon + l_0)
\end{aligned}
$$

# Termination fails

$G = \{g_0, g_1\}, L = \{l_0, l_1\}$

$R = \{ g_0\, l_0 \rightarrow g_0\,,\; g_1\, l_1 \rightarrow g_0\,,\; g_1\, l_1 \rightarrow g_1\, l_1\, l_0 \}$

$$
\begin{aligned}
S_0 &= D && = g_0\, l_0\, l_1^*\, l_0 + g_1\, l_1 \\
S_1 &= S_0 \cup pre(S_0) && = g_0\, (l_0 + l_0^2)\, l_1^*\, l_0 + \\
& && \quad g_1\, l_1\, (\epsilon + l_0)\, l_1^*\, (\epsilon + l_0) \\
& \ldots \\
S_i &= S_{i-1} \cup pre(S_{i-1}) && = g_0\, (l_0 + \ldots + l_0^{i+1})\, l_1^*\, l_0 + \\
& && \quad g_1\, l_1\, (\epsilon + l_0 + \ldots + l_0^i)\, l_1^* (\epsilon + l_0) \\
& \ldots
\end{aligned}
$$

However, the fixpoint

$$
\begin{aligned}
pre^*(D) &= g_0\, l_0^+ l_1^* l_0 + \\
& \quad g_1\, l_1\, l_0^* l_1^* (\epsilon + l_0)
\end{aligned}
$$

is regular.

*How can we compute it?*

# Accelerations

By definition, $pre(D) = \bigcup_{i \geq 0} S_i$
where $S_0 = D$ and $S_{i+1} = S_i \cup pre(S_i)$ for every $i \geq 0$

If convergence fails, try to compute an acceleration :
a sequence $T_0 \subseteq T_1 \subseteq T_2 \ldots$ such that

(a) $\forall i \geq 0 : S_i \subseteq T_i$

(b) $\forall i \geq 0 : T_i \subseteq \bigcup_{j \geq 0} S_j = pre(D)$

Property (a) ensures capture of (at least) the whole set $pre(D)$

Property (b) ensures that only elements of $pre(D)$ are captured

The acceleration guarantees termination if

(c) $\exists i \geq 0 : T_{i+1} = T_i$

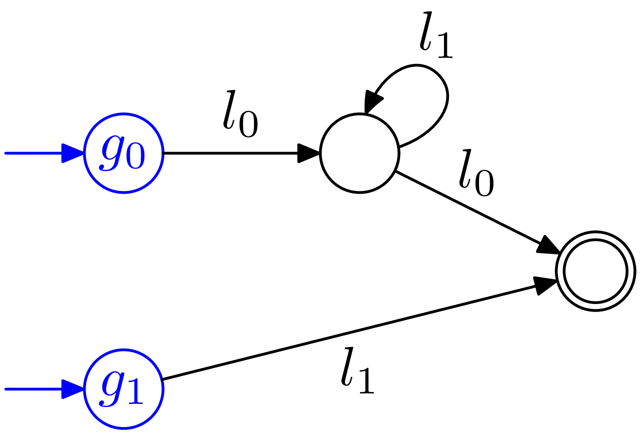
# An acceleration for prefix rewriting

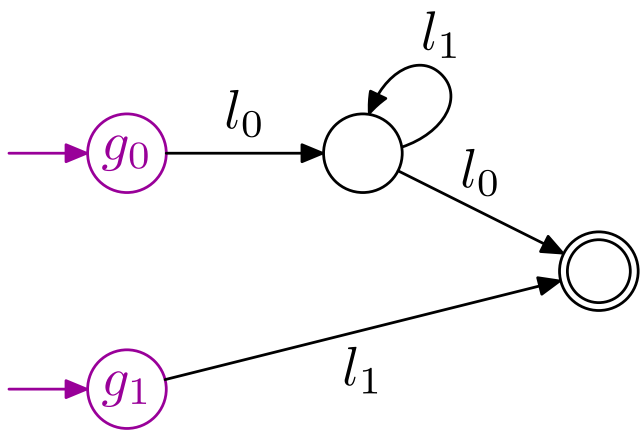
Idea: reuse the same states

# An acceleration for prefix rewriting

Idea: reuse the same states

$$R = \{g_0 l_0 \to g_0,\ g_1 l_1 \to g_0,\ g_1 l_1 \to g_1 l_1 l_0\}$$

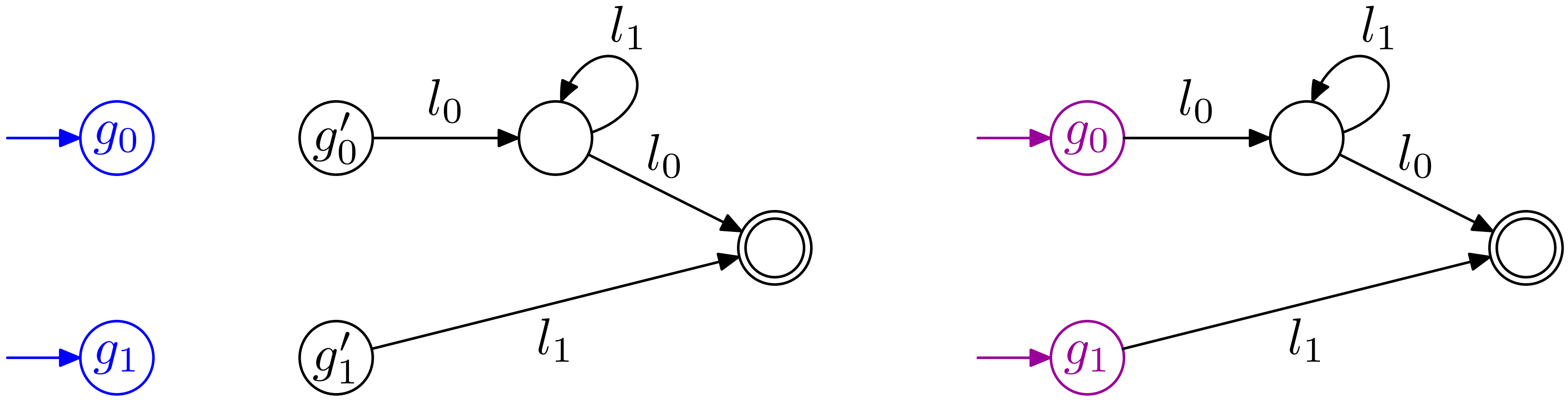


# An acceleration for prefix rewriting

Idea: reuse the same states

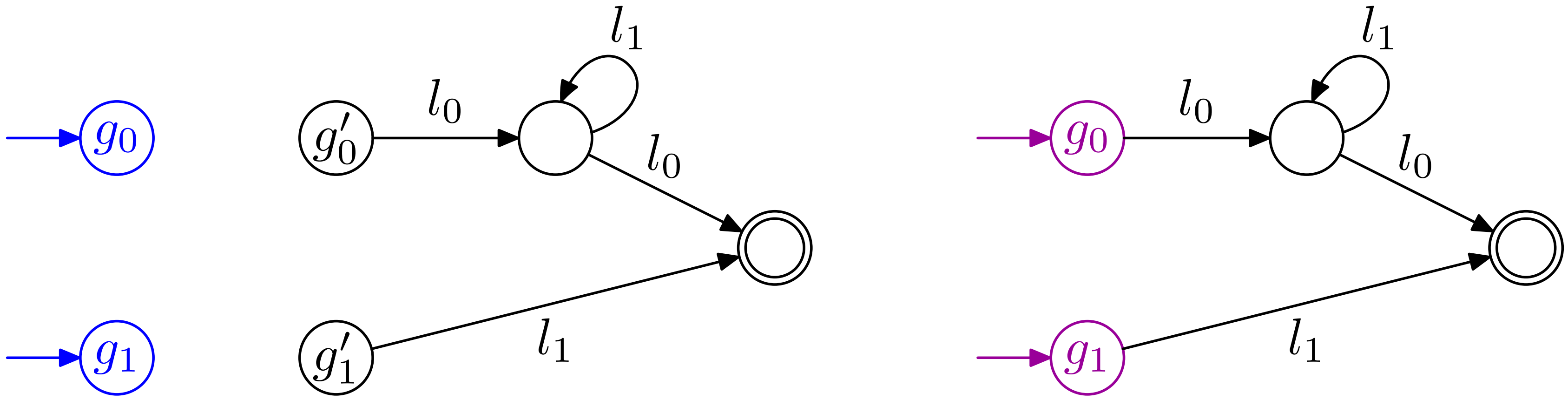
$$R = \{ g_0 l_0 \rightarrow g_0 , g_1 l_1 \rightarrow g_0 , g_1 l_1 \rightarrow g_1 l_1 l_0 \}$$



# An acceleration for prefix rewriting

Idea: reuse the same states

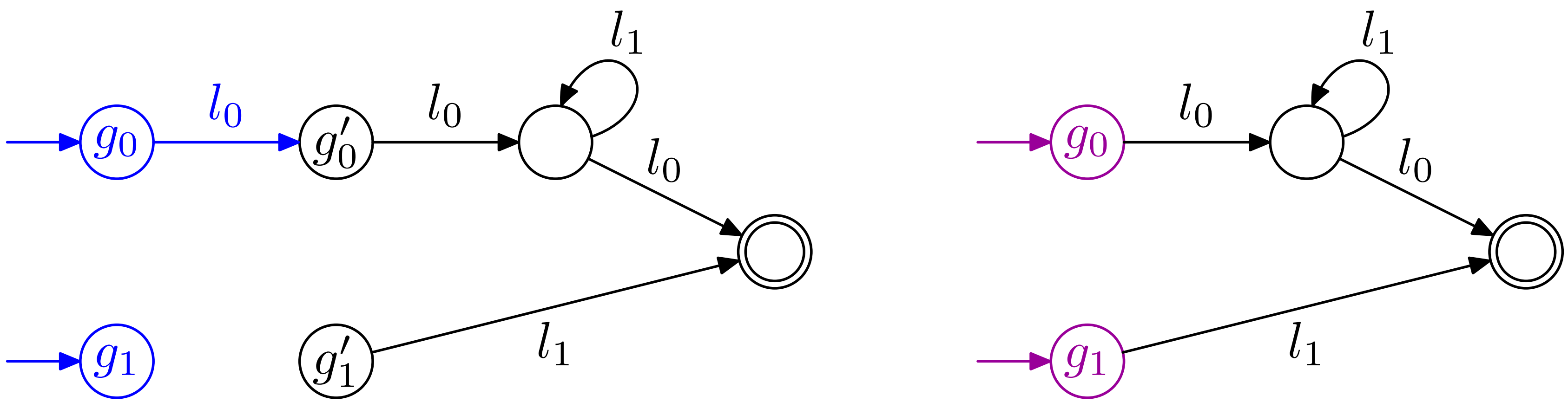
$$g_0\, l_0 \rightarrow g_0$$



# An acceleration for prefix rewriting

Idea: reuse the same states

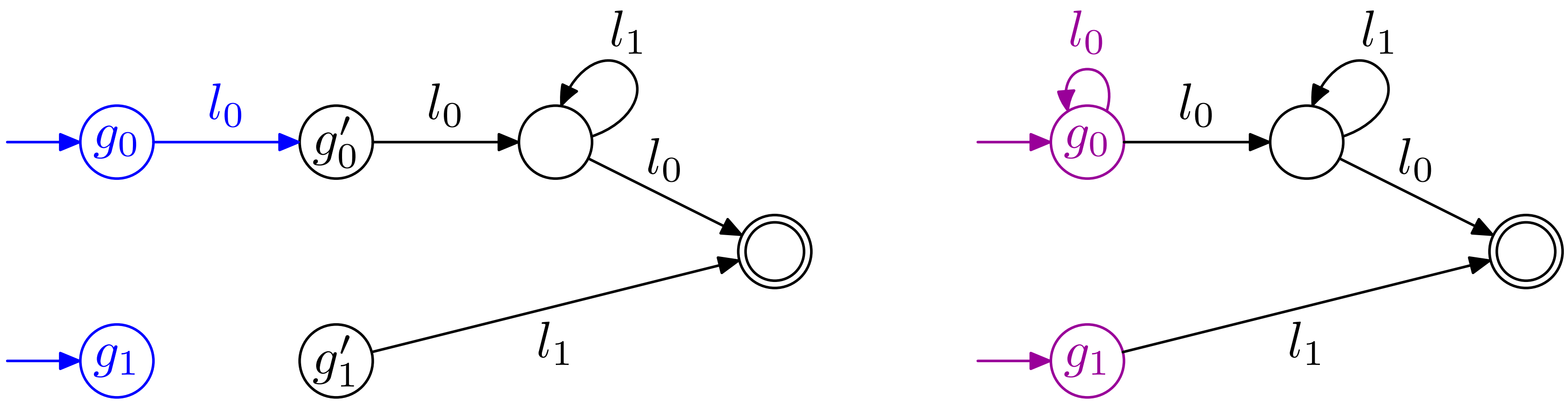
$$g_0 \, l_0 \rightarrow g_0$$



# An acceleration for prefix rewriting

Idea: reuse the same states

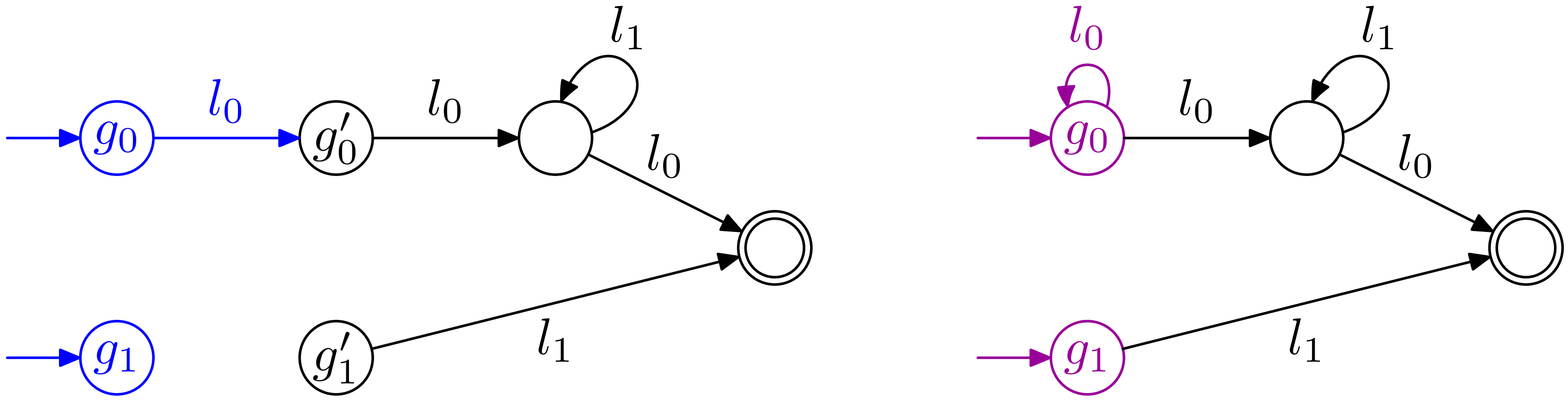
$$g_0\, l_0 \rightarrow g_0$$



# An acceleration for prefix rewriting

Idea: reuse the same states

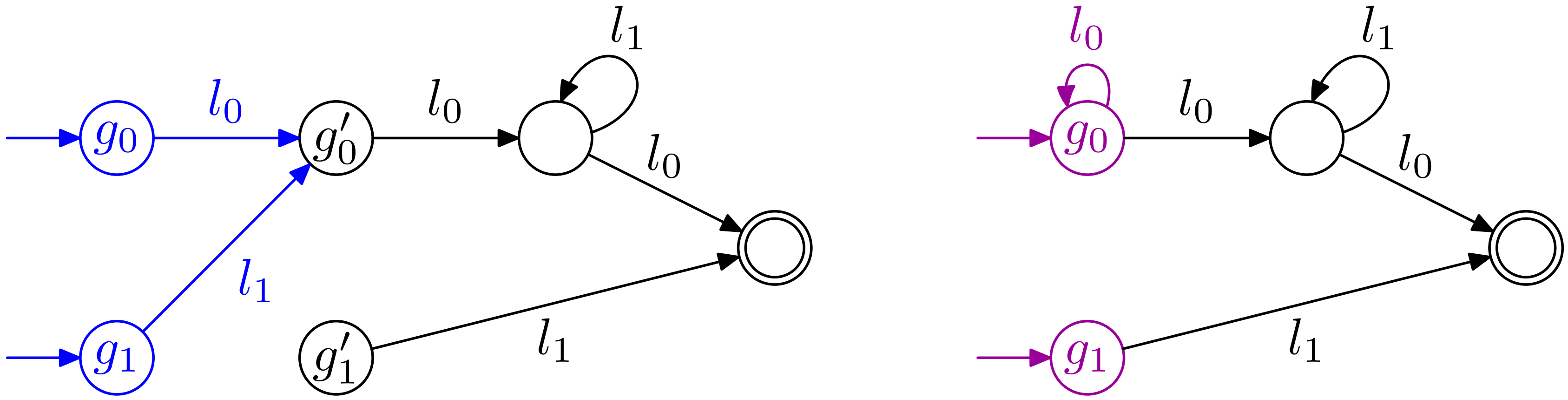
$$g_1\, l_1 \rightarrow g_0$$



# An acceleration for prefix rewriting

Idea: reuse the same states

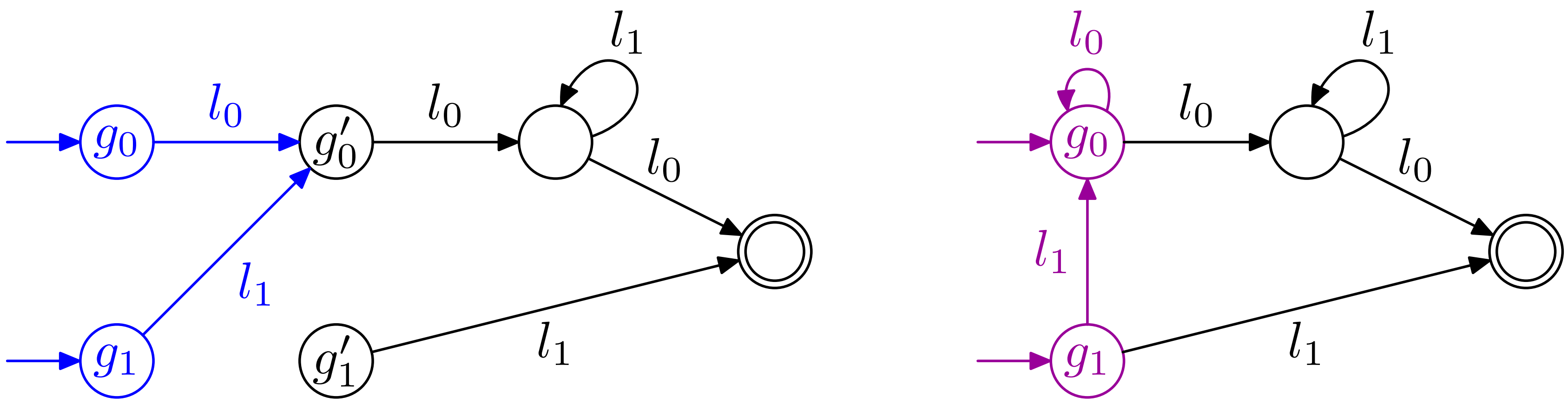
$$g_1 l_1 \rightarrow g_0$$



# An acceleration for prefix rewriting

Idea: reuse the same states

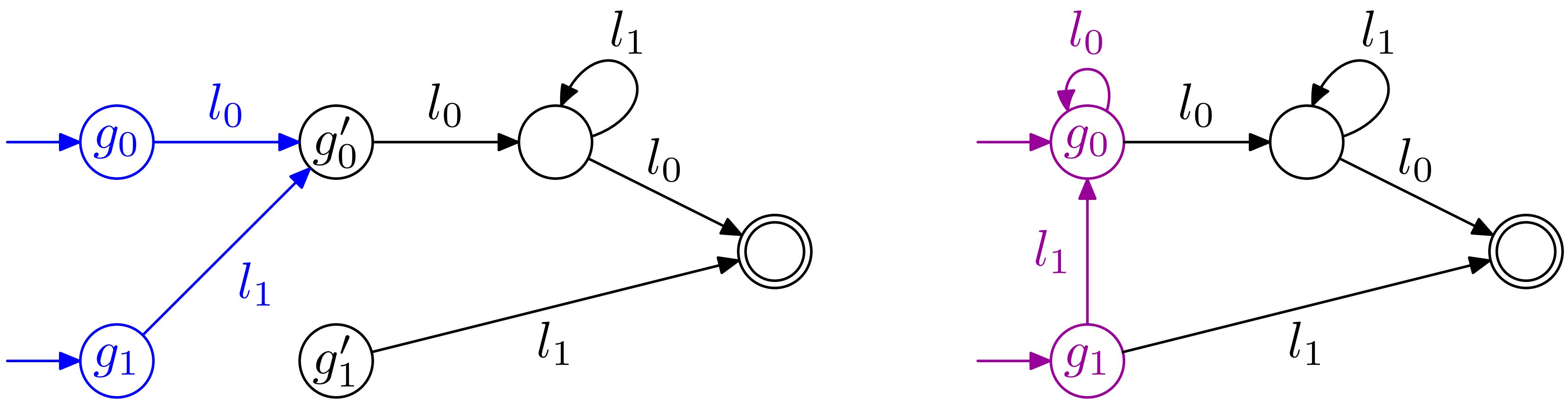
$$g_1 l_1 \rightarrow g_0$$



# An acceleration for prefix rewriting

Idea: reuse the same states

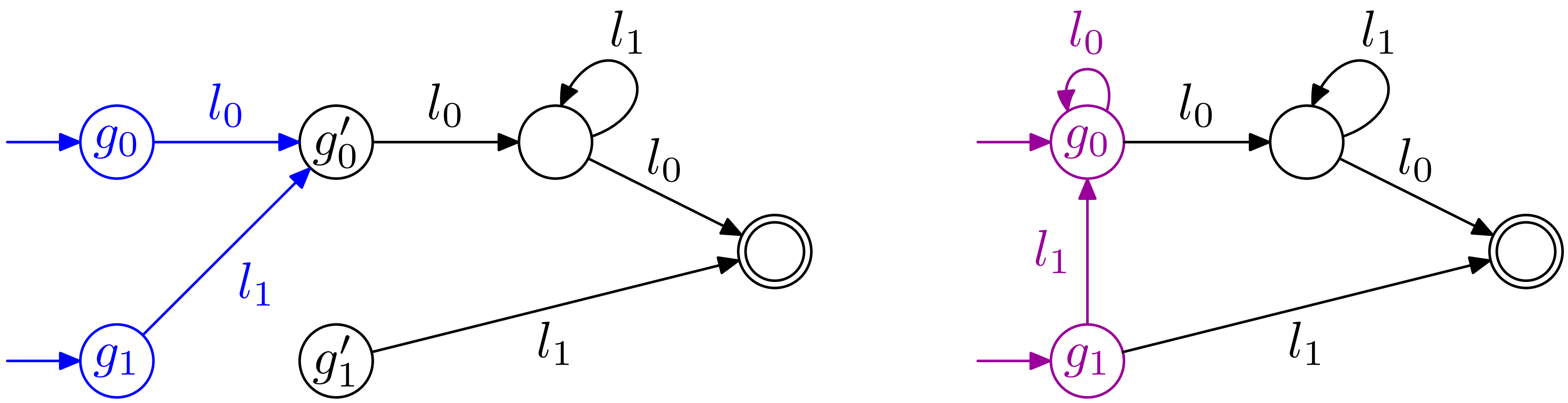
$g_1\, l_1 \rightarrow g_1\, l_1\, l_0$



# An acceleration for prefix rewriting

Idea: reuse the same states

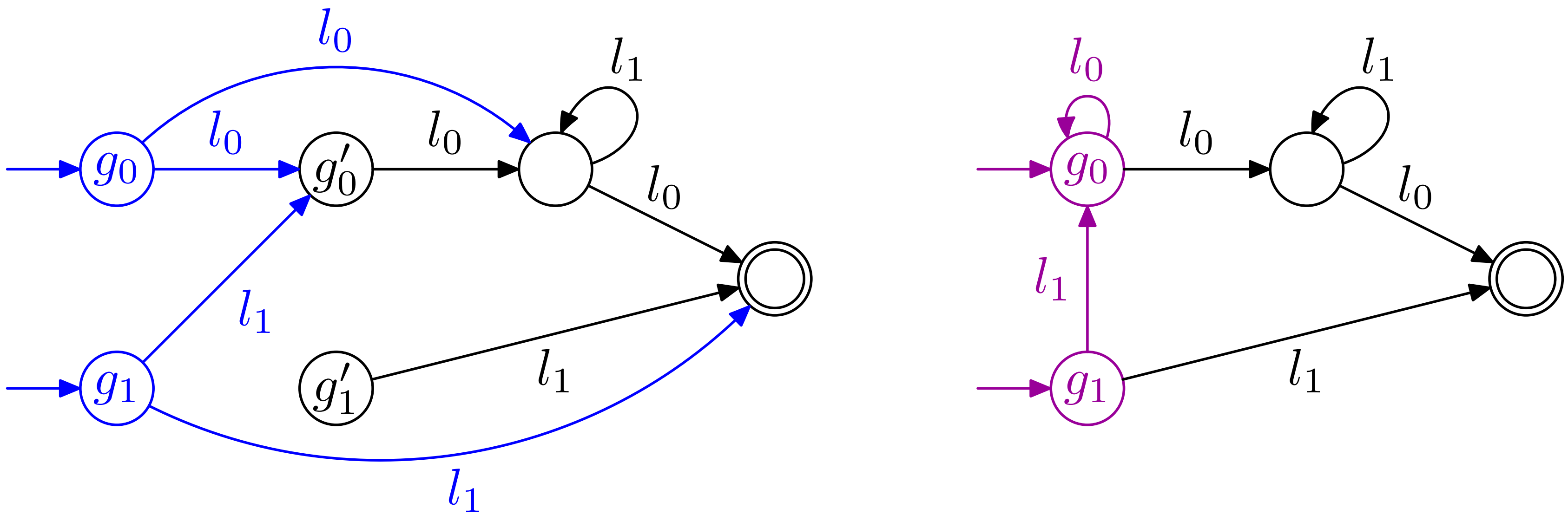
$$R = \{ g_0 l_0 \rightarrow g_0 ,\ g_1 l_1 \rightarrow g_0 ,\ g_1 l_1 \rightarrow g_1 l_1 l_0 \}$$



# An acceleration for prefix rewriting

Idea: reuse the same states

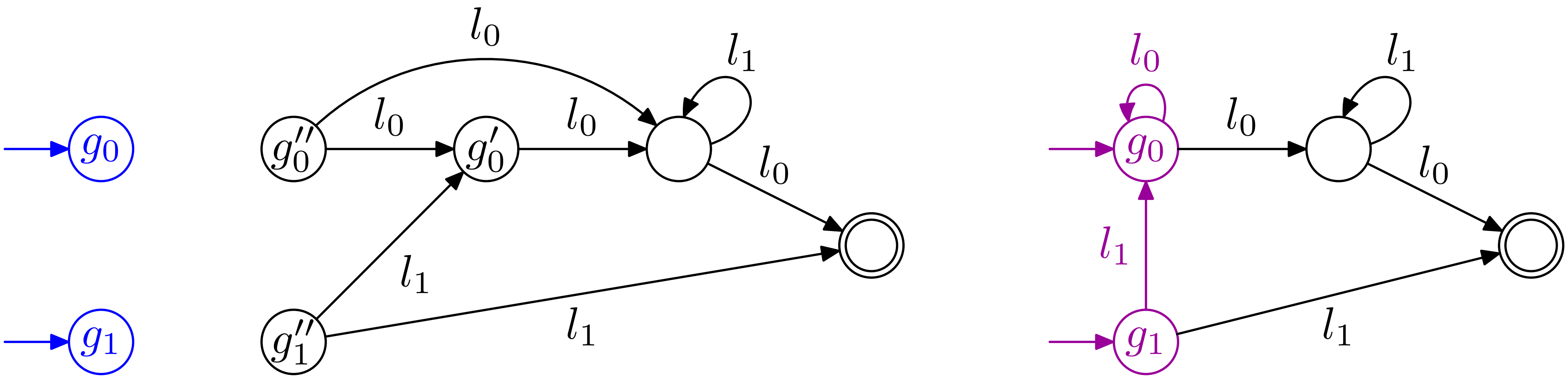
$$R = \{ g_0 l_0 \rightarrow g_0 , g_1 l_1 \rightarrow g_0 , g_1 l_1 \rightarrow g_1 l_1 l_0 \}$$



# An acceleration for prefix rewriting

Idea: reuse the same states

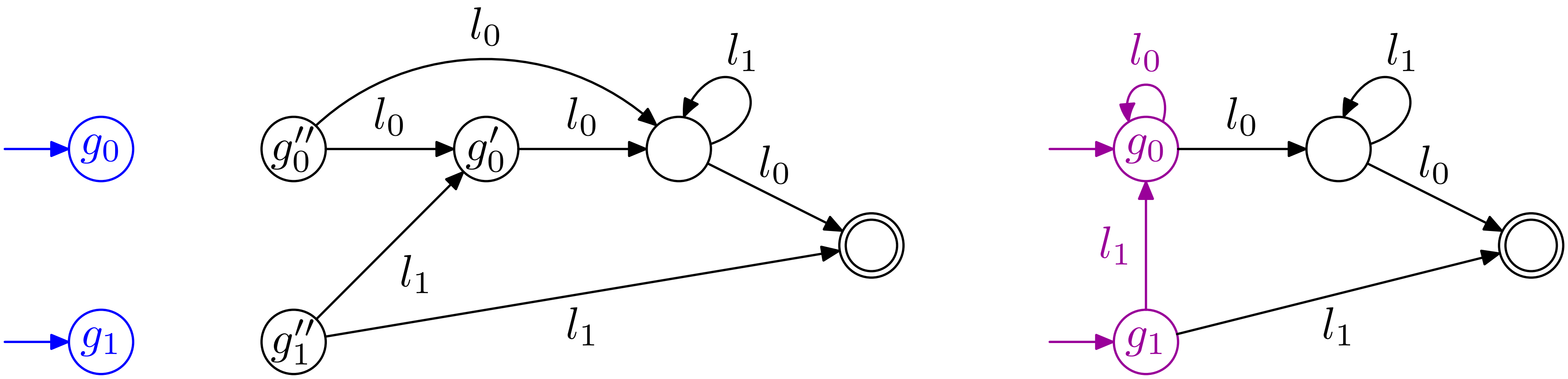
$$R = \{ g_0 l_0 \to g_0 ,\ g_1 l_1 \to g_0 ,\ g_1 l_1 \to g_1 l_1 l_0 \}$$



# An acceleration for prefix rewriting

Idea: reuse the same states

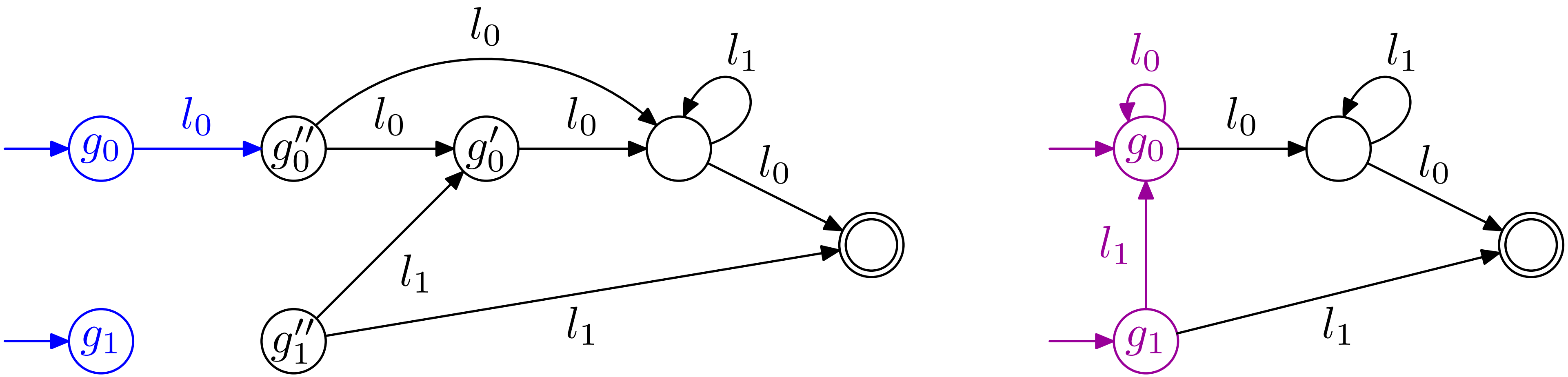
$$g_0 l_0 \rightarrow g_0$$



# An acceleration for prefix rewriting

Idea: reuse the same states

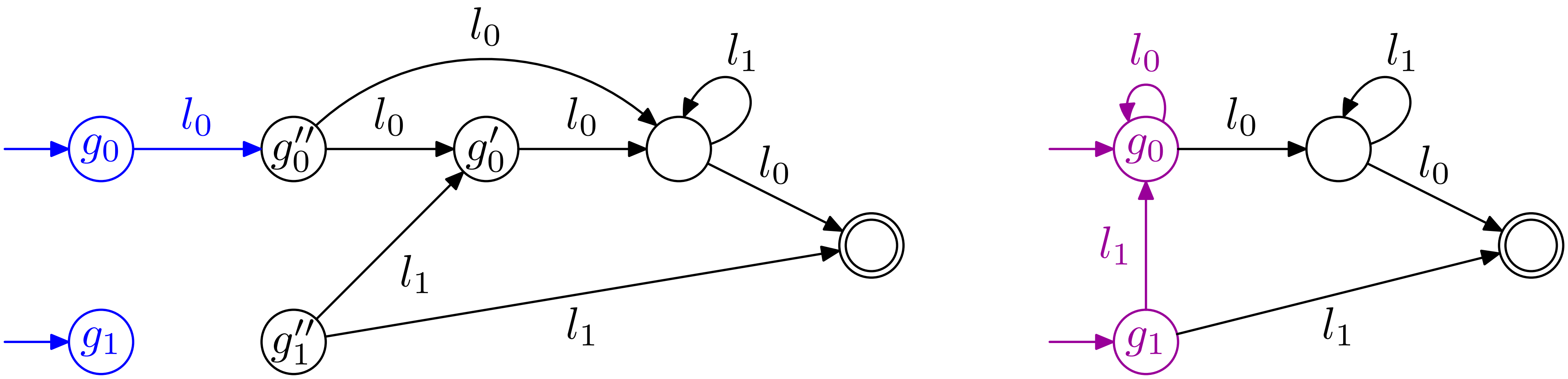
$$g_0 l_0 \to g_0$$



# An acceleration for prefix rewriting

Idea: reuse the same states

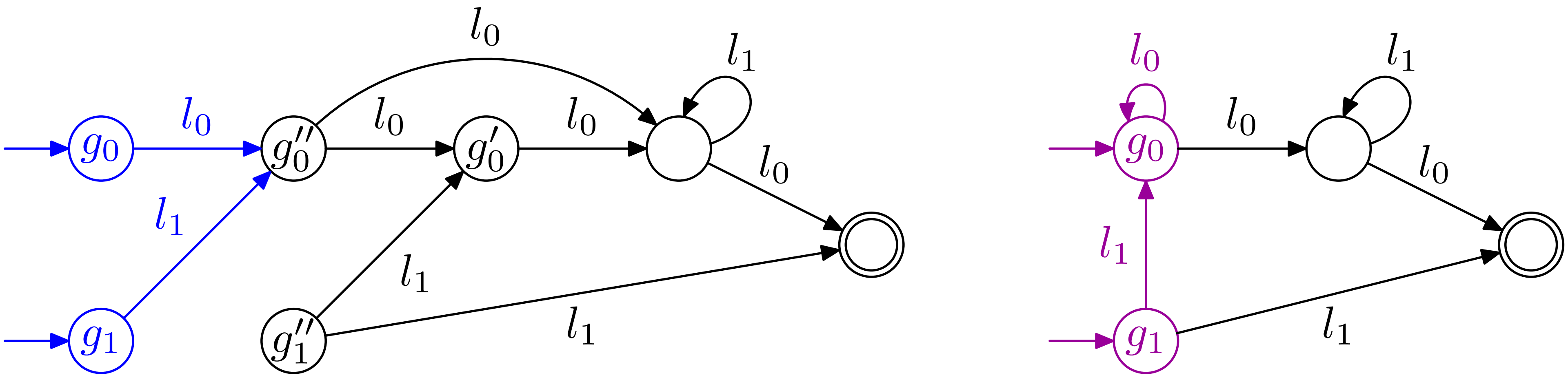
$$g_1 l_1 \to g_0$$



# An acceleration for prefix rewriting

Idea: reuse the same states

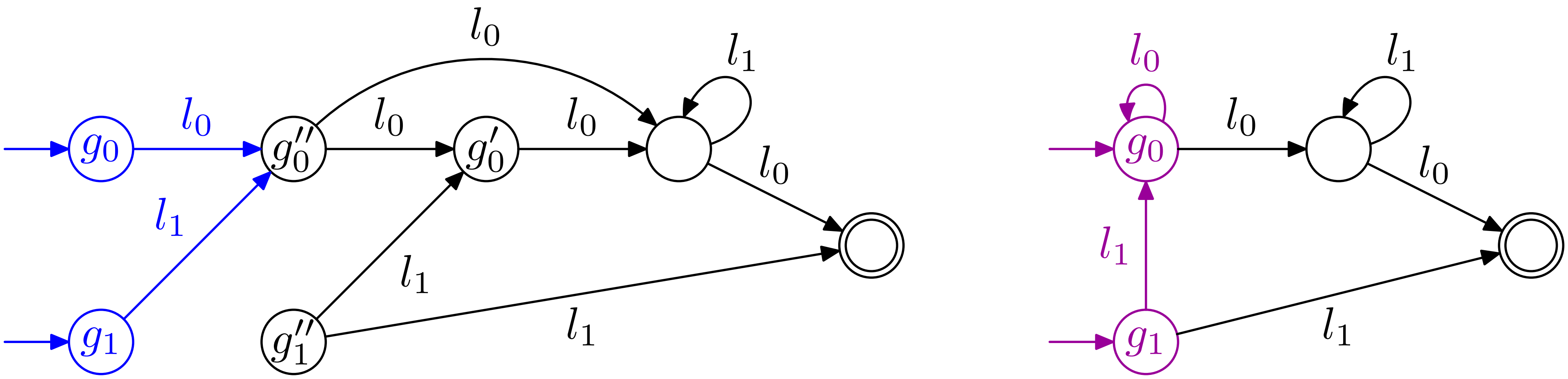
$$g_1 l_1 \to g_0$$



# An acceleration for prefix rewriting

Idea: reuse the same states

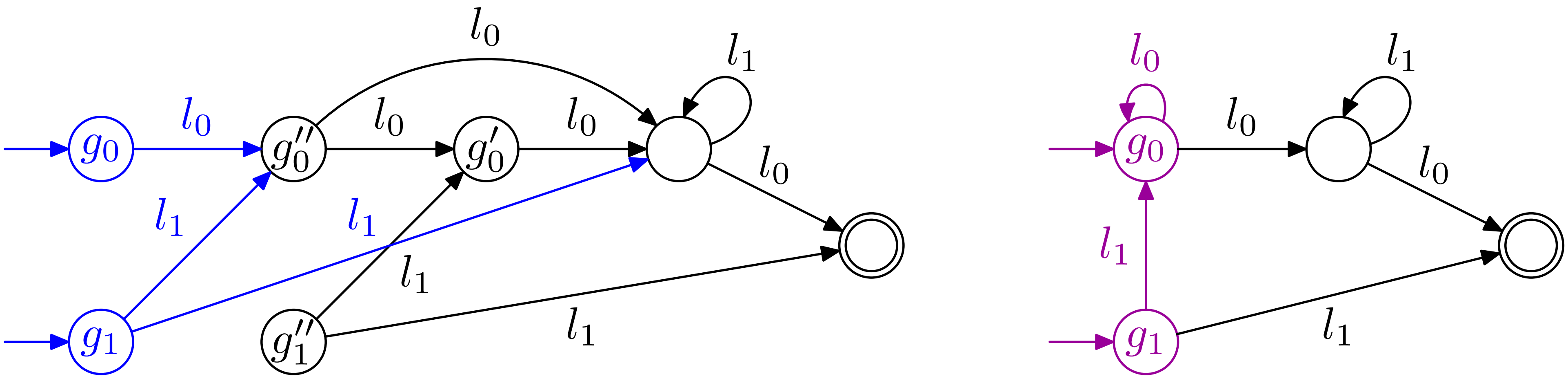
$g_1\, l_1 \rightarrow g_1\, l_1\, l_0$



# An acceleration for prefix rewriting

Idea: reuse the same states

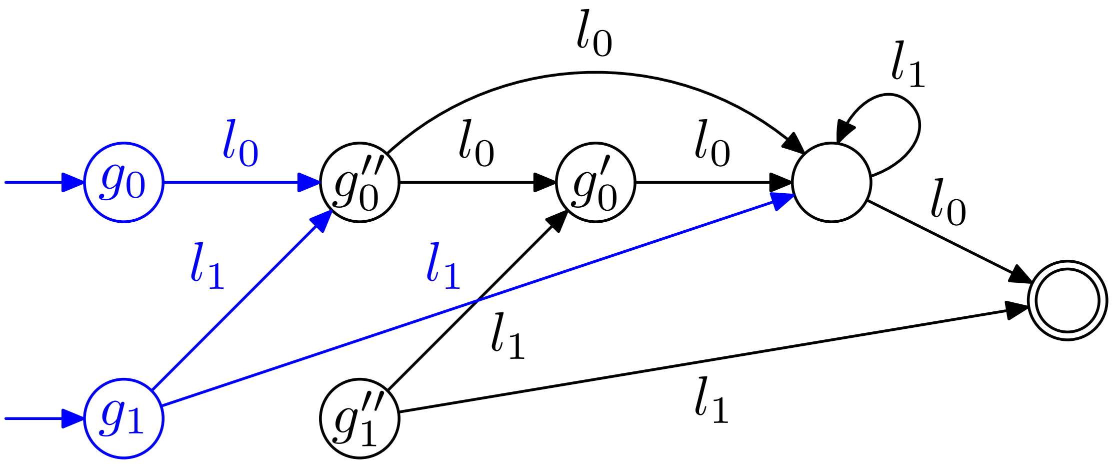
$g_1 l_1 \rightarrow g_1 l_1 l_0$

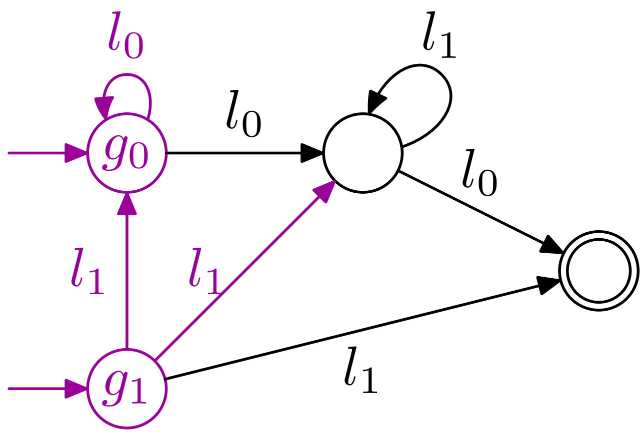


# An acceleration for prefix rewriting

Idea: reuse the same states

$g_1 \, l_1 \rightarrow g_1 \, l_1 \, l_0$

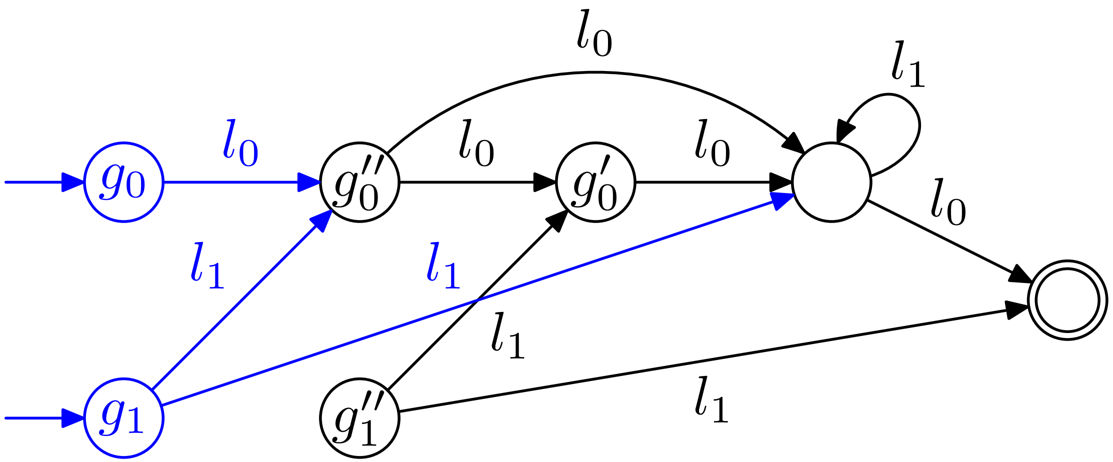


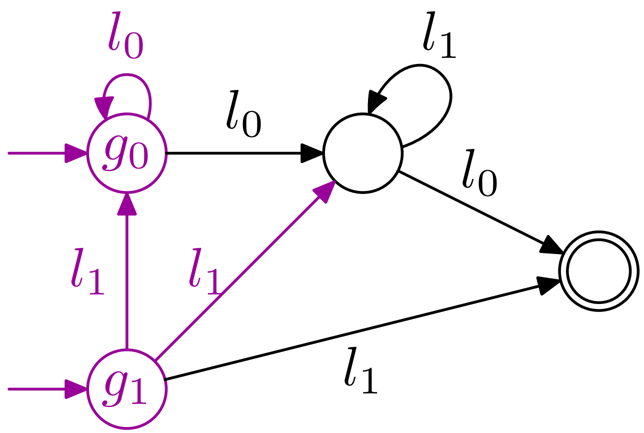



# An acceleration for prefix rewriting

Idea: reuse the same states

$$R = \{ g_0 l_0 \to g_0 , g_1 l_1 \to g_0 , g_1 l_1 \to g_1 l_1 l_0 \}$$

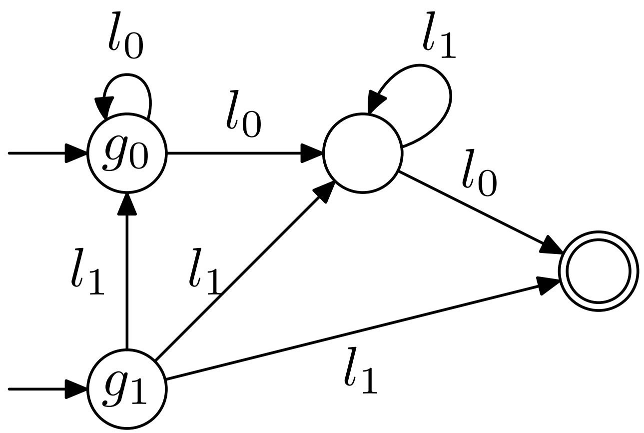


# An acceleration for prefix rewriting

Idea: reuse the same states

$$R = \{ g_0 l_0 \to g_0 ,\ g_1 l_1 \to g_0 ,\ g_1 l_1 \to g_1 l_1 l_0 \}$$



# But does it work ...?

All predecessors are computed, and termination guaranteed

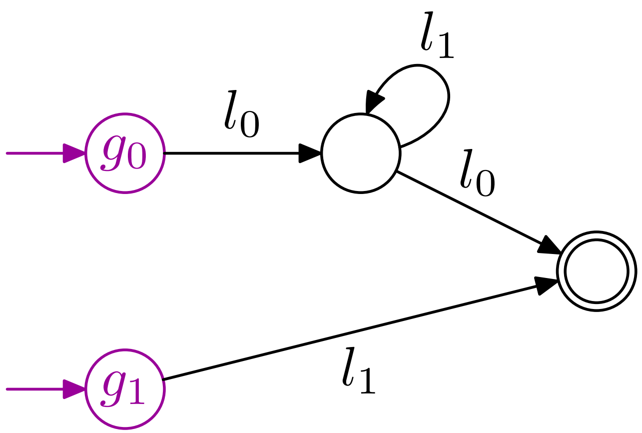
But: we might be adding non-predecessors

# But does it work ...?

All predecessors are computed, and termination guaranteed

But: we might be adding non-predecessors

$$R = \{ g_0 l_0 \to g_0 ,\ g_1 l_1 \to g_0 ,\ g_1 l_1 \to g_1 l_1 l_0 \}$$

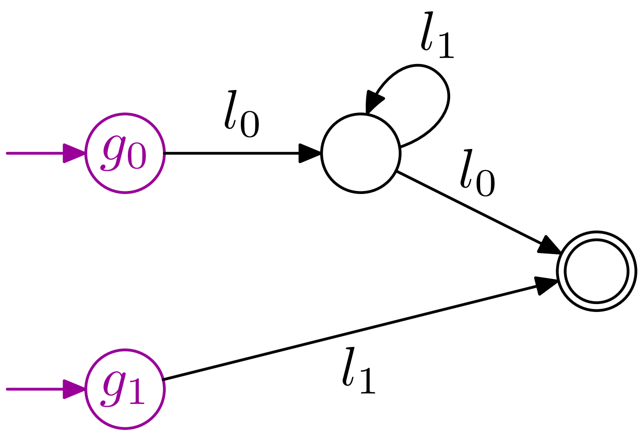


# But does it work ...?

All predecessors are computed, and termination guaranteed

But: we might be adding non-predecessors

$$g_0 \, l_0 \rightarrow g_0$$

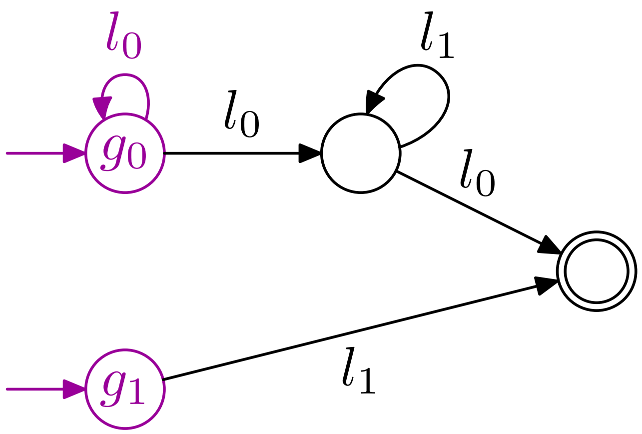


# But does it work ...?

All predecessors are computed, and termination guaranteed

But: we might be adding non-predecessors

$$g_0 \, l_0 \rightarrow g_0$$

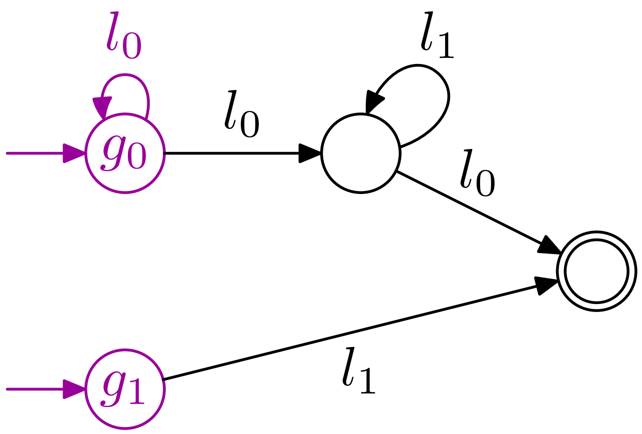


# But does it work ...?

All predecessors are computed, and termination guaranteed

But: we might be adding non-predecessors

$$R = \{ g_0 l_0 \to g_0 \,,\, g_1 l_1 \to g_0 \,,\, g_1 l_1 \to g_1 l_1 l_0 \}$$




Fortunately: correct if initial states have no incoming arcs.

# Forward search and complexity

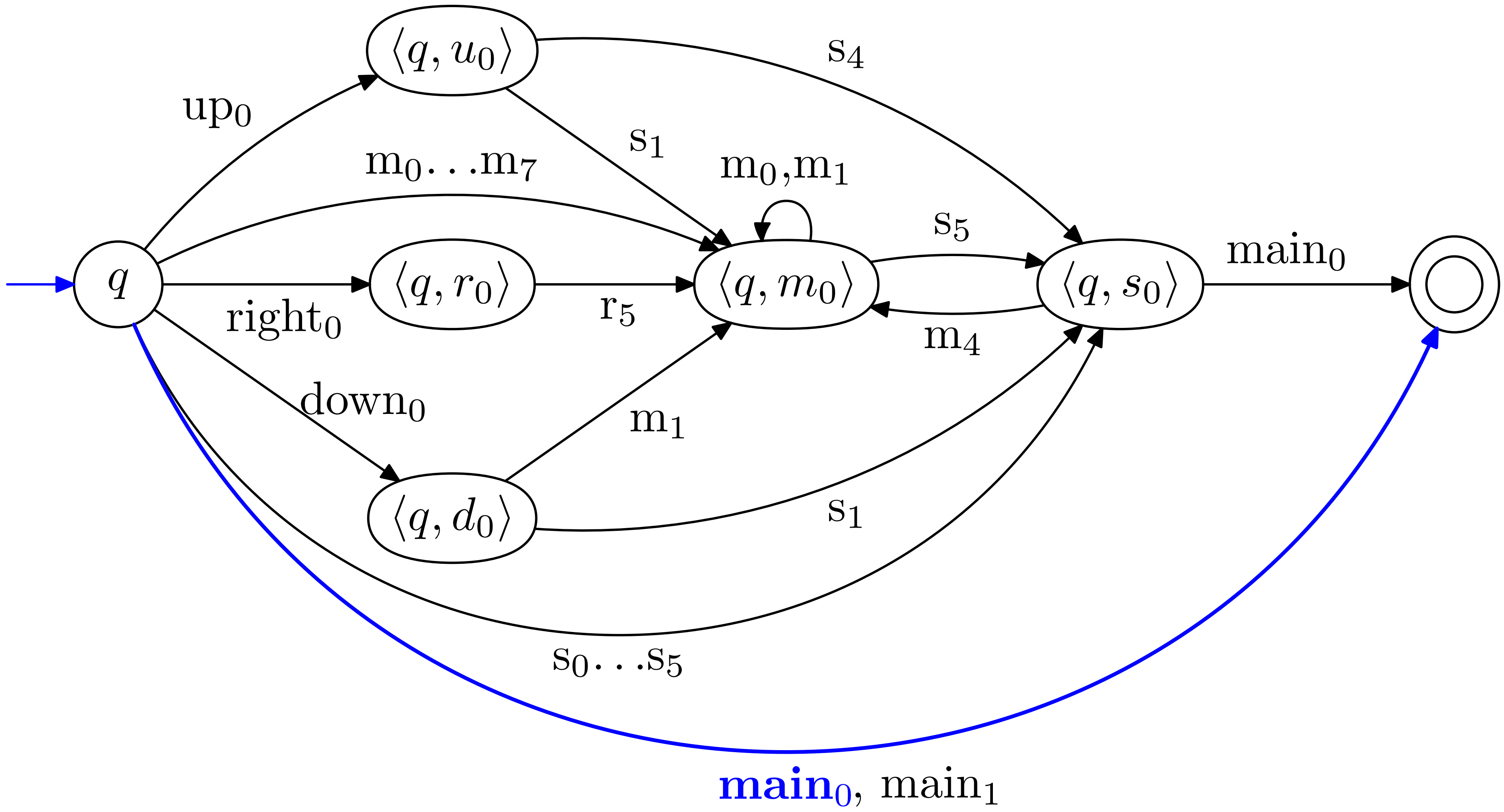
Symbolic forward search with regular sets can be accelerated in a similar way

Recall input: Alphabet $\Sigma = G \cup L$, set $R$ of rules, NFA $\mathcal{A} = (Q, L, \rightarrow_0, G, F)$ recognizing subset of $G L^*$.

Complexity of backward search: $O(|Q|^2 \cdot |R|)$ time, $O(|Q| \cdot |R| + |\rightarrow_0|)$ space.

Complexity of forward search: $O(|G| \cdot |R| \cdot (|Q \setminus G| + |R|) + |G| \cdot |\rightarrow_0|)$ time and space.

# Reachable configurations of the plotter program



# Repeated reachability for prefix rewriting

Let $I = g_0\, l_0$ and $D = g\, L^*$.

$D$ can be repeatedly reached from $I$ iff

$$g_0\, l_0 \longrightarrow^* g'\, l\, w$$

and

$$g'\, l \longrightarrow^* g\, v \longrightarrow^* g'\, l\, u$$

for some $g', l, w, v, u$.

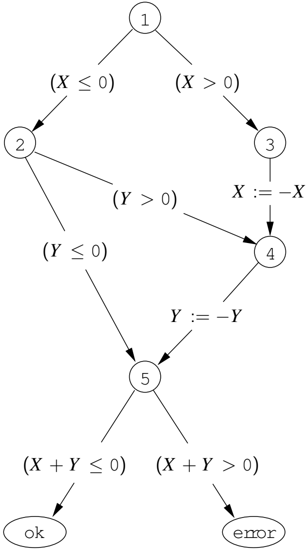
Repeated reachability can be reduced to computing several $pre^*$.

# Part III: Abstraction Refinement

Javier Esparza

Technische Universität München

# Example



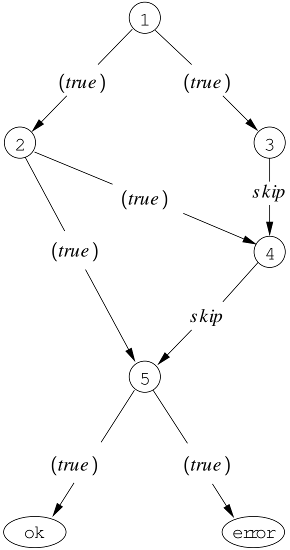

**The problem:**

- Is the error label reachable?

**The approach:**

- Upgrade a BDD checker with abstraction refinement

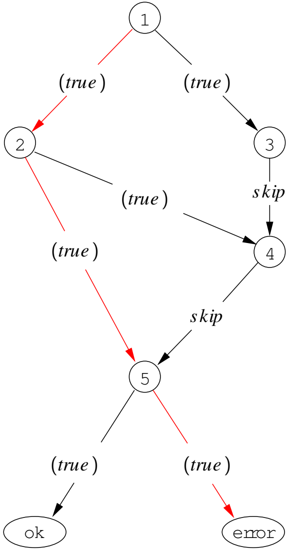
# Example




**Model-check the abstract program:**

- Is the error label reachable considering only control flow?

# Example

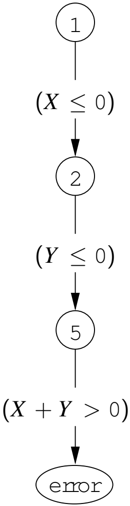



**Model-check the abstract program:**

- Is the error label reachable considering only control flow?

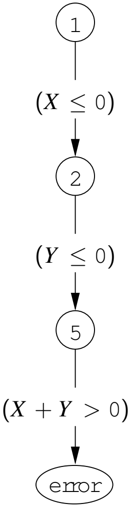
Yes!

# Example




The concrete instructions are inserted again.

# Example

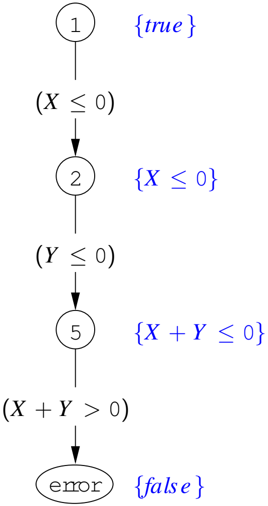



The concrete instructions are inserted again.

**Analysis of the trace**

- Is it real or spurious?

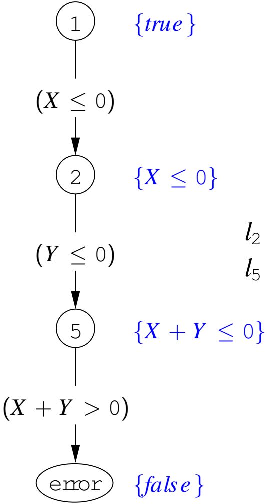
# Example



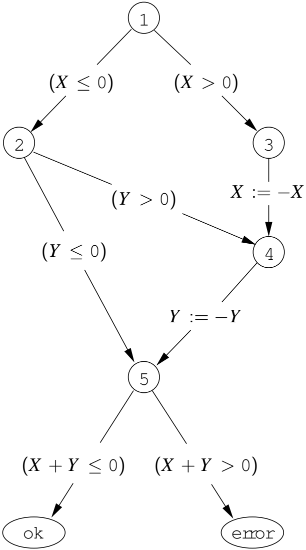

The concrete instructions are inserted again.

**Analysis of the trace**

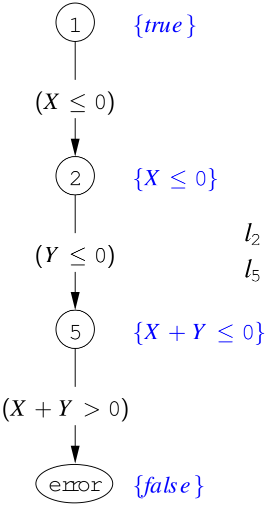
- Is it real or spurious?
- Spurious! ⇒ Hoare proof



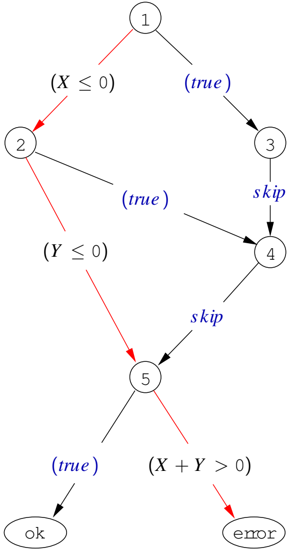

$$l_2 = (X \leq 0)$$
$$l_5 = (X + Y \leq 0)$$



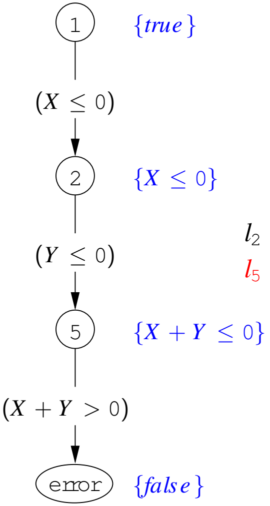
# Example




$$l_2 = (X \leq 0)$$
$$l_5 = (X + Y \leq 0)$$

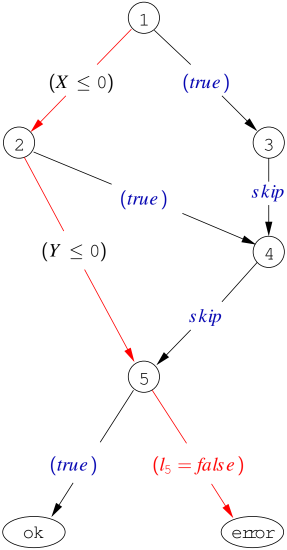


# Example

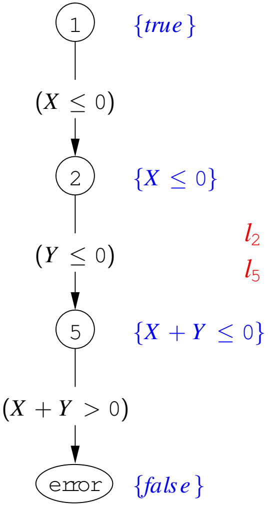



$l_2 = (X \leq 0)$

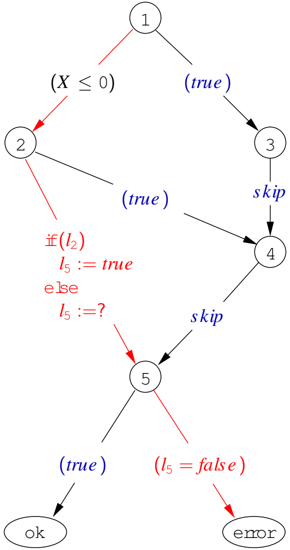
$l_5 = (X + Y \leq 0)$



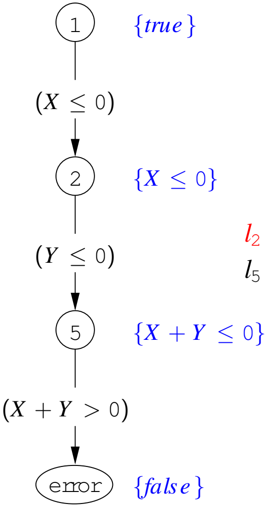
# Example



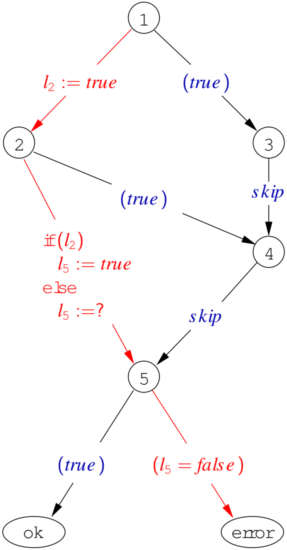

$$l_2 = (X \leq 0)$$
$$l_5 = (X + Y \leq 0)$$

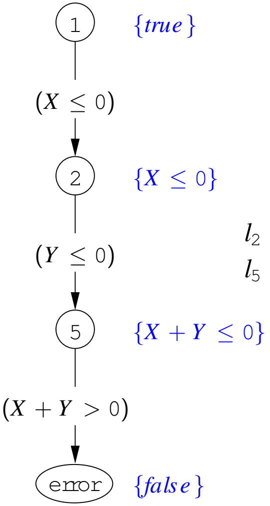


# Example

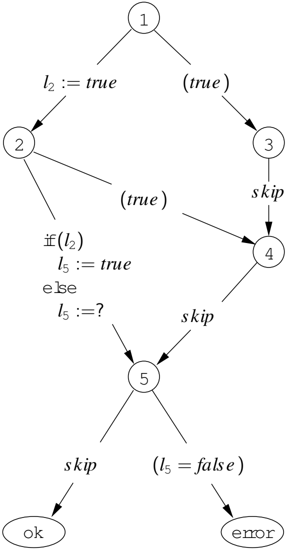



$$l_2 = (X \leq 0)$$
$$l_5 = (X + Y \leq 0)$$

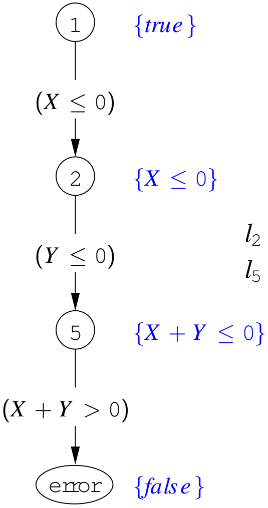




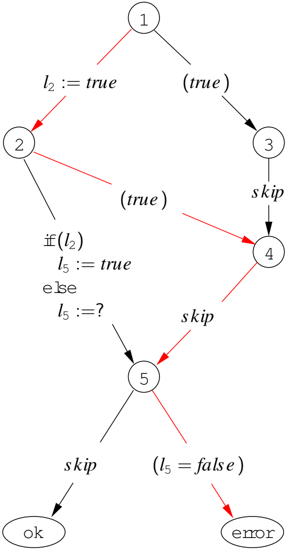

$$l_2 = (X \leq 0)$$
$$l_5 = (X + Y \leq 0)$$

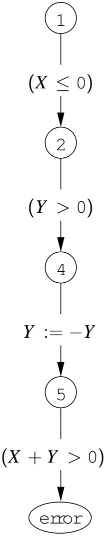


# Example



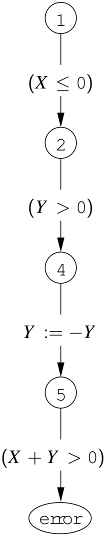

$$l_2 = (X \leq 0)$$
$$l_5 = (X + Y \leq 0)$$






The concrete instructions are inserted again.

# Example

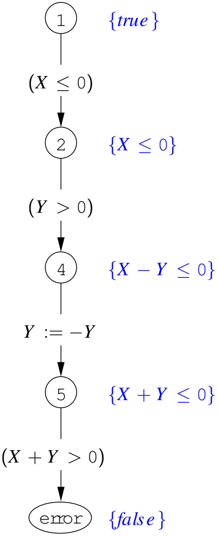



The concrete instructions are inserted again.

**Analysis of the trace**

- Is it real or spurious?

# Example

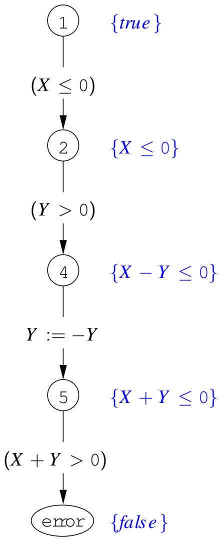



The concrete instructions are inserted again.

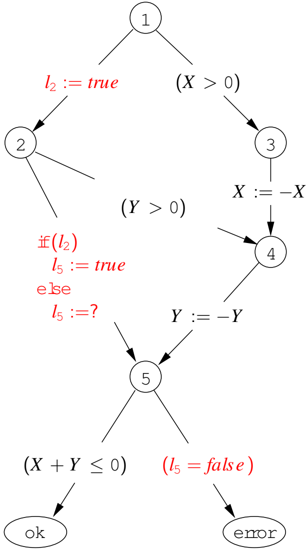
**Analysis of the trace**

- Is it real or spurious?
- Spurious! ⇒ Hoare-like proof

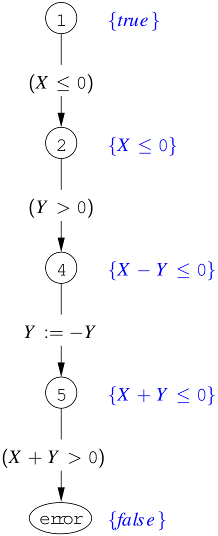
# Example



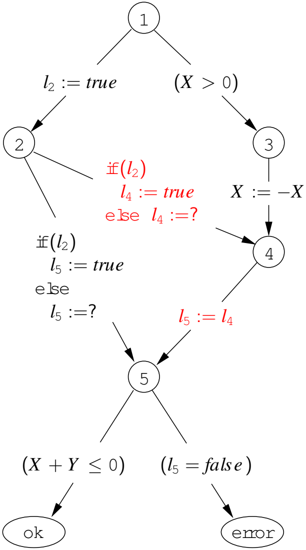

$$l_2 \leftrightarrow (X \leq 0)$$
$$l_4 \leftrightarrow (X - Y \leq 0)$$
$$l_5 \leftrightarrow (X + Y \leq 0)$$



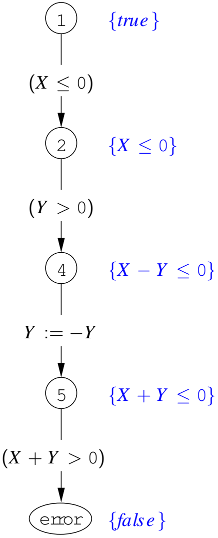
# Example



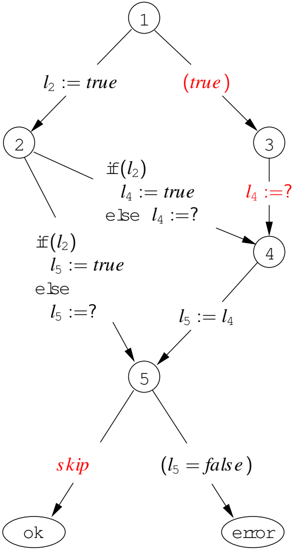

$$l_2 \leftrightarrow (X \leq 0)$$
$$l_4 \leftrightarrow (X - Y \leq 0)$$
$$l_5 \leftrightarrow (X + Y \leq 0)$$



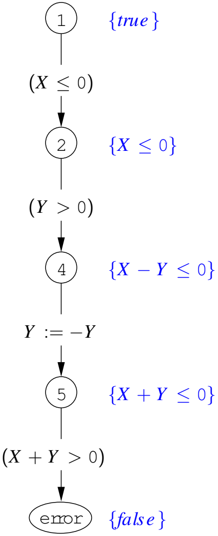
# Example



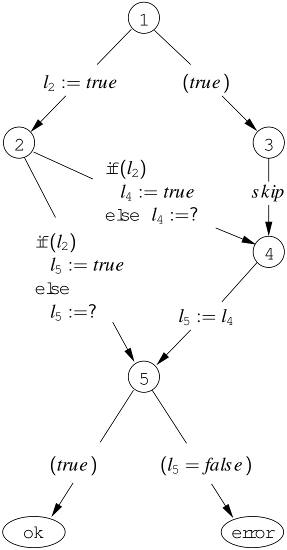

$$l_2 \leftrightarrow (X \leq 0)$$
$$l_4 \leftrightarrow (X - Y \leq 0)$$
$$l_5 \leftrightarrow (X + Y \leq 0)$$



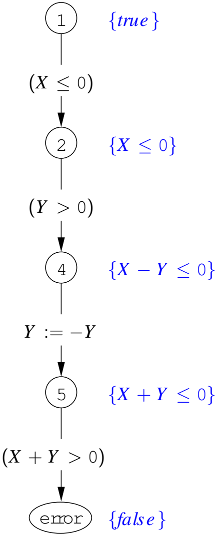
# Example



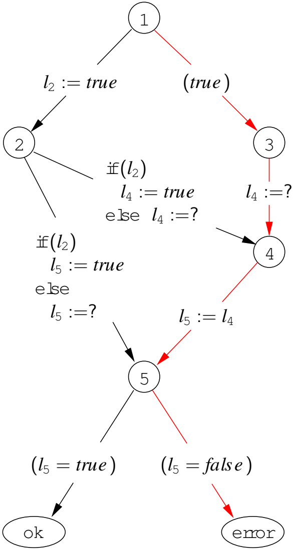

$$l_2 \leftrightarrow (X \leq 0)$$
$$l_4 \leftrightarrow (X - Y \leq 0)$$
$$l_5 \leftrightarrow (X + Y \leq 0)$$



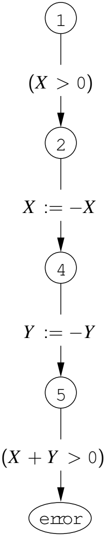
# Example




$$l_2 \leftrightarrow (X \leq 0)$$
$$l_4 \leftrightarrow (X - Y \leq 0)$$
$$l_5 \leftrightarrow (X + Y \leq 0)$$

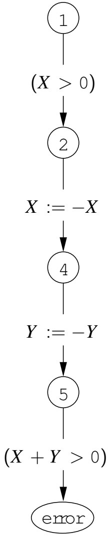


# Example




The concrete instructions are inserted again.

# Example

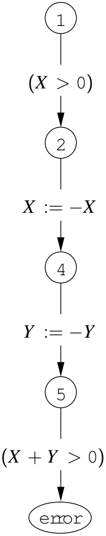



The concrete instructions are inserted again.

**Analysis of the trace**

- Is it real or spurious?

# Example

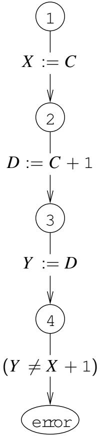



The concrete instructions are inserted again.

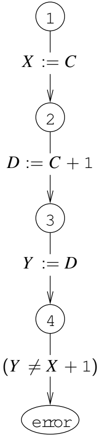
**Analysis of the trace**

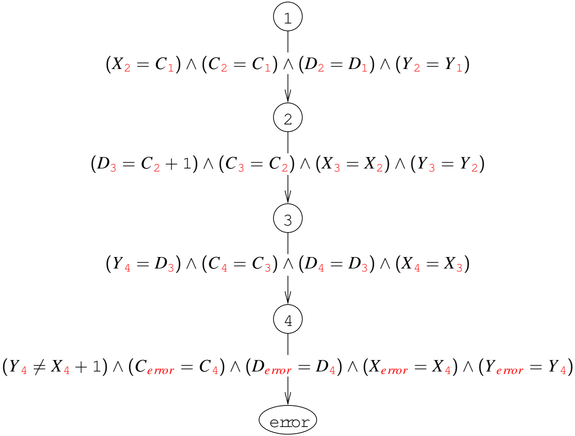
- Is it real or spurious?
- Real! $\Rightarrow$ Report it to the user!

# A Spurious Trace is an Unsatisfiable Formula.

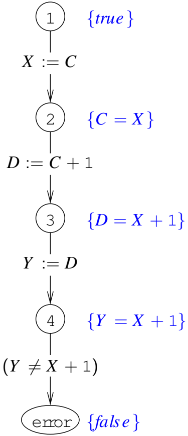


# A Spurious Trace is an Unsatisfiable Formula.



# What is a Hoare-Proof of Spuriousness?



# What is a Hoare-Proof of Spuriousness?

1 $\{true\}$

$X := C$

2 $\{C = X\}$

$D := C + 1$

3 $\{D = X + 1\}$

$Y := D$

4 $\{Y = X + 1\}$

$(Y \neq X + 1)$

error $\{false\}$

1 $\{true\}$

$(X_2 = C_1) \wedge (C_2 = C_1) \wedge (D_2 = D_1) \wedge (Y_2 = Y_1)$

2 $\{C_2 = X_2\}$

$(D_3 = C_2 + 1) \wedge (C_3 = C_2) \wedge (X_3 = X_2) \wedge (Y_3 = Y_2)$

3 $\{D_3 = X_3 + 1\}$

$(Y_4 = D_3) \wedge (C_4 = C_3) \wedge (D_4 = D_3) \wedge (X_4 = X_3)$

4 $\{Y_4 = X_4 + 1\}$

$(Y_4 \neq X_4 + 1) \wedge (C_{error} = C_4) \wedge (D_{error} = D_4) \wedge (X_{error} = X_4) \wedge (Y_{error} = Y_4)$

error $\{false\}$

# What is a Hoare Proof of Spuriousness?

## Observations

- A blue predicate $\{\cdots\}$ is implied by the conjunction of the **instructions above.**

# What is a Hoare Proof of Spuriousness?

## Observations

- A blue predicate $\{\cdots\}$ is implied by the conjunction of the **instructions above**.
- A blue predicate is unsatisfiable together with the conjunction of the **instructions below**.

# What is a Hoare Proof of Spuriousness?

## Observations

- A blue predicate $\{\cdots\}$ is implied by the conjunction of the **instructions above**.
- A blue predicate is unsatisfiable together with the conjunction of the **instructions below**.
- A blue predicate, together with the **next instruction**, implies the next blue predicate.

The last property is called Tracking Property.

**Definition (Craig interpolant)**

Let $(F,G)$ be a pair of formulas with $F \wedge G$ unsatisfiable.
An interpolant for $(F,G)$ is a formula $I$
with the following properties:

- $F \models I$,
- $I \wedge G$ is unsatisfiable and
- $I$ refers only to the common variables of $F$ and $G$.

**Definition (Craig interpolant)**

Let $(F, G)$ be a pair of formulas with $F \wedge G$ unsatisfiable.
An interpolant for $(F, G)$ is a formula $I$
with the following properties:

- $F \models I$,
- $I \wedge G$ is unsatisfiable and
- $I$ refers only to the common variables of $F$ and $G$.

**Example**

$$F = x \wedge y \qquad G = \neg x \wedge z$$

$I = x$ is an interpolant for $(F, G)$.

# Craig Interpolation: Our Application

1 $\{true\}$

$$(X_2 = C_1) \wedge (C_2 = C_1) \wedge (D_2 = D_1) \wedge (Y_2 = Y_1)$$

2 $\{C_2 = X_2\}$

$$(D_3 = C_2 + 1) \wedge (C_3 = C_2) \wedge (X_3 = X_2) \wedge (Y_3 = Y_2)$$

3 $\{D_3 = X_3 + 1\}$

$$(Y_4 = D_3) \wedge (C_4 = C_3) \wedge (D_4 = D_3) \wedge (X_4 = X_3)$$

4 $\{Y_4 = X_4 + 1\}$

$$(Y_4 \neq X_4 + 1) \wedge (C_{error} = C_4) \wedge (D_{error} = D_4) \wedge (X_{error} = X_4) \wedge (Y_{error} = Y_4)$$

error $\{false\}$

1

$$(X_2 = C_1) \wedge (C_2 = C_1) \wedge (D_2 = D_1) \wedge (Y_2 = Y_1)$$

2 $\{C_2 = X_2\}$

$$(D_3 = C_2 + 1) \wedge (C_3 = C_2) \wedge (X_3 = X_2) \wedge (Y_3 = Y_2)$$

3

$$(Y_4 = D_3) \wedge (C_4 = C_3) \wedge (D_4 = D_3) \wedge (X_4 = X_3)$$

4

$$(Y_4 \neq X_4 + 1) \wedge (C_{error} = C_4) \wedge (D_{error} = D_4) \wedge (X_{error} = X_4) \wedge (Y_{error} = Y_4)$$

error

# Craig Interpolation: Our Application

1

$$(X_2 = C_1) \wedge (C_2 = C_1) \wedge (D_2 = D_1) \wedge (Y_2 = Y_1)$$

2

$$(D_3 = C_2 + 1) \wedge (C_3 = C_2) \wedge (X_3 = X_2) \wedge (Y_3 = Y_2)$$

3 $\{D_3 = X_3 + 1\}$

$$(Y_4 = D_3) \wedge (C_4 = C_3) \wedge (D_4 = D_3) \wedge (X_4 = X_3)$$

4

$$(Y_4 \neq X_4 + 1) \wedge (C_{error} = C_4) \wedge (D_{error} = D_4) \wedge (X_{error} = X_4) \wedge (Y_{error} = Y_4)$$

error

# Craig Interpolation: Our Application

1

$(X_2 = C_1) \wedge (C_2 = C_1) \wedge (D_2 = D_1) \wedge (Y_2 = Y_1)$

2

$(D_3 = C_2 + 1) \wedge (C_3 = C_2) \wedge (X_3 = X_2) \wedge (Y_3 = Y_2)$

3

$(Y_4 = D_3) \wedge (C_4 = C_3) \wedge (D_4 = D_3) \wedge (X_4 = X_3)$

4 $\{Y_4 = X_4 + 1\}$

$(Y_4 \neq X_4 + 1) \wedge (C_{error} = C_4) \wedge (D_{error} = D_4) \wedge (X_{error} = X_4) \wedge (Y_{error} = Y_4)$

error

# Craig Interpolation: Our Application

1 $\{true\}$

$$(X_2 = C_1) \wedge (C_2 = C_1) \wedge (D_2 = D_1) \wedge (Y_2 = Y_1)$$

2 $\{C_2 = X_2\}$

$$(D_3 = C_2 + 1) \wedge (C_3 = C_2) \wedge (X_3 = X_2) \wedge (Y_3 = Y_2)$$

3 $\{D_3 = X_3 + 1\}$

$$(Y_4 = D_3) \wedge (C_4 = C_3) \wedge (D_4 = D_3) \wedge (X_4 = X_3)$$

4 $\{Y_4 = X_4 + 1\}$

$$(Y_4 \neq X_4 + 1) \wedge (C_{error} = C_4) \wedge (D_{error} = D_4) \wedge (X_{error} = X_4) \wedge (Y_{error} = Y_4)$$

error $\{false\}$

# Summary

- Spurious traces $\leftrightarrow$ unsatisfiable formula
- Craig interpolants satisfying the tracking property $\rightarrow$ Hoare proofs of spuriousness
- 'Clean' Hoare proofs of spuriousness $\rightarrow$ Craig interpolants

# Weakest and Strongest Interpolants

**Definition (weakest interpolant)**

The weakest interpolant for $(F, G)$ is the interpolant for $(F, G)$ that is implied by all interpolants for $(F, G)$.
It is denoted by $WI(F, G)$.

**Definition (strongest interpolant)**

The strongest interpolant for $(F, G)$ is the interpolant for $(F, G)$ that implies all interpolants for $(F, G)$.
It is denoted by $SI(F, G)$.

We show how to compute them and that they satisfy the tracking property.

# A Charaterization of Weakest Interpolants

**Theorem (weakest interpolant)**

- *Let* $(F, G)$ *be a pair of formulas with* $F \wedge G$ *unsatisfiable.*
- *Let* $Z$ *be the variables that occur in* $G$*, but not in* $F$*.*

*Then* $WI(F, G) \equiv \forall Z . \neg G$.

# A Charaterization of Weakest Interpolants

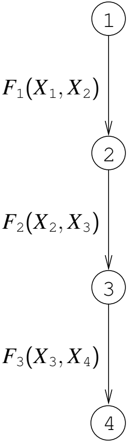
**Theorem (weakest interpolant)**

- *Let $(F, G)$ be a pair of formulas with $F \wedge G$ unsatisfiable.*
- *Let $Z$ be the variables that occur in $G$, but not in $F$.*

*Then $WI(F, G) \equiv \forall Z . \neg G$.*

Very adequate for computation with BDDs.

# Efficient Computation of Weakest Interpolants




**Theorem**

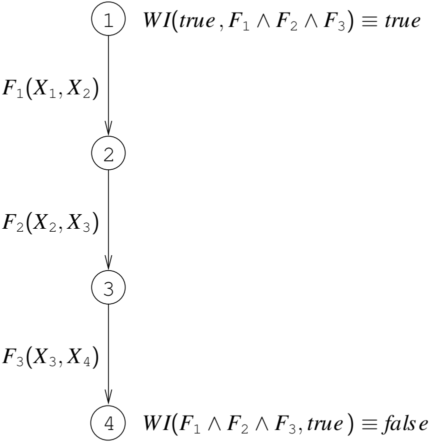
- *Let* $F_1 \wedge F_2 \wedge F_3$ *be unsatisfiable.*
- *Let* $X_3$ *be the variables that occur in* $F_2$*, but not in* $F_1$*.*

*Then*

$$WI(F_1, F_2 \wedge F_3) \equiv \forall X_3 (F_2 \rightarrow WI(F_1 \wedge F_2, F_3)).$$

**Corollary (Tracking Property)**

$$WI(F_1, F_2 \wedge F_3) \wedge F_2 \quad \models \quad WI(F_1 \wedge F_2, F_3).$$


1
$WI(true, F_1 \wedge F_2 \wedge F_3) \equiv true$
$F_1(X_1, X_2)$
2
$F_2(X_2, X_3)$
3
$F_3(X_3, X_4)$
4
$WI(F_1 \wedge F_2 \wedge F_3, true) \equiv false$

# Interpolants Computation for a Spurious Trace

1 $WI(true, F_1 \wedge F_2 \wedge F_3) \equiv true$

$F_1(X_1, X_2)$

2

$F_2(X_2, X_3)$

3 $WI(F_1 \wedge F_2, F_3) \equiv J_3(X_3) \equiv \forall X_4 (F_3 \rightarrow false)$

$F_3(X_3, X_4)$

4 $WI(F_1 \wedge F_2 \wedge F_3, true) \equiv false$

# Interpolants Computation for a Spurious Trace

1: $WI(true, F_1 \wedge F_2 \wedge F_3) \equiv true$

$F_1(X_1, X_2)$ ↓

2: $WI(F_1, F_2 \wedge F_3) \equiv J_2(X_2) \equiv \forall X_3 (F_2 \rightarrow J_3)$

$F_2(X_2, X_3)$ ↓

3: $WI(F_1 \wedge F_2, F_3) \equiv J_3(X_3) \equiv \forall X_4 (F_3 \rightarrow false)$

$F_3(X_3, X_4)$ ↓

4: $WI(F_1 \wedge F_2 \wedge F_3, true) \equiv false$

# Interpolants Computation for a Spurious Trace

(1) $WI(true, F_1 \wedge F_2 \wedge F_3) \equiv J_1(X_1) \equiv \forall X_2(F_1 \rightarrow J_2) \equiv true$

$\downarrow$ $F_1(X_1, X_2)$

(2) $WI(F_1, F_2 \wedge F_3) \equiv J_2(X_2) \equiv \forall X_3(F_2 \rightarrow J_3)$

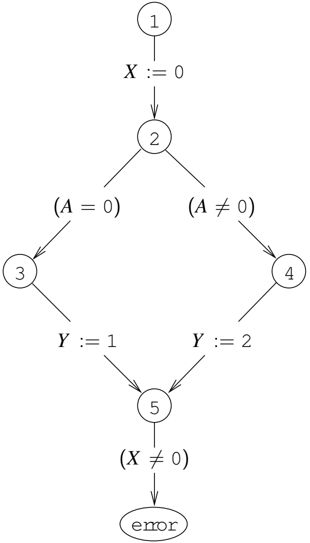
$\downarrow$ $F_2(X_2, X_3)$

(3) $WI(F_1 \wedge F_2, F_3) \equiv J_3(X_3) \equiv \forall X_4(F_3 \rightarrow false)$

$\downarrow$ $F_3(X_3, X_4)$

(4) $WI(F_1 \wedge F_2 \wedge F_3, true) \equiv false$

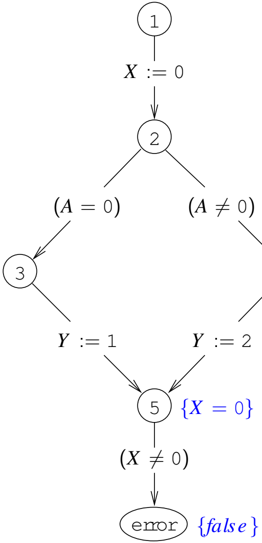
# DAGs of Spurious Counterexamples



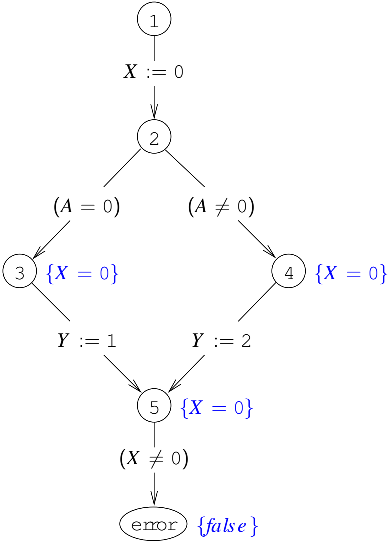

## Spurious Counterexample DAGs

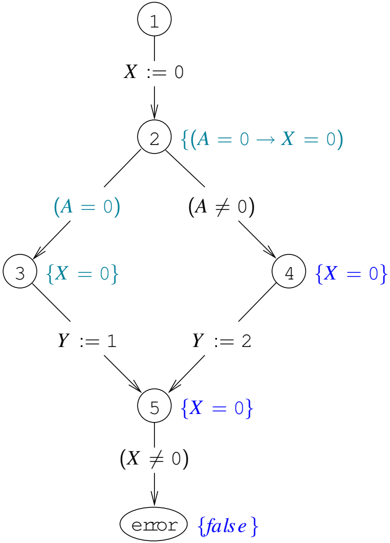
- Each path through the DAG is a spurious counterexample.
- Each path through the DAG corresponds to an unsatisfiable formula.
- The disjunction of the trace formulas is unsatisfiable.

# DAGs of Spurious Counterexamples

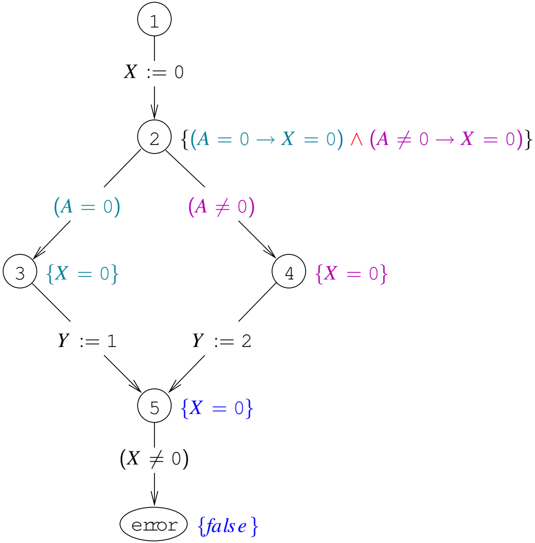


# DAGs of Spurious Counterexamples

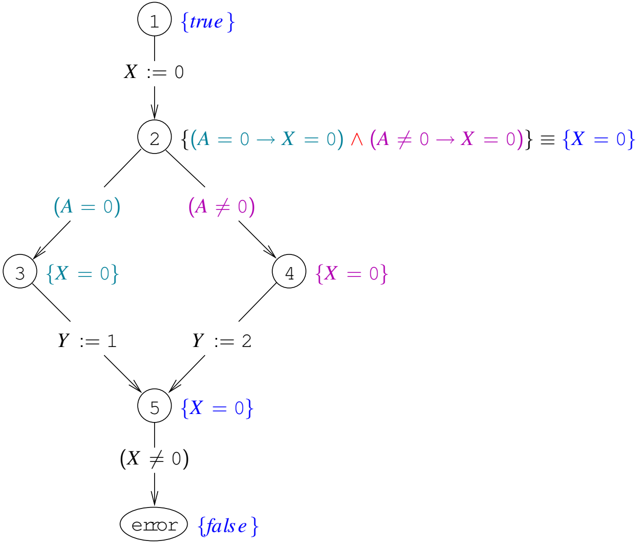



1
X := 0
2
{(A = 0 → X = 0)
(A = 0)
(A ≠ 0)
3
{X = 0}
4
{X = 0}
Y := 1
Y := 2
5
{X = 0}
(X ≠ 0)
error
{false}

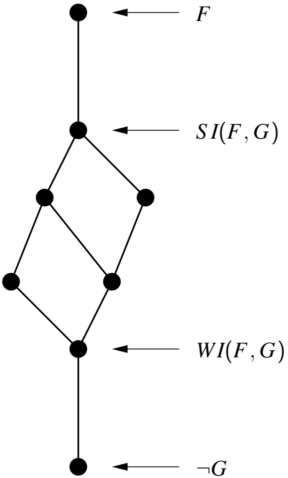
# DAGs of Spurious Counterexamples



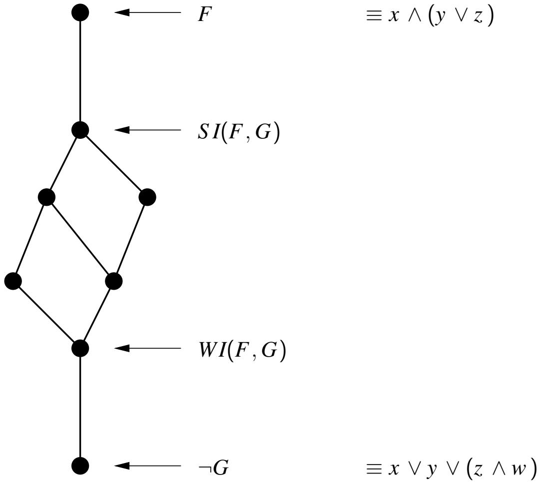
# DAGs of Spurious Counterexamples

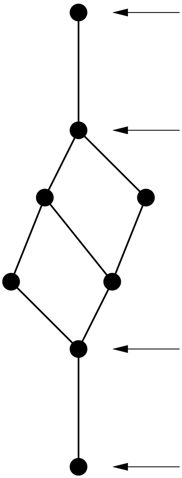


# There Are Many Interpolants.



# There Are Many Interpolants.



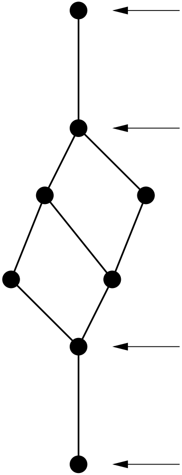


$F \quad \equiv x \wedge (y \vee z)$

$SI(F,G) \quad \equiv \exists\{\}.F \equiv x \wedge (y \vee z)$

$WI(F,G) \quad \equiv \forall\{w\}.\neg G \equiv x \vee y$

$\neg G \quad \equiv x \vee y \vee (z \wedge w)$

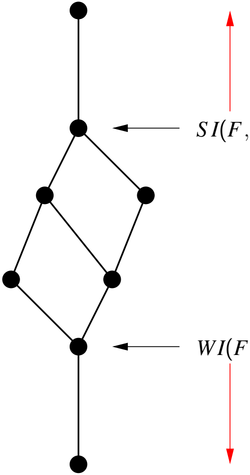


$F \equiv x \wedge (y \vee z)$

$SI(F,G) \equiv \exists\{\}.F \equiv x \wedge (y \vee z)$

$WI(F,G) \equiv \forall\{w\}.\neg G \equiv x \vee y$

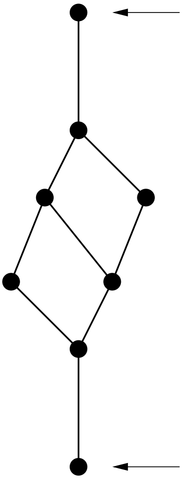
$\neg G \equiv x \vee y \vee (z \wedge w)$



$SI(F,G) \equiv \exists\{\}.F \equiv x \wedge (y \vee z)$

$WI(F,G) \equiv \forall\{w\}.\neg G \equiv x \vee y$

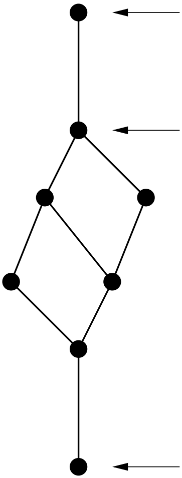
# There Are Many Interpolants.



$SI(F,G) \equiv x \wedge (y \vee z)$

$WI(F,G) \equiv x \vee y$

# There Are Many Interpolants.

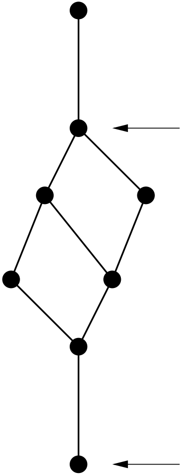


$SI(F, G) \equiv x \wedge (y \vee z)$

$SI(x \wedge (y \vee z), \neg(x \vee y)) \equiv \exists\{z\}.(x \wedge (y \vee z)) \equiv x$

$WI(F, G) \equiv x \vee y$

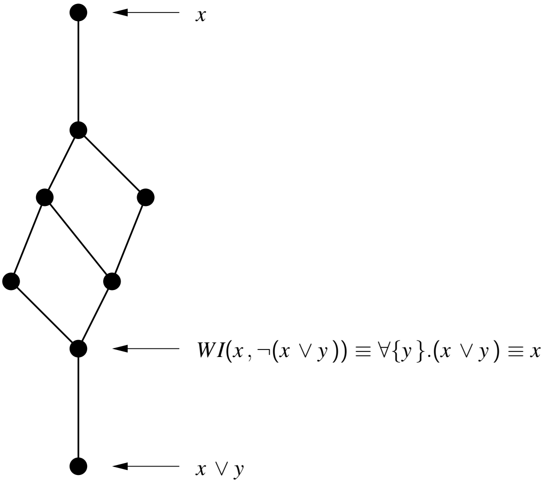
# There Are Many Interpolants.

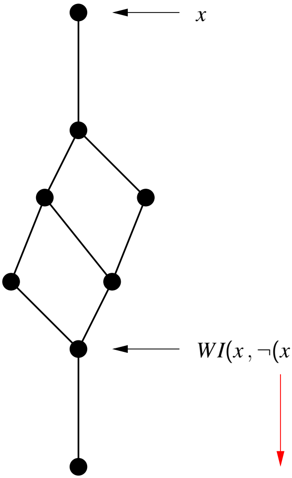


$SI(x \wedge (y \vee z), \neg(x \vee y)) \equiv \exists\{z\}.(x \wedge (y \vee z)) \equiv x$

$WI(F, G) \equiv x \vee y$

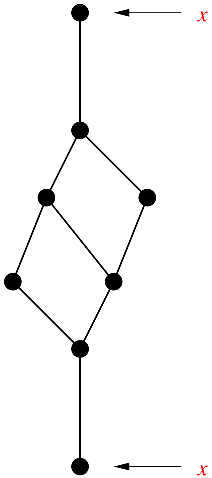
# There Are Many Interpolants.



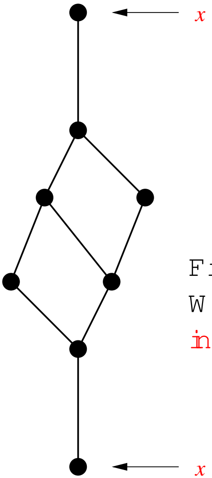



$WI(x, \neg(x \vee y)) \equiv \forall\{y\}.(x \vee y) \equiv x$

# There Are Many Interpolants.



# There Are Many Interpolants.



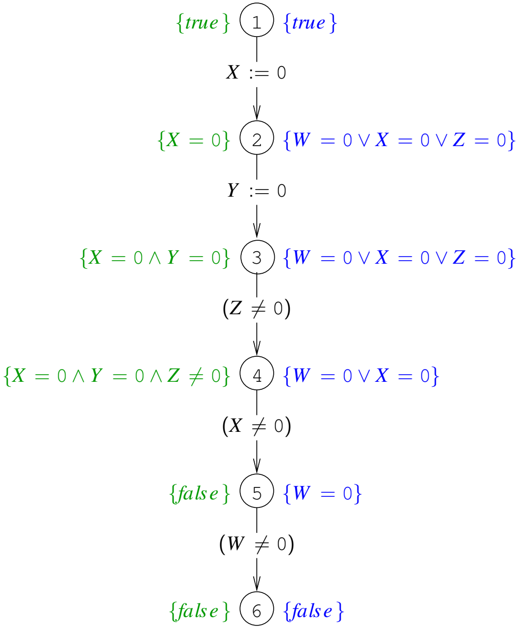

Fixed points have been reached.
We call them conciliated interpolants.

# Conciliated Interpolants

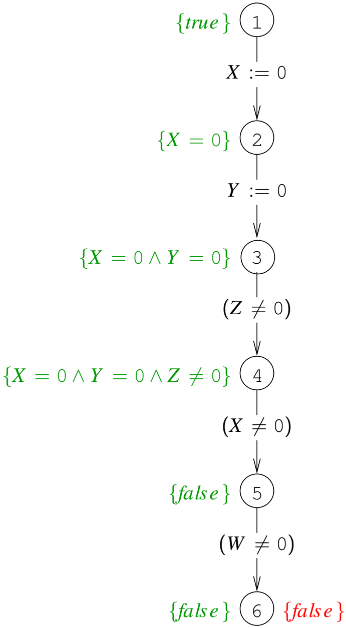
**What about the Tracking Property?**

- Conciliated interpolants by themselves do not necessarily satisfy the tracking property.
- Therefore, we
  1. apply a strongest interpolants computation (forward),
  2. apply a backward computation and conciliate after each step with the strongest interpolant.
- The resulting interpolants satisfy the tracking property.

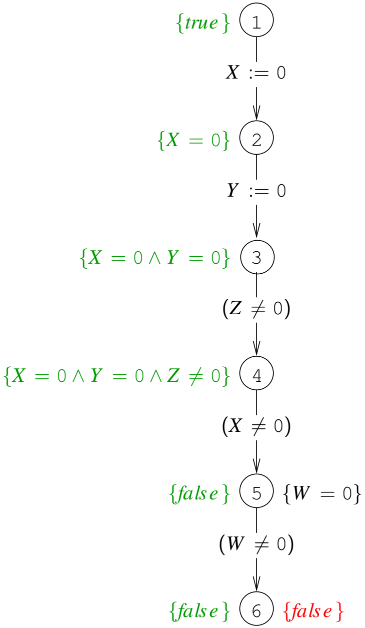
# Conciliated Interpolants as a Simplification Procedure



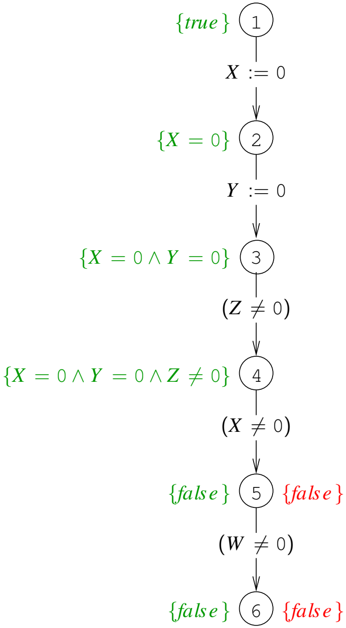
# Conciliated Interpolants as a Simplification Procedure



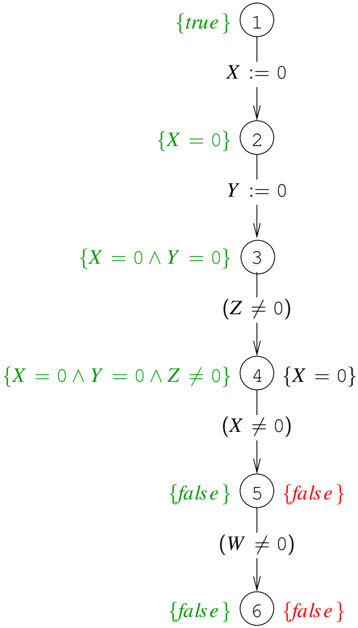
# Conciliated Interpolants as a Simplification Procedure



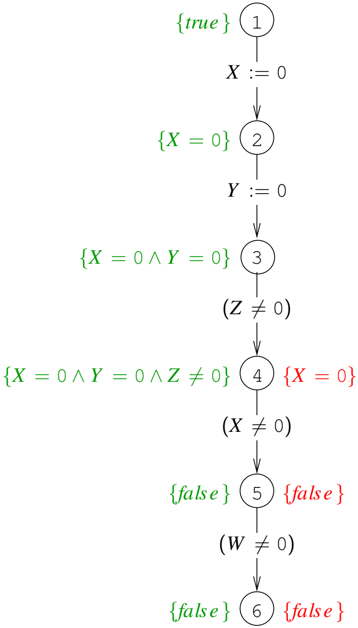
# Conciliated Interpolants as a Simplification Procedure



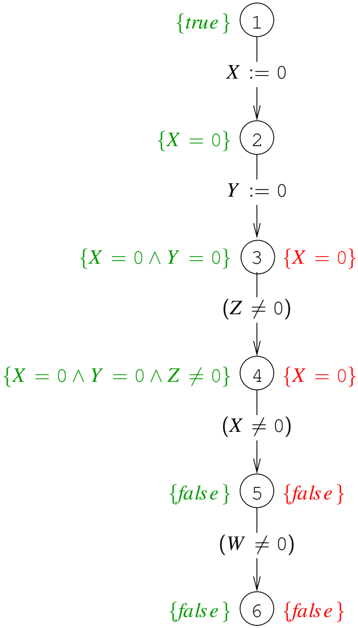
# Conciliated Interpolants as a Simplification Procedure



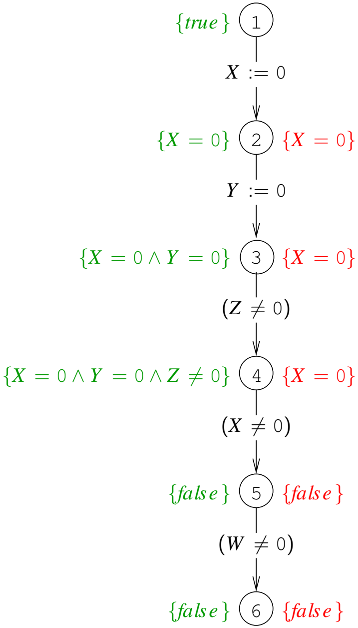
# Conciliated Interpolants as a Simplification Procedure



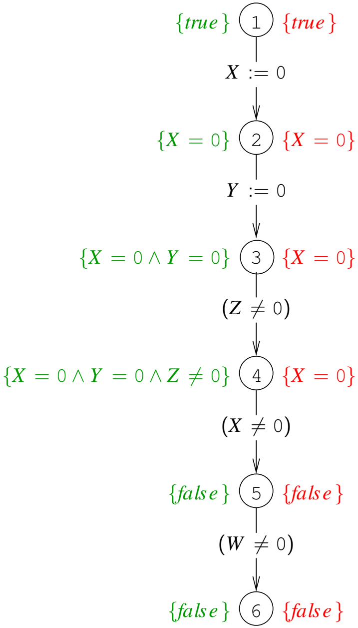
# Conciliated Interpolants as a Simplification Procedure



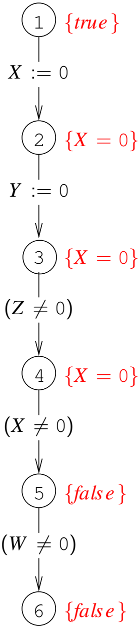
# Conciliated Interpolants as a Simplification Procedure



# Conciliated Interpolants as a Simplification Procedure



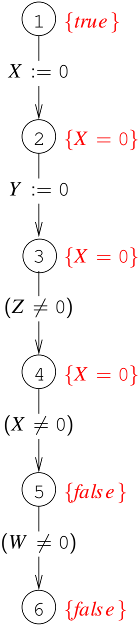
# Conciliated Interpolants




**Conciliated Interpolants**

lead to predicates on fewer variables

# Conciliated Interpolants

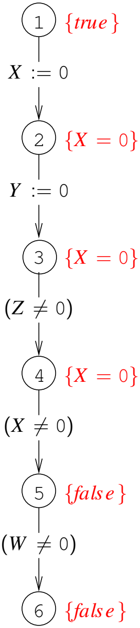



**Conciliated Interpolants**

lead to predicates on fewer variables

⟹ faster computation

# Conciliated Interpolants




**Conciliated Interpolants**

lead to predicates on fewer variables

⇒ faster computation

⇒ more meaningful predicates

# A Locking Example

```
struct file {
    bool locked;
    int pos;
};
open (file f) {
    assert(¬f.locked);
    f.locked = true;
    f.pos = 0;
}
close (file f) {
    assert(f.locked ∨
          f.pos==0);
    f.locked = false;
}
```

```
rw (file f) {
    assert(f.locked ∨ f.pos==0);
    f.pos = f.pos + 1;
}
main () {
    file f1, f2;
    f1.locked = f2.locked = false;
    open (f1);
    while (*) {
        open (f2);
        while (*) { rw (f2); rw (f1); }
        close (f2);
    }
    close (f1);
}
```

# Experimental Results

| | time/s | memory (BDD nodes) | # cycles |
|---|---|---|---|
| w/o abstraction | 460 | 440482 | n/a |
| weakest interp. | 0.43 | 89936 | 14 |
| concil. interp. | 0.29 | 80738 | 10 |

# Summary

- Craig interpolation goes well with CEGAR if the program is given in terms of BDDs.
- Multiple counterexamples can be excluded at once.
- There are heuristics to enhance predicate generation.
- The model-checking process can be speeded up.