Verification of Java Programs using jMoped

Stefan Schwoon

ENS Cachan, INRIA Saclay

Outline

Introduction to pushdown model checking

A look at jMoped

An extension for concurrency

Credits: Work on Moped together with

Javier Esparza, Dejvuth Suwimonteerabuth, David Hansel,
Stefan Kiefer, Felix Berger, Christian Kern

Infinite-state systems

Reasons for infinite state-systems:

Data: integers, reals, lists, trees, pointer structures, ...

Control: (recursive) procedures, dynamic thread creation

Communication: unbounded FIFO channels, ...

Unknown parameters: number of participants in a protocol

Real time: discrete or continuous time

Some of these features lead to Turing-powerful computation models (and thus undecidable verification problems).

Here: approach for procedures, infinity due to presence of stack

Pushdown Systems

A pushdown system $\mathcal{P} = (P, \Gamma, \Delta)$ features

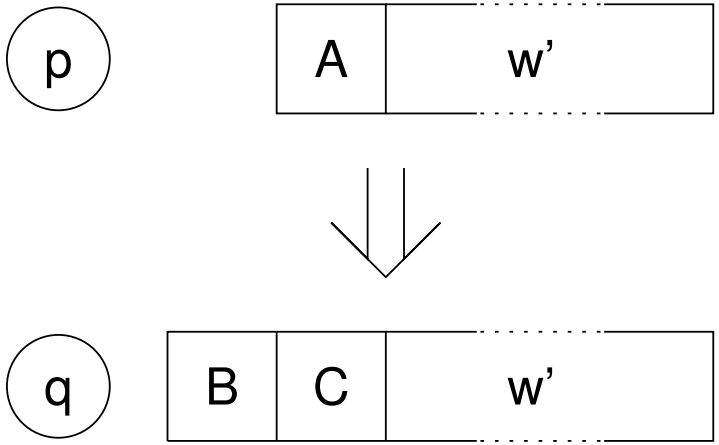
a finite set of control locations P;

a finite stack alphabet Γ;

(a pair $\langle p, w \rangle$, $p \in P$, $w \in \Gamma^*$ is called configuration of $\mathcal{P}$)

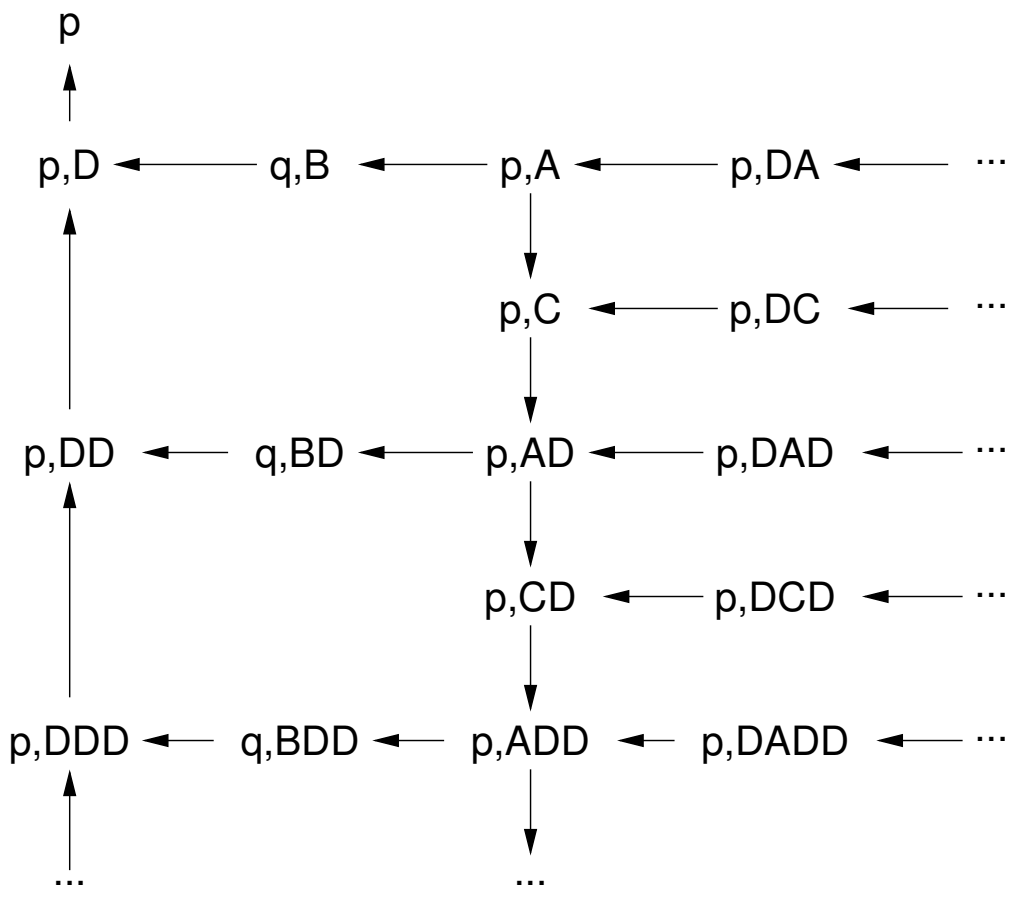
a finite set of rules $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$.

Meaning of a rule $\langle p, A\rangle \hookrightarrow \langle q, BC\rangle$:



Pushdown Graph

$\langle p, A\rangle \hookrightarrow \langle q, B\rangle$
$\langle p, A\rangle \hookrightarrow \langle p, C\rangle$
$\langle q, B\rangle \hookrightarrow \langle p, D\rangle$
$\langle p, C\rangle \hookrightarrow \langle p, AD\rangle$
$\langle p, D\rangle \hookrightarrow \langle p, \varepsilon\rangle$



Reachability

A well-known result (Büchi, Caucal):

If the set of configuration C is regular, then so are $pre^*(C)$ and $post^*(C)$.

$$pre^*(C) = \{\, c \mid c' \in C : c \Rightarrow^* c' \,\}$$
$$post^*(C) = \{\, c \mid c' \in C : c' \Rightarrow^* c \,\}$$

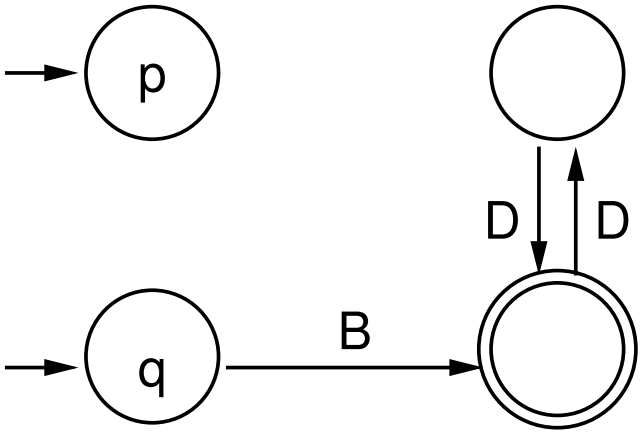
How it works (for pre^*):

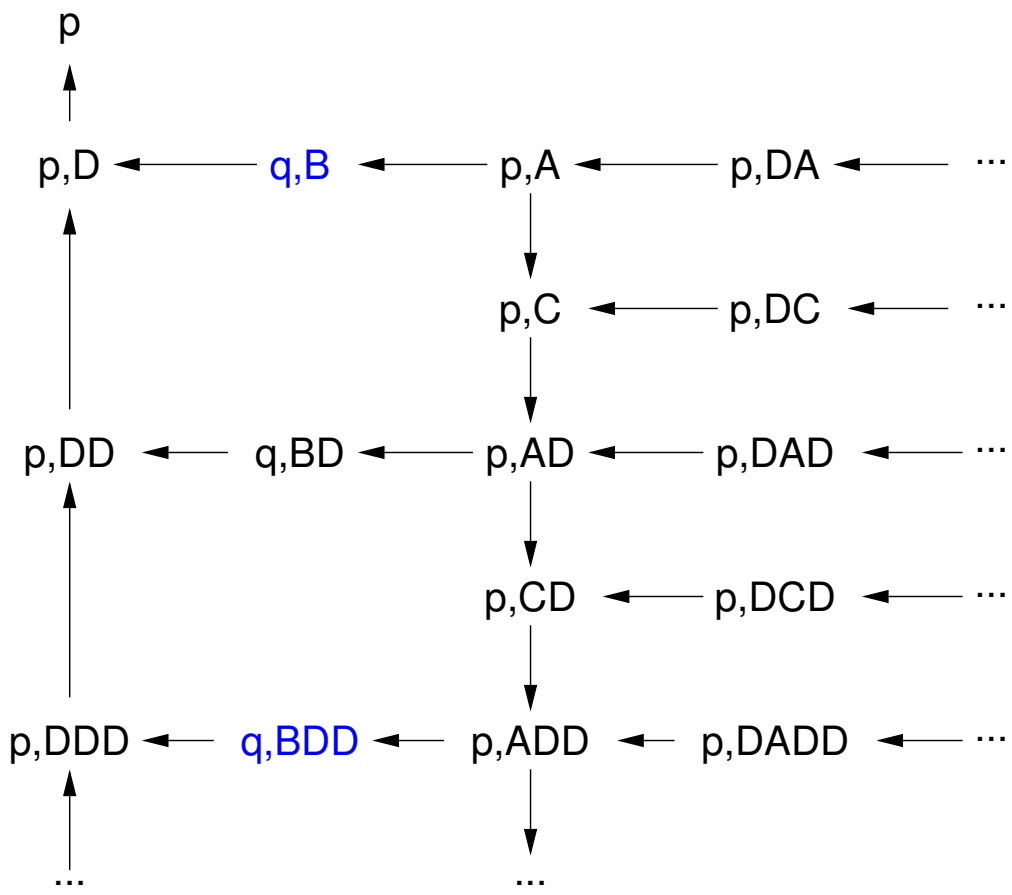
Construct automaton $\mathcal{A}$ accepting C.

Extend $\mathcal{A}$ to $\mathcal{A}'$ by adding transitions (according to some rule).

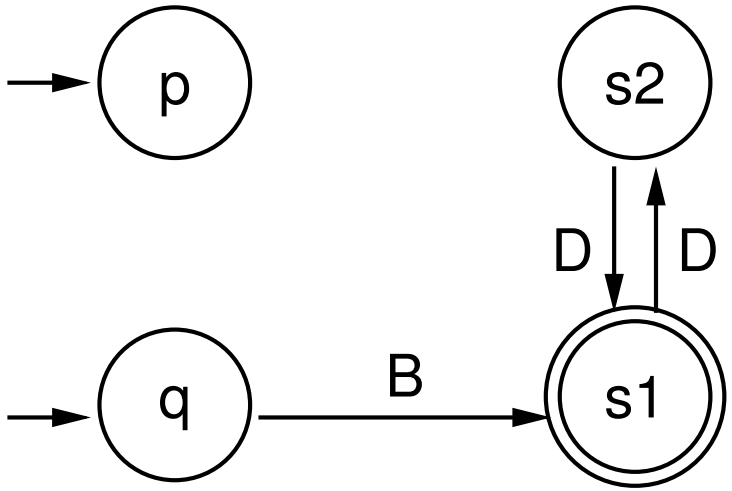
$post^*$ works in a similar fashion.

Construct automaton $\mathcal{A}$ accepting C



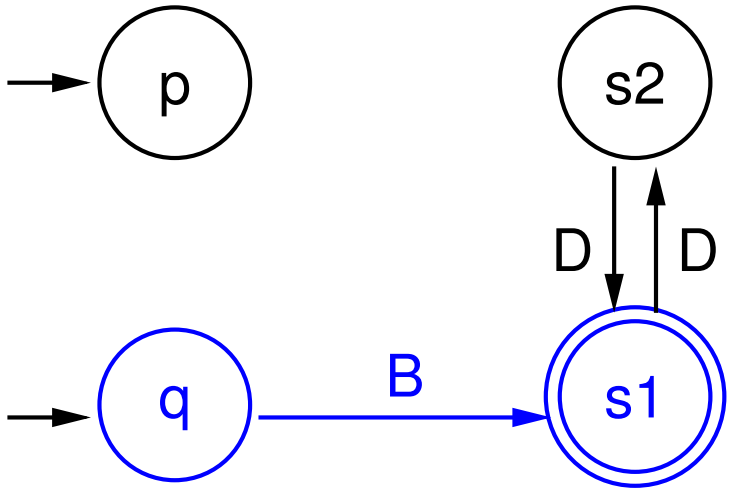


Extend $\mathcal{A}$ by adding transitions



Extend $\mathcal{A}$ by adding transitions

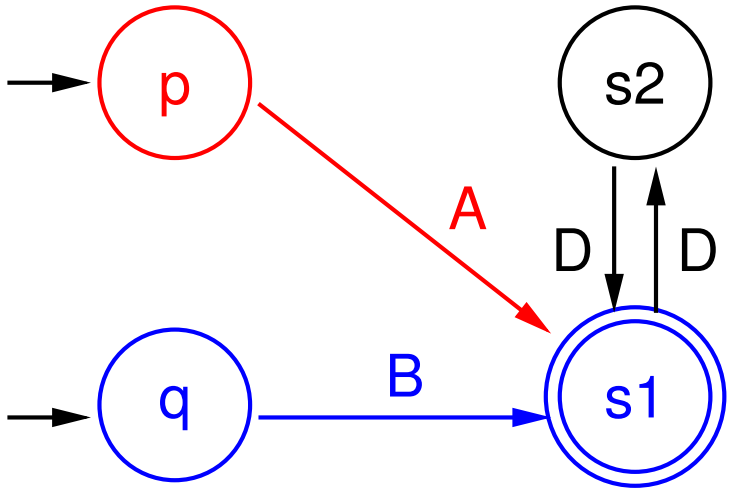
If the right-hand side of a rule has a path, .



Rule: $\langle p, A\rangle \hookrightarrow \langle q, B\rangle$ Path: $q \xrightarrow{B} s_1$

Extend $\mathcal{A}$ by adding transitions

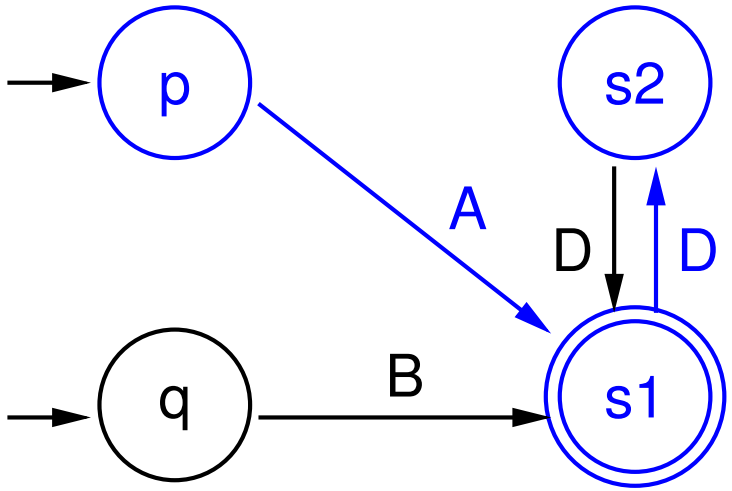
If the right-hand side of a rule has a path, add the left-hand side.



Rule: $\langle p, A\rangle \hookrightarrow \langle q, B\rangle$ Path: $q \xrightarrow{B} s_1$ New Path: $p \xrightarrow{A} s_1$

Extend $\mathcal{A}$ by adding transitions

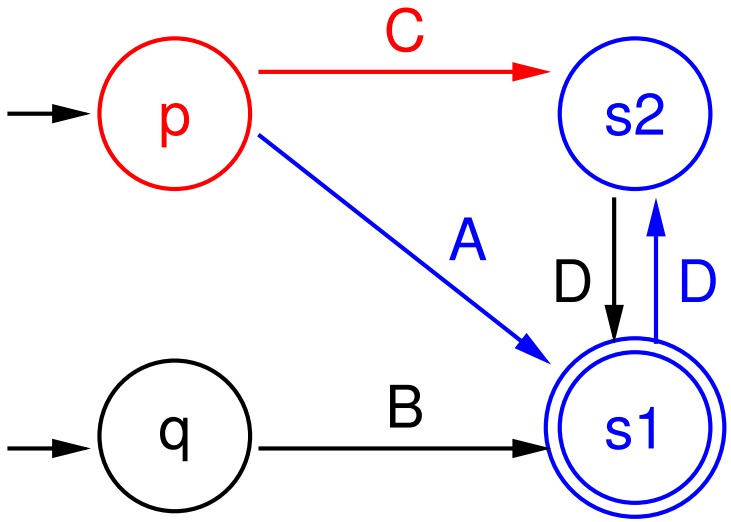
If the right-hand side of a rule has a path, .



Rule: $\langle p, C\rangle \hookrightarrow \langle p, AD\rangle$ Path: $p \xrightarrow{a} s_1 \xrightarrow{D} s_2$

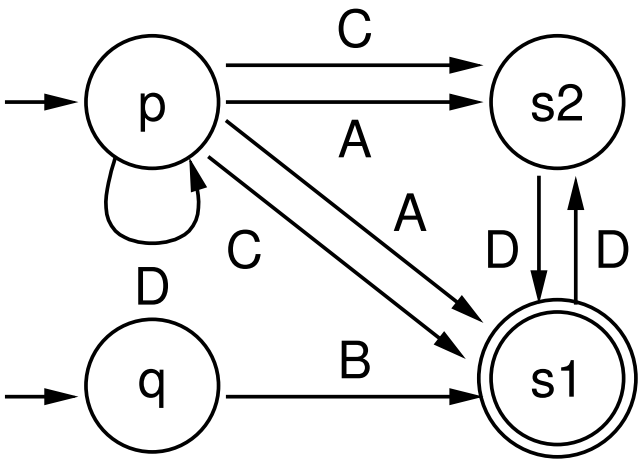
Extend $\mathcal{A}$ by adding transitions

If the right-hand side of a rule has a path, add the left-hand side.

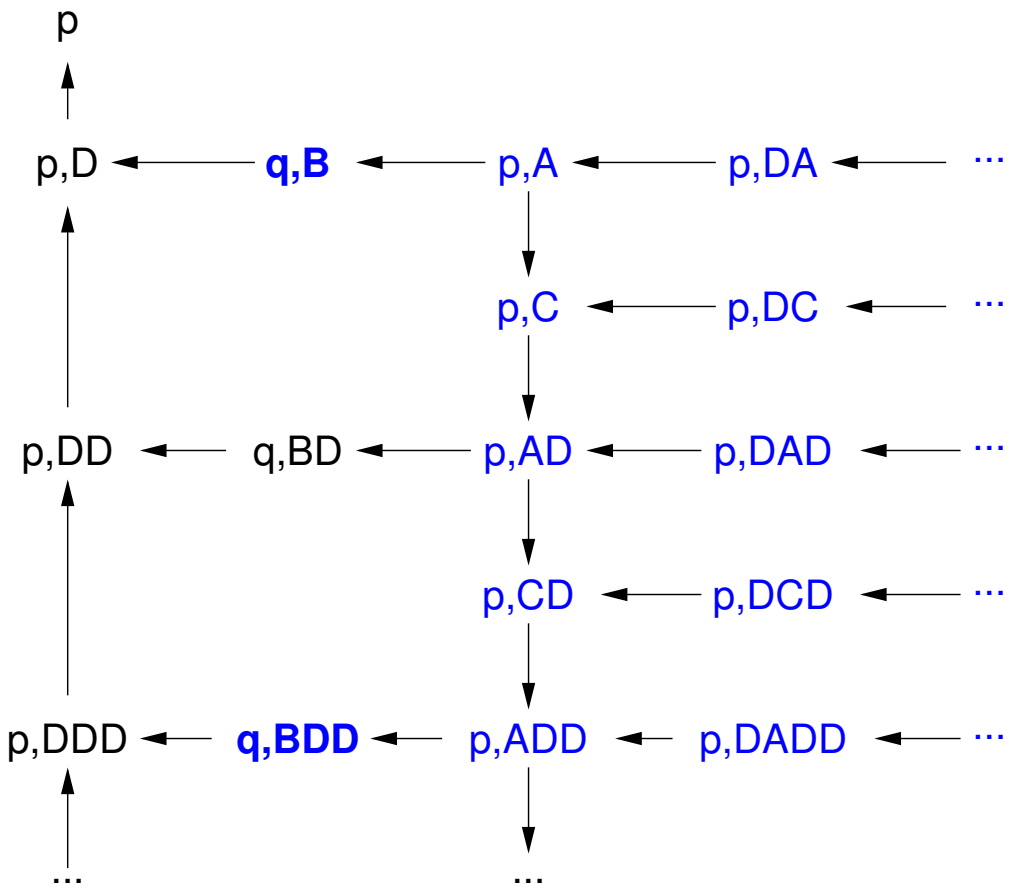


Rule: $\langle p, C\rangle \hookrightarrow \langle p, AD\rangle$ Path: $p \xrightarrow{a} s_1 \xrightarrow{D} s_2$ New Path: $p \xrightarrow{C} s_2$

Final result



Computation takes
$\mathcal{O}(|Q|^2 \cdot |\Delta|)$ time.
[EHRS00]



Interprocedural programs

PDS = finite state machine + (unbounded) stack

$$\langle q_1, a \rangle \hookrightarrow \langle q_2, b\ c \rangle$$

Model for programs with (recursive) procedures

- $\langle p, A \rangle \hookrightarrow \langle p, B \rangle \ \widehat{=}$ intraprocedural statement
- $\langle p, A \rangle \hookrightarrow \langle p, BC \rangle \ \widehat{=}$ procedure call
- $\langle p, A \rangle \hookrightarrow \langle p, \varepsilon \rangle \ \widehat{=}$ return statement

Interprocedural programs with Data

SPDS = PDS + global variables + local variables

Variables have finite ranges

One copy of global variables

Each stack symbol owns a copy of local variables

$$\langle g, (a, l_a)\rangle \hookrightarrow \langle g', (b, l_b)\ (c, l_c)\rangle$$
$$a \hookrightarrow b\ c \qquad R(g, g', l_a, l_b, l_c)$$

Sequential programs with data can be naturally encoded into SPDS.

Moped works on SPDS using BDDs.

Development of Moped

Moped v1:

text-based tool, implements reachability and LTL on SPDS

interface to Boolean programs, used for verifying device drivers
(large control, not much data)

Moped v2:

re-implementation of Moped v1, cleaner, more extensible

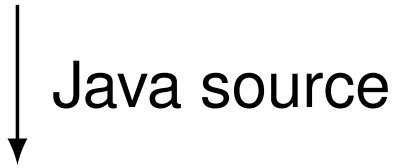
based on weighted PDS library, implements reachability only

jMoped:

Moped equipped with a Java frontend, Eclipse plugin

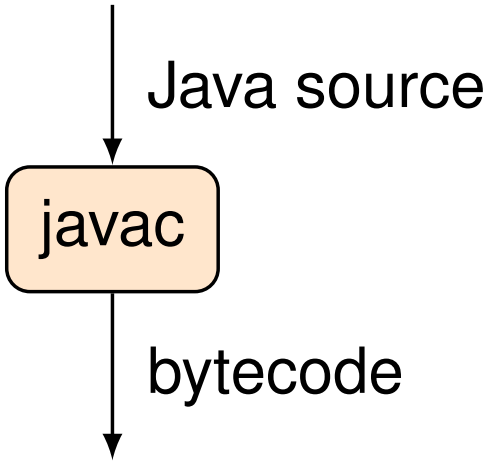
aimed at small, but data-intensive programs (testing single procedures)

Architecture of jMoped




```
static int x;
void f(){x=0; f();}
```

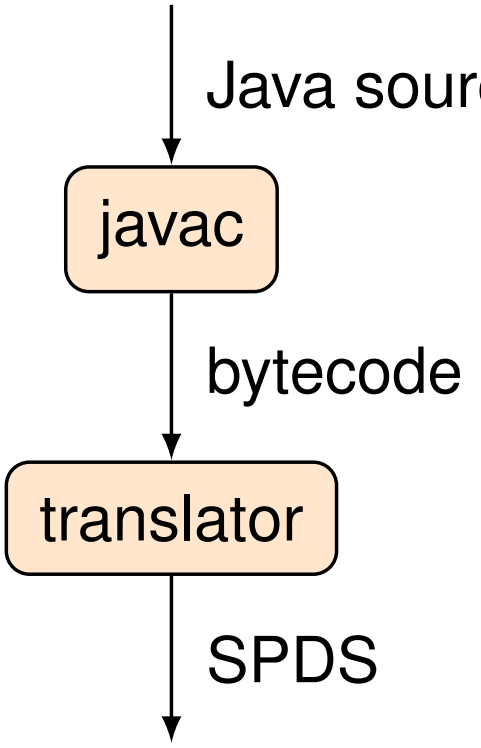
Architecture of jMoped




```
static int x;
void f(){x=0; f();}
```

```
f0:   iconst_0
f1:   putstatic x
f2:   invokevirtual f()
f3:   return
```

Architecture of jMoped

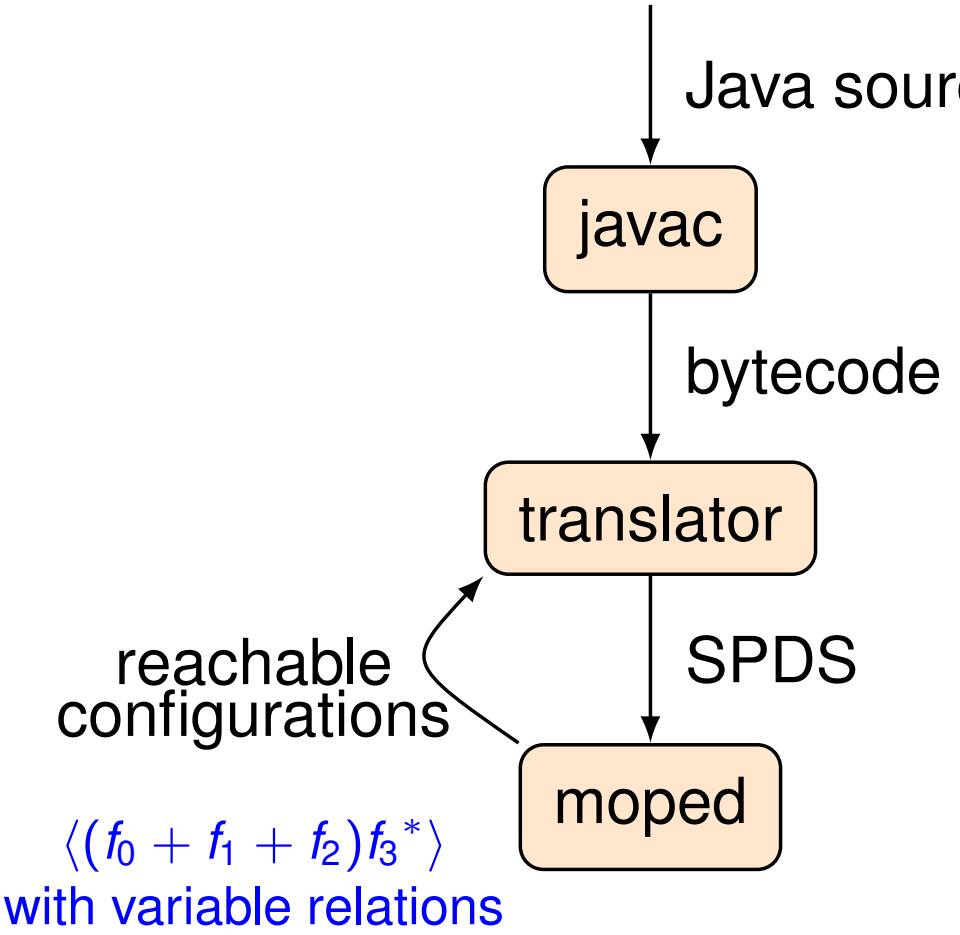



```
static int x;
void f(){x=0; f();}
```

```
f0:  iconst_0
f1:  putstatic x
f2:  invokevirtual f()
f3:  return
```

$$
\begin{array}{llll}
\langle f_0 \rangle & \hookrightarrow & \langle f_1 \rangle & (s_0' = 0) \\
\langle f_1 \rangle & \hookrightarrow & \langle f_2 \rangle & (x' = s_0) \\
\langle f_2 \rangle & \hookrightarrow & \langle f_0 \ \ f_3 \rangle & (x' = x) \\
\langle f_3 \rangle & \hookrightarrow & \langle \rangle & (x' = x)
\end{array}
$$

Architecture of jMoped

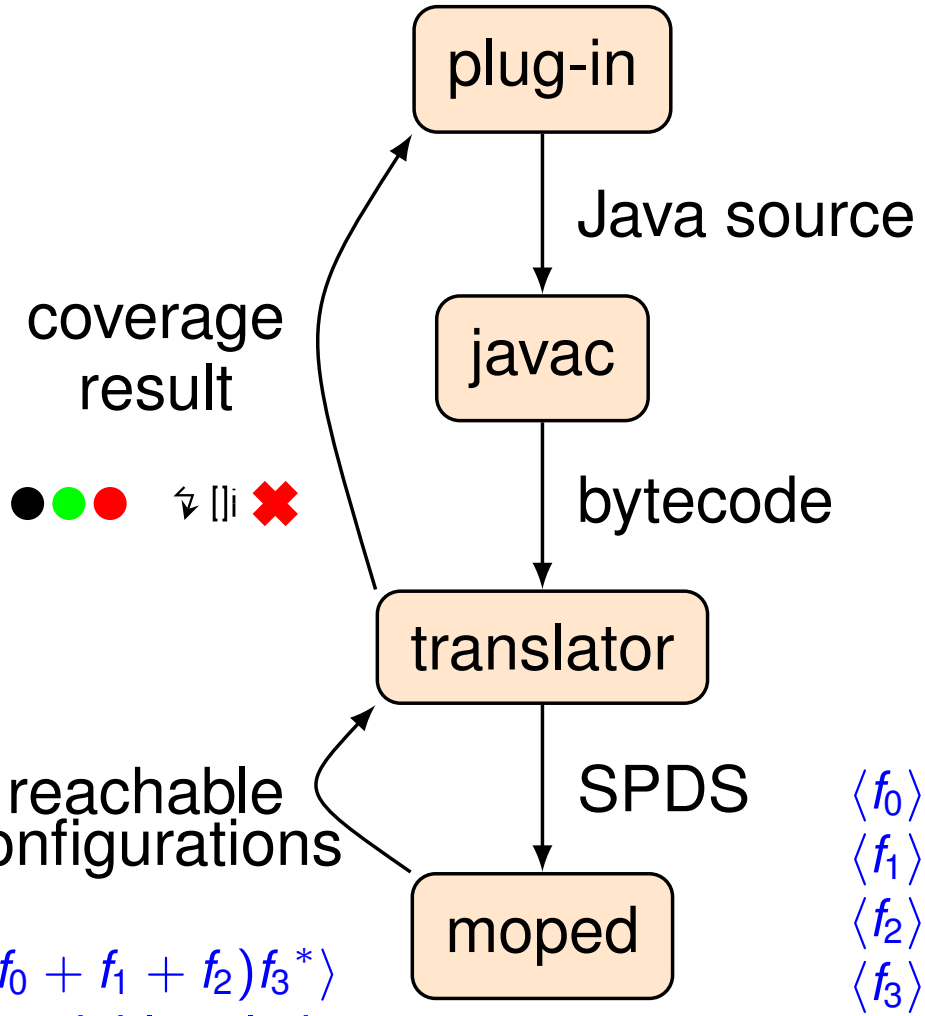



```
static int x;
void f(){x=0; f();}
```

```
f0:  iconst_0
f1:  putstatic x
f2:  invokevirtual f()
f3:  return
```

$$
\begin{array}{llll}
\langle f_0 \rangle & \hookrightarrow & \langle f_1 \rangle & (s_0' = 0) \\
\langle f_1 \rangle & \hookrightarrow & \langle f_2 \rangle & (x' = s_0) \\
\langle f_2 \rangle & \hookrightarrow & \langle f_0 \ \ f_3 \rangle & (x' = x) \\
\langle f_3 \rangle & \hookrightarrow & \langle \rangle & (x' = x)
\end{array}
$$

Architecture of jMoped




```
static int x;
void f(){x=0; f();}
```

```
f0:   iconst_0
f1:   putstatic x
f2:   invokevirtual f()
f3:   return
```

$$
\begin{array}{llll}
\langle f_0 \rangle & \hookrightarrow & \langle f_1 \rangle & (s_0' = 0) \\
\langle f_1 \rangle & \hookrightarrow & \langle f_2 \rangle & (x' = s_0) \\
\langle f_2 \rangle & \hookrightarrow & \langle f_0 \;\; f_3 \rangle & (x' = x) \\
\langle f_3 \rangle & \hookrightarrow & \langle\rangle & (x' = x)
\end{array}
$$

Extensions to multithreaded programs

Reachability problem is undecidable for concurrent pushdown systems with shared memory

The usual remedies: approximative techniques or restricted models

[Kahlon, Ivančić, Gupta 07] Communication via nested locks

[Bouajjani, Müller-Olm, Touili 05] Dynamic Pushdown Network (DPN): PDS + dynamic thread creation, no communication

$$\langle q_1, a\rangle \hookrightarrow \langle q_2, b\rangle \rhd \langle q_3, c\rangle$$

[Bouajjani, Esparaza, Touili 03] compute an (over-)approximation of each thread individually, then intersect

[Qadeer, Rehof 05] under-approximation by context-bounded analysis

Extensions to multithreaded programs

Reachability problem is undecidable for concurrent pushdown systems with shared memory

The usual remedies: approximative techniques or restricted models

[Kahlon, Ivančić, Gupta 07] Communication via nested locks

[Bouajjani, Müller-Olm, Touili 05] Dynamic Pushdown Network (DPN): PDS + dynamic thread creation, no communication

$$\langle q_1, a\rangle \hookrightarrow \langle q_2, b\rangle \rhd \langle q_3, c\rangle$$

[Bouajjani, Esparaza, Touili 03] compute an (over-)approximation of each thread individually, then intersect

[Qadeer, Rehof 05] under-approximation by context-bounded analysis

Extensions to multithreaded programs

Reachability problem is undecidable for concurrent pushdown systems with shared memory

The usual remedies: approximative techniques or restricted models

[Kahlon, Ivančić, Gupta 07] Communication via nested locks

[Bouajjani, Müller-Olm, Touili 05] Dynamic Pushdown Network (DPN): PDS + dynamic thread creation, no communication

$$\langle q_1, a\rangle \hookrightarrow \langle q_2, b\rangle \rhd \langle q_3, c\rangle$$

[Bouajjani, Esparaza, Touili 03] compute an (over-)approximation of each thread individually, then intersect

[Qadeer, Rehof 05] under-approximation by context-bounded analysis

Extensions to multithreaded programs

Reachability problem is undecidable for concurrent pushdown systems with shared memory

The usual remedies: approximative techniques or restricted models

[Kahlon, Ivančić, Gupta 07] Communication via nested locks

[Bouajjani, Müller-Olm, Touili 05] Dynamic Pushdown Network (DPN): PDS + dynamic thread creation, no communication

$$\langle q_1, a\rangle \hookrightarrow \langle q_2, b\rangle \triangleright \langle q_3, c\rangle$$

[Bouajjani, Esparaza, Touili 03] compute an (over-)approximation of each thread individually, then intersect

[Qadeer, Rehof 05] under-approximation by context-bounded analysis

Extensions to multithreaded programs

Reachability problem is undecidable for concurrent pushdown systems with shared memory

The usual remedies: approximative techniques or restricted models

[Kahlon, Ivančić, Gupta 07] Communication via nested locks

[Bouajjani, Müller-Olm, Touili 05] Dynamic Pushdown Network (DPN): PDS + dynamic thread creation, no communication

$$\langle q_1, a\rangle \hookrightarrow \langle q_2, b\rangle \triangleright \langle q_3, c\rangle$$

[Bouajjani, Esparaza, Touili 03] compute an (over-)approximation of each thread individually, then intersect

[Qadeer, Rehof 05] under-approximation by context-bounded analysis

Extensions to multithreaded programs

Reachability problem is undecidable for concurrent pushdown systems with shared memory

The usual remedies: approximative techniques or restricted models

[Kahlon, Ivančić, Gupta 07] Communication via nested locks

[Bouajjani, Müller-Olm, Touili 05] Dynamic Pushdown Network (DPN): PDS + dynamic thread creation, no communication

$$\langle q_1, a\rangle \hookrightarrow \langle q_2, b\rangle \rhd \langle q_3, c\rangle$$

[Bouajjani, Esparaza, Touili 03] compute an (over-)approximation of each thread individually, then intersect

[Qadeer, Rehof 05] under-approximation by context-bounded analysis

Context-bounded reachability analysis

Multiple stacks and one common control state

Global configurations have the form $(g, \alpha_1, \ldots, \alpha_n)$

A transition of process i modifies g and α_i

Context – a sequence of transitions performed by a single thread

Compute reachability for a fixed number of contexts

Context-bounded reachability analysis

Multiple stacks and one common control state

Global configurations have the form $(g, \alpha_1, \ldots, \alpha_n)$

A transition of process i modifies g and α_i

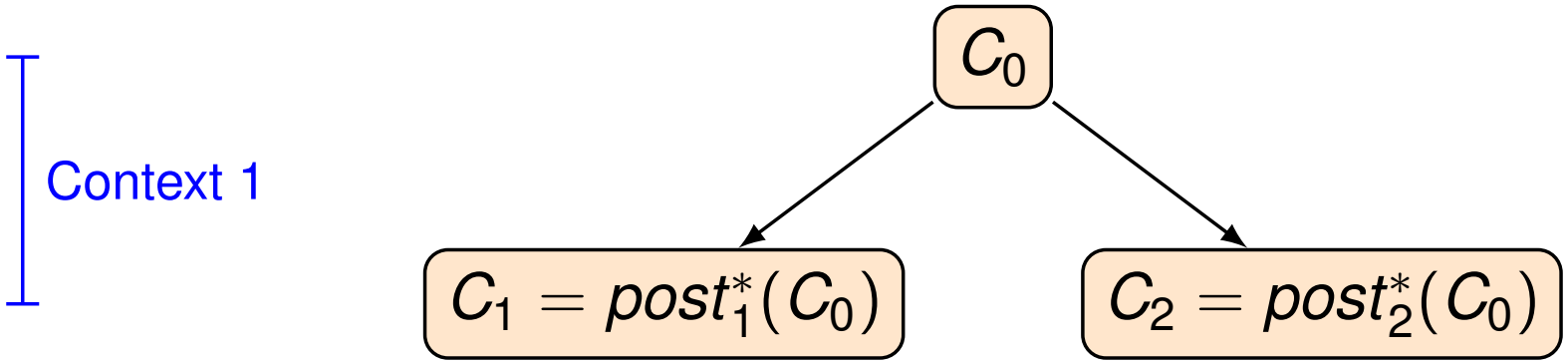
Context – a sequence of transitions performed by a single thread

Compute reachability for a fixed number of contexts

Context-bounded reachability

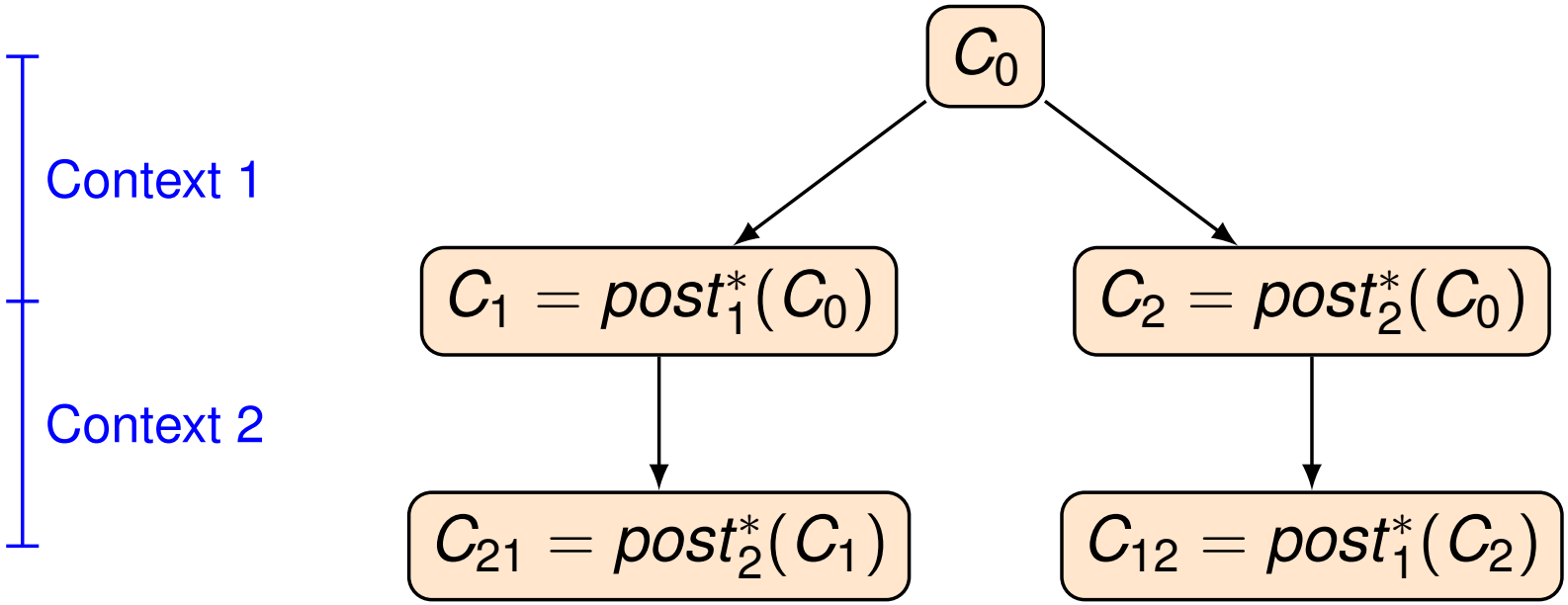
Initial set of global configurations C_0

Context-bounded reachability



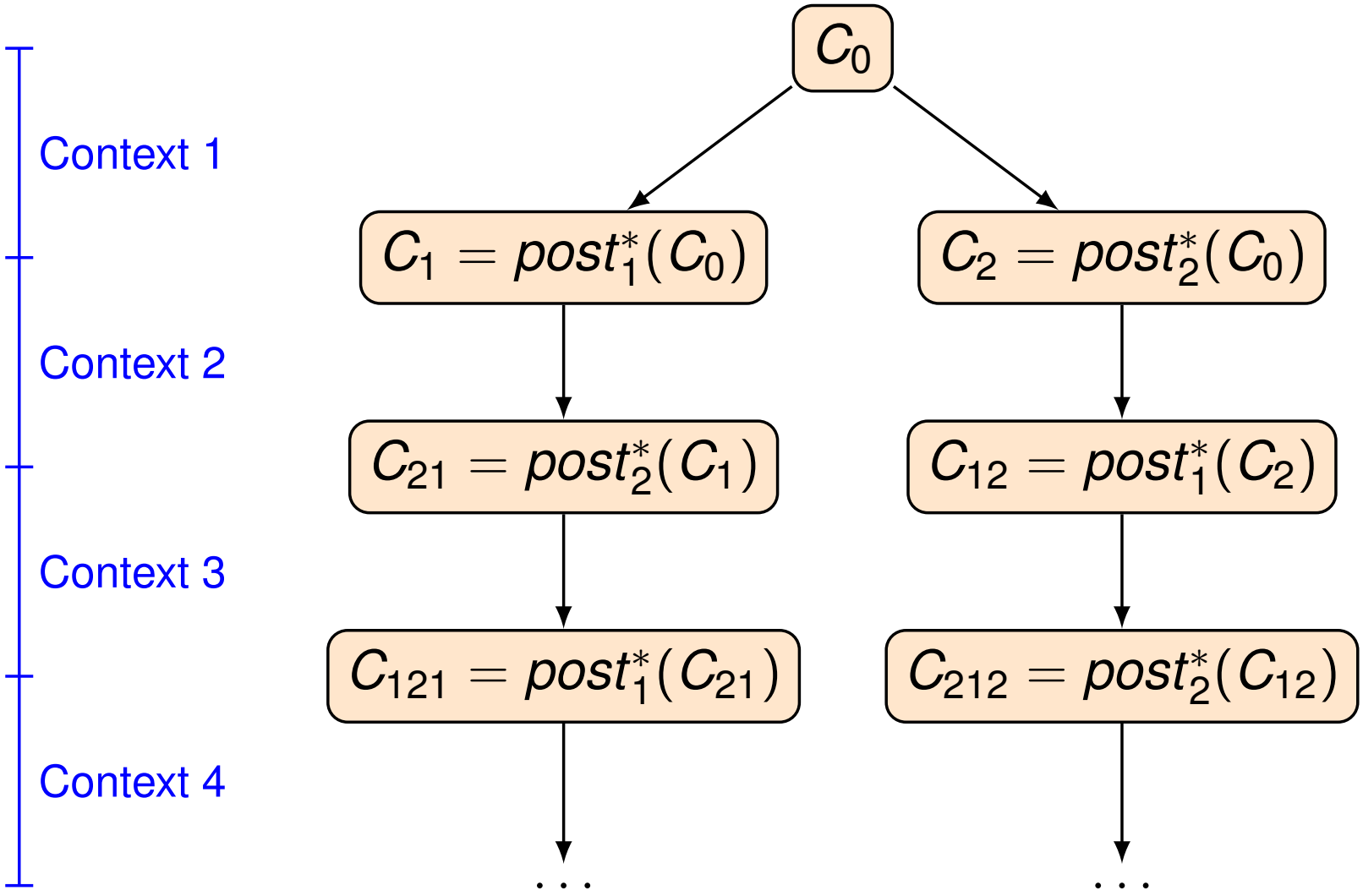
Repeatedly compute $post^*$ for each thread

Context-bounded reachability



Repeatedly compute $post^*$ for each thread

Context-bounded reachability



Repeatedly compute $post^*$ for each thread

More on context-bounded analysis

CBA is a useful concept, because

- many bugs can be exposed with few context switches
- most states reachable within few context switches; diminishing returns from exploring more

Additional work on CBA:

- [Bouajjani, Esparza, S., Strejček 05]: process creation, improved algorithm
- [Bouajjani, Fratani, Qadeer 07]: heap structures
- [La Torre, Madhudusan, Parlato 08]: FIFO queues
- [Lal, Touili, Kidd, Reps 08]: infinite data domains, different algorithm

A lot of work on this topic. . . but implementation is problematic.

More on context-bounded analysis

CBA is a useful concept, because

- many bugs can be exposed with few context switches
- most states reachable within few context switches; diminishing returns from exploring more

Additional work on CBA:

- [Bouajjani, Esparza, S., Strejček 05]: process creation, improved algorithm
- [Bouajjani, Fratani, Qadeer 07]: heap structures
- [La Torre, Madhudusan, Parlato 08]: FIFO queues
- [Lal, Touili, Kidd, Reps 08]: infinite data domains, different algorithm

A lot of work on this topic. . . but implementation is problematic.

More on context-bounded analysis

CBA is a useful concept, because

- many bugs can be exposed with few context switches
- most states reachable within few context switches; diminishing returns from exploring more

Additional work on CBA:

- [Bouajjani, Esparza, S., Strejček 05]: process creation, improved algorithm
- [Bouajjani, Fratani, Qadeer 07]: heap structures
- [La Torre, Madhudusan, Parlato 08]: FIFO queues
- [Lal, Touili, Kidd, Reps 08]: infinite data domains, different algorithm

A lot of work on this topic. . . but implementation is problematic.

View tuples

Problem: we need a data structure for sets of global configurations $C(g, \alpha_1, \alpha_2)$.

What we have:

1. Data structure for (possibly infinite) sets of local configurations $c_1(g, \alpha_1)$ and $c_2(g, \alpha_2)$
2. *post** for $c_1(g, \alpha_1)$ and $c_2(g, \alpha_2)$

Definition A view tuple $T = (c_1, c_2)$ represents the following set of global configurations:

$$[\![T]\!] = \{(g, \alpha_1, \alpha_2) \mid (g, \alpha_1) \in c_1 \text{ and } (g, \alpha_2) \in c_2\}$$

E.g. If $c_1 = \{(g, a), (g, a')\}$ and $c_2 = \{(g, b), (g', b')\}$, we have $[\![T]\!] = \{(g, a, b), (g, a', b)\}$.

View tuples

Problem: we need a data structure for sets of global configurations $C(g, \alpha_1, \alpha_2)$.

What we have:

1. Data structure for (possibly infinite) sets of local configurations $c_1(g, \alpha_1)$ and $c_2(g, \alpha_2)$
2. $post^*$ for $c_1(g, \alpha_1)$ and $c_2(g, \alpha_2)$

Definition A view tuple $T = (c_1, c_2)$ represents the following set of global configurations:

$$[\![T]\!] = \{(g, \alpha_1, \alpha_2) \mid (g, \alpha_1) \in c_1 \text{ and } (g, \alpha_2) \in c_2\}$$

E.g. If $c_1 = \{(g, a), (g, a')\}$ and $c_2 = \{(g, b), (g', b')\}$, we have $[\![T]\!] = \{(g, a, b), (g, a', b)\}$.

View tuples

Problem: we need a data structure for sets of global configurations $C(g, \alpha_1, \alpha_2)$.

What we have:

1. Data structure for (possibly infinite) sets of local configurations $c_1(g, \alpha_1)$ and $c_2(g, \alpha_2)$
2. *post** for $c_1(g, \alpha_1)$ and $c_2(g, \alpha_2)$

Definition A view tuple $T = (c_1, c_2)$ represents the following set of global configurations:

$$[\![T]\!] = \{(g, \alpha_1, \alpha_2) \mid (g, \alpha_1) \in c_1 \text{ and } (g, \alpha_2) \in c_2\}$$

E.g. If $c_1 = \{(g, a), (g, a')\}$ and $c_2 = \{(g, b), (g', b')\}$, we have $[\![T]\!] = \{(g, a, b), (g, a', b)\}$.

View tuples

Problem: not every set can be represented as a view tuple.

E.g. to represent

$$C = \{(g, a, a), (g', b, a), (g'', a, a), (g'', b, b)\}$$

as a view tuple $T = (c_1, c_2)$ would require $(g'', a) \in c_1$ and $(g'', b) \in c_2$. Then, $(g'', a, b) \in [\![T]\!]$, but $(g'', a, b) \notin C$.

However, a union of view tuples will do, i.e. we split C into several view tuples.

E.g. $C = \{(g, a, a), (g'', a, a)\} \cup \{(g', b, a), (g'', b, b)\}$

View tuples

Problem: not every set can be represented as a view tuple.

E.g. to represent

$$C = \{(g, a, a), (g', b, a), (g'', a, a), (g'', b, b)\}$$

as a view tuple $T = (c_1, c_2)$ would require $(g'', a) \in c_1$ and $(g'', b) \in c_2$. Then, $(g'', a, b) \in [\![T]\!]$, but $(g'', a, b) \notin C$.

However, a union of view tuples will do, i.e. we split C into several view tuples.

E.g. $C = \{(g, a, a), (g'', a, a)\} \cup \{(g', b, a), (g'', b, b)\}$

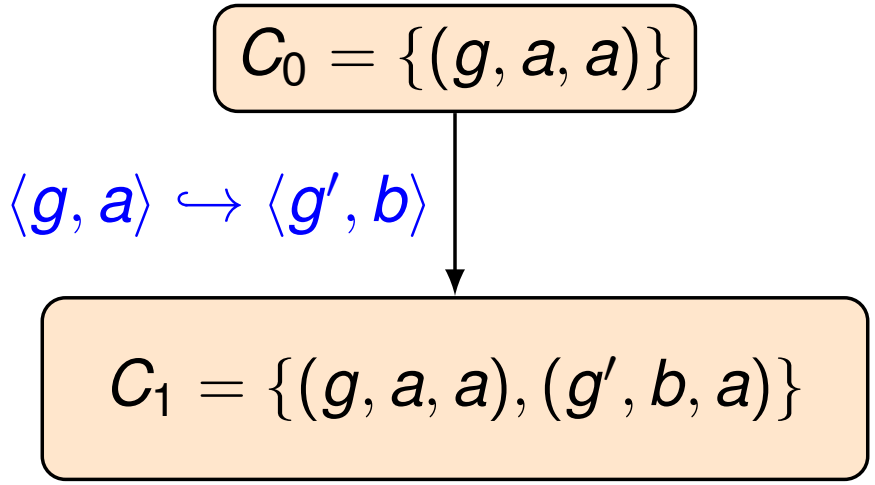
View tuples

Worse problem: sets representable by view tuples not closed under reachability.

$$C_0 = \{(g, a, a)\}$$

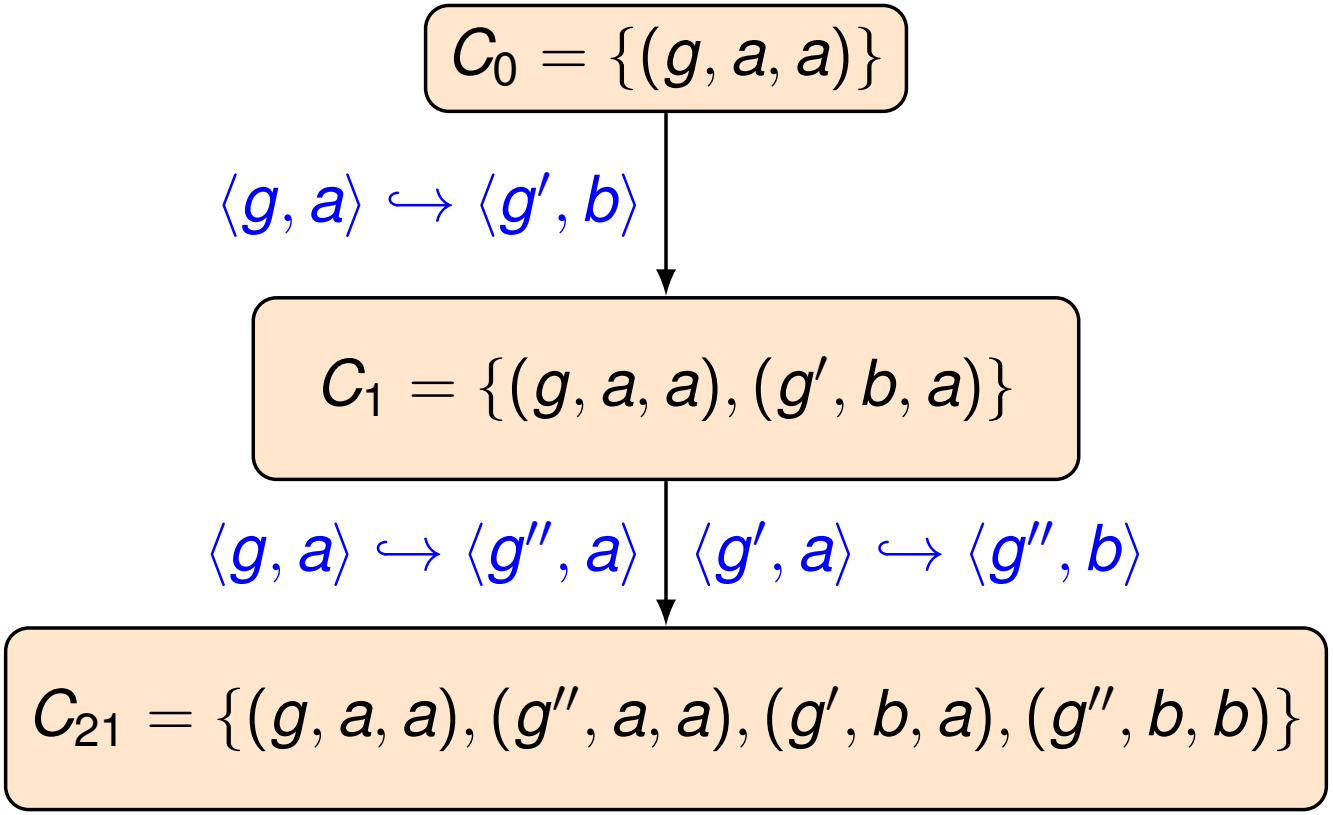
View tuples

Worse problem: sets representable by view tuples not closed under reachability.



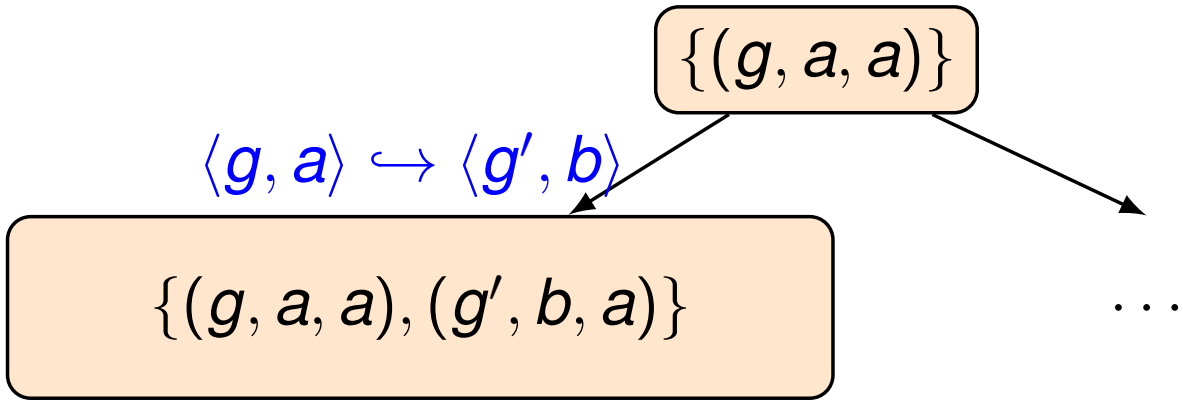
View tuples

Worse problem: sets representable by view tuples not closed under reachability.



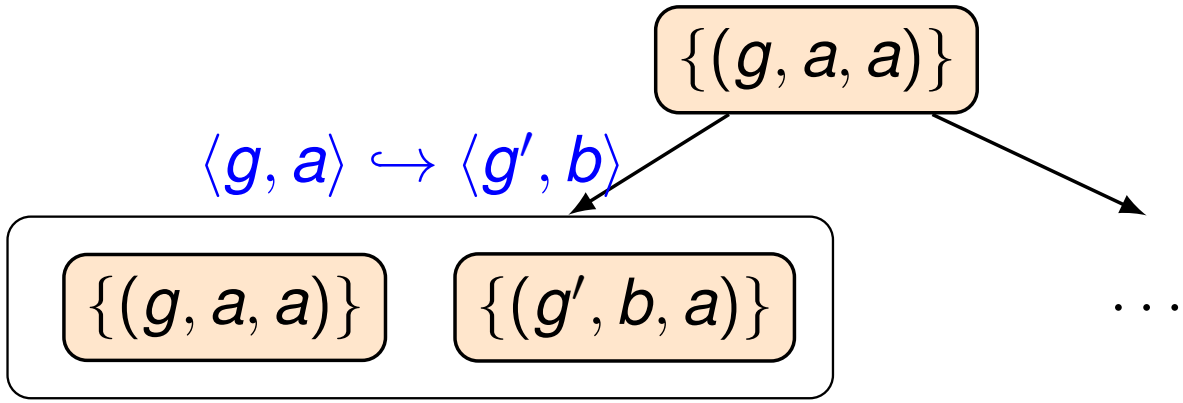
How to split?

Eager splitting: split for every poss. value of globals [Qadeer, Rehof 05]



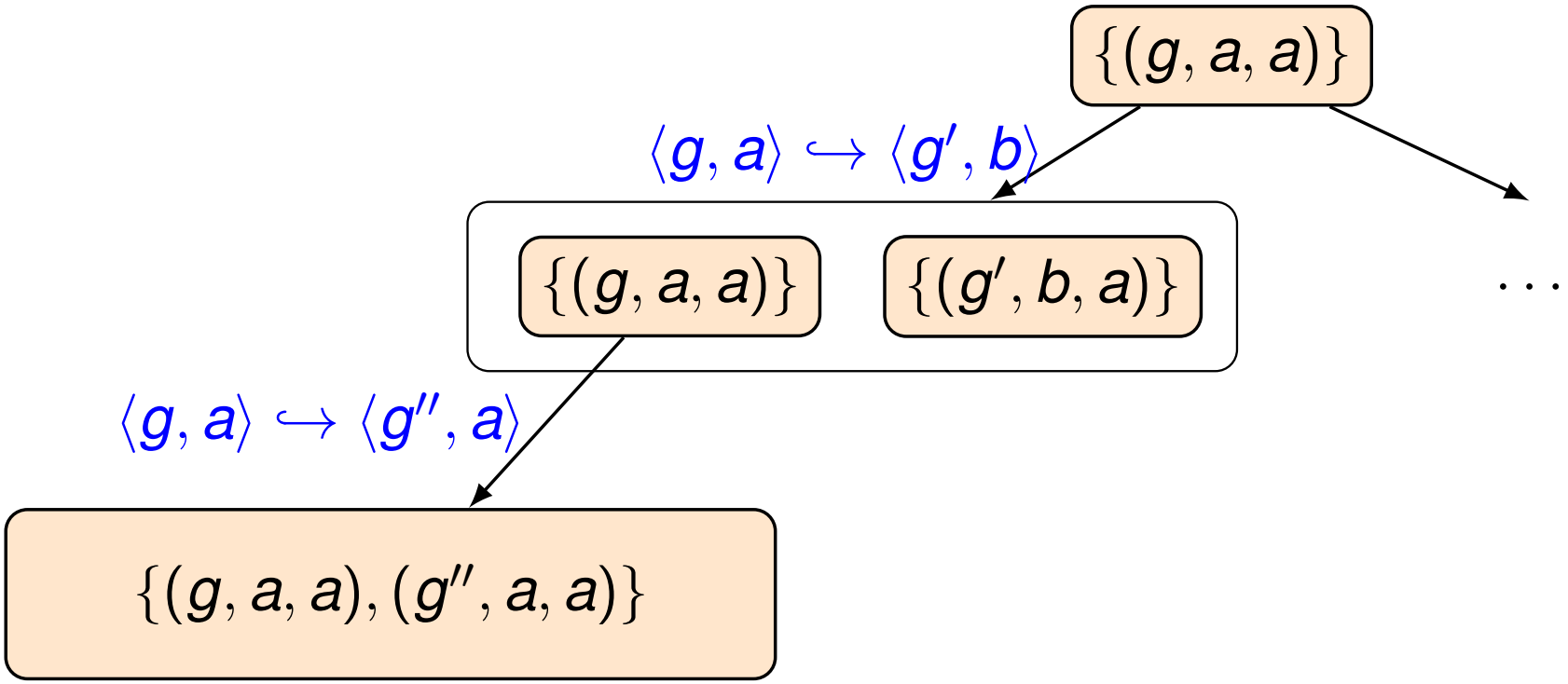
How to split?

Eager splitting: split for every poss. value of globals [Qadeer, Rehof 05]



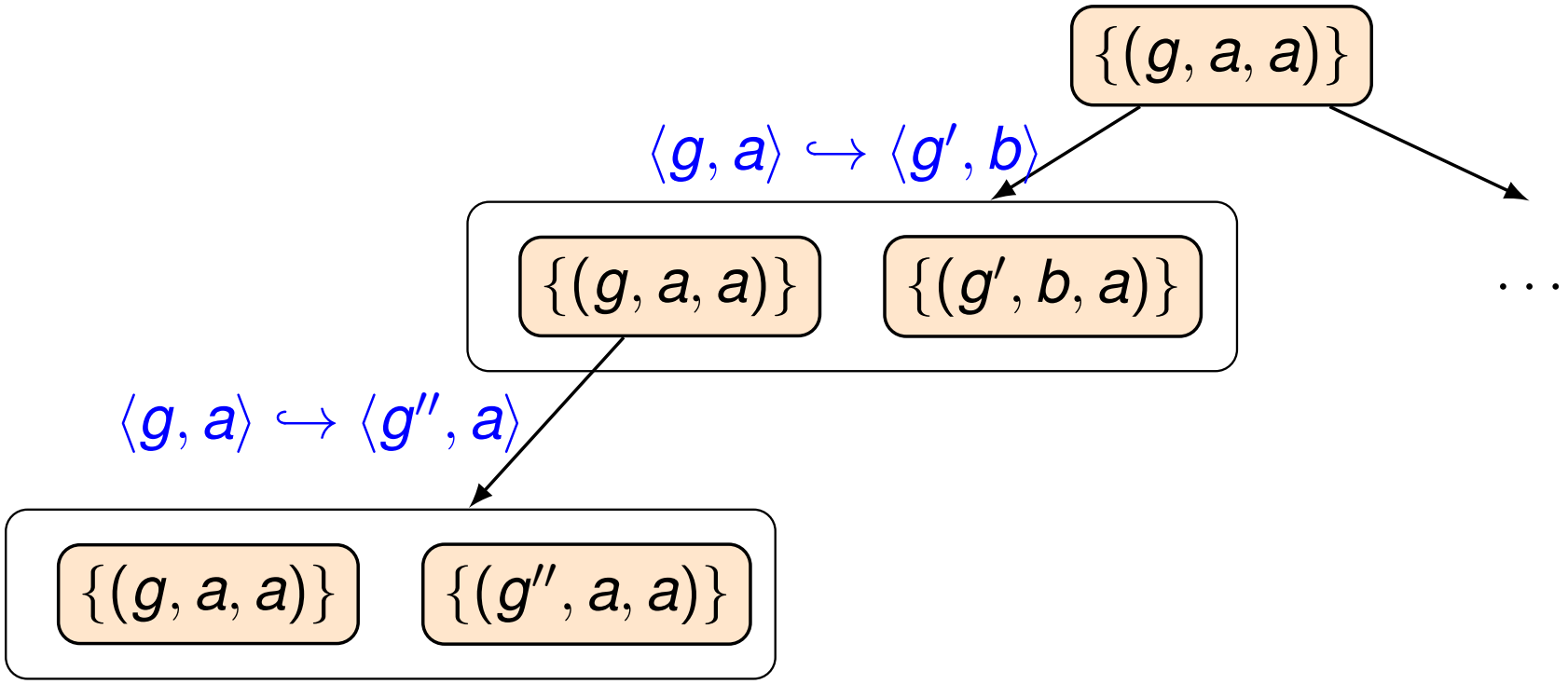
How to split?

Eager splitting: split for every poss. value of globals [Qadeer, Rehof 05]



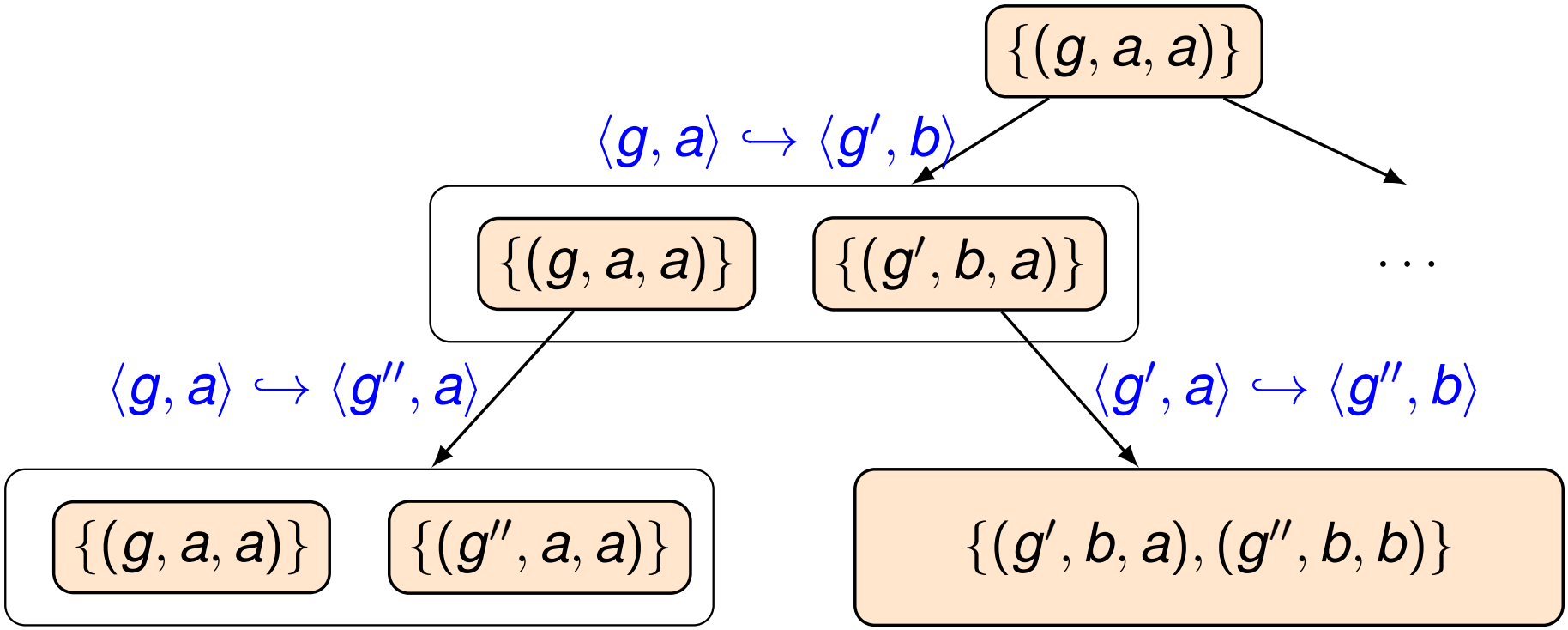
How to split?

Eager splitting: split for every poss. value of globals [Qadeer, Rehof 05]



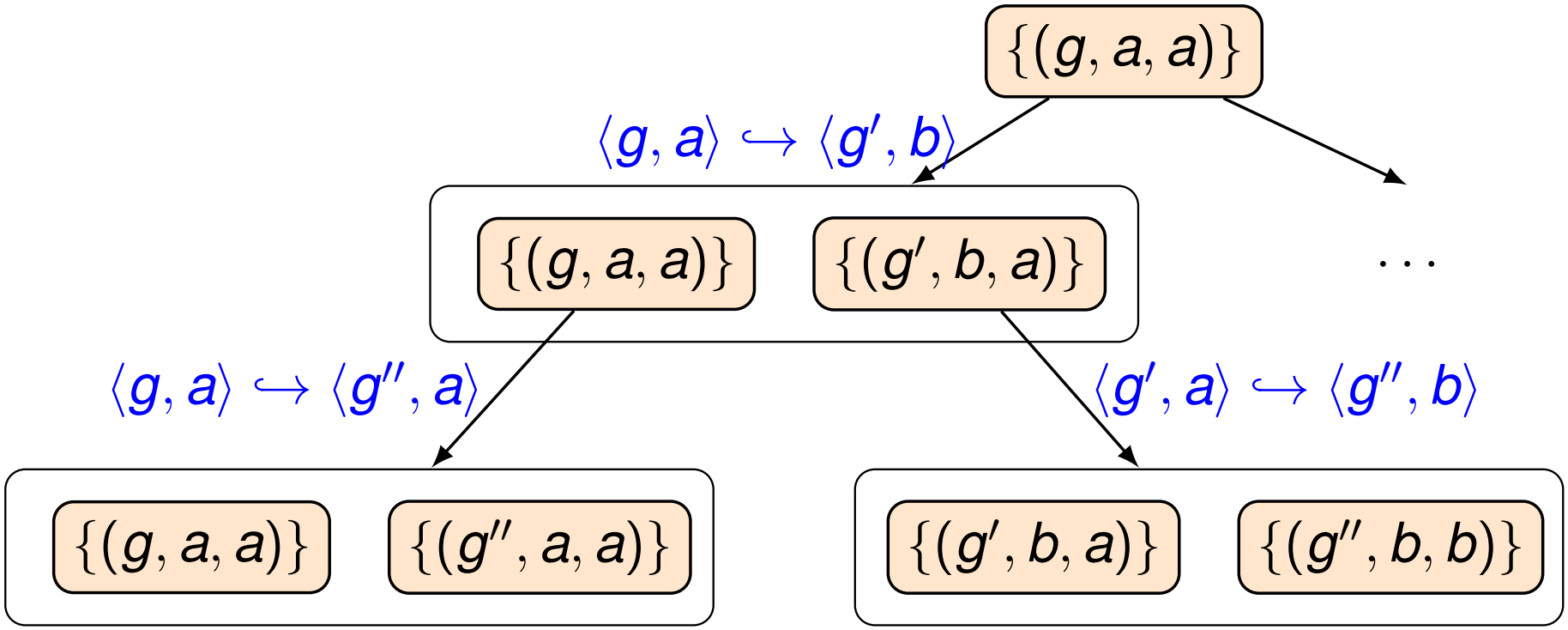
How to split?

Eager splitting: split for every poss. value of globals [Qadeer, Rehof 05]



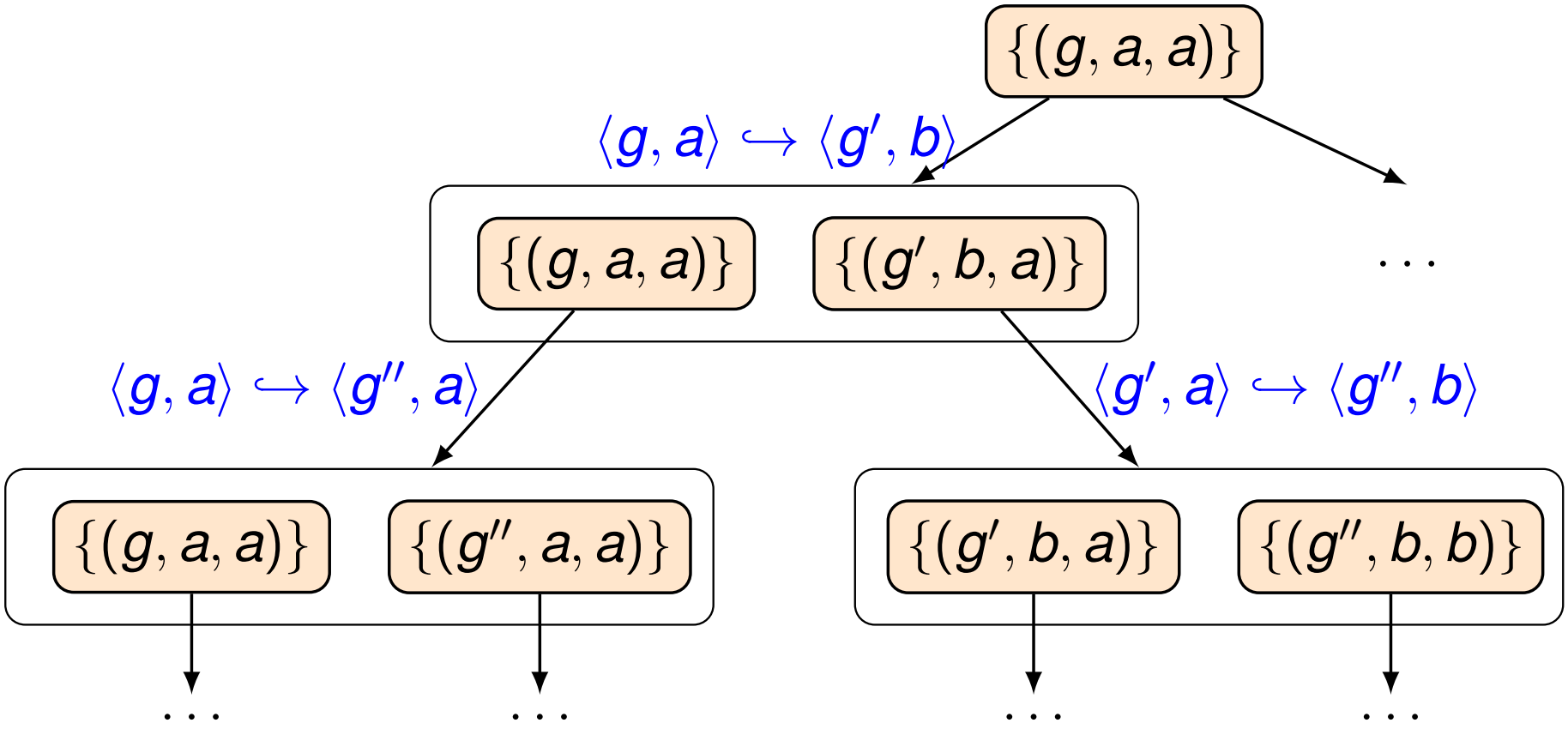
How to split?

Eager splitting: split for every poss. value of globals [Qadeer, Rehof 05]



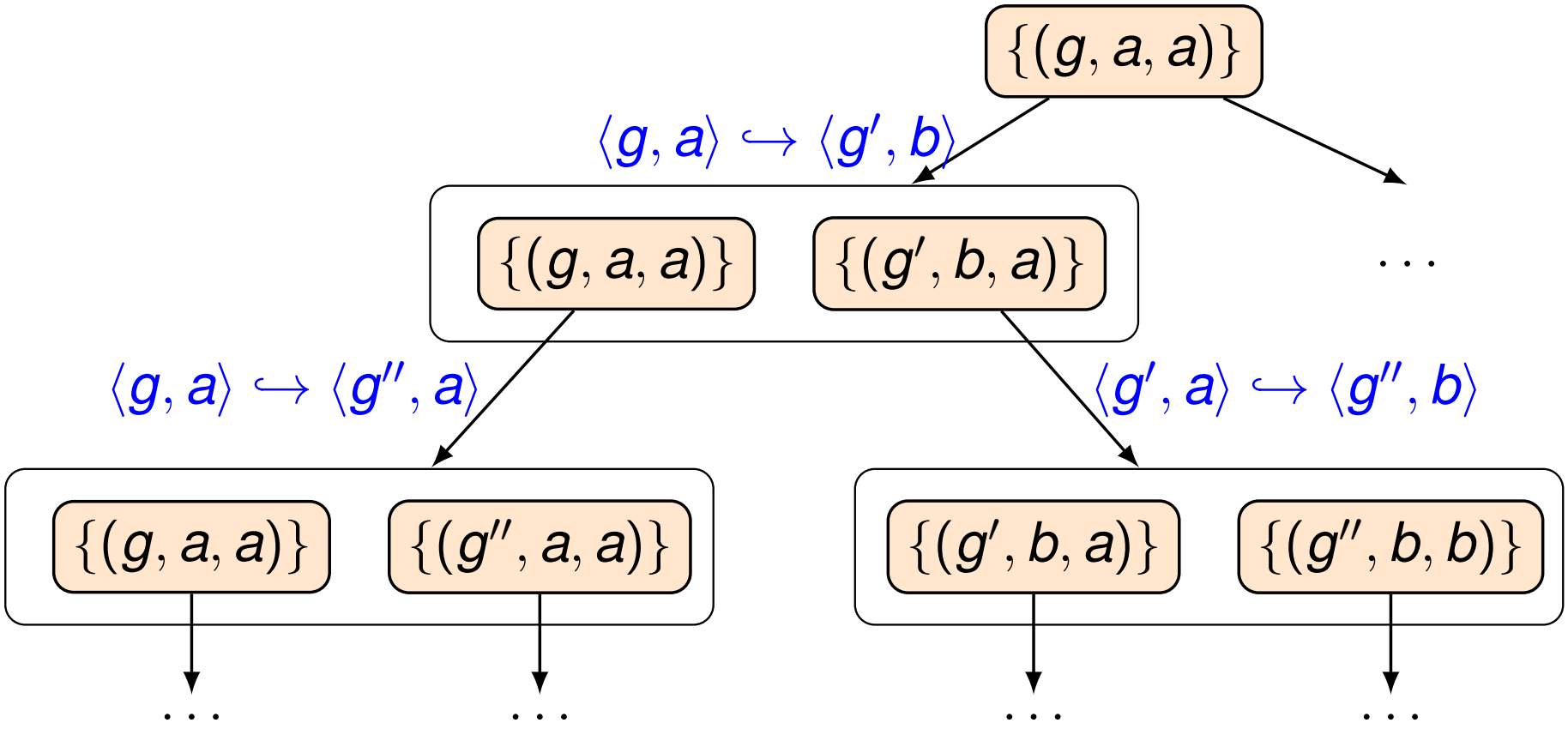
How to split?

Eager splitting: split for every poss. value of globals [Qadeer, Rehof 05]



How to split?

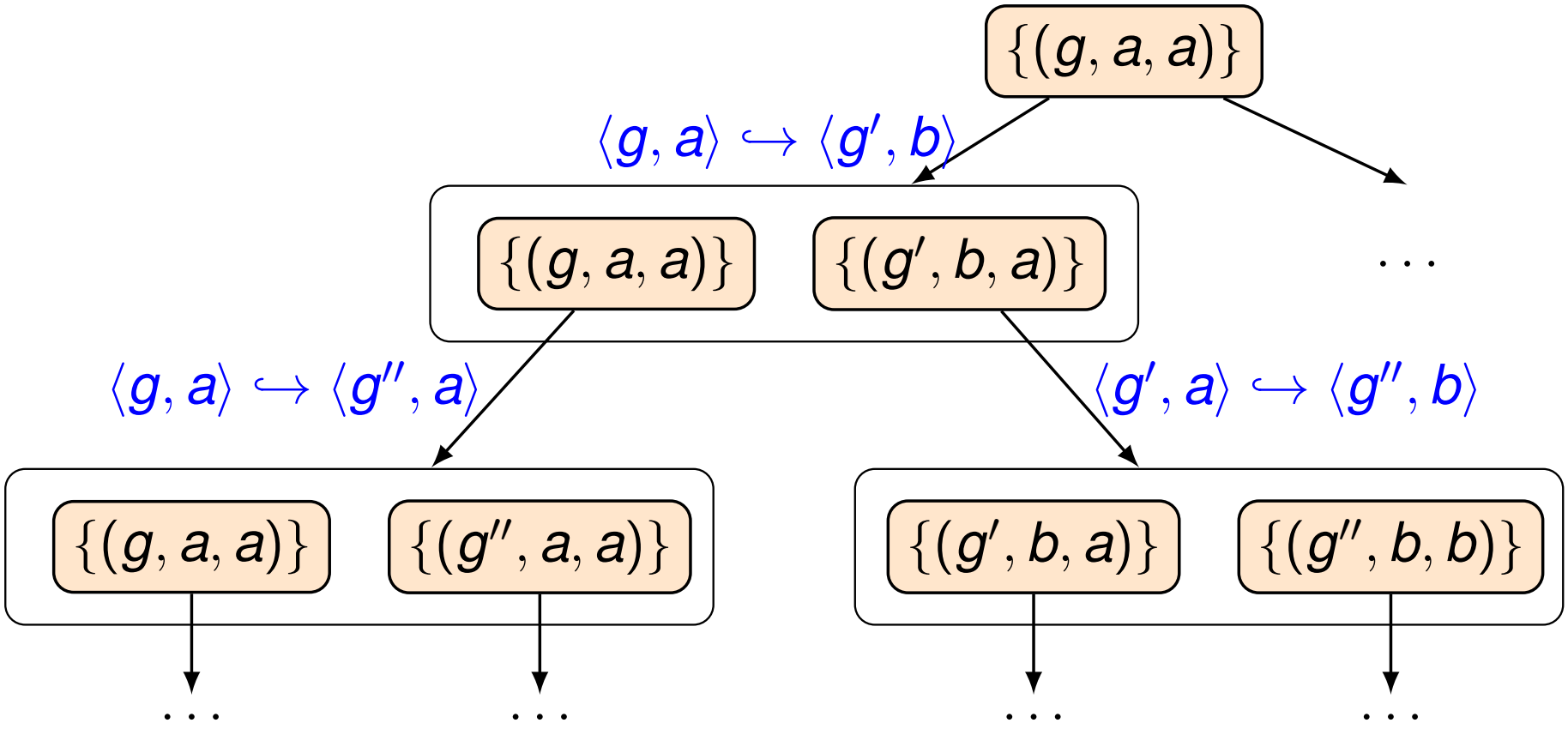
Eager splitting: split for every poss. value of globals [Qadeer, Rehof 05]



Bad news: exponential blow-up, non-symbolic

How to split?

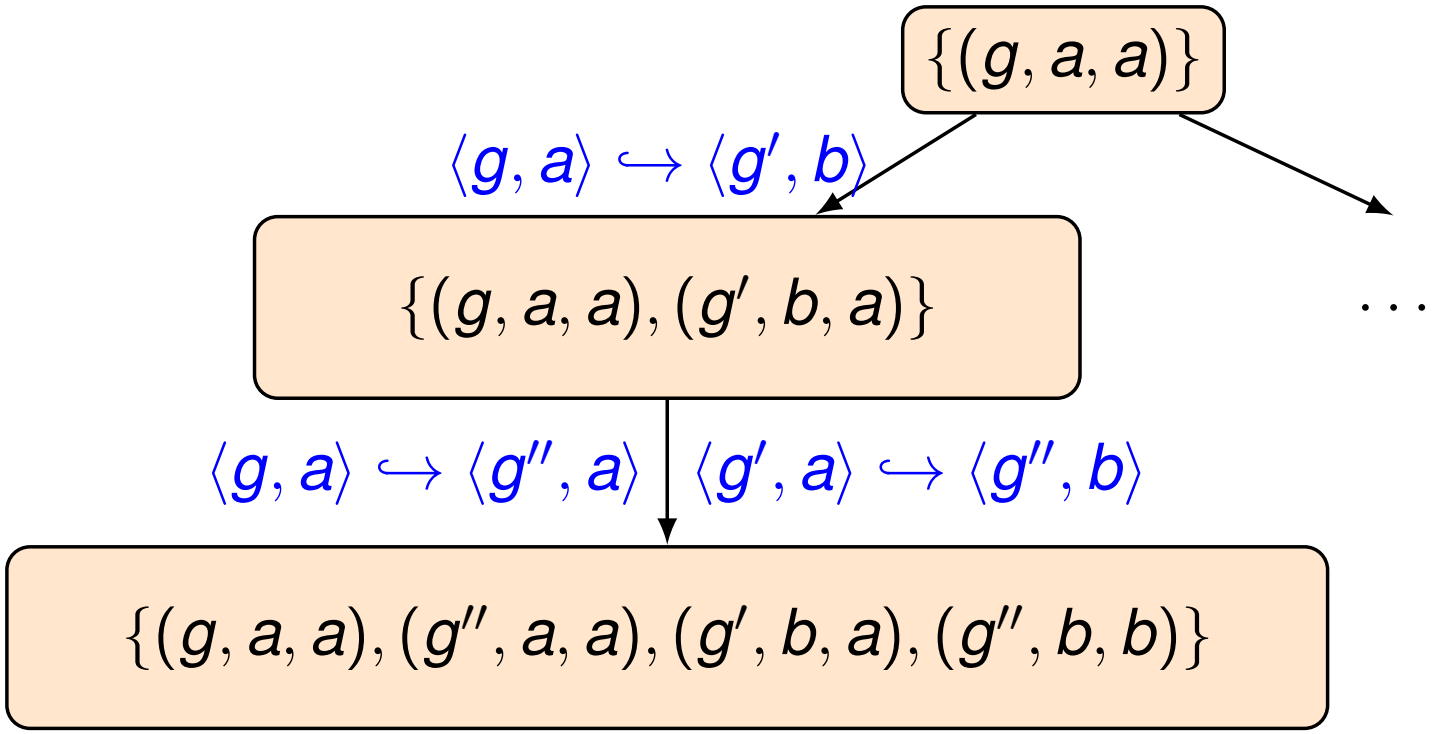
Eager splitting: split for every poss. value of globals [Qadeer, Rehof 05]



Can we group some values together? $\longrightarrow$ Lazy splitting!

How to split?

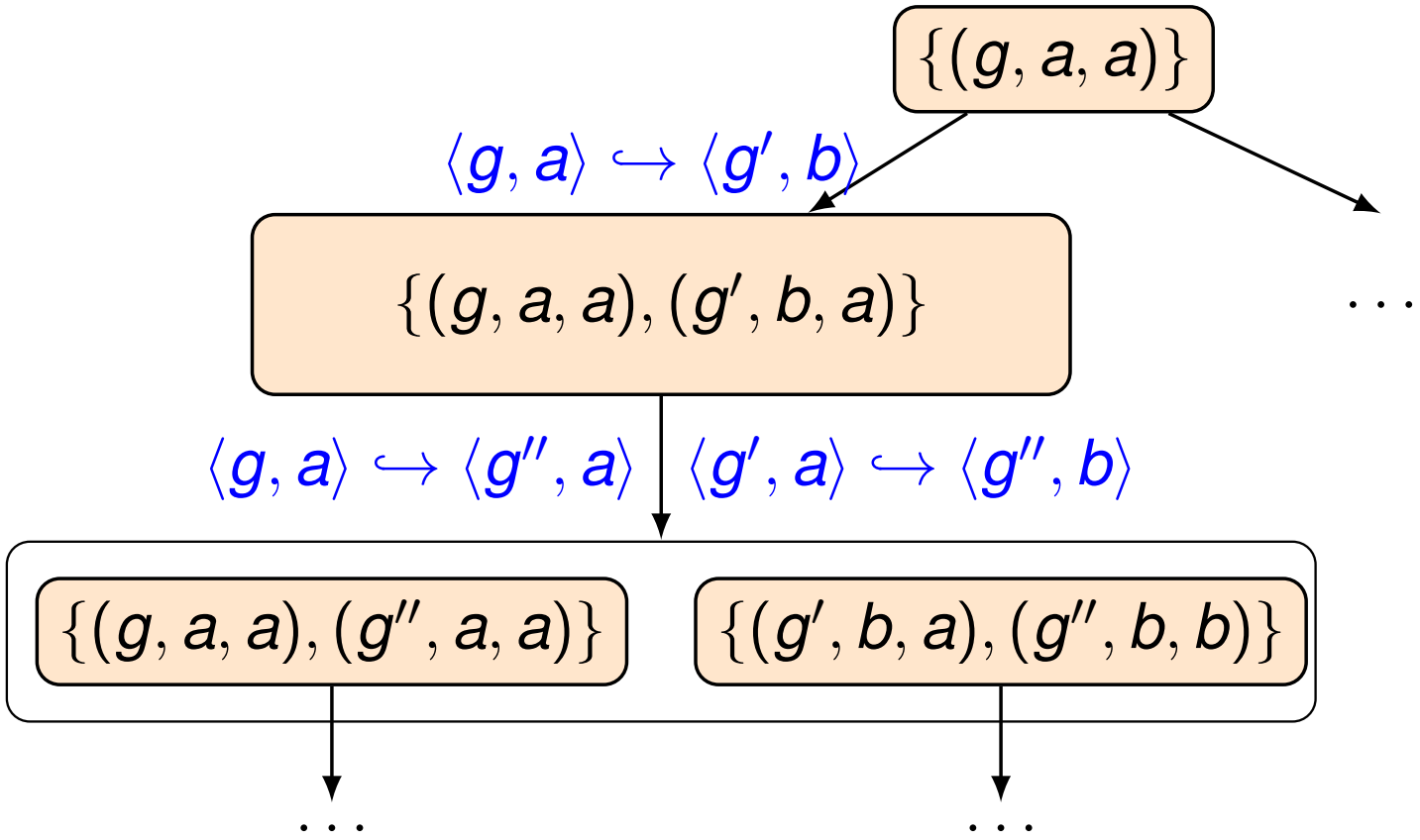
Eager splitting: split for every poss. value of globals [Qadeer, Rehof 05]



Can we group some values together? $\longrightarrow$ Lazy splitting!

How to split?

Eager splitting: split for every poss. value of globals [Qadeer, Rehof 05]



Can we group some values together? → Lazy splitting!

Lazy splitting

Which values of the global can be grouped together for symbolic computation?

Eager: only consider values after $post^*$
Lazy: consider relations of values before and after $post^*$; split on values before $post^*$

Definition Two different values are confluent, if they may get updated to the same value by $post^*$.

Definition A partition of values is safe if none of its groups contains two confluent values.

Theorem Given a set of global configurations C, it is possible to split C (into unions of view tuples) by *any* safe partition (on values of the global in C).

Lazy splitting

Which values of the global can be grouped together for symbolic computation?

Eager: only consider values after $post^*$
Lazy: consider relations of values before and after $post^*$; split on values before $post^*$

Definition Two different values are confluent, if they may get updated to the same value by $post^*$.

Definition A partition of values is safe if none of its groups contains two confluent values.

Theorem Given a set of global configurations C, it is possible to split C (into unions of view tuples) by *any* safe partition (on values of the global in C).

Lazy splitting

Which values of the global can be grouped together for symbolic computation?

Eager: only consider values after $post^*$
Lazy: consider relations of values before and after $post^*$; split on values before $post^*$

Definition Two different values are confluent, if they may get updated to the same value by $post^*$.

Definition A partition of values is safe if none of its groups contains two confluent values.

Theorem Given a set of global configurations C, it is possible to split C (into unions of view tuples) by *any* safe partition (on values of the global in C).

Lazy splitting

Which values of the global can be grouped together for symbolic computation?

Eager: only consider values after $post^*$
Lazy: consider relations of values before and after $post^*$; split on values before $post^*$

Definition Two different values are confluent, if they may get updated to the same value by $post^*$.

Definition A partition of values is safe if none of its groups contains two confluent values.

Theorem Given a set of global configurations C, it is possible to split C (into unions of view tuples) by *any* safe partition (on values of the global in C).

Lazy splitting

How to compute a safe partition?

By introducing another copy of globals, we can compute the relation of globals before and after $post^*$: $U(G, G')$.

Formally, confluence relation:

$$C(g_1, g_2) := g_1 \neq g_2 \wedge \exists g' : U(g_1, g') \wedge U(g_2, g')$$

A partition is safe if and only if its sets are cliques of $\neg C$.

However, finding the best cliques is NP-complete.

We propose a heuristic algorithm for finding a safe partition in a symbolic setting.

Lazy splitting

How to compute a safe partition?

By introducing another copy of globals, we can compute the relation of globals before and after $post^*$: $U(G, G')$.

Formally, confluence relation:

$$C(g_1, g_2) := g_1 \neq g_2 \wedge \exists g' : U(g_1, g') \wedge U(g_2, g')$$

A partition is safe if and only if its sets are cliques of $\neg C$.

However, finding the best cliques is NP-complete.

We propose a heuristic algorithm for finding a safe partition in a symbolic setting.

Lazy splitting

How to compute a safe partition?

By introducing another copy of globals, we can compute the relation of globals before and after $post^*$: $U(G, G')$.

Formally, confluence relation:

$$C(g_1, g_2) := g_1 \neq g_2 \wedge \exists g' : U(g_1, g') \wedge U(g_2, g')$$

A partition is safe if and only if its sets are cliques of $\neg C$.

However, finding the best cliques is NP-complete.

We propose a heuristic algorithm for finding a safe partition in a symbolic setting.

Lazy splitting

How to compute a safe partition?

By introducing another copy of globals, we can compute the relation of globals before and after $post^*$: $U(G, G')$.

Formally, confluence relation:

$$C(g_1, g_2) := g_1 \neq g_2 \wedge \exists g' : U(g_1, g') \wedge U(g_2, g')$$

A partition is safe if and only if its sets are cliques of $\neg C$.

However, finding the best cliques is NP-complete.

We propose a heuristic algorithm for finding a safe partition in a symbolic setting.

Lazy splitting

How to compute a safe partition?

By introducing another copy of globals, we can compute the relation of globals before and after $post^*$: $U(G, G')$.

Formally, confluence relation:

$$C(g_1, g_2) := g_1 \neq g_2 \wedge \exists g' : U(g_1, g') \wedge U(g_2, g')$$

A partition is safe if and only if its sets are cliques of $\neg C$.

However, finding the best cliques is NP-complete.

We propose a heuristic algorithm for finding a safe partition in a symbolic setting.

Experiments: `java.util.Vector` class

x (bits)			1	2	3	4	5	6	7	8
Java 5.0	Eager	T	9.3	10.8	16.9	31.1	67.9	117.8	225.7	457.9
		N	0.4	0.5	0.8	1.4	2.5	5.2	9.0	18.1
		V	48	87	167	327	648	1348	2567	5126
	Lazy	T	19.7	17.7	19.6	17.5	17.2	18.9	16.7	18.8
		N	1.2	1.2	1.2	1.3	1.2	1.3	1.3	1.3
		V	3	3	3	3	3	3	3	3
Java 6.0	Eager	T	15.1	18.6	37.5	64.3	147.7	301.7	642.0	1732.0
		N	0.4	0.7	1.1	2.0	3.7	7.1	13.9	27.9
		V	105	209	417	833	1655	3329	6657	13313
	Lazy	T	20.9	20.8	19.4	22.3	20.8	18.8	23.4	23.2
		N	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3
		V	3	3	3	3	3	3	3	3

T = Time (s), N = BDD Nodes ($\times 10^6$), V = View tuples

Experiments: Windows NT Bluetooth driver

	Version 1		Version 2		Version 3	
	Eager	Lazy	Eager	Lazy	Eager	Lazy
Time (s)	1.1	1.3	51.7	36.0	11.9	6.0
Nodes ($\times 10^3$)	46	88	720	1851	195	518
View tuples	21	4	1460	154	234	16
Contexts	3		5		4	

Conclusion

- Model checking in a software devolopment environment
- Symbolic testing: uses a BDD-based model checker for testing a large set of inputs.
- Symbolic context-switching for multithreaded programs
- Generates coverage information and finds common errors
- User-friendly interface, model checker is hidden.
- Can be used as a complement to JUnit.

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