Parallelising Symbolic State-Space Generators:
Frustration & Hope

Gerald Luettgen, University of York
www.cs.york.ac.uk/~luettgen

In collaboration with

Jonathan Ezekiel, Imperial College
Gianfranco Ciardo, University of California at Riverside
Radu Siminiceanu, National Institute of Aerospace, Virginia

Research funded by EPSRC under grant no. GR/S86211/01

© 2008 G. Luettgen, Department of Computer Science, University of York

FORMED GALOP GT-VMT HFL LDTA LSCITS MBT OPENCERT

ACCAT ARSPA-WITS BYTECODE COCV COMPASS FLNCO

QAPL RC SAFECERT TAASN TERMGRAPH VAMP WINMC

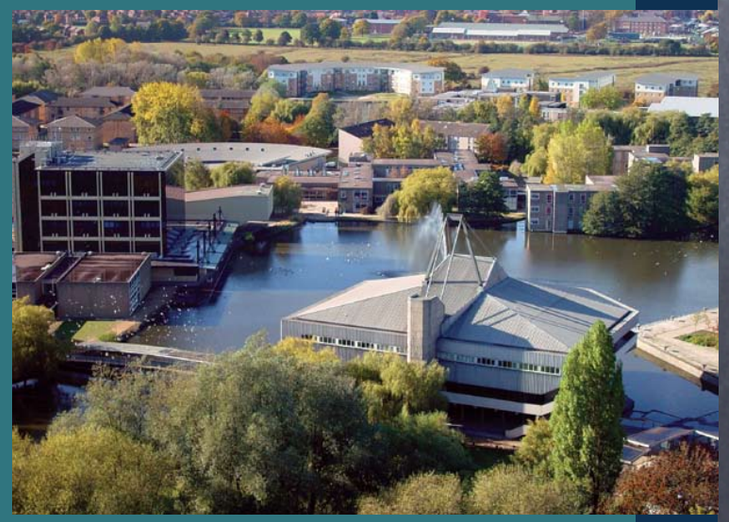
ETAPS 2009
22-29 MARCH
YORK ENGLAND

European Joint Conferences on Theory and Practice of Software

CC:	International Conference on Compiler Construction
ESOP:	European Symposium on Programming
FASE:	Fundamental Approaches to Software Engineering
FOSSACS:	Foundations of Software Science and Computation Structures
TACAS:	Tools and Algorithms for the Construction and Analysis of Systems

Satellite Events
4 Tutorials, 21 Workshops

Invited Speakers
Rajeev Alur, Pennsylvania
Jean-Marc Eber, Paris
Stephen Gilmore, Edinburgh
Steven Miller, Rockwell Collins
John Reynolds, Carnegie Mellon
Vivek Sarkar, Rice
Wolfgang Thomas, Aachen



WWW.ETAPS.ORG

EASST EAPLS

THE UNIVERSITY of York

Le Menu

Le Menu

Saturation Séquentièlle

A taste of symbolic state-space generation

Le Menu

Saturation Séquentièlle

A taste of symbolic state-space generation

Saturation Parallèle

2 Cilk-flavoured implementations

1 Pthreads-flavoured implementation

Le Menu

Saturation Séquentièlle

A taste of symbolic state-space generation

Saturation Parallèle

2 Cilk-flavoured implementations

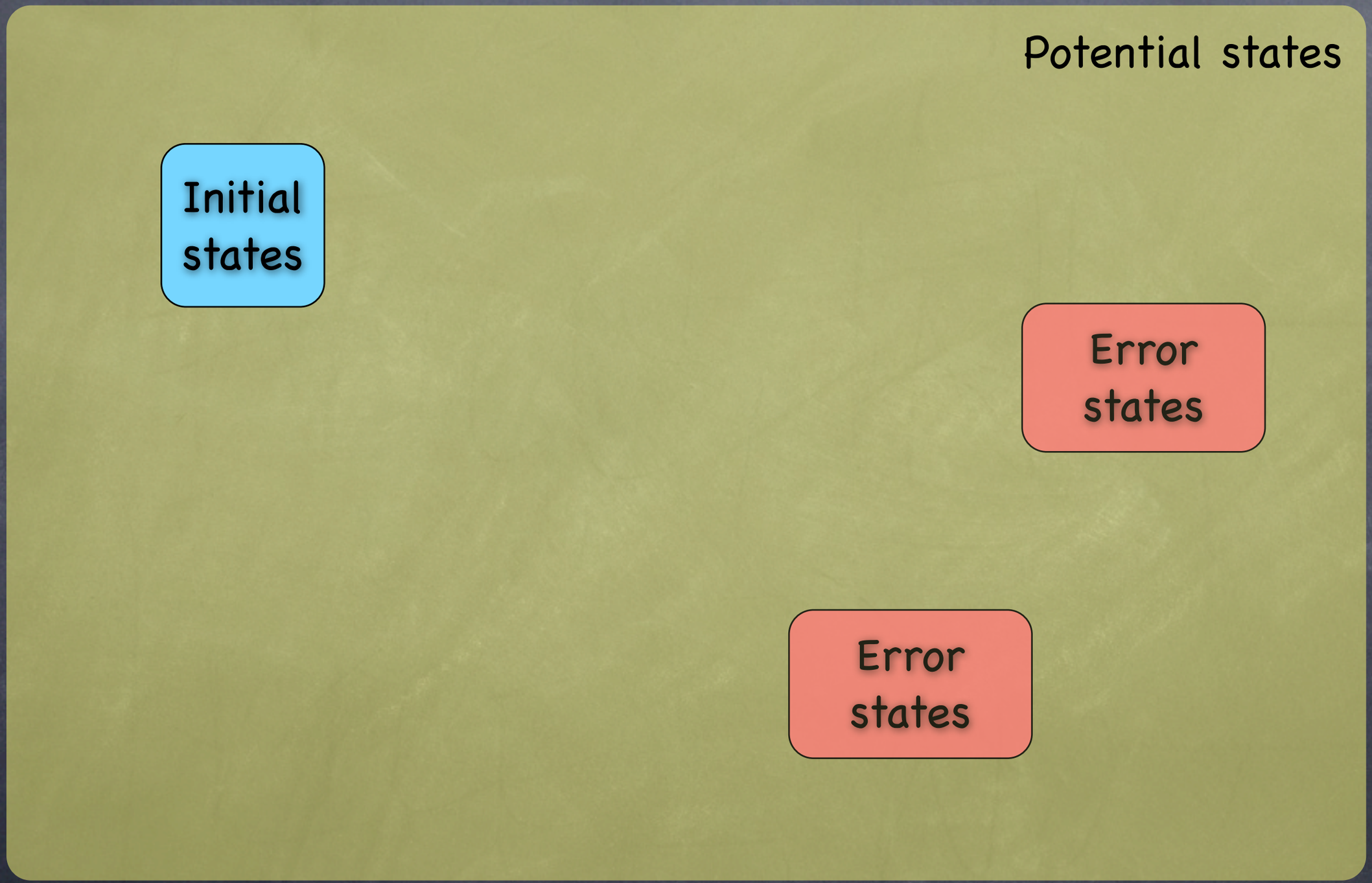
1 Pthreads-flavoured implementation

A crunch of experimental results

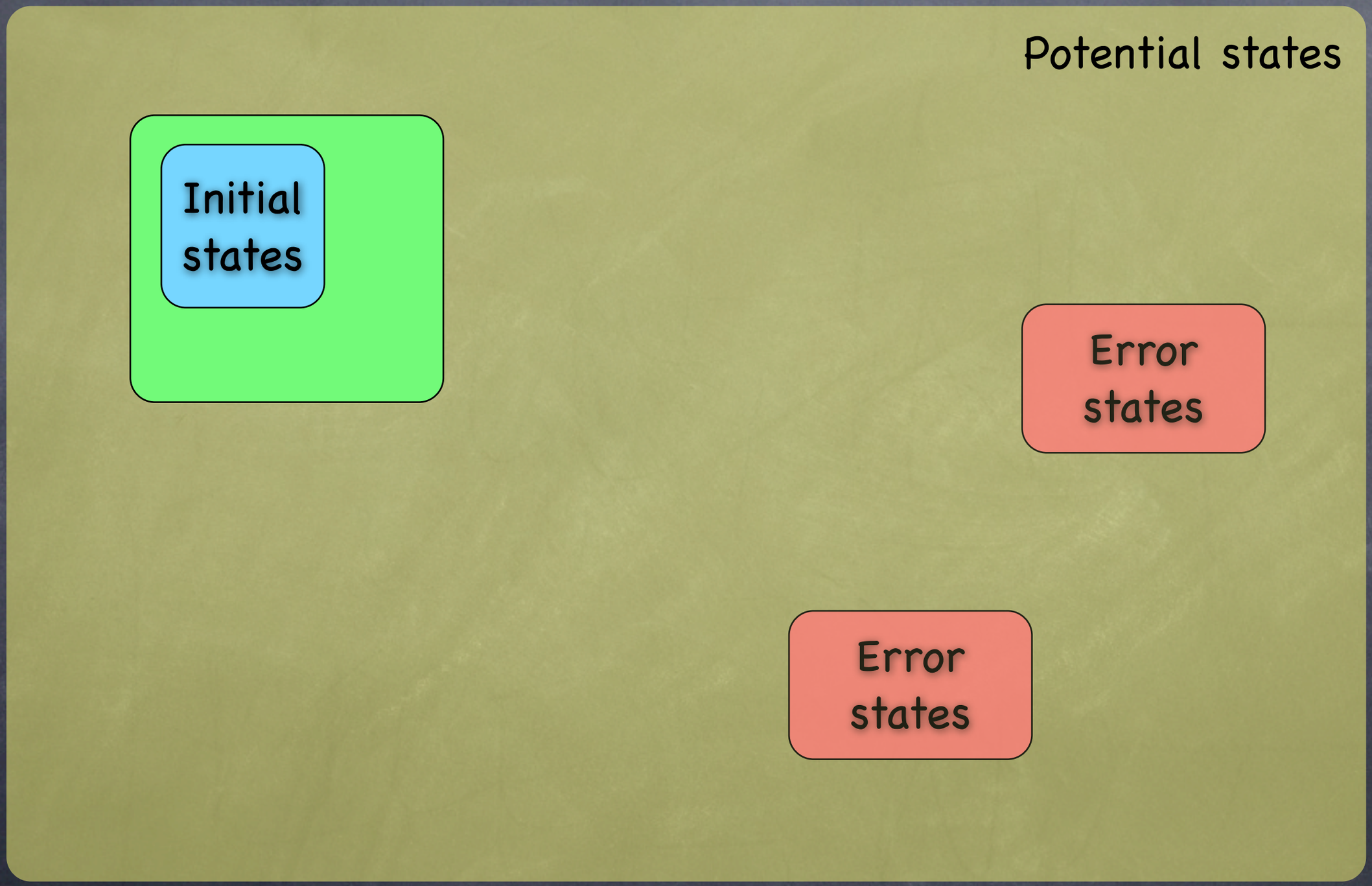
Frustrations and hopes: Can you do soufflé?

Conclusions: What have we learned?

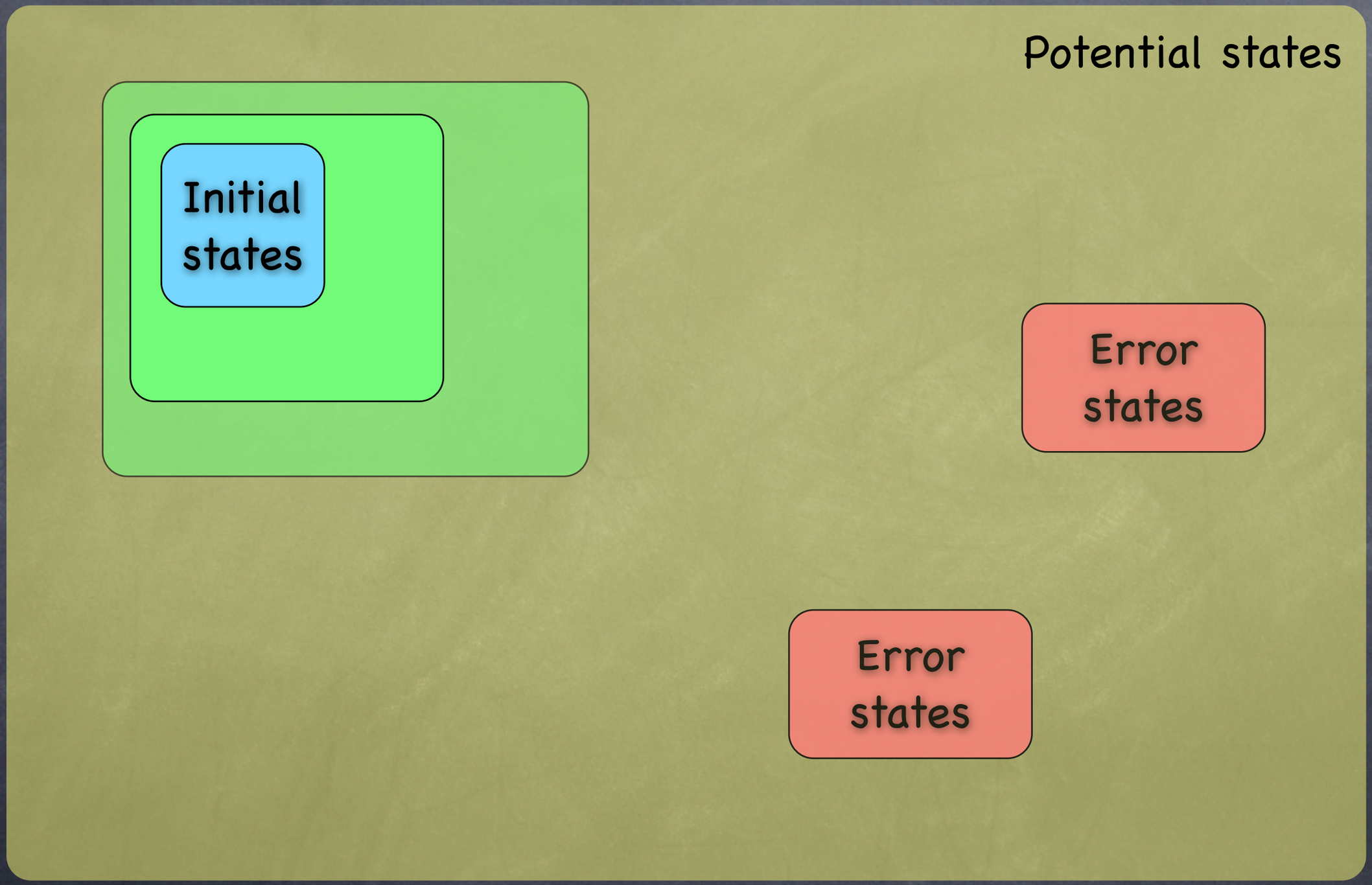
Symbolic Reachability Analysis



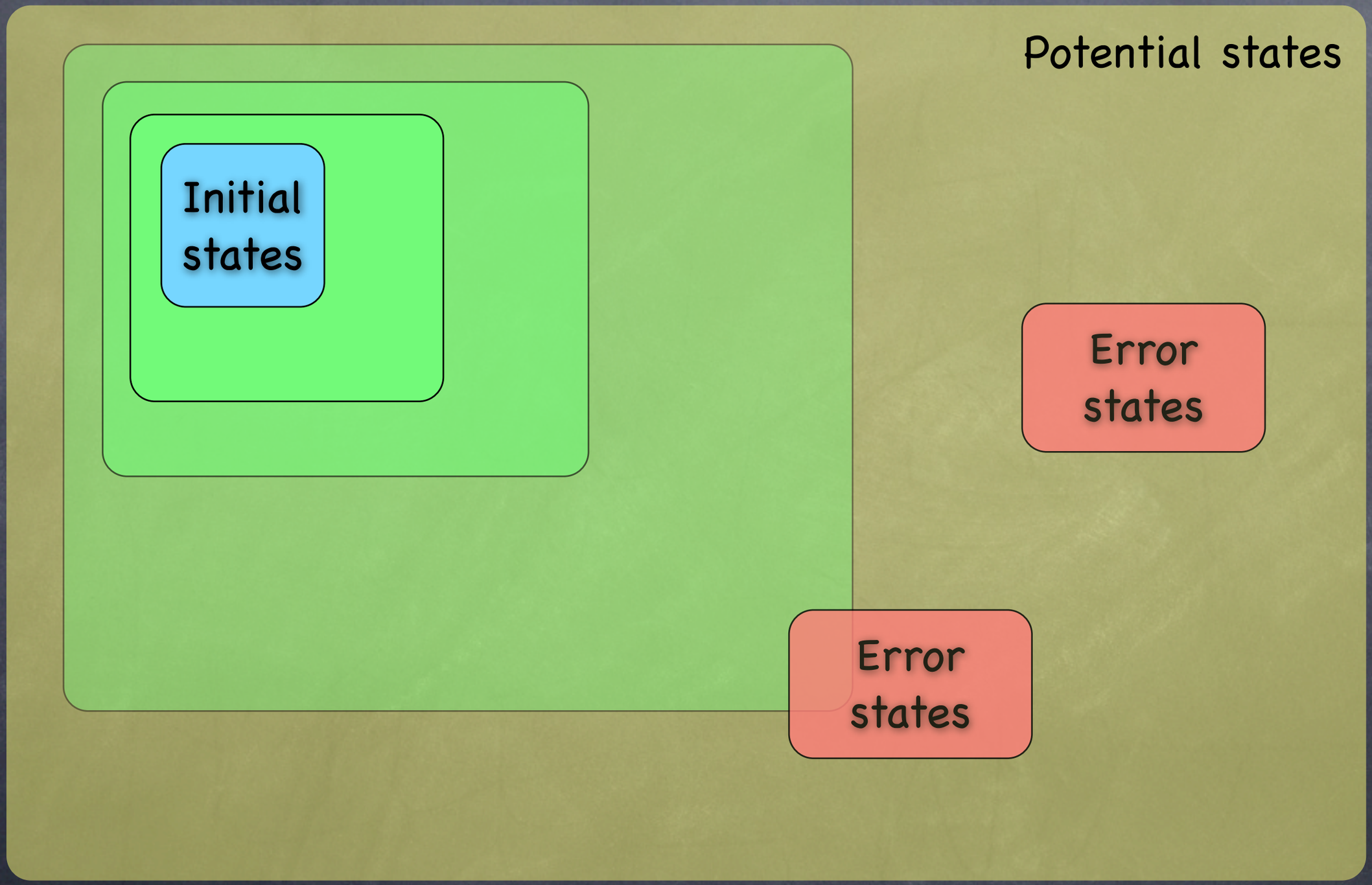
Symbolic Reachability Analysis



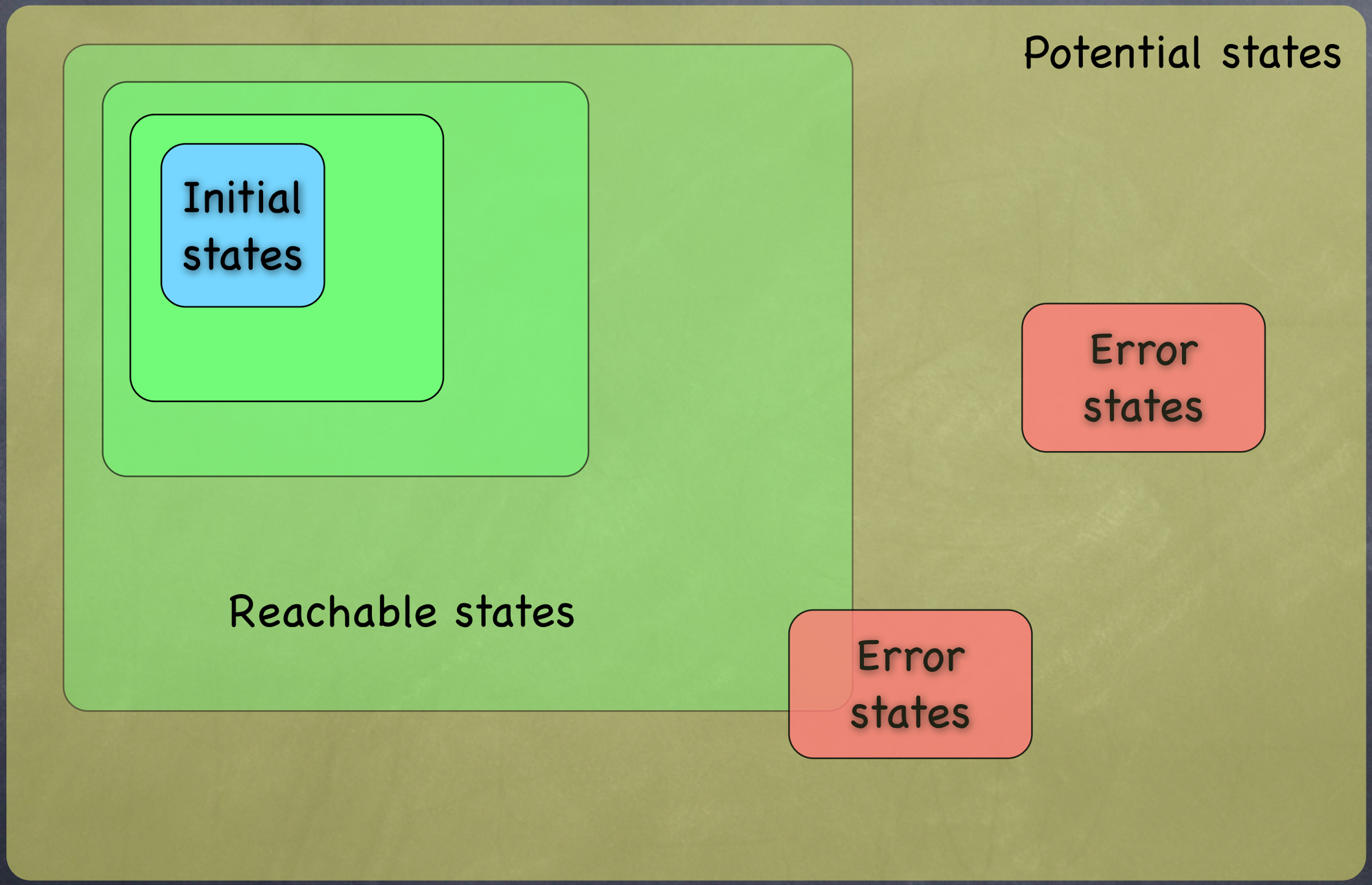
Symbolic Reachability Analysis



Symbolic Reachability Analysis



Symbolic Reachability Analysis



State Spaces as Fixed Points

State Spaces as Fixed Points

- What is a system/model?
  - A set of state variables $X = \{x_K,\ldots,x_1\}$, with the set States of potential states being the set of assignments over X
  - A set of initial states S_{init}
  - A next-state function $N : States \rightarrow 2^{States}$

State Spaces as Fixed Points

- What is a system/model?
  - A set of state variables $X = \{x_K,\ldots,x_1\}$, with the set States of potential states being the set of assignments over X
  - A set of initial states S_{init}
  - A next-state function $N : States \rightarrow 2^{States}$
- The reachable state space is the result of a least fixed point computation
  - $S_0 = S_{init}$
  - $S_{i+1} = Union(S_i, N(S_i))$

Asynchronous Models & Breadth-First Search

Asynchronous Models & Breadth-First Search

- Asynchronous models are typically event-based
  - A set Events of events with representative e
  - A disjunctively partitioned next-state function

$$N(X) = \bigcup_{e \in Events} N_e(X)$$

Asynchronous Models & Breadth-First Search

- Asynchronous models are typically event-based
  - A set Events of events with representative e
  - A disjunctively partitioned next-state function

$$N(X) = \bigcup_{e \in Events} N_e(X)$$

- BFS-based state-space generation with chaining

```
S = Sinit
do
    for each e∈Events do S = Union(S, Ne(S))
while S does not change
```

Saturation: A Better Search Strategy

Saturation: A Better Search Strategy

- Asynchronous systems exhibit event locality
  - The firing of an event only updates a small subset of the state variables (cf. interleaving semantics)
  - This is exploited in our Saturation algorithm

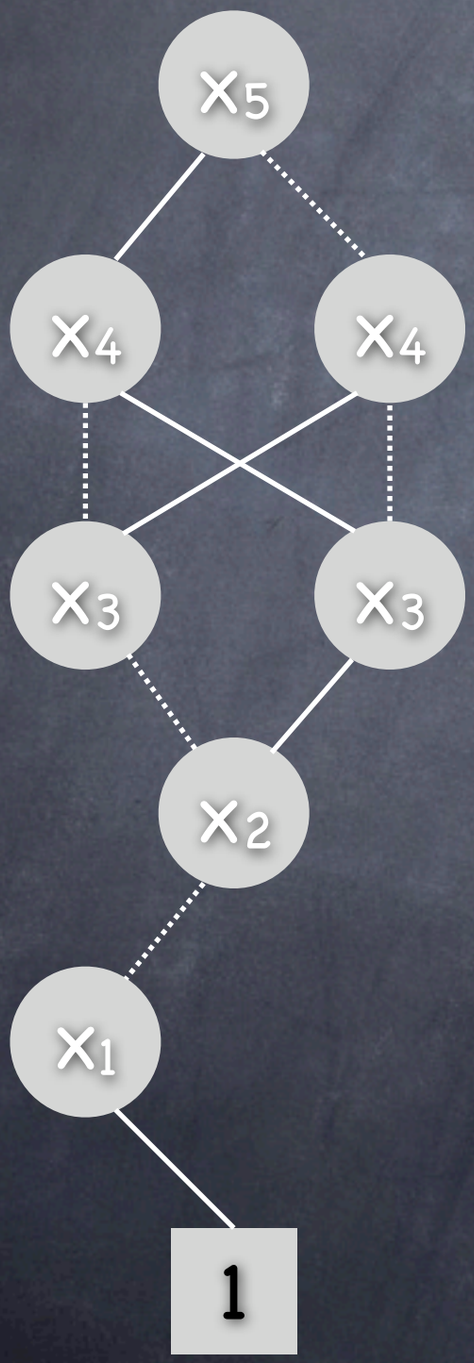
Saturation: A Better Search Strategy

- Asynchronous systems exhibit event locality
  - The firing of an event only updates a small subset of the state variables (cf. interleaving semantics)
  - This is exploited in our Saturation algorithm

- Saturation, explained via an example model having
  - A state vector of 5 variables: x_5, x_4, x_3, x_2 and x_1
  - Three events e_{21}, e_2 and e_1 which affect variables x_2 and x_1, only variable x_2 and only variable x_1, respectively

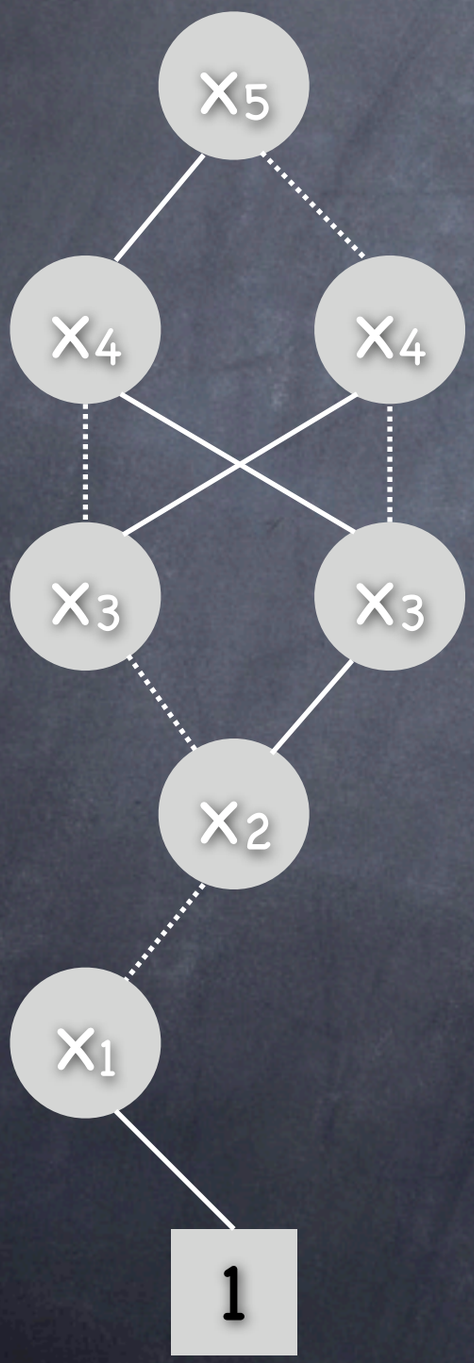
Decision Diagrams + Partitioning by Event

State space



Decision Diagrams + Partitioning by Event

State space



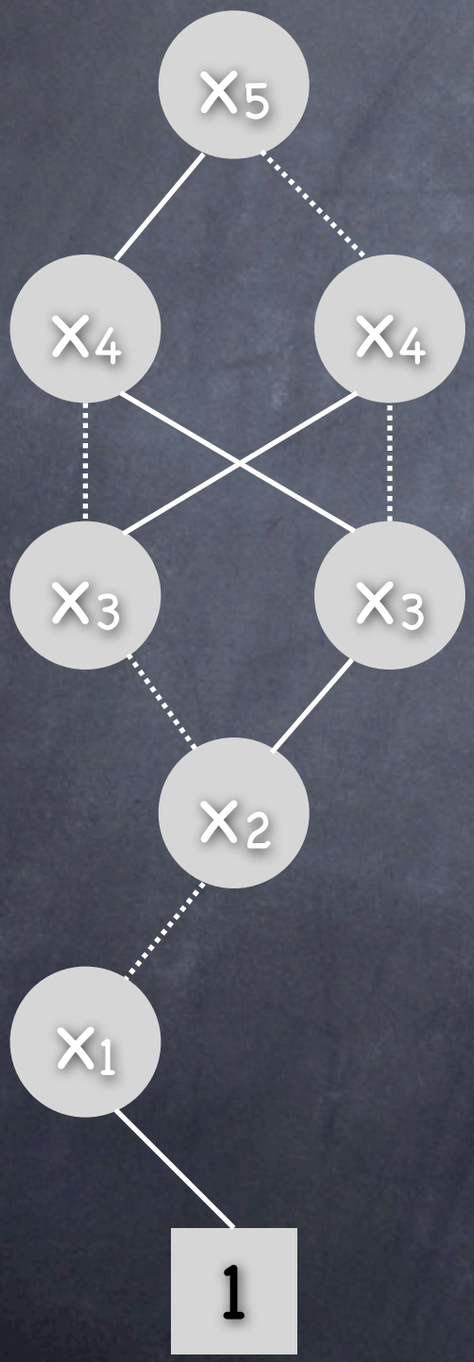
Next-state function

$$N(X) = U_{e \in Events}\ N_e(X)$$

$$= N_{e21}(X) \cup N_{e2}(X) \cup N_{e1}(X)$$

Decision Diagrams + Partitioning by Event

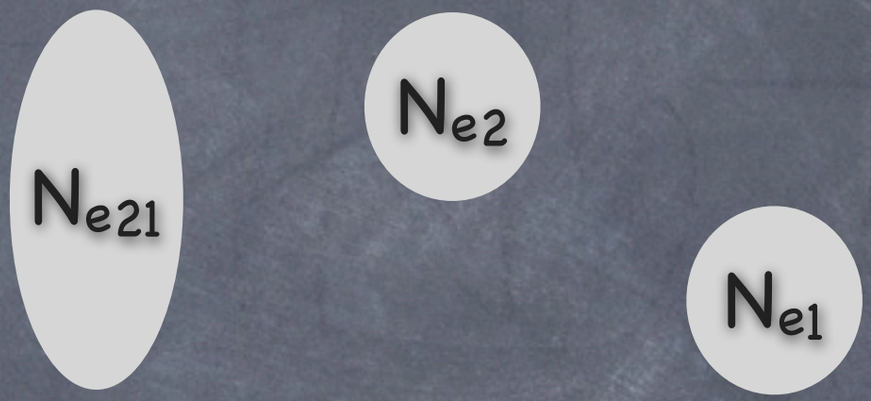
State space



Next-state function

$$N(X) = U_{e \in Events}\ N_e(X)$$

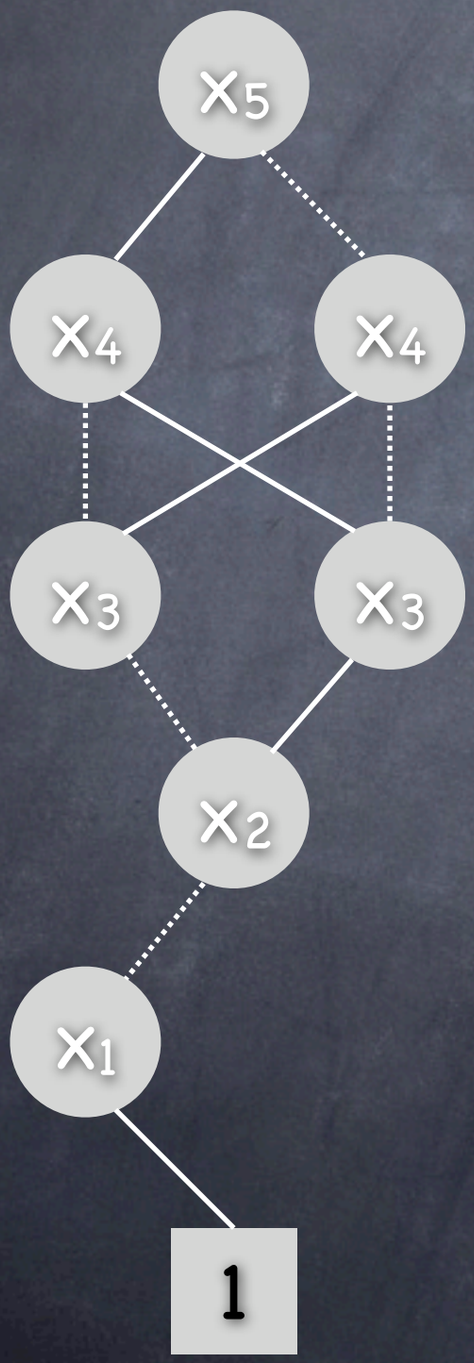
$$= N_{e21}(X) \cup N_{e2}(X) \cup N_{e1}(X)$$



In the symbolic encoding of decision diagrams, each event is assigned a top-level: $Top(e_{21}) = 2$, $Top(e_2) = 2$ and $Top(e_1) = 1$

Partitioning by Top + Identity Reduction

State space



$$N(X) = U_{i=K,\ldots,1}\ N_i(X)$$

$N_5(X) = X$

$N_4(X) = X$

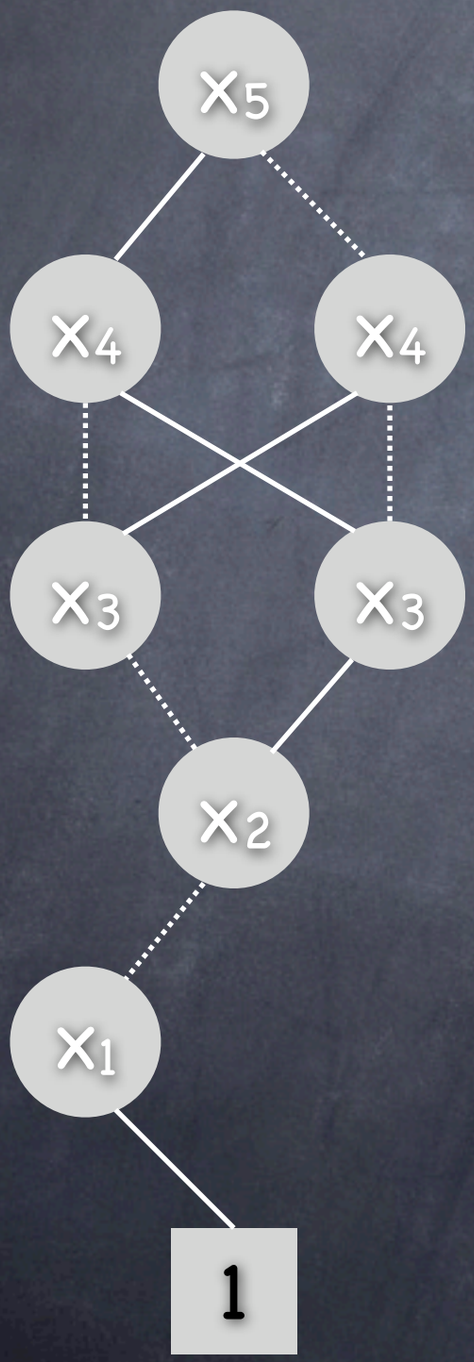
$N_3(X) = X$

$N_2(X) = N_{e21}(X) \cup N_{e2}(X)$

$N_1(X) = N_{e1}(X)$

Partitioning by Top + Identity Reduction

State space



$N(X) = U_{i=K,\ldots,1}\ N_i(X)$

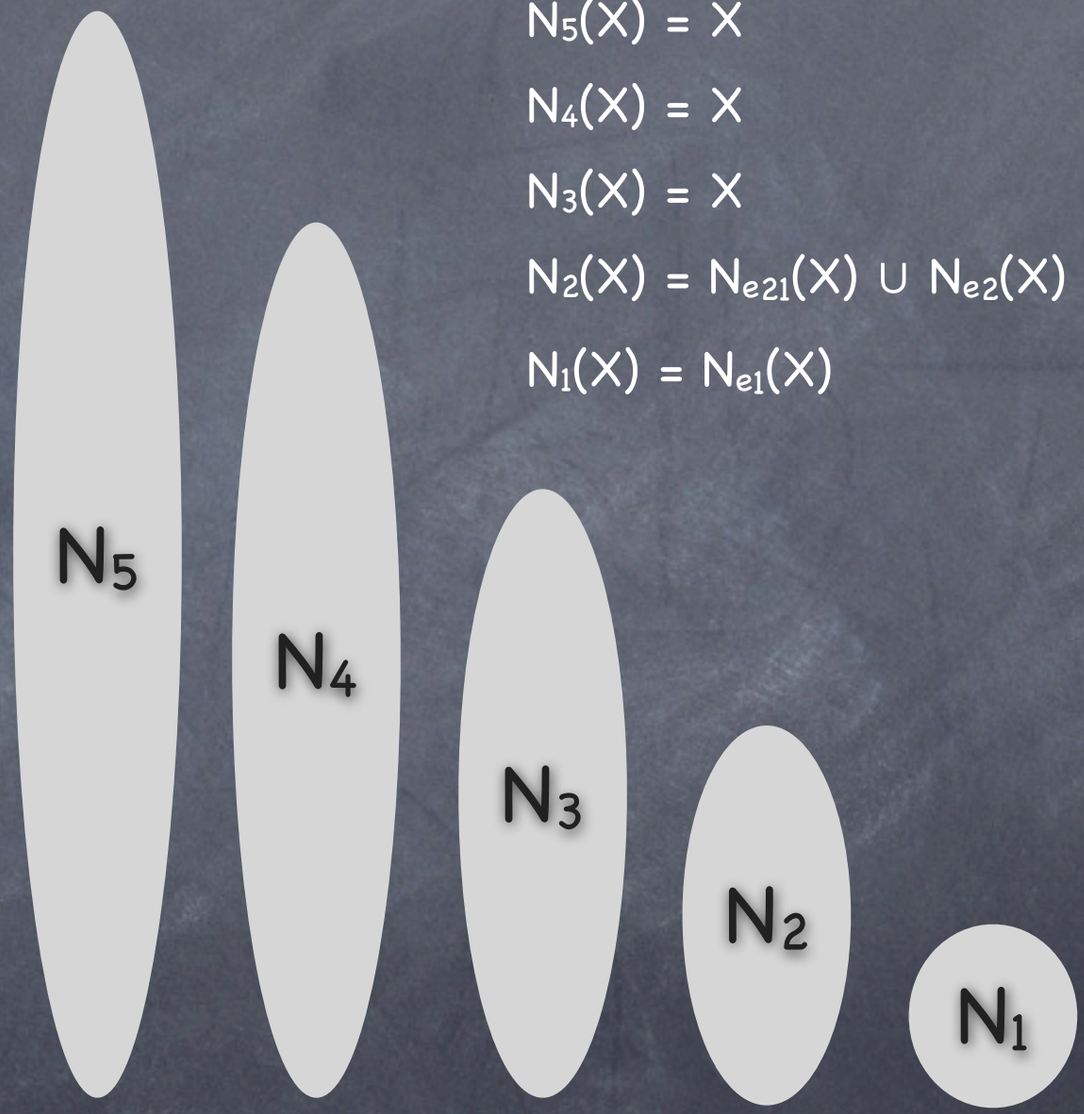
$N_5(X) = X$

$N_4(X) = X$

$N_3(X) = X$

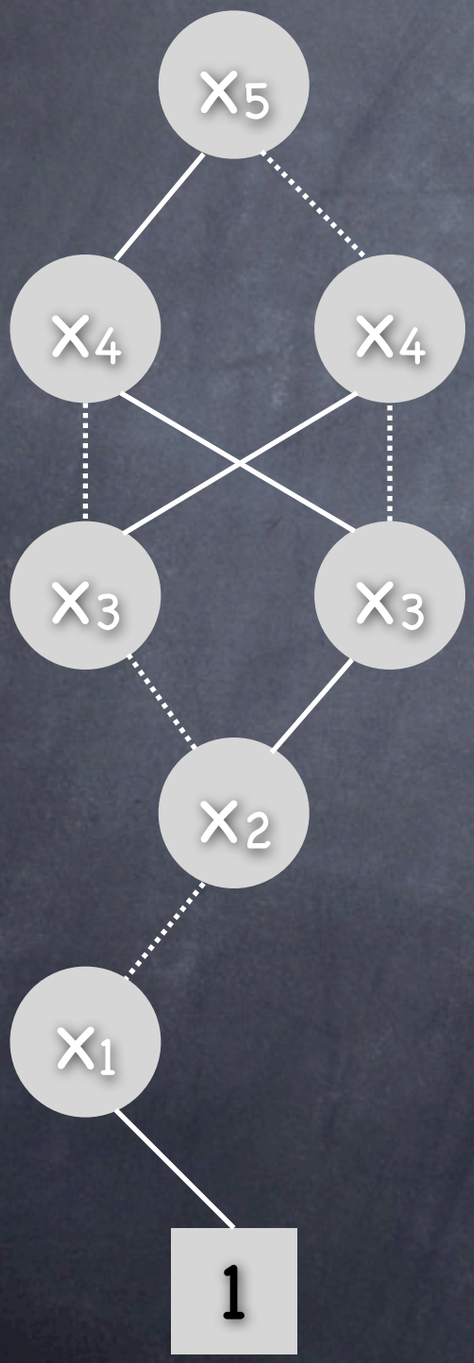
$N_2(X) = N_{e21}(X) \cup N_{e2}(X)$

$N_1(X) = N_{e1}(X)$



Partitioning by Top + Identity Reduction

State space



$N(X) = U_{i=K,\ldots,1}\ N_i(X)$

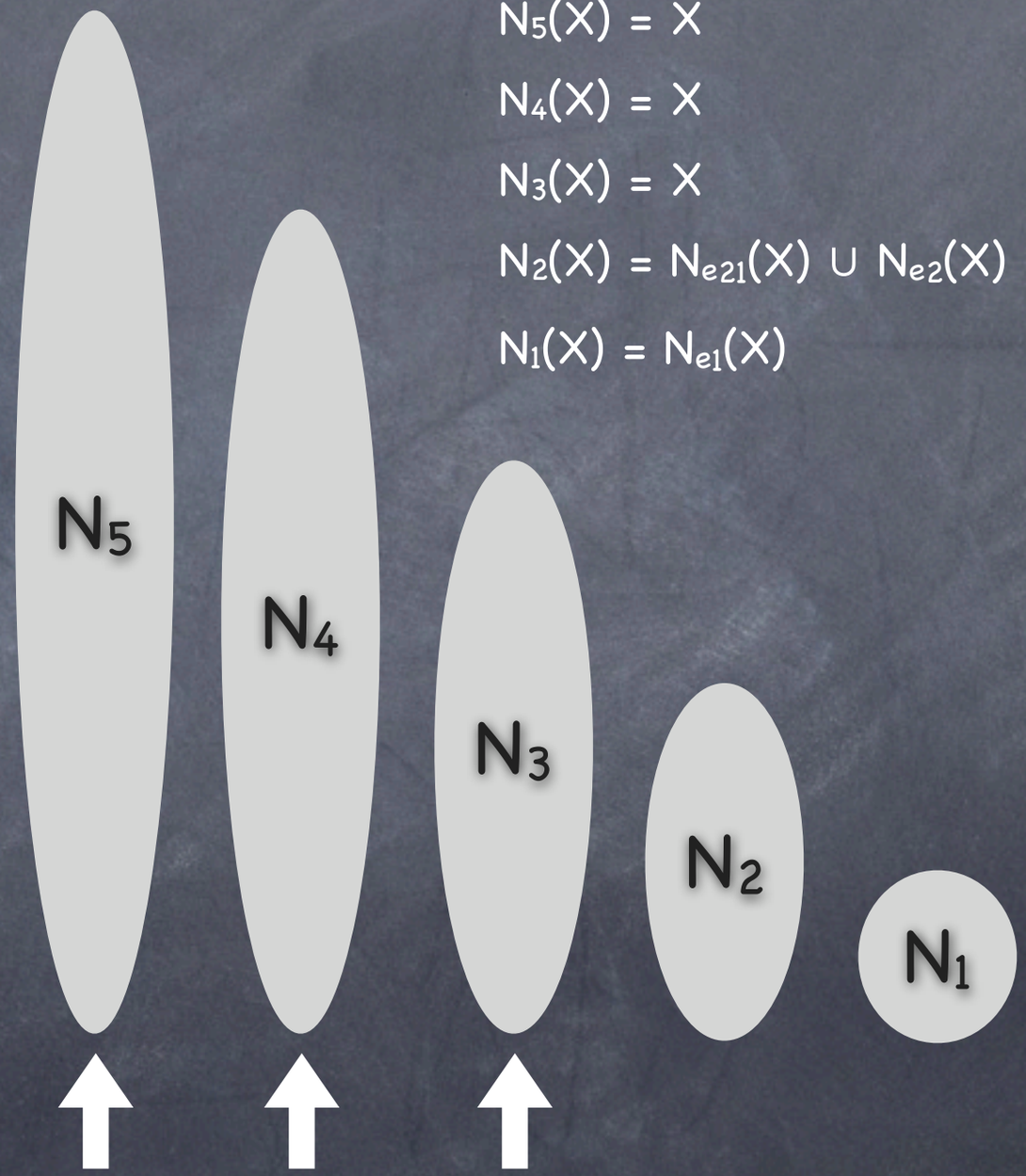
$N_5(X) = X$

$N_4(X) = X$

$N_3(X) = X$

$N_2(X) = N_{e21}(X) \cup N_{e2}(X)$

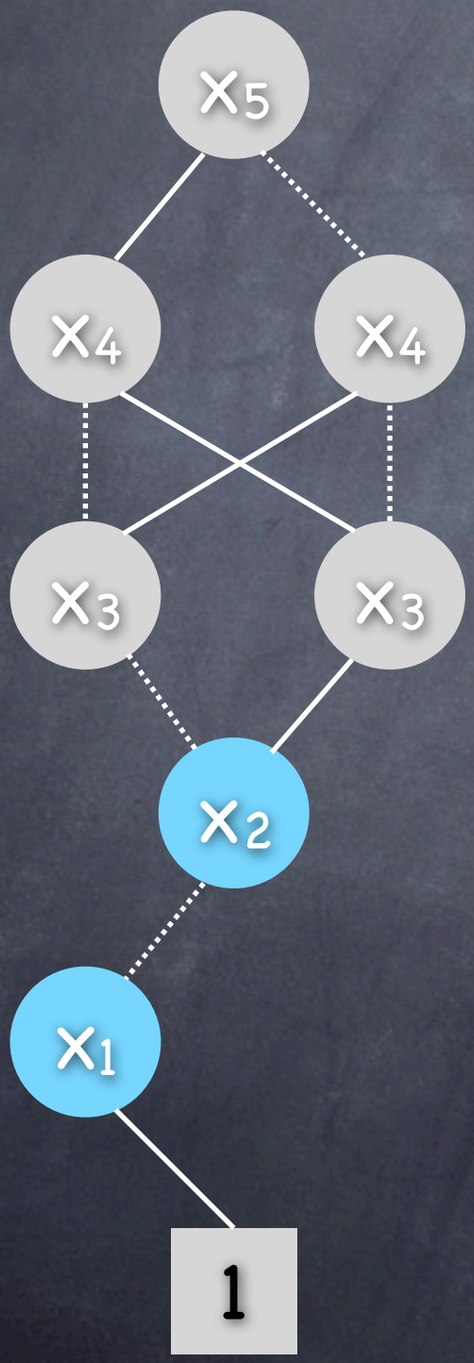
$N_1(X) = N_{e1}(X)$



Identity transformations are skipped

Standard BFS-based Image Computation

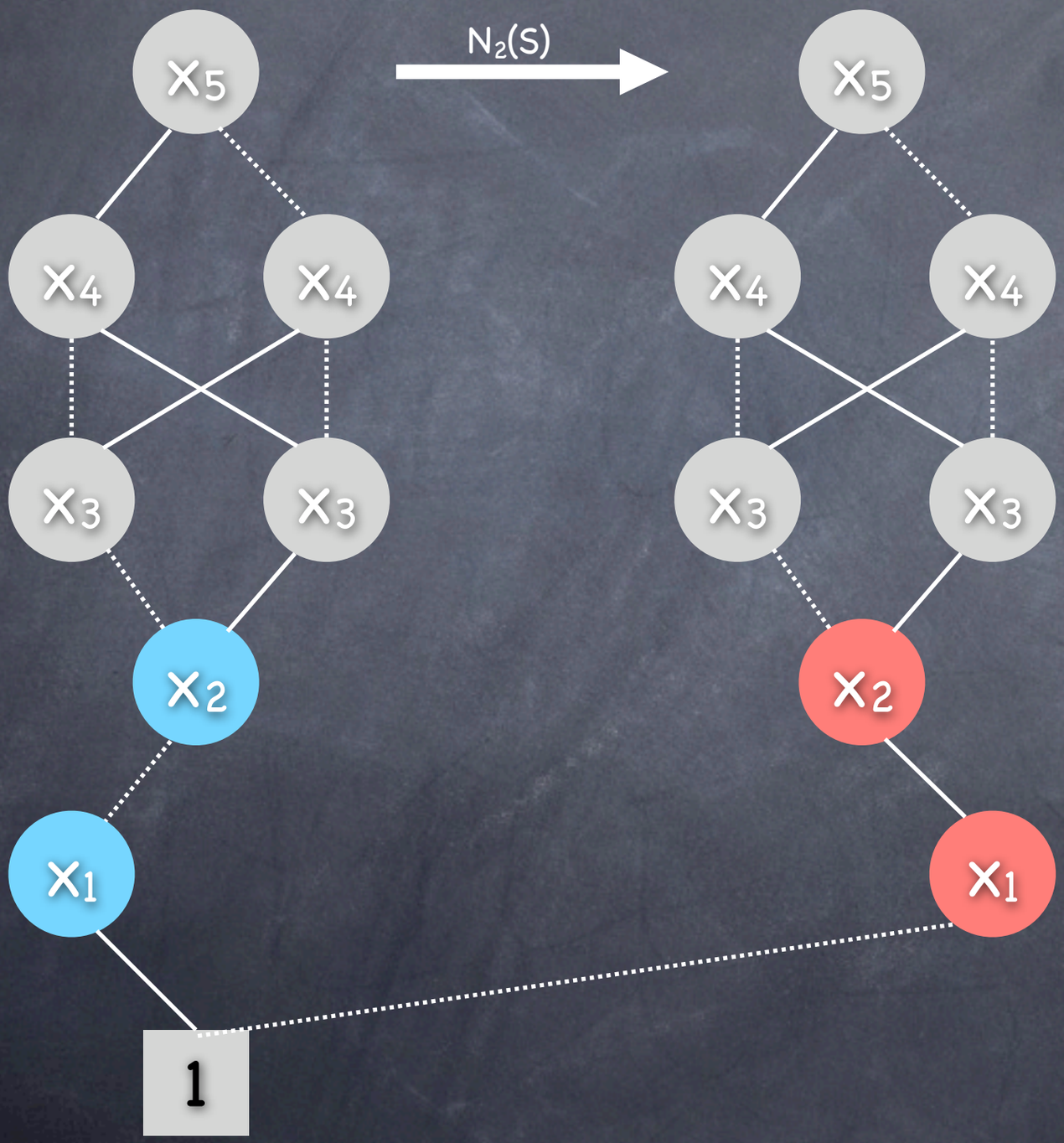
Current states S



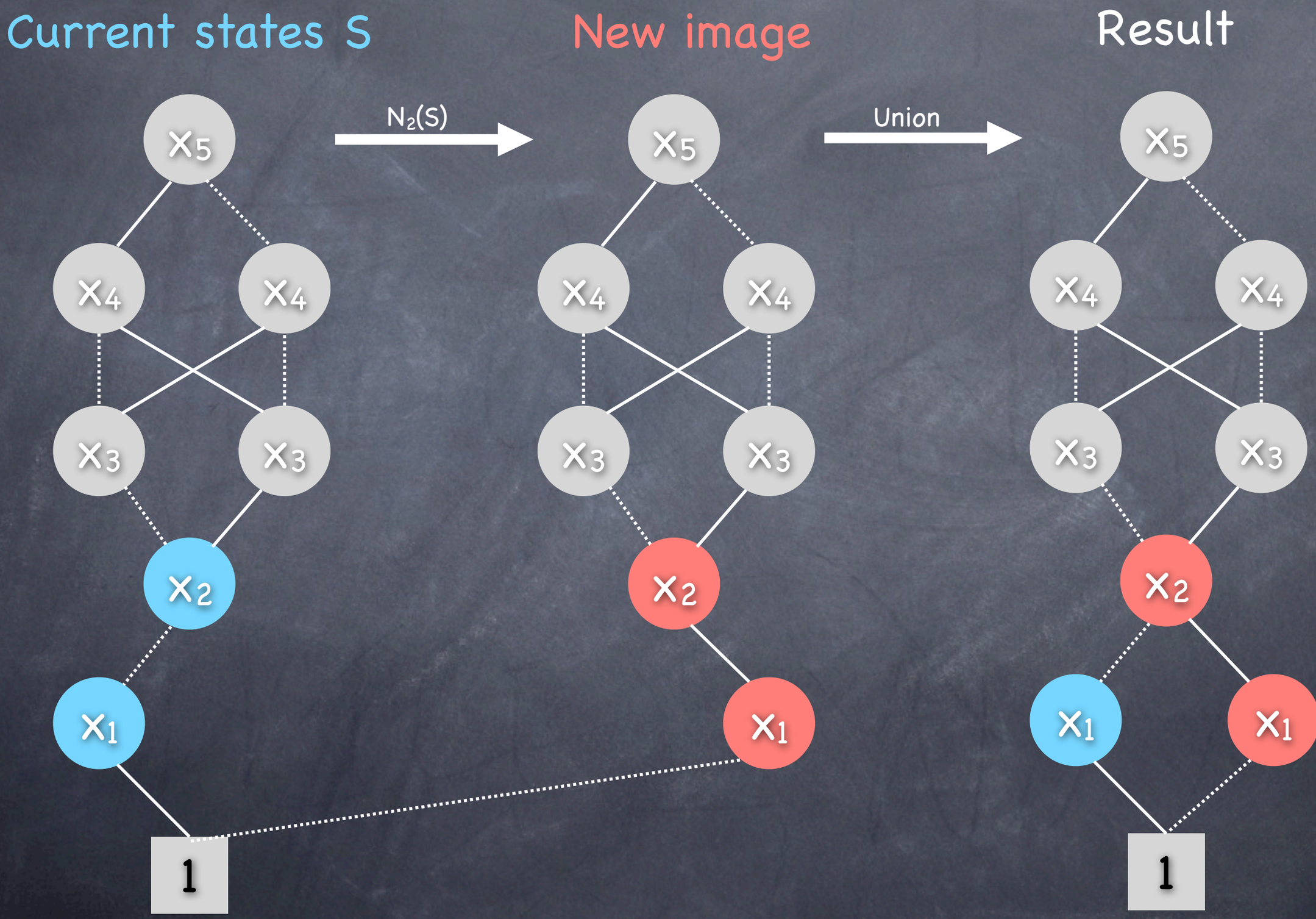
Standard BFS-based Image Computation

Current states S

New image

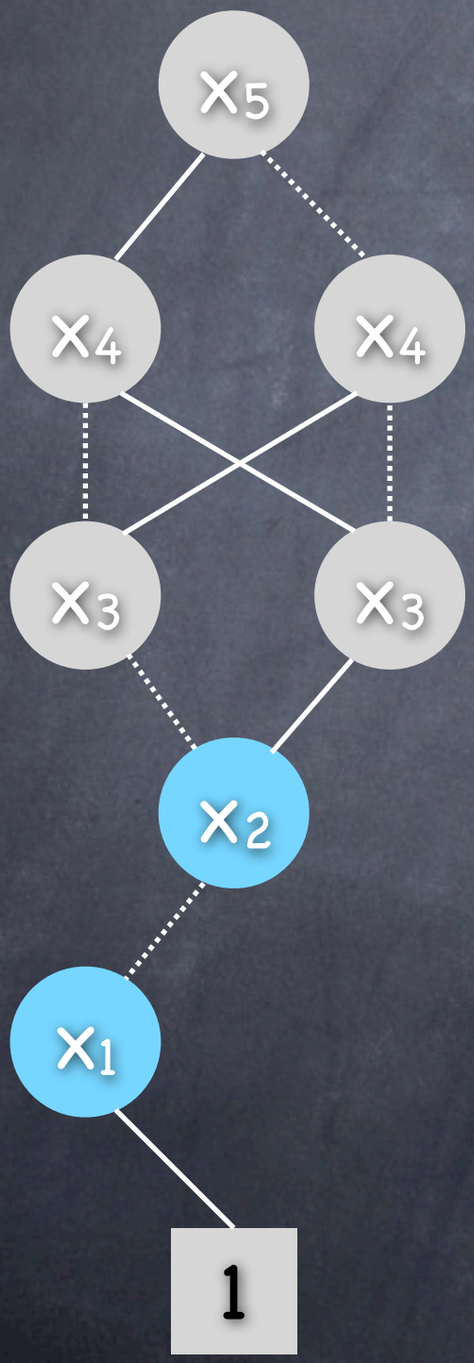


Standard BFS-based Image Computation



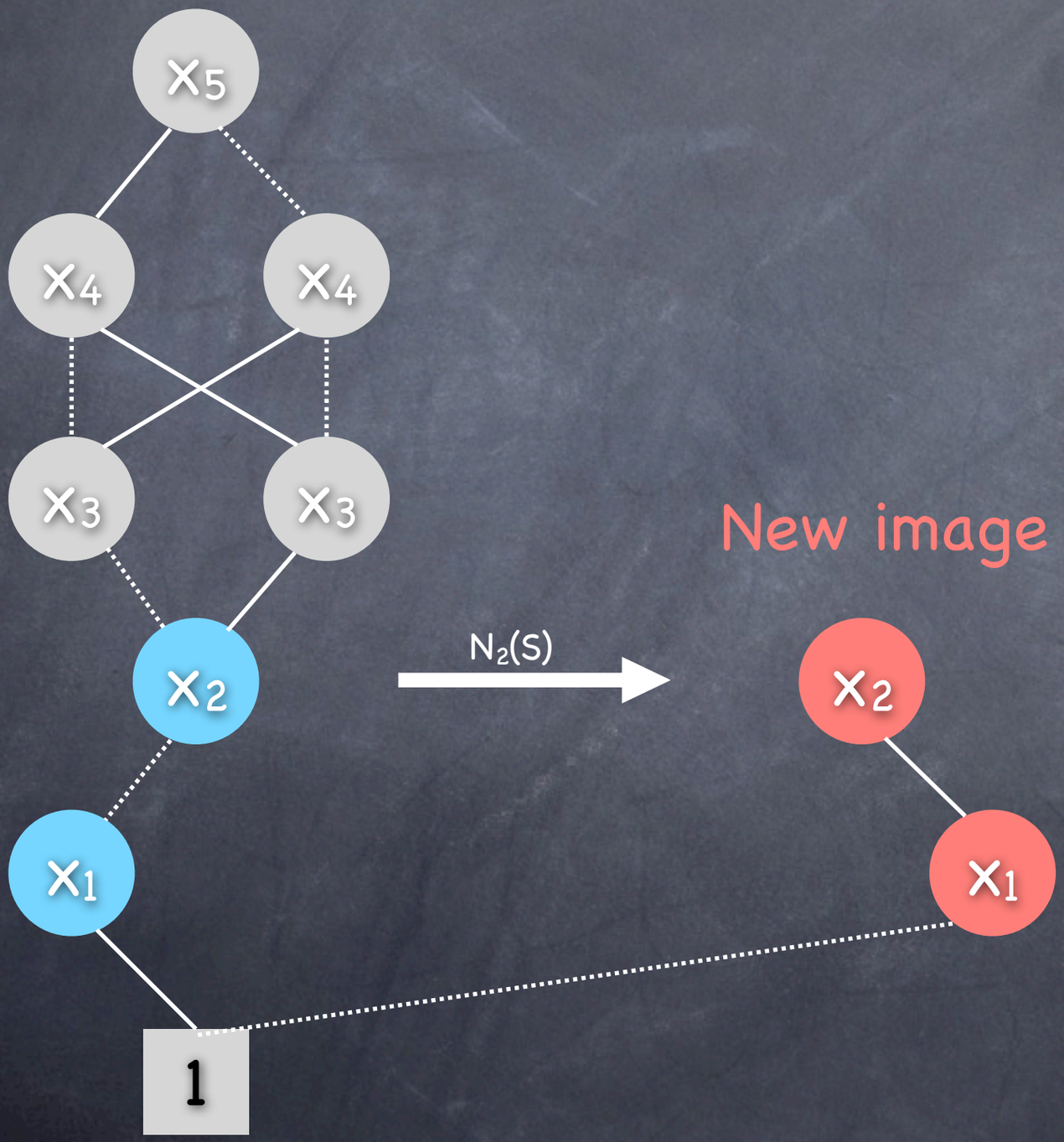
Exploiting Event Locality in Image Computation

Current states S



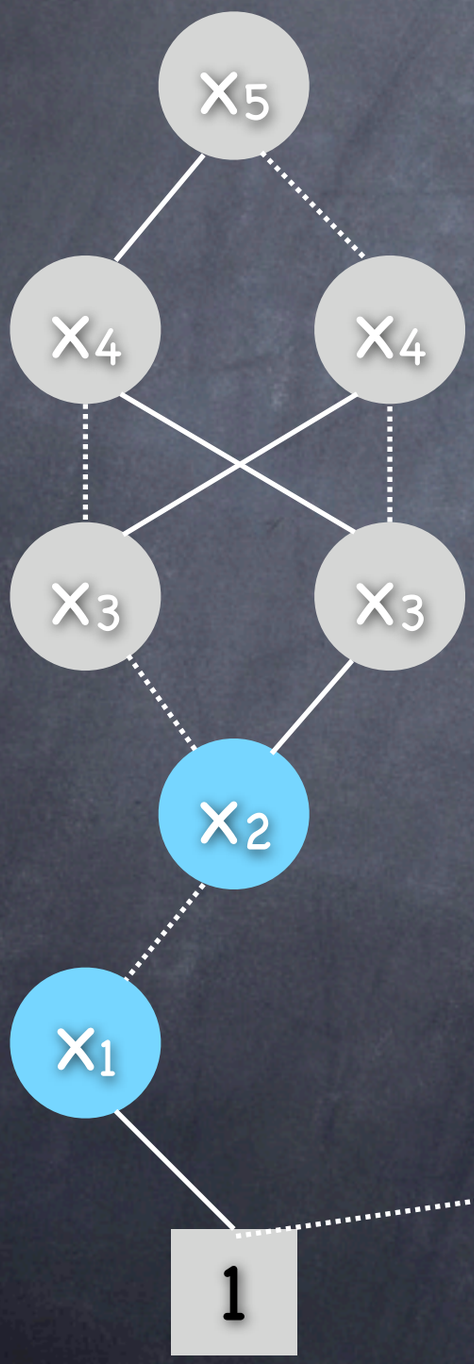
Exploiting Event Locality in Image Computation

Current states S



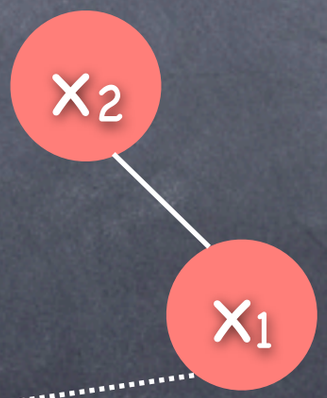
Exploiting Event Locality in Image Computation

Current states S



New image

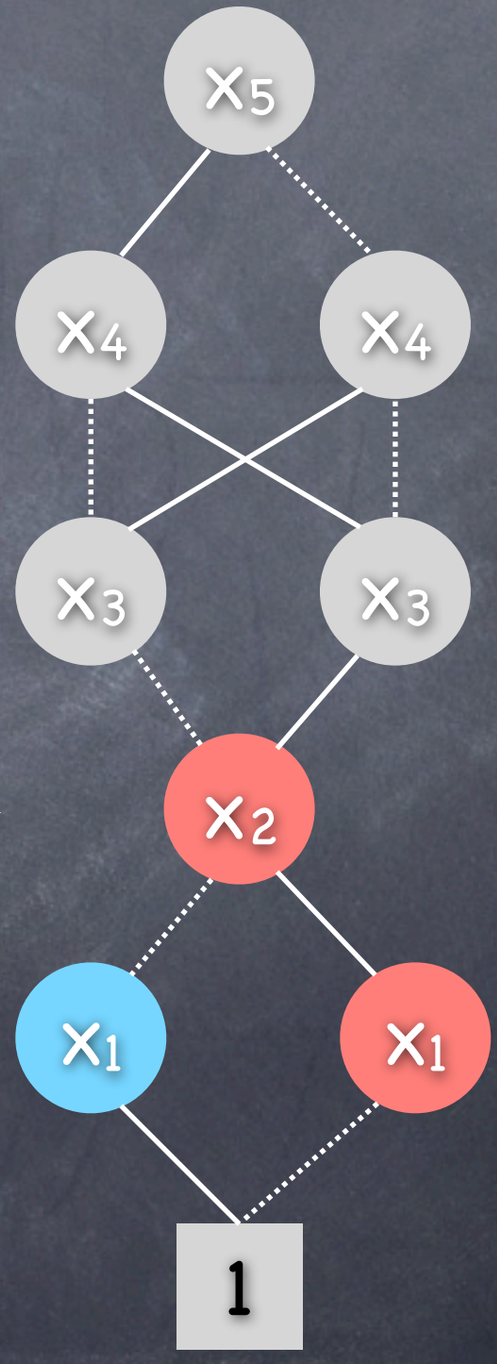
$N_2(S)$



Union +

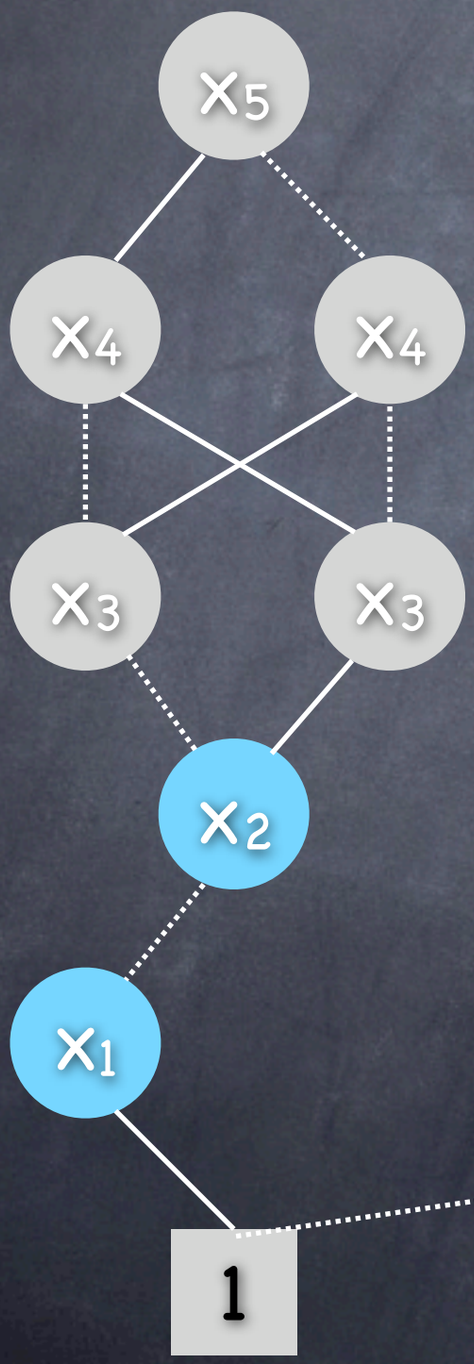
In-place update

Result



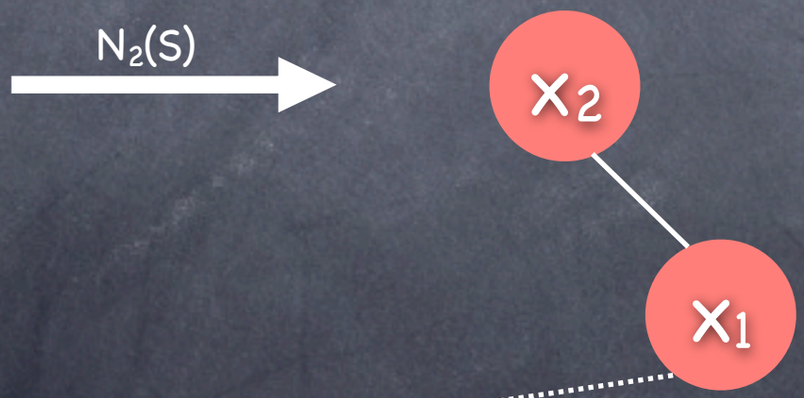
Exploiting Event Locality in Image Computation

Current states S



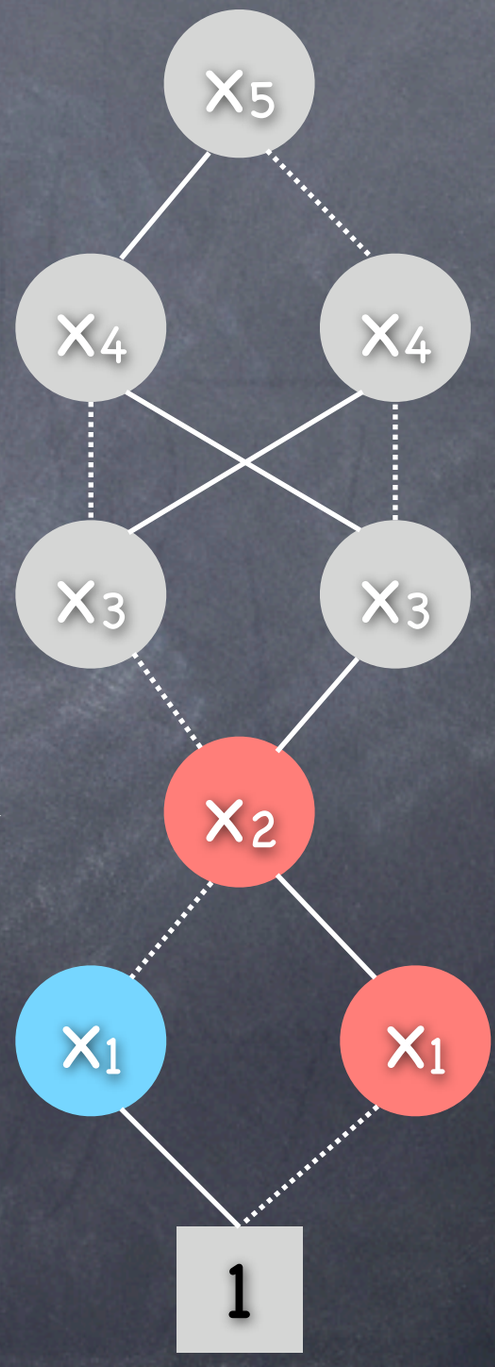
Reduces peak number of nodes by a large factor!

New image



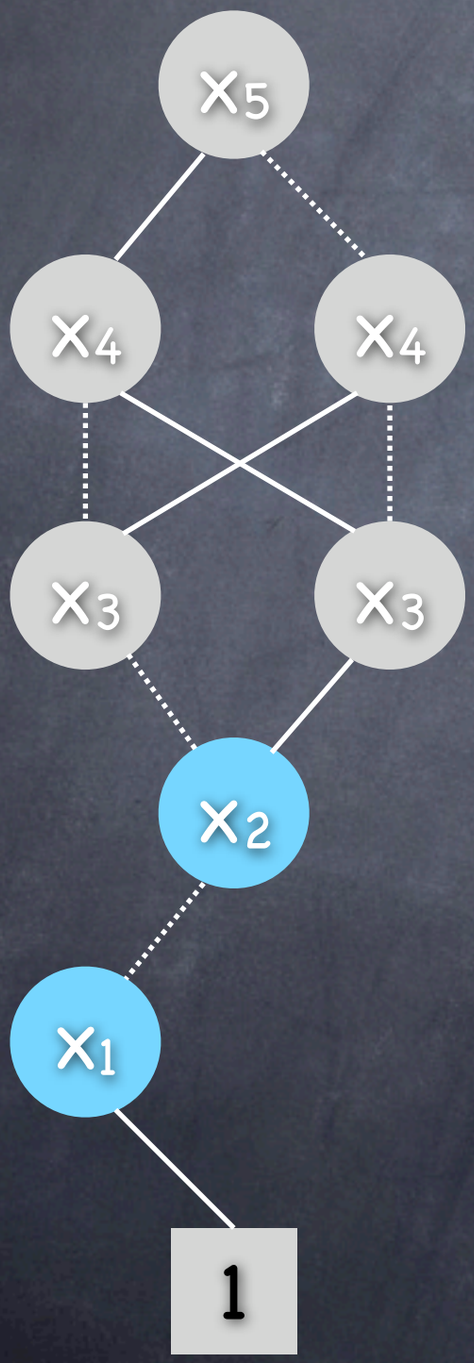
Union +
In-place update

Result



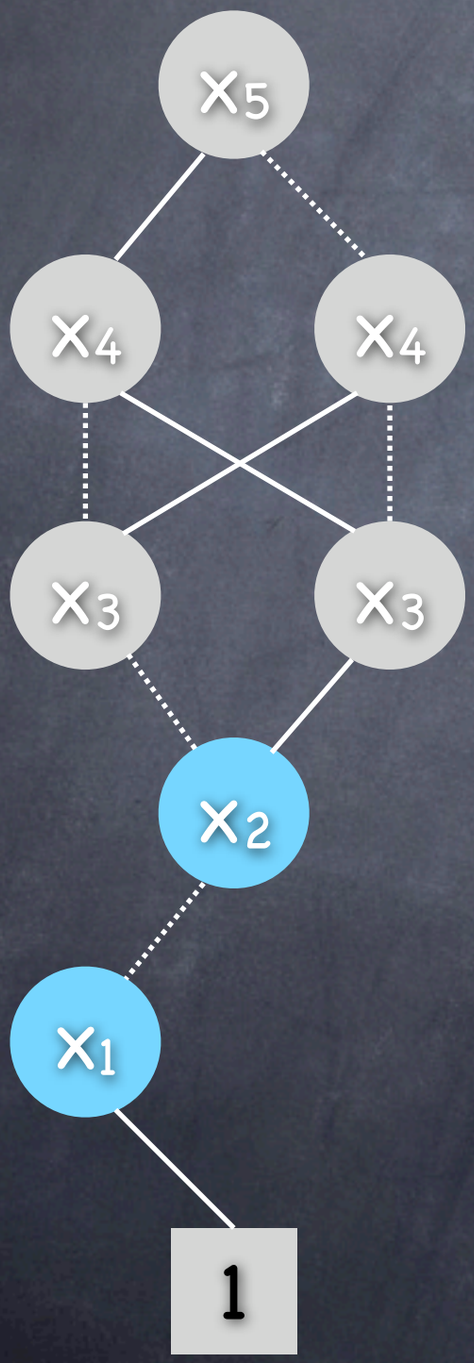
Saturation: Computing Fixed Point at Each Node

Current states



Saturation: Computing Fixed Point at Each Node

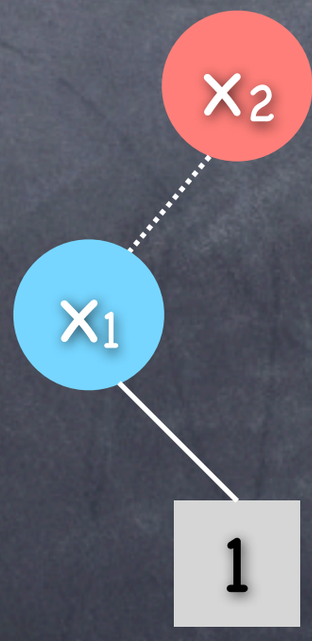
Current states



Fire e_{21} + e_2 repeatedly +

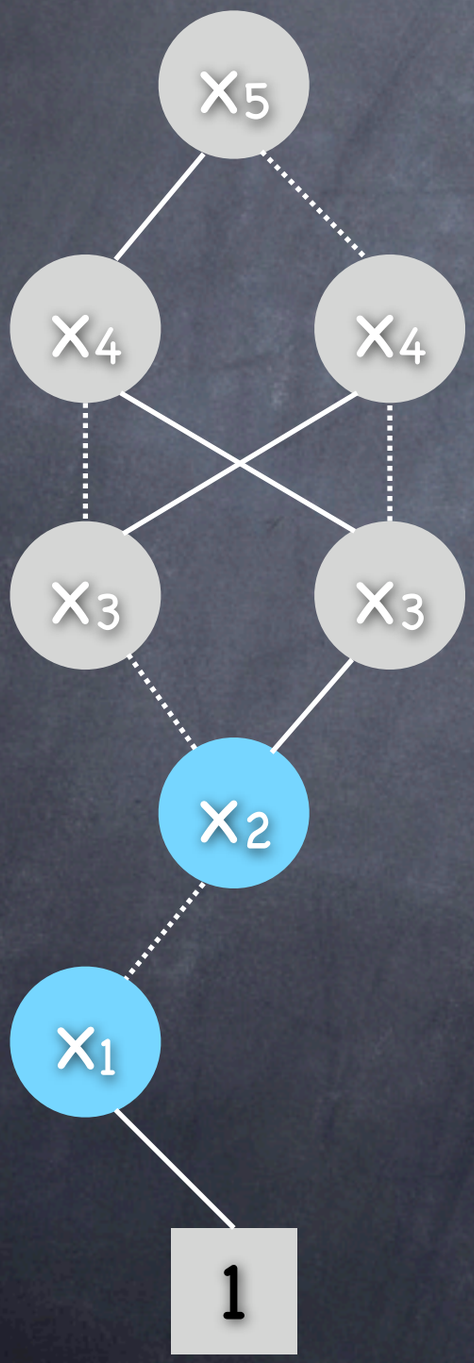
Saturate new node

fixed point e_1



Saturation: Computing Fixed Point at Each Node

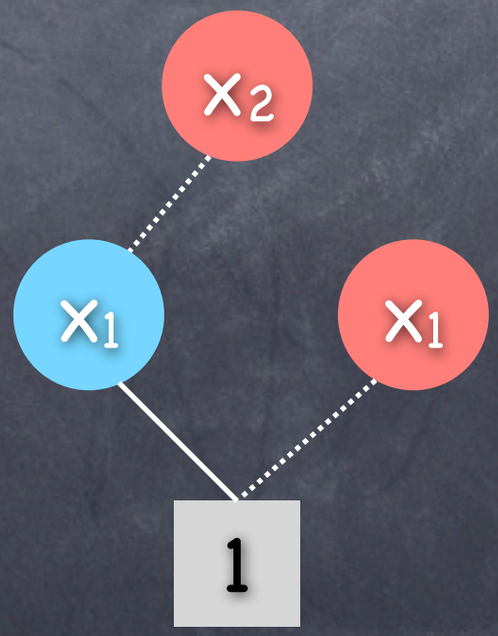
Current states



Fire e_{21} + e_2 repeatedly +

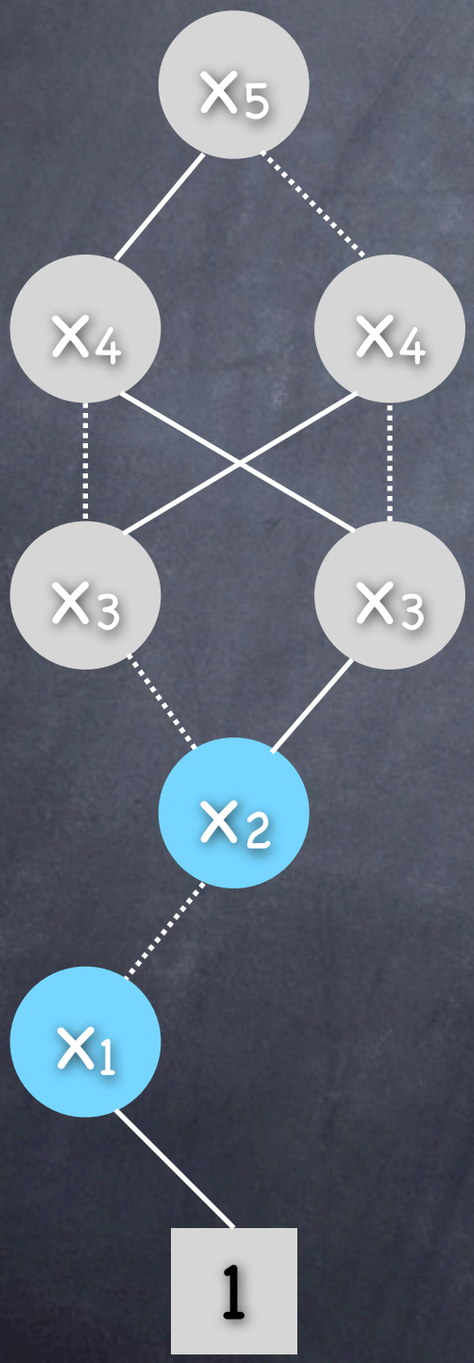
Saturate new node

fixed point e_1



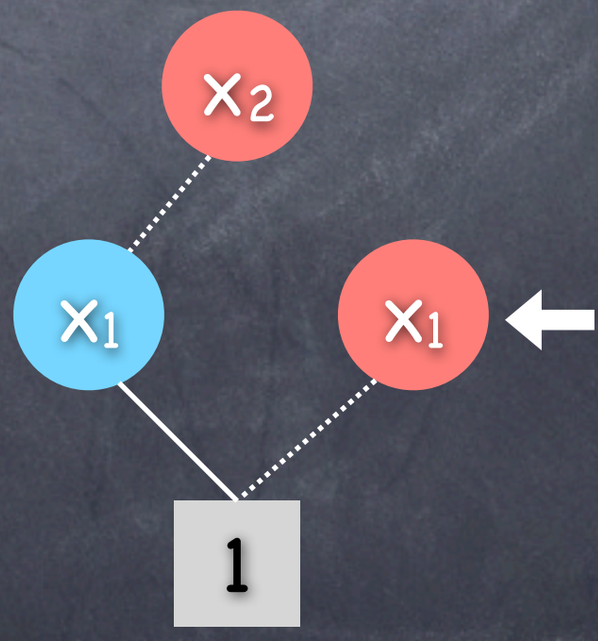
Saturation: Computing Fixed Point at Each Node

Current states



Fire $e_{21} + e_2$ repeatedly +
Saturate new node

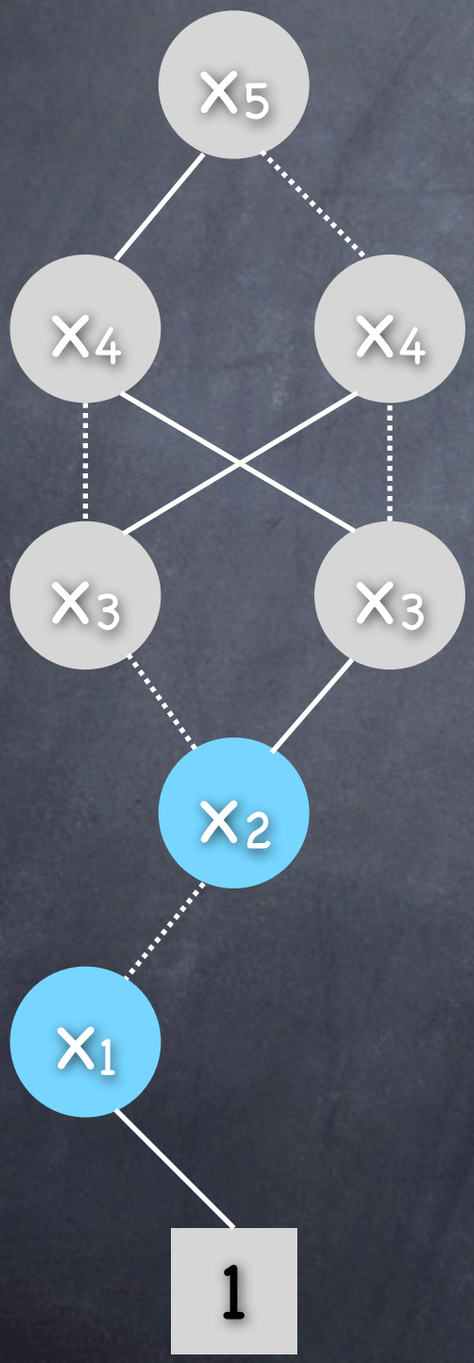
fixed point e_1



fixed point e_1

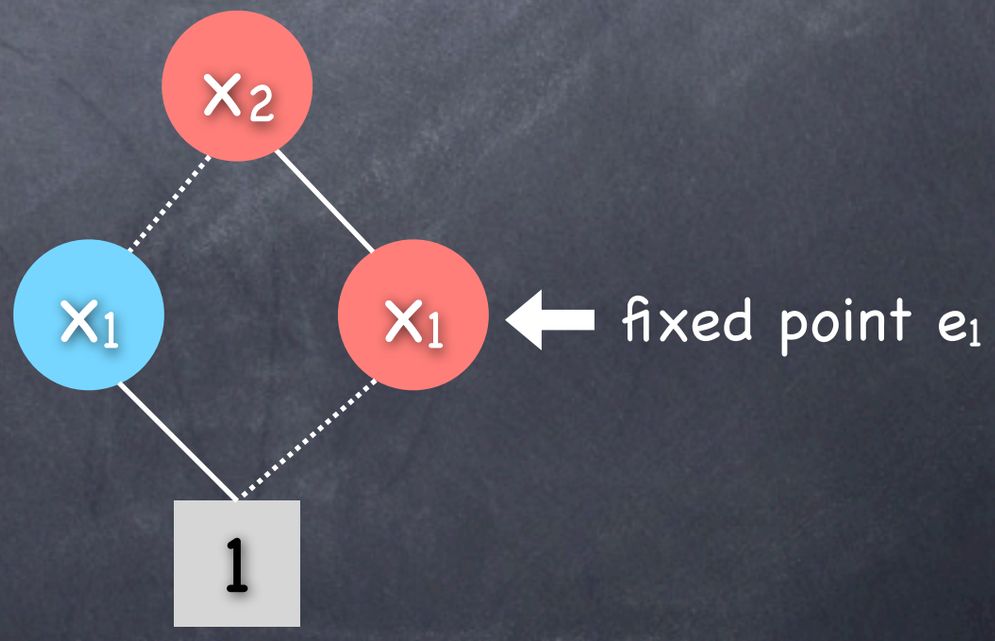
Saturation: Computing Fixed Point at Each Node

Current states



Fire e_{21} + e_2 repeatedly +
Saturate new node

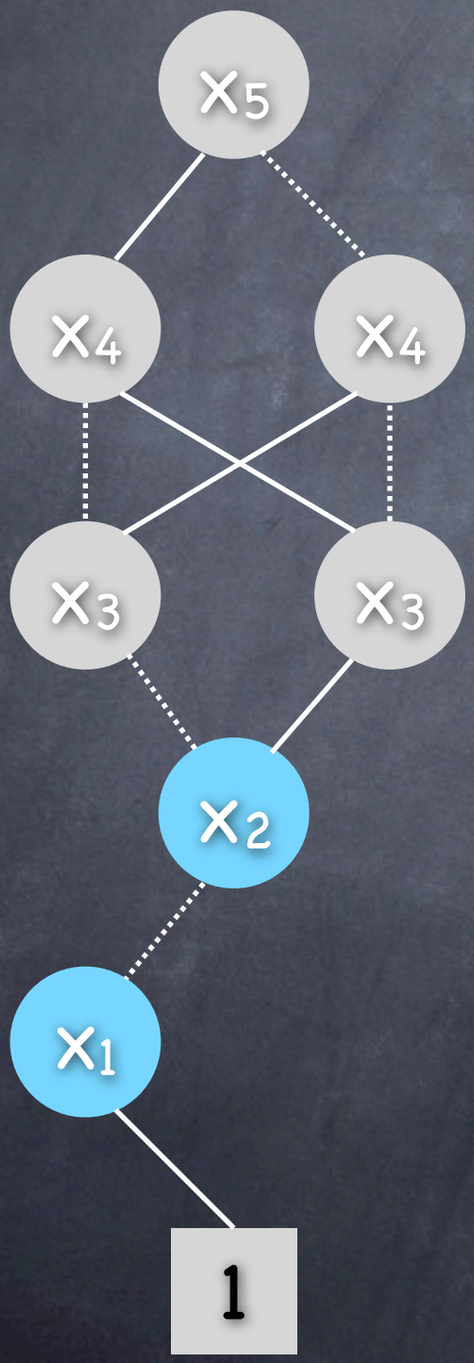
fixed point e_1



fixed point e_1

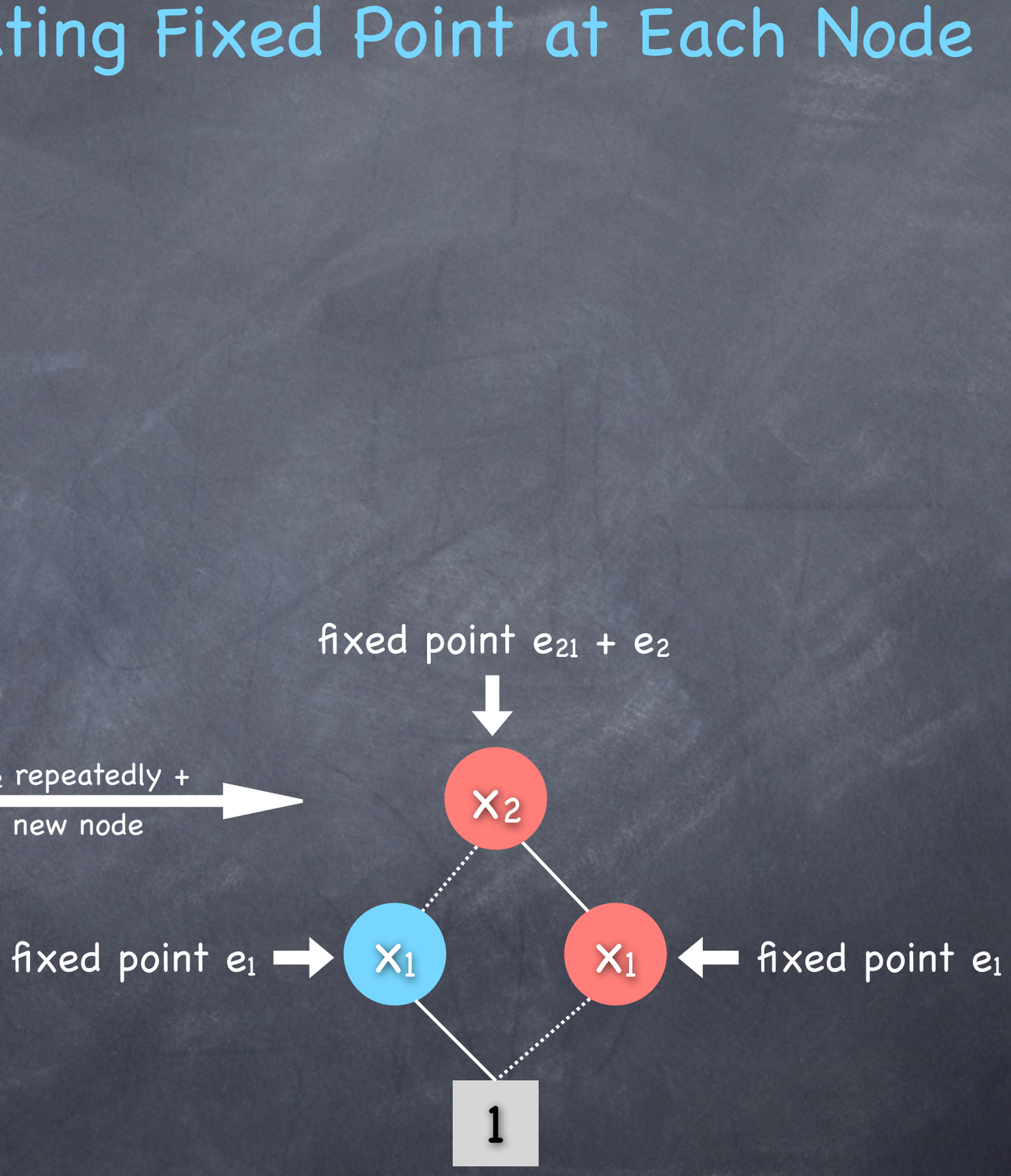
Saturation: Computing Fixed Point at Each Node

Current states



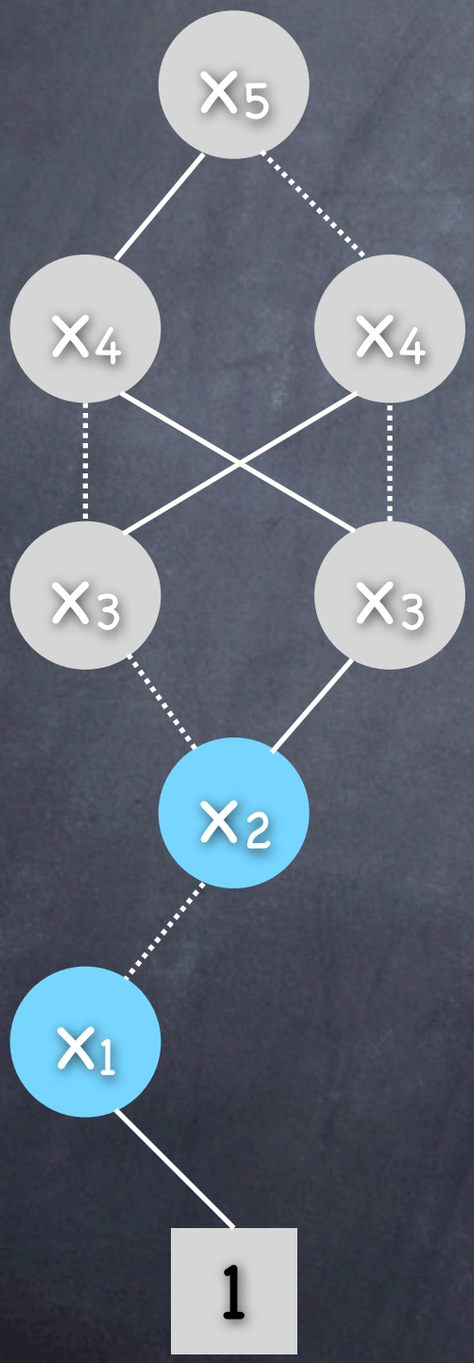
Fire $e_{21} + e_2$ repeatedly +

Saturate new node



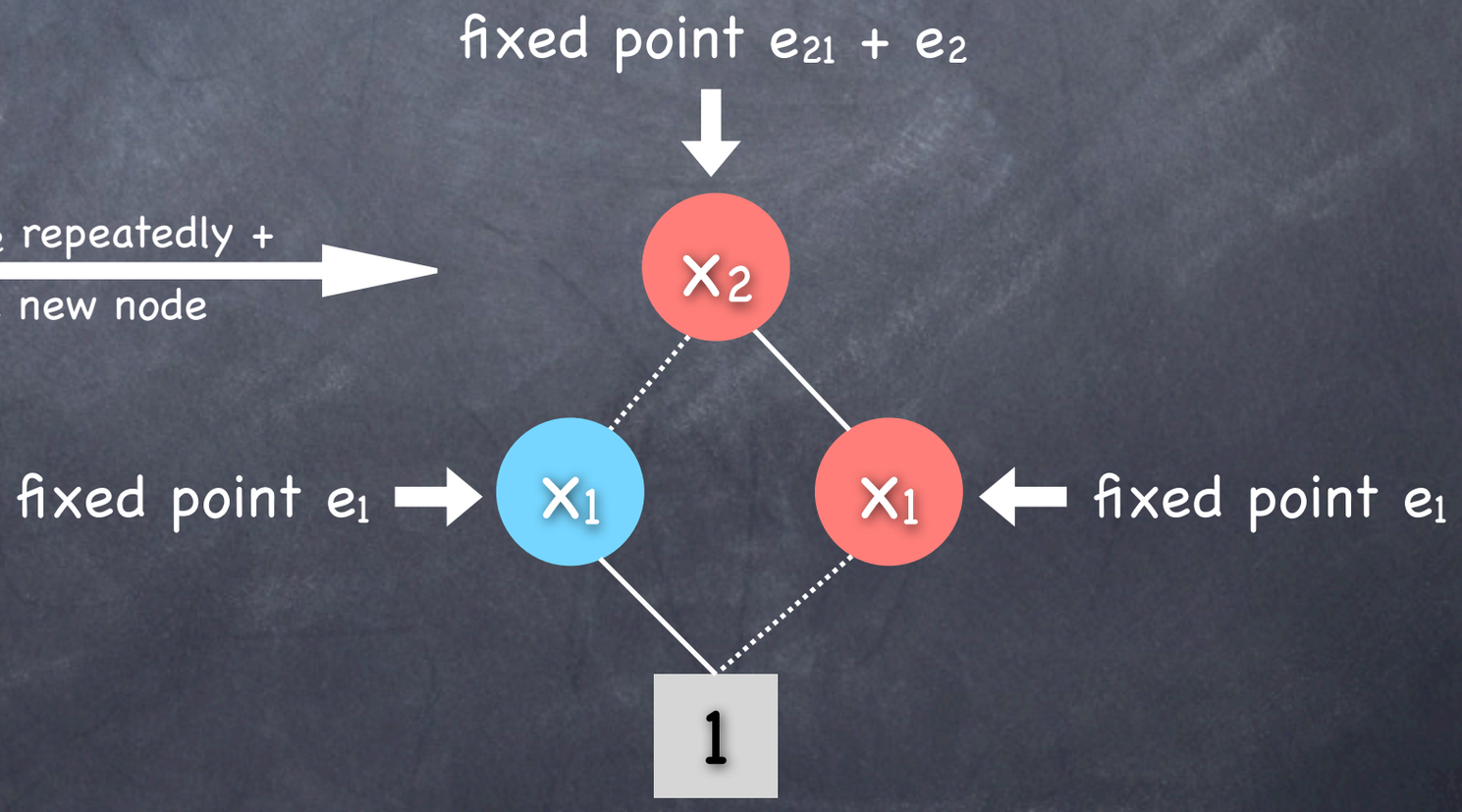
Saturation: Computing Fixed Point at Each Node

Current states



Reduces run-time and memory usage by several orders of magnitude!

Fire $e_{21} + e_2$ repeatedly + Saturate new node



Parallelising Saturation

Parallelising Saturation

- ... for shared-memory machines such as multi-core PCs

Parallelising Saturation

- ... for shared-memory machines such as multi-core PCs
- Idea:
  - When saturating a node, fire several events simultaneously (including the same event enabled in several local states)

Parallelising Saturation

- ... for shared-memory machines such as multi-core PCs
- Idea:
  - When saturating a node, fire several events simultaneously (including the same event enabled in several local states)
- Two implementation strategies:
  - Use a dedicated parallel language, such as Cilk (MIT)
  - Write parallelisation by hand, using Pthreads (POSIX threads)

Parallelising Saturation

- ... for shared-memory machines such as multi-core PCs
- Idea:
  - When saturating a node, fire several events simultaneously (including the same event enabled in several local states)
- Two implementation strategies:
  - Use a dedicated parallel language, such as Cilk (MIT)
  - Write parallelisation by hand, using Pthreads (POSIX threads)
- Questions:
  - Does it work? Is it worthwhile?

Parallelising Saturation on Multi-Core PCs

Parallelising Saturation on Multi-Core PCs

- Analysing Saturation with respect to its characteristics and their implications for parallel overheads ...

Parallelising Saturation on Multi-Core PCs

- Analysing Saturation with respect to its characteristics and their implications for parallel overheads ...

- Symbolic state-space generation is irregular in nature
  - Random access of data structures (hash tables & caches)
    - Hash table for each decision-diagram level, for storing nodes
    - Caches for union and firing operations
  - Unpredictable sizes of work (firing an event on a node)

Parallelising Saturation on Multi-Core PCs

- Analysing Saturation with respect to its characteristics and their implications for parallel overheads ...

- Symbolic state-space generation is irregular in nature
  - Random access of data structures (hash tables & caches)
    - Hash table for each decision-diagram level, for storing nodes
    - Caches for union and firing operations
  - Unpredictable sizes of work (firing an event on a node)

➡ Load balancing becomes a major issue!

Parallelising Saturation on Multi-Core PCs

Parallelising Saturation on Multi-Core PCs

- Saturation uses lightweight operations when firing events
  - Cache hits, hashing ~ 500 ns
  - Local state manipulation via in-place updates ~ 800 ns

Parallelising Saturation on Multi-Core PCs

- Saturation uses lightweight operations when firing events
  - Cache hits, hashing ~ 500 ns
  - Local state manipulation via in-place updates ~ 800 ns

➡ Synchronisation overheads, due to locking on data structures, are non-negligible!

Parallelising Saturation on Multi-Core PCs

- Saturation uses lightweight operations when firing events
  - Cache hits, hashing ~ 500 ns
  - Local state manipulation via in-place updates ~ 800 ns

➡ Synchronisation overheads, due to locking on data structures, are non-negligible!

➡ Scheduling must be done inexpensively!

Parallelising Saturation on Multi-Core PCs

- Saturation uses lightweight operations when firing events
  - Cache hits, hashing ~ 500 ns
  - Local state manipulation via in-place updates ~ 800 ns

➡ Synchronisation overheads, due to locking on data structures, are non-negligible!

➡ Scheduling must be done inexpensively!

- Operating system threads are not an option
  - Thread creation ~ 12,000 ns; thread allocation ~ 3,000 ns

Related Work - A Rough Overview

Related Work - A Rough Overview

- Most research on symbolic state exploration parallelises decision diagrams on PC clusters
  - Two-tiered implementations of hash tables [e.g., Stornetta & Brewer - DAC'96]
  - Partitioning of decision diagrams [e.g., Grumberg et al - CAV'00, CAV'03]

Related Work - A Rough Overview

- Most research on symbolic state exploration parallelises decision diagrams on PC clusters
  - Two-tiered implementations of hash tables [e.g., Stornetta & Brewer - DAC'96]
  - Partitioning of decision diagrams [e.g., Grumberg et al - CAV'00, CAV'03]
- No-one parallelises the underlying algorithms but instead distributes the main data structures!

Related Work - A Rough Overview

- Most research on symbolic state exploration parallelises decision diagrams on PC clusters
  - Two-tiered implementations of hash tables [e.g., Stornetta & Brewer - DAC'96]
  - Partitioning of decision diagrams [e.g., Grumberg et al - CAV'00, CAV'03]
- No-one parallelises the underlying algorithms but instead distributes the main data structures!
- Everyone parallelises "by hand", mostly using MPI or Pthreads; no-one employs a dedicated parallel language!

The Parallel Language Cilk

The Parallel Language Cilk

- Is a language for multi-threaded parallel programming based on ANSI C

The Parallel Language Cilk

- Is a language for multi-threaded parallel programming based on ANSI C
- Has been developed by Charles E. Leiserson and his team at MIT in the 1990's

The Parallel Language Cilk

- Is a language for multi-threaded parallel programming based on ANSI C
- Has been developed by Charles E. Leiserson and his team at MIT in the 1990's
- Is effective for exploiting dynamic, highly asynchronous parallelism on symmetric processors

The Parallel Language Cilk

- Is a language for multi-threaded parallel programming based on ANSI C
- Has been developed by Charles E. Leiserson and his team at MIT in the 1990's
- Is effective for exploiting dynamic, highly asynchronous parallelism on symmetric processors
- Employs a scheduler that allows the performance of programs to be estimated accurately based on abstract complexity measures

The Parallel Language Cilk

- Is a language for multi-threaded parallel programming based on ANSI C
- Has been developed by Charles E. Leiserson and his team at MIT in the 1990's
- Is effective for exploiting dynamic, highly asynchronous parallelism on symmetric processors
- Employs a scheduler that allows the performance of programs to be estimated accurately based on abstract complexity measures
- Is aimed at divide-and-conquer problems rather than producer-consumer problems

Cilk Language Constructs

Cilk Language Constructs

- Specifying a Cilk function by using the keyword cilk in front of a C function

Cilk Language Constructs

- Specifying a Cilk function by using the keyword cilk in front of a C function
- Spawning a Cilk function by using the keyword spawn when calling the function
- Multiple functions can be spawned within a calling function
- Calling function continues to execute in parallel

Cilk Language Constructs

- Specifying a Cilk function by using the keyword cilk in front of a C function
- Spawning a Cilk function by using the keyword spawn when calling the function
  - Multiple functions can be spawned within a calling function
  - Calling function continues to execute in parallel
- Synchronising spawned threads by using the keyword sync, which prevents the calling function to continue until all of its spawned functions have completed
  - Cilk functions contain an implicit sync before they are allowed to return

More Cilk Language Constructs

More Cilk Language Constructs

- Handling return values either left implicit or to an inlet
  - An inlet is a function local to a Cilk function, which handles the result of a spawned function
  - Only one completed Cilk function can be handled at a time by an inlet, i.e., inlets are atomic
  - Inlets may not use further spawn or sync statements, i.e., no expression of pipelining or producer-consumer problems

More Cilk Language Constructs

- Handling return values either left implicit or to an inlet
  - An inlet is a function local to a Cilk function, which handles the result of a spawned function
  - Only one completed Cilk function can be handled at a time by an inlet, i.e., inlets are atomic
  - Inlets may not use further spawn or sync statements, i.e., no expression of pipelining or producer-consumer problems
- Specifying explicit mutex locks is supported in Cilk
  - So we have control over locking our hash tables and caches

Parallel Saturation in Cilk: First Variant

cilk $Saturate$(in k:lvl, p:$node$)

Update p, a node at level k not in the hash table, in–place, to encode $\mathcal{N}^*_{\leq k}(\mathcal{B}(p))$.

declare $pCng : bool$; $e : event$; $i, j : lcl$;
declare $\mathcal{L}$: set of lcl; $u : node$;

1. inlet void $DoUnion(f : lcl)$ {
2. if $f \neq \mathbf{0}$ then
3. foreach $j \in \mathcal{N}_{k,e}(i)$ do
4. $u \Leftarrow Union(k-1, f, p[j])$;
5. if $u \neq p[j]$ then
6. $p[j] \Leftarrow u$; $pCng = true$;
7. if $\mathcal{N}_{k,e}(j) \neq 0$ then
8. $\mathcal{L} = \mathcal{L} \cup \{j\}$;
9. }
10. repeat
11. $pCng \Leftarrow$ false;
12. for each $e \in \mathcal{E}_k$ do
13. $\mathcal{L} = Locals(e,k,p)$;
14. while $\mathcal{L} \neq \emptyset$ do
15. $i = Pick(\mathcal{L})$;
16. $DoUnion$(spawn $Fire(e, k-1, p[i])$);
17. sync;
18. until $pCng = false$;

cilk $Fire$(in e:$event$, l:lvl, q:$node$):$node$

Build an MDD rooted at level l, encoding $\mathcal{N}^*_{\leq l}(\mathcal{N}_e(\mathcal{B}(q)))$.

declare $\mathcal{L}$: set of lcl;
declare $i, j : lcl$;
declare $f, u, s : node$;
declare $sCng : bool$;

1. if $l < Last(e)$ then return q;
2. if $Find(FireCache[l], \{q, e\}, s)$ return s;
3. $s \Leftarrow NewNode(l)$; $sCng \Leftarrow false$;
4. $\mathcal{L} \Leftarrow Locals(e, l, q)$;
5. while $\mathcal{L} \neq \emptyset$ do
6. $i \Leftarrow Pick(\mathcal{L})$;
7. $f \Leftarrow Fire(e, l-1, q[i])$;
8. if $f \neq \mathbf{0}$ then
9. foreach $j \in \mathcal{N}_{l,e}(i)$ do
10. $u \Leftarrow Union(l-1, f, s[j])$;
11. if $u \neq s[j]$ then
12. $s[j] \Leftarrow u$; $sCng \Leftarrow true$;
13. if $sCng$ then $Saturate(l, s)$;
14. $CheckIntoHashTable(l, s)$;
15. $Insert(FireCache[l]\{q, e\}, s)$;
16. return s;

Features of this Algorithm

Features of this Algorithm

- Firing of events on a saturating node are parallelised

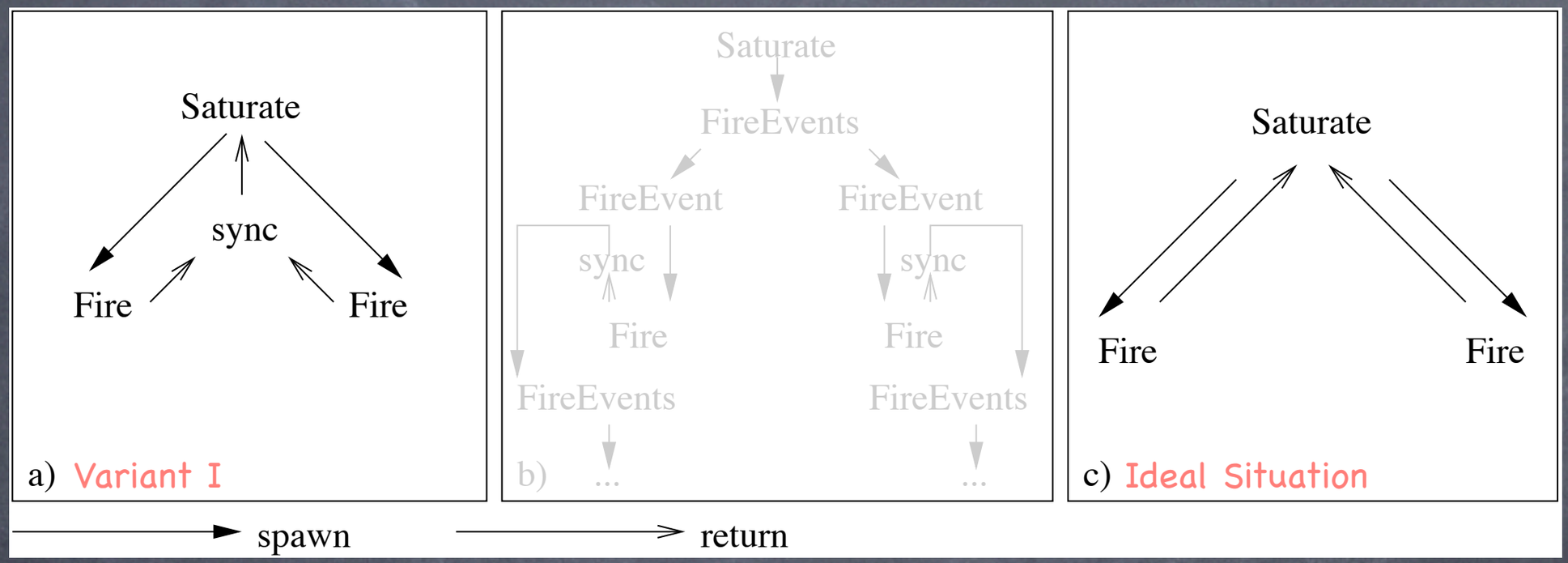
Features of this Algorithm

- Firing of events on a saturating node are parallelised
- Inlet handles computing unions and conducting in-place updates, as soon as a spawned firing returns

Features of this Algorithm

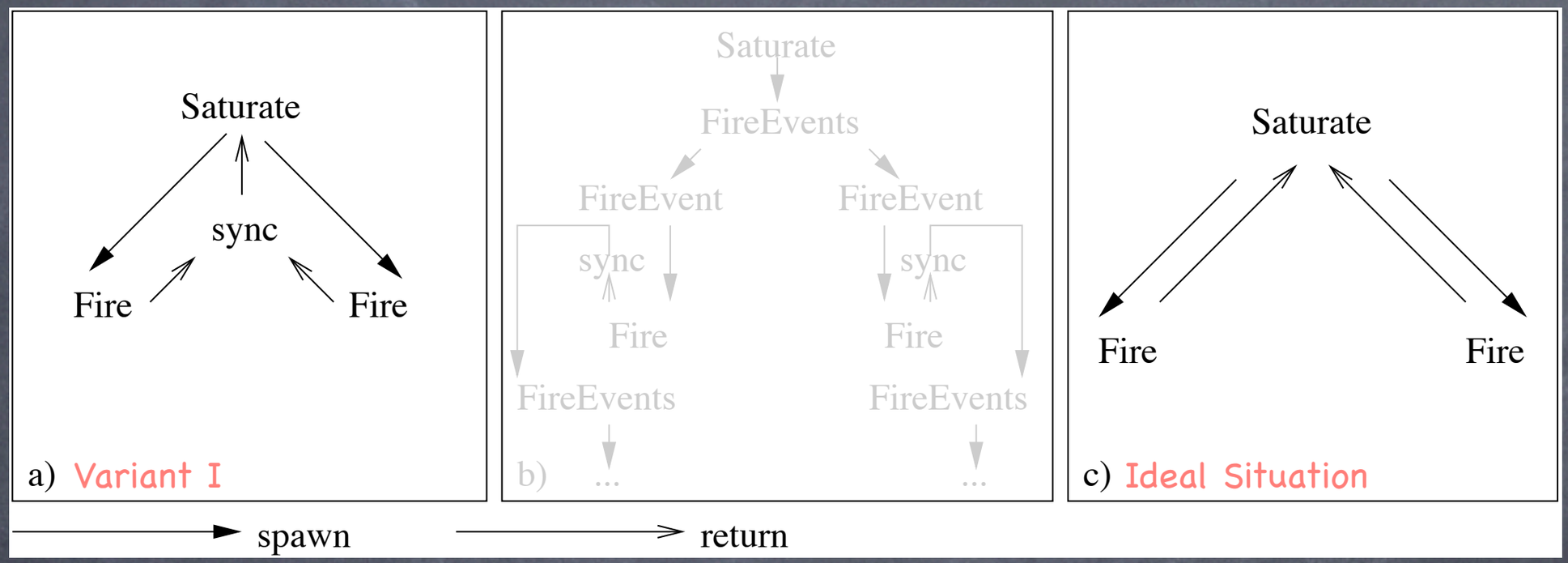
- Firing of events on a saturating node are parallelised
- Inlet handles computing unions and conducting in-place updates, as soon as a spawned firing returns
- Saturate function synchronises on the spawned firings; this is essential for
  - Determining when the firing loop shall terminate
  - Clearing up call stacks and thus controlling memory

Discussion


Saturate
sync
Fire
Fire
a) Variant I
Saturate
FireEvents
FireEvent
FireEvent
sync
sync
Fire
Fire
FireEvents
FireEvents
b) ...
...
Saturate
Fire
Fire
c) Ideal Situation
spawn
return

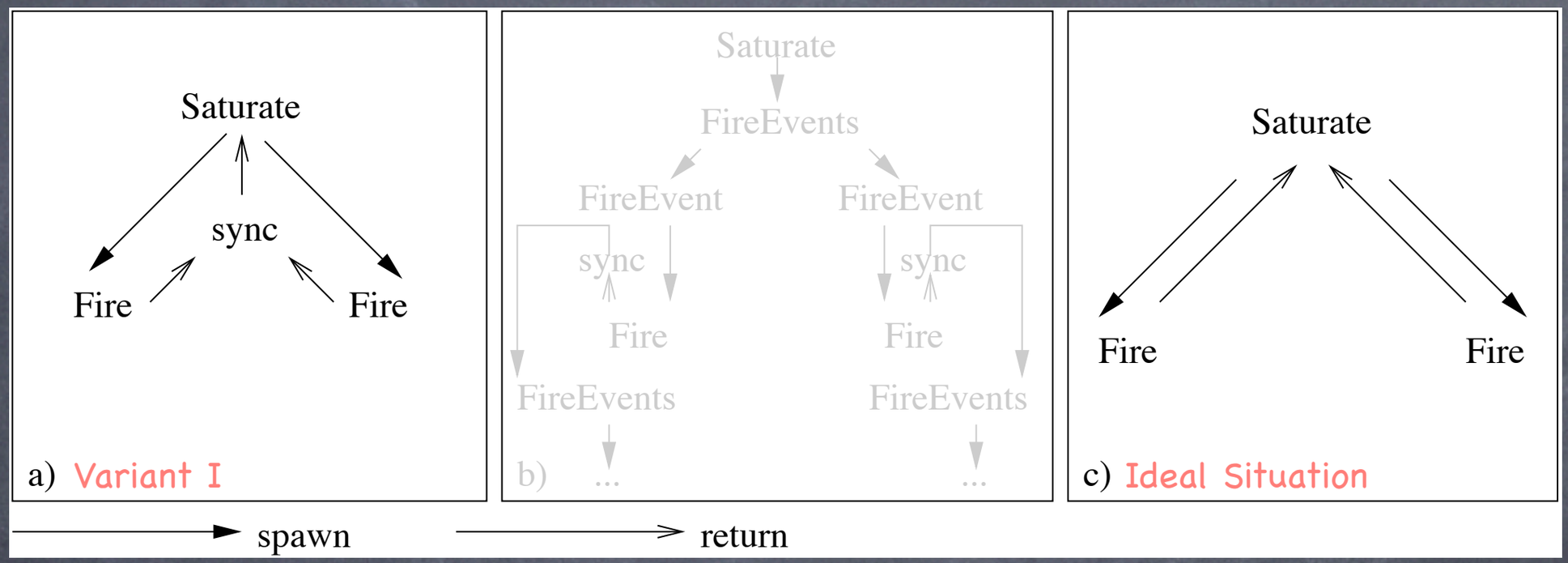
Discussion

- The different sizes of firing calls create load imbalance as parallel firings must synchronise before further firing



Discussion

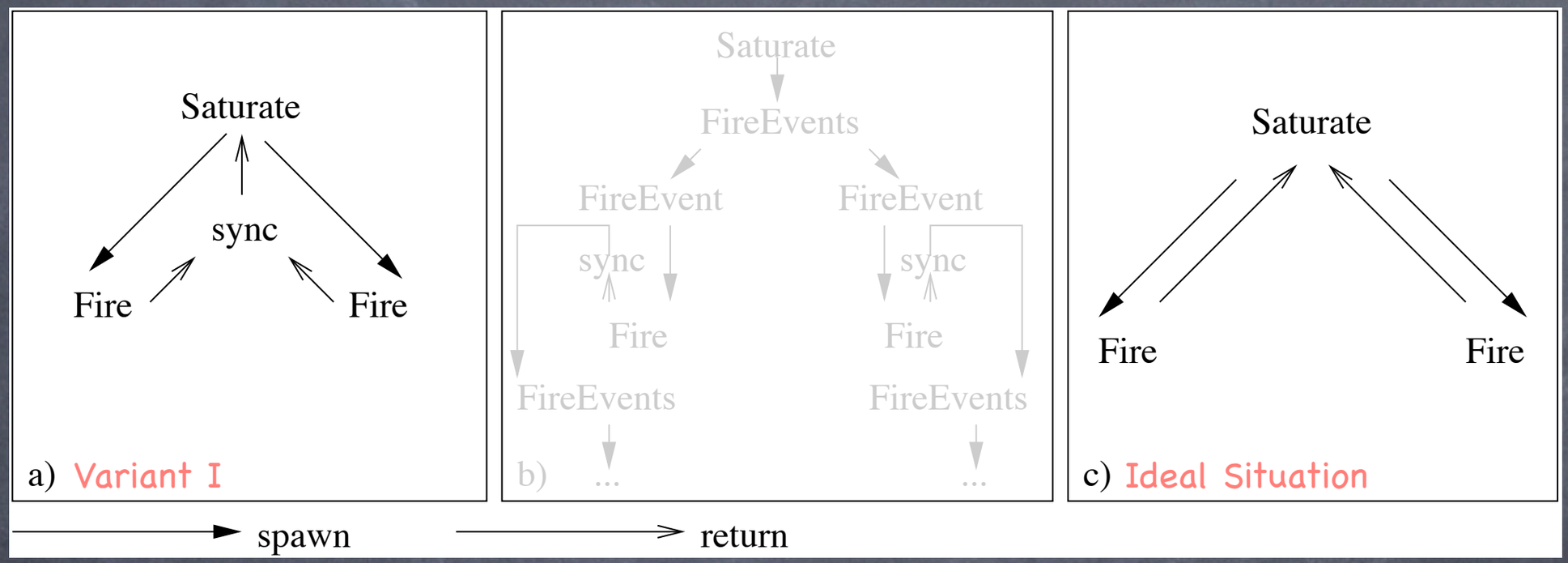
- The different sizes of firing calls create load imbalance as parallel firings must synchronise before further firing



- It would be desirable to spawn more firings after a union, but inlets are not allowed to spawn functions

Discussion

- The different sizes of firing calls create load imbalance as parallel firings must synchronise before further firing



- It would be desirable to spawn more firings after a union, but inlets are not allowed to spawn functions

- Flags can also not be used to signal completed firings, as these would need to be checked within a busy-wait loop

Parallel Saturation in Cilk: Second Variant

cilk $Saturate$(in k:lvl, p:$node$)

Update p, a node at level k not in the hash table, in–place, to encode $\mathcal{N}^*_{\leq k}(\mathcal{B}(p))$.

declare $i : lcl$;

1. foreach $i \in \mathcal{S}_k$ do
2. if $p[i] \neq \mathbf{0}$ then
3. spawn $FireEvents(k, p, i)$;

cilk $FireEvents$(in k:lvl, p:$node$, i:lcl)

Fire e on $p[i]$ when $\mathcal{N}_{k,e}(i) \neq \mathbf{0}$.

declare $e : event$;

1. foreach $e \in \mathcal{E}_k$ do
2. if $\mathcal{N}_{k,e}(i) \neq \mathbf{0}$ then
3. spawn $FireEvent(k, p, i, e)$;

cilk $FireEvent$(in k:lvl, p:$node$, i:lcl, e:$event$)

Fire e on node $p[i]$ at level k.

declare $j : lcl$; $f : node$;

1. $f \Leftarrow Fire(e, k-1, p[i])$;
2. if $f \neq \mathbf{0}$ then
3. foreach $j \in \mathcal{N}_{k,e}(i)$ do
4. spawn $DoUnion(k, p, j, f)$;

cilk $DoUnion$(in k:lvl, p:$node$, j:lcl, f:$node$)

Fire events on $p[j]$ when $p[j]$ changes.

declare $u : node$;

1. $u \Leftarrow Union(k-1, f, p[j])$;
2. if $u \neq p[j]$ then
3. $p[j] \Leftarrow u$; spawn $FireEvents(k, p, j)$;

Features of This Algorithm

Features of This Algorithm

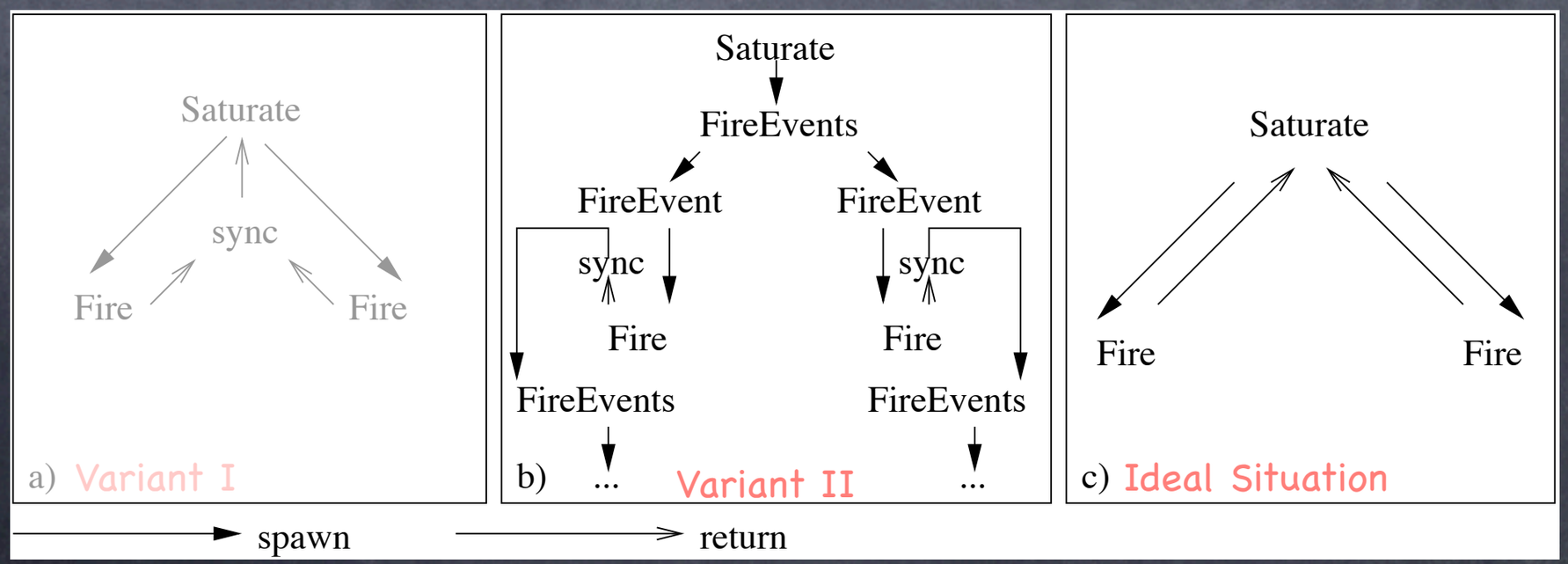
- Continues spawning functions for firing events
  - No explicit sync statements are employed, and all spawn statements occur in the last line of each Cilk function

Features of This Algorithm

- Continues spawning functions for firing events
  - No explicit sync statements are employed, and all spawn statements occur in the last line of each Cilk function
- Breaks the Saturation function into several components
  - Child initiates the continued firing of the saturating node, but many call frames are left unnecessarily on the call stack

Features of This Algorithm

- Continues spawning functions for firing events
  - No explicit sync statements are employed, and all spawn statements occur in the last line of each Cilk function
- Breaks the Saturation function into several components
  - Child initiates the continued firing of the saturating node, but many call frames are left unnecessarily on the call stack



Discussion

Discussion

- The problem is that we must relinquish functions from the call stack while spawned work executes
  - Parts of the call stack frame include necessary information, e.g., the parent-child relationship
  - Parts of the call stack frame include redundant information, e.g., control information

Discussion

- The problem is that we must relinquish functions from the call stack while spawned work executes
  - Parts of the call stack frame include necessary information, e.g., the parent-child relationship
  - Parts of the call stack frame include redundant information, e.g., control information
- All the required information can be stored via an upward arc between a firing child node and a saturating parent node within a decision diagram

Discussion

- The problem is that we must relinquish functions from the call stack while spawned work executes
  - Parts of the call stack frame include necessary information, e.g., the parent-child relationship
  - Parts of the call stack frame include redundant information, e.g., control information
- All the required information can be stored via an upward arc between a firing child node and a saturating parent node within a decision diagram

➡ Hand-written version of Saturation using Pthreads ...

Parallel Saturation by Hand: Threadpool Version

Saturate(in k:*lvl*, p:*node*)

declare i : *lcl*;

1. foreach $i \in \mathcal{S}_k$ do
2. if $p[i] \neq \mathbf{0}$ then
3. $FireEvents(k, p, i)$;
4. if $Tasks(k, p) = 0$ then
5. $NodeSaturated(k, p)$;

FireEvents(in k:*lvl*, p:*node*, i:*lcl*)

declare e : *event*;

1. foreach $e \in \mathcal{E}_k$ do
2. if $\mathcal{N}_{k,e}(i) \neq \mathbf{0}$ then
3. $Fire(e, k, p, p[i], i)$;

Fire(in e:*event*, k:*lvl*, p:*node*, q:*node*, i:*lcl*):*node*

declare s : *node*; j : *lcl*;

...

4. $s{=}CreateNode(k{-}1)$;
5. foreach $j \in \mathcal{N}_{k,e}(i)$ do
6. $AddTask(k, p)$; $SetUpArc(k{-}1, s, j, p)$;

...

14. $AddQueue(Saturate(k{-}1, s))$;

...

NodeSaturated(in k:*lvl*, p:*node*)

declare q : *node*;

1. while $GetUpArc(k, p, i, q)$
2. $DoUnion(k{+}1, q, i, p)$;
3. if $Tasks(k{+}1, q) = 0$ then
4. $NodeSaturated(k{+}1, q)$;

Parallel Saturation by Hand: Threadpool Version

Saturate(in k:*lvl*, p:*node*)

declare i : *lcl*;

1. foreach $i \in \mathcal{S}_k$ do
2. if $p[i] \neq \mathbf{0}$ then
3. $FireEvents(k, p, i)$;
4. if $Tasks(k, p) = 0$ then
5. $NodeSaturated(k, p)$;

FireEvents(in k:*lvl*, p:*node*, i:*lcl*)

declare e : *event*;

1. foreach $e \in \mathcal{E}_k$ do
2. if $\mathcal{N}_{k,e}(i) \neq \mathbf{0}$ then
3. $Fire(e, k, p, p[i], i)$;

Fire(in e:*event*, k:*lvl*, p:*node*, q:*node*, i:*lcl*):*node*

declare s : *node*; j : *lcl*;

...

4. $s{=}CreateNode(k{-}1)$;
5. foreach $j \in \mathcal{N}_{k,e}(i)$ do
6. $AddTask(k, p)$; $SetUpArc(k{-}1, s, j, p)$;

...

14. $AddQueue(Saturate(k{-}1, s))$;

...

NodeSaturated(in k:*lvl*, p:*node*)

declare q : *node*;

1. while $GetUpArc(k, p, i, q)$
2. $DoUnion(k{+}1, q, i, p)$;
3. if $Tasks(k{+}1, q) = 0$ then
4. $NodeSaturated(k{+}1, q)$;

DoUnion spawns further FireEvents, if needed

Features of This Algorithm

Features of This Algorithm

- Employs upward arcs
  - A completed firing function initiates the continuation of firing in the originating node, i.e., it creates the continuing task
  - This implies some work for termination & saturation detection

Features of This Algorithm

- Employs upward arcs
  - A completed firing function initiates the continuation of firing in the originating node, i.e., it creates the continuing task
  - This implies some work for termination & saturation detection
- Uses a threadpool to schedule and load-balance work
  - Implemented in Pthreads with one thread per processor core
  - Pool is implemented as a FIFO queue
  - Idle thread picks work from the head of the queue

Features of This Algorithm

- Employs upward arcs
  - A completed firing function initiates the continuation of firing in the originating node, i.e., it creates the continuing task
  - This implies some work for termination & saturation detection
- Uses a threadpool to schedule and load-balance work
  - Implemented in Pthreads with one thread per processor core
  - Pool is implemented as a FIFO queue
  - Idle thread picks work from the head of the queue

➡ "Pipelined" firings while managing memory and load balancing responsibly

Experimental Studies

Experimental Studies

- Four experimental algorithms
  - Cilk vs. Threadpool, with and without chaining

Experimental Studies

- Four experimental algorithms
  - Cilk vs. Threadpool, with and without chaining
- 10 parameterised models
  - Academic - from queens' problem to dining philosophers
  - Practical - from leader election to manufacturing systems
  - Industrial - NASA/Lockheed Martin's Runway Safety Monitor

Experimental Studies

- Four experimental algorithms
  - Cilk vs. Threadpool, with and without chaining
- 10 parameterised models
  - Academic - from queens' problem to dining philosophers
  - Practical - from leader election to manufacturing systems
  - Industrial - NASA/Lockheed Martin's Runway Safety Monitor
- Implementation in C using Cilk/Pthreads running on
  - Dual-processor, dual-core PC (Xeon 3.06 GHz, 512 KB cache)
  - Redhat Linux AS4, kernel 2.6.9-22.ELsmp, glibc 2.3.4-2.13

Experimental Results: Parallelisability

Experimental Results: Parallelisability

- 7 of the 10 models exhibit parallelism
  - Speedups from just over 1x to superlinear speedups over 4x
  - For parallelisable models, parallelism increases when increasing the model's size

Experimental Results: Parallelisability

- 7 of the 10 models exhibit parallelism
  - Speedups from just over 1x to superlinear speedups over 4x
  - For parallelisable models, parallelism increases when increasing the model's size
- Cilk is superior to our Threadpool in exploiting parallelism
  - Threadpool efficiently parallelises only 3 of the 10 models
  - This is due to Cilk's efficient load balancing and scheduling

Experimental Results: Parallelisability

- 7 of the 10 models exhibit parallelism
  - Speedups from just over 1x to superlinear speedups over 4x
  - For parallelisable models, parallelism increases when increasing the model's size
- Cilk is superior to our Threadpool in exploiting parallelism
  - Threadpool efficiently parallelises only 3 of the 10 models
  - This is due to Cilk's efficient load balancing and scheduling
- The arguably most relevant model, NASA's Runway Safety Monitor, exhibits a superlinear speedup because of the additional positive effect of chaining

Experimental Results: Memory

Experimental Results: Memory

- Threadpool outperforms Cilk regarding memory
  - Memory increase for Threadpool version - roughly 2-3x
  - Memory increase for Cilk version - roughly 10-20x

Experimental Results: Memory

- Threadpool outperforms Cilk regarding memory
  - Memory increase for Threadpool version - roughly 2-3x
  - Memory increase for Cilk version - roughly 10-20x
- The Cilk stack becomes huge!
  - This is the result of parallel Saturation relying on pipelining; but many verification algorithms are producer-consumer problems!
  - Our first variant of Saturation in Cilk, which adheres to the Cilk philosophy, is memory-efficient but hopelessly inefficient regarding run-time

Conclusions: Frustration & Hope

Conclusions: Frustration & Hope

- Many models do not exhibit (much) parallelism, which is a property that cannot be predicted!
  - Exploiting parallelism requires super-efficient techniques at operating-system level and computer-architecture level

Conclusions: Frustration & Hope

- Many models do not exhibit (much) parallelism, which is a property that cannot be predicted!
  - Exploiting parallelism requires super-efficient techniques at operating-system level and computer-architecture level
- Specialised parallel programming languages such as Cilk show much promise
  - Can Cilk be extended to support pipelining and hence our kind of producer-consumer problem?

Conclusions: Frustration & Hope

- Many models do not exhibit (much) parallelism, which is a property that cannot be predicted!
  - Exploiting parallelism requires super-efficient techniques at operating-system level and computer-architecture level
- Specialised parallel programming languages such as Cilk show much promise
  - Can Cilk be extended to support pipelining and hence our kind of producer-consumer problem?

Maybe our time is better spent improving sequential algorithms (cf. BFS→Saturation) rather then parallelising them?!