SHIM

Verification Challenges in the SHIM Concurrent Language

Stephen A. Edwards

Columbia University

Definition

shim \'shim\ *n*

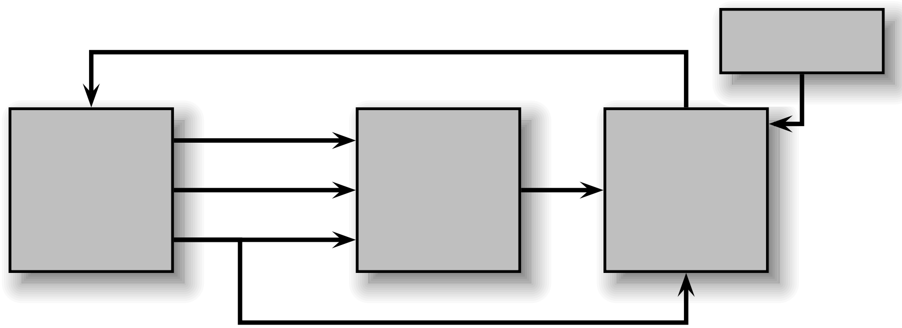
1 : a thin often tapered piece of material (as wood, metal, or stone) used to fill in space between things (as for support, leveling, or adjustment of fit).

2 : *Software/Hardware Integration Medium*, a model for describing hardware/software systems

The SHIM Wishlist

- *Concurrent*
  Hardware always concurrent
- *Mixes synchronous and asynchronous styles*
  Need multi-rate for hardware/software systems
- *Only requires bounded resources*
  Hardware always bounded; simplifies verification
- *Formal semantics*
  Clarifies verification problem; want to avoid arguments
- *Scheduling-independent*
  Renders simulated behavior informative and definitive
  Solves some verification problems "by theorem"
  Verify functionality and performance separately

The SHIM Model



- Sequential processes
- Unbuffered one-to-many communication channels exchange data tokens
- Dynamic topology with an easily-defined static subset
- Asynchronous process execution rates
- Blocking synchronous rendezvous communication
- Scheduling-independent: sequence of data through any channel independent of scheduling policy (the Kahn principle)
- "Kahn networks with rendezvous communication"

The SHIM Language

An imperative language with familiar C/Java-like syntax

```
int32 gcd(int32 a, int32 b)
{
  while (a != b) {
    if (a > b)
      a -= b;
    else
      b -= a;
  }
  return a;
}
```

```
struct foo {        // Composite types
  int x;
  bool y;
  uint15 z;         // Explicit-width integers
  int<-3,5> w;      // Explicit-range integers
  int8 p[10];       // Arrays
  bar q;            // Recursive types
};
```

Three Additional Constructs

$stmt_1$ `par` $stmt_2$	Run $stmt_1$ and $stmt_2$ concurrently
`send` *var*	Communicate on channel *var*
`recv` *var*	
`next` *var*	
`try`	Define the scope of an exception
⋮	
`throw` *exc*	Raise an exception
⋮	
`catch(` *exc* `)` *stmt*	

Concurrency & *par*

Par statements run concurrently and asynchronously

Terminate when all terminate

Each thread gets private copies of variables; no sharing

Writing thread sets the variable's final value

```
void main() {
  int a = 3, b = 7, c = 1;
  {
    a = a + c;          // a ← 4, b = 7, c = 1
    a = a + b;          // a ← 11, b = 7, c = 1
  } par {
    b = b - c;          // a = 3, b ← 6, c = 1
    b = b + a;          // a = 3, b ← 9, c = 1
  }
                        // a ← 11, b ← 9, c = 1
}
```

Restrictions

Both pass-by-reference and pass-by-value arguments
Simple syntactic rules avoid races

```
void f(int &x) { x = 1; }    // x passed by reference
void g(int  x) { x = 2; }    // x passed by value

void main() {
  int a = 0, b = 0;

  a = 1; par b = a;   // OK: a and b modified separately
  a = 1; par a = 2;   // Error: a modified by both

  f(a);  par f(b);    // OK: a and b modified separately
  f(a);  par g(a);    // OK: a modified by f only
  g(a);  par g(a);    // OK: a not modified
  f(a);  par f(a);    // Error: a passed by reference twice
}
```

Communication

Blocking: wait for all processes connected to *a*

```
void f(chan int a) {  // a is a copy of c
  a = 3;              // change local copy
  recv a;             // receive (wait for g)
                      // a now 5
}

void g(chan int &b) { // b is an alias of c
  next b = 5;         // sets c and send (wait for f)
                      // b now 5
}

void main() {
  chan int c = 0;
  f(c); par g(c);
}
```

Synchronization, Deadlocks

Blocking communication makes for potential deadlock

```
{ next a; next b; } par { next b; next a; }  // deadlocks
```

Only threads responsible for a variable must synchronize

```
{ next a; next b; } par next b; par next a;  // OK
```

When a thread terminates, it is no longer responsible

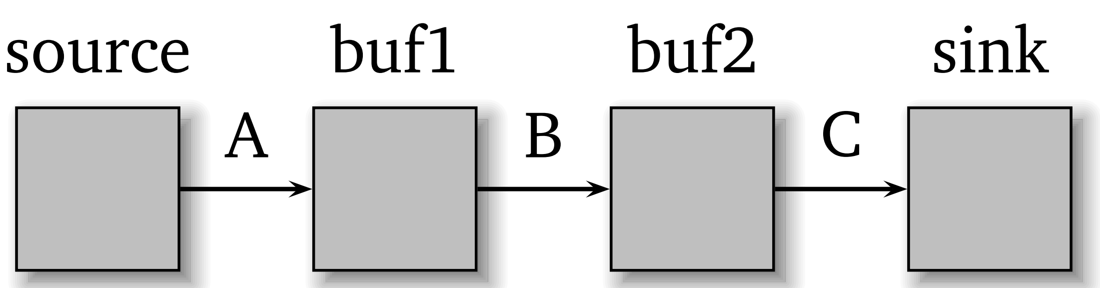
```
{ next a; next a; } par next a;  // OK
```

Philosophy: deadlocks easy to detect; races are too subtle

SHIM prefers deadlocks to races (always reproducible)

An Example

```
void main() {
  chan uint8 A, B, C;
  {                     // source: generate four values
    next A = 17;
    next A = 42;
    next A = 157;
    next A = 8;
  } par {               // buf1: copy from input to output
    for (;;)
      next B = next A;
  } par {               // buf2: copy, add 1 alternately
    for (;;) {
      next C = next B;
      next C = next B + 1;
    }
  } par {               // sink
    for (;;)
      recv C;
  }
}
```

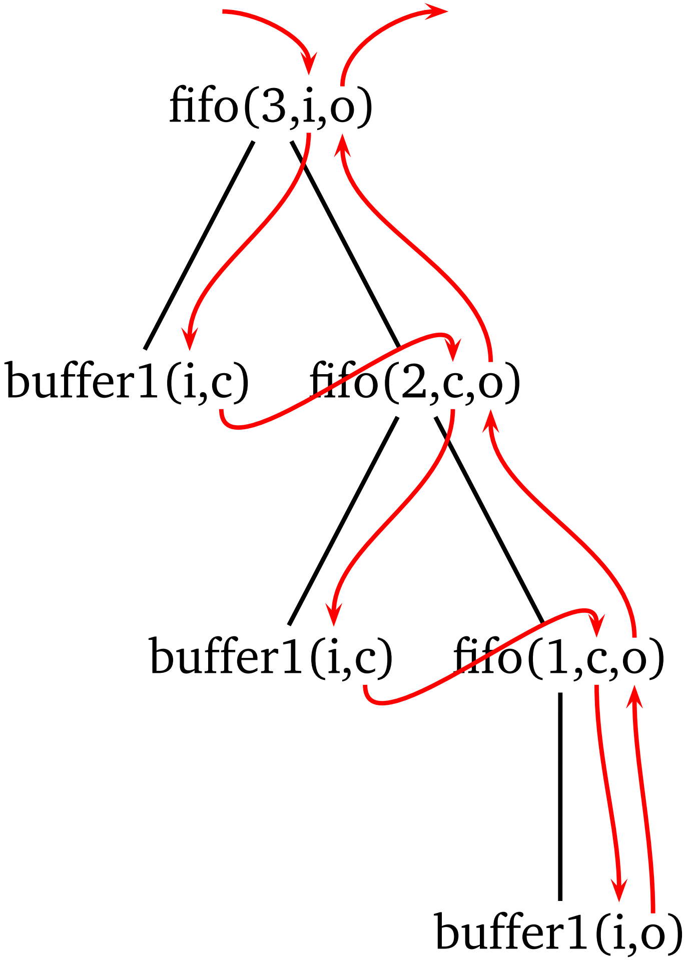


Recursion & Concurrency

A bounded FIFO: compiler analyzes & expands

```
void buffer1(chan int in, chan int &out) {
  for (;;) next out = next in;
}

void fifo(int n, chan int in,
                 chan int &out) {
  if (n == 1)
    buffer1(in, out);
  else {
    chan int channel;
      buffer1(in, channel);
    par
      fifo(n-1, channel, out);
  }
}
```



Exceptions

Sequential semantics are classical

```
void main() {
  int i = 1;
  try {
    throw T;
    i = i * 2;              // Not executed
  } catch (T) {
     i = i * 3;             // Executed by throw T
  }
                            // i = 3 on exit
}
```

Exceptions & Concurrency

```
void main() {
  chan int i = 0, j = 0;
  try {
    while (i < 5)
      next i = i + 1;
    throw T;
  } par {
    for (;;) {
      next j =
        next i + 1;
    }
  } par {
    for (;;)
      recv j;
  } catch (T) {}
}
```

Exceptions propagate through communication actions to preserve determinism

Idea: “transitive poisoning”

Raising an exception “poisons” a process

Any process attempting to communicate with a poisoned process is itself poisoned (within exception scope)

“Best effort preemption”

SHIM Verification Challenges

- *Can a particular program deadlock?*
  General answer is data-dependent
  Many systems exhibit regular patterns

SHIM Verification Challenges

- *Can a particular program deadlock?*
  General answer is data-dependent
  Many systems exhibit regular patterns

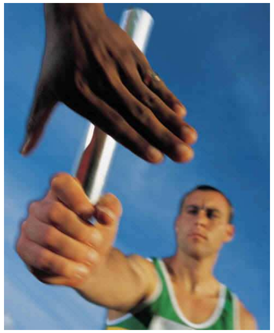
- *Can a program achieve a particular performance?*
  Related to worst-case execution time analysis
  Complicated by communication behaviors
  Precise answer depends on particular implementation

SHIM Verification Challenges

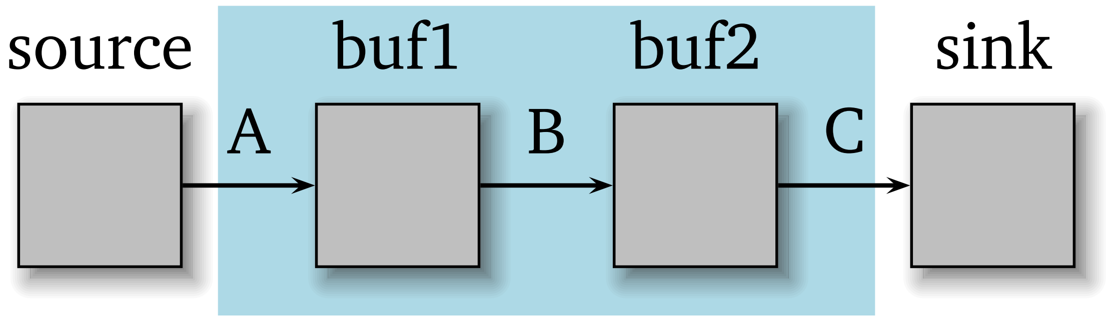
- *Can a particular program deadlock?*
  General answer is data-dependent
  Many systems exhibit regular patterns

- *Can a program achieve a particular performance?*
  Related to worst-case execution time analysis
  Complicated by communication behaviors
  Precise answer depends on particular implementation

- *Does a translation faithfully implement SHIM semantics?*
  Pthreads implementation nondeterministic
  Many opportunities for inadvertant races



A Partial Evaluation Approach



Build an automaton through abstract simulation

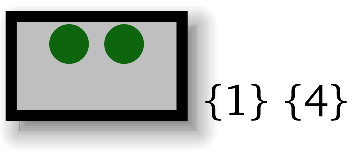
State signature:

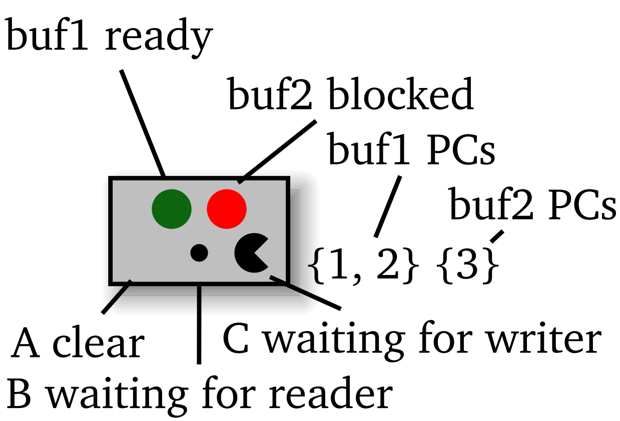
- Running/blocked status of each process
- Blocked on reading/writing status of each channel

Trick: does not include control or data state of each process

Abstract Simulation

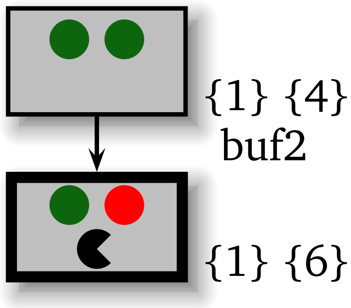
```
{                              // buf1
   1for (;;)
      2next B = 3next A;
} par {                        // buf2
   4for (;;) {
      5next C = 6next B;
      7next C = 8next B + 1;
   }
}
```

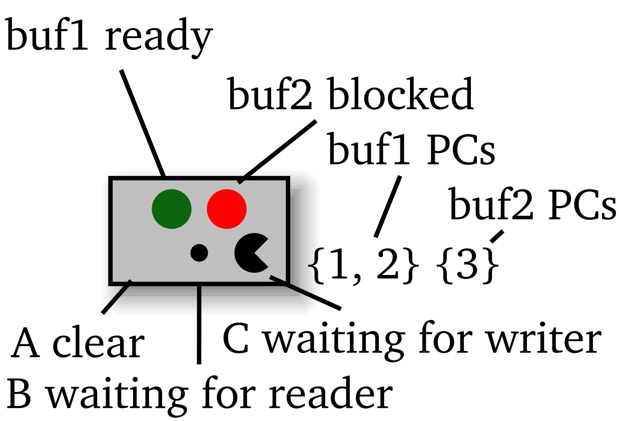




Abstract Simulation

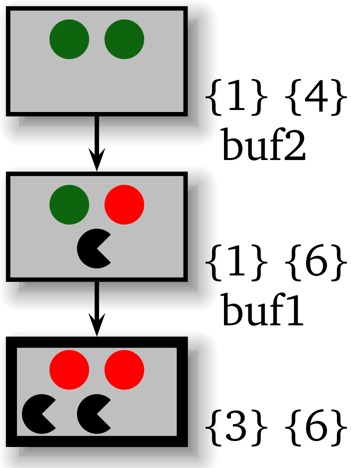
```
{                              // buf1
   1 for (;;)
      2 next B = 3 next A;
} par {                        // buf2
   4 for (;;) {
      5 next C = 6 next B;
      7 next C = 8 next B + 1;
   }
}
```

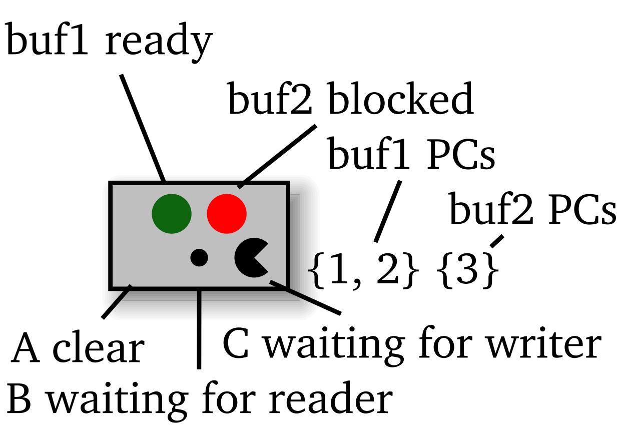




Abstract Simulation

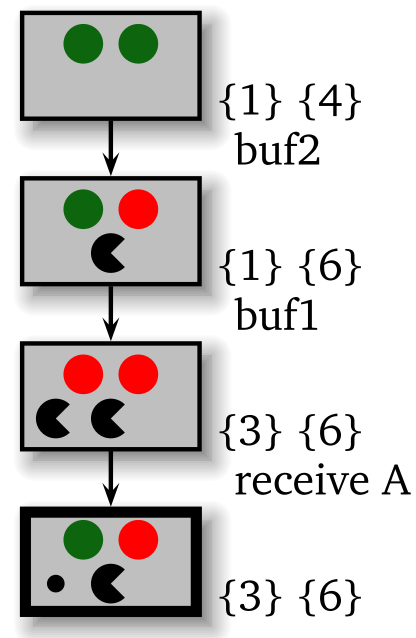
```
{                              // buf1
   1for (;;)
      2next B = 3next A;
} par {                        // buf2
   4for (;;) {
      5next C = 6next B;
      7next C = 8next B + 1;
   }
}
```

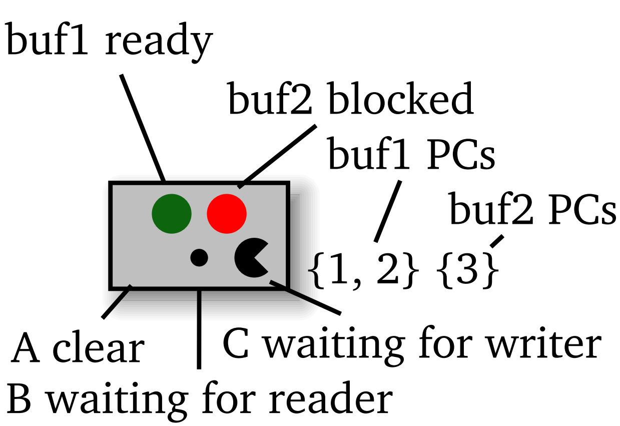




Abstract Simulation

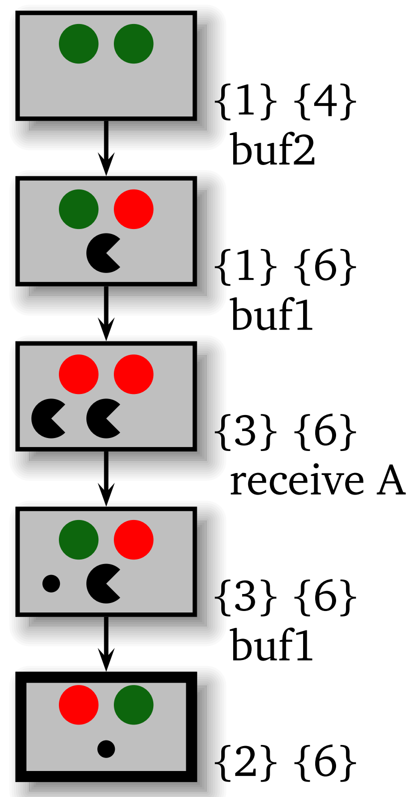
```
{                          // buf1
  1for (;;)
     2next B = 3next A;
} par {                    // buf2
  4for (;;) {
     5next C = 6next B;
     7next C = 8next B + 1;
  }
}
```

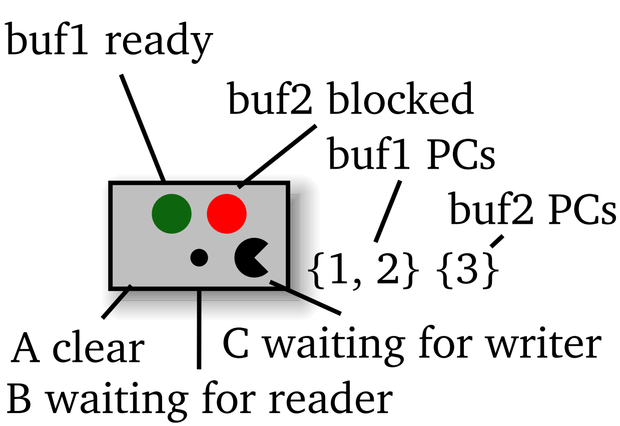




Abstract Simulation

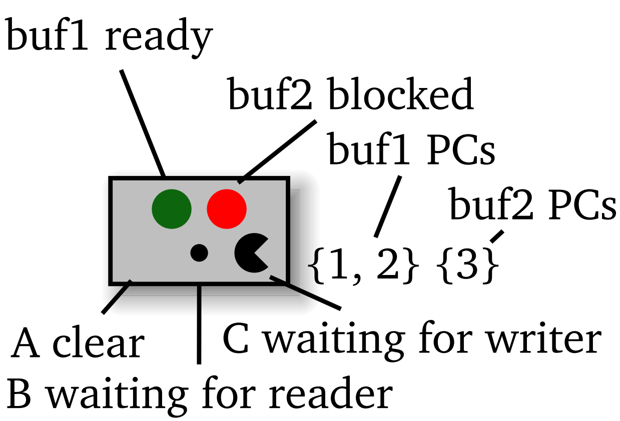
```
{                              // buf1
   1for (;;)
     2next B = 3next A;
} par {                        // buf2
   4for (;;) {
     5next C = 6next B;
     7next C = 8next B + 1;
   }
}
```

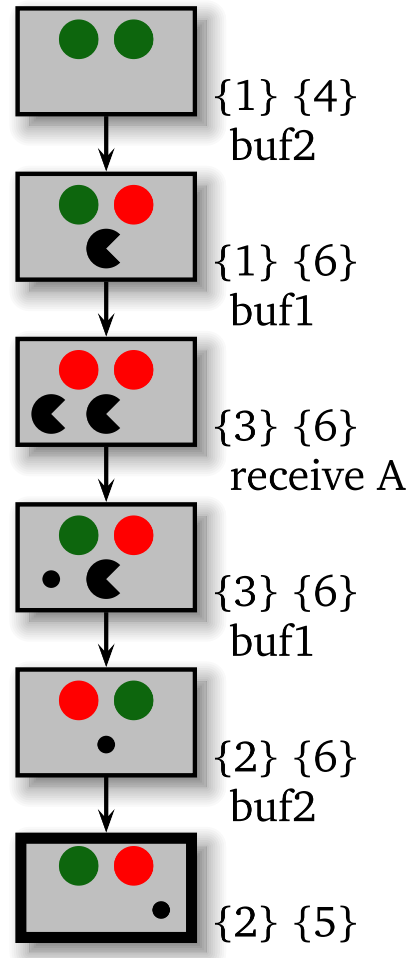




Abstract Simulation

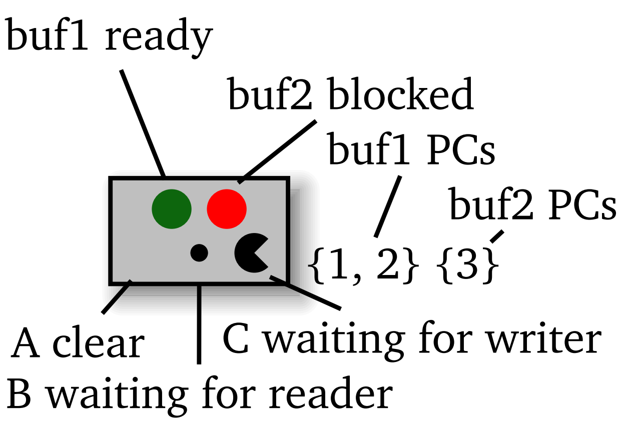
```
{                       // buf1
  1for (;;)
    2next B = 3next A;
} par {                 // buf2
  4for (;;) {
    5next C = 6next B;
    7next C = 8next B + 1;
  }
}
```

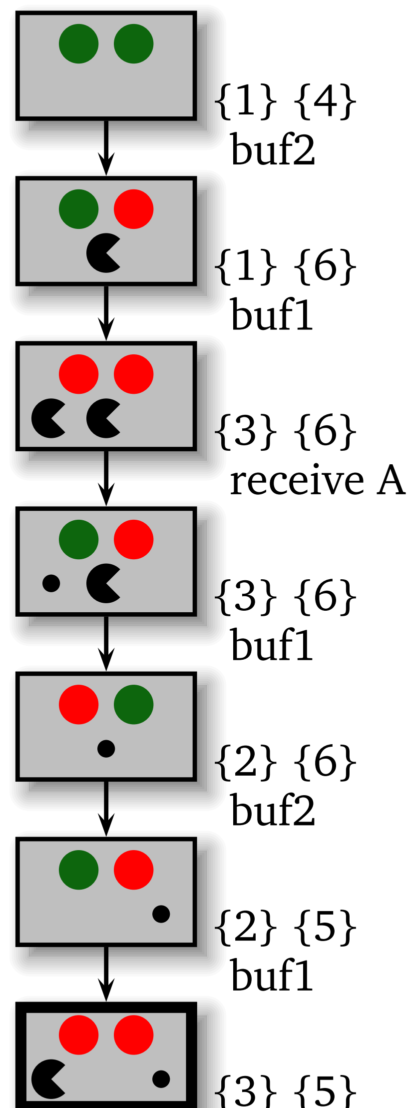




Abstract Simulation

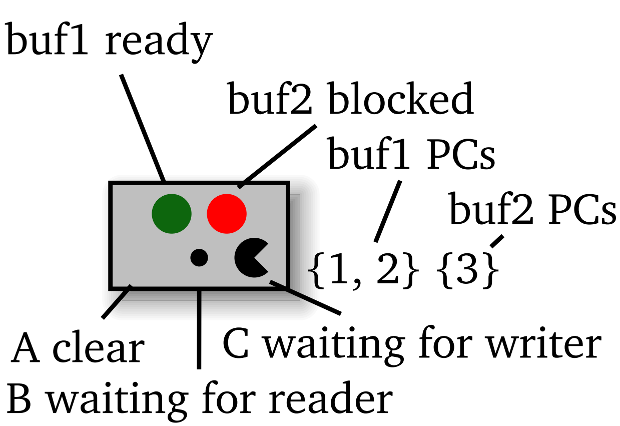
```
{                          // buf1
  1for (;;)
    2next B = 3next A;
} par {                    // buf2
  4for (;;) {
    5next C = 6next B;
    7next C = 8next B + 1;
  }
}
```

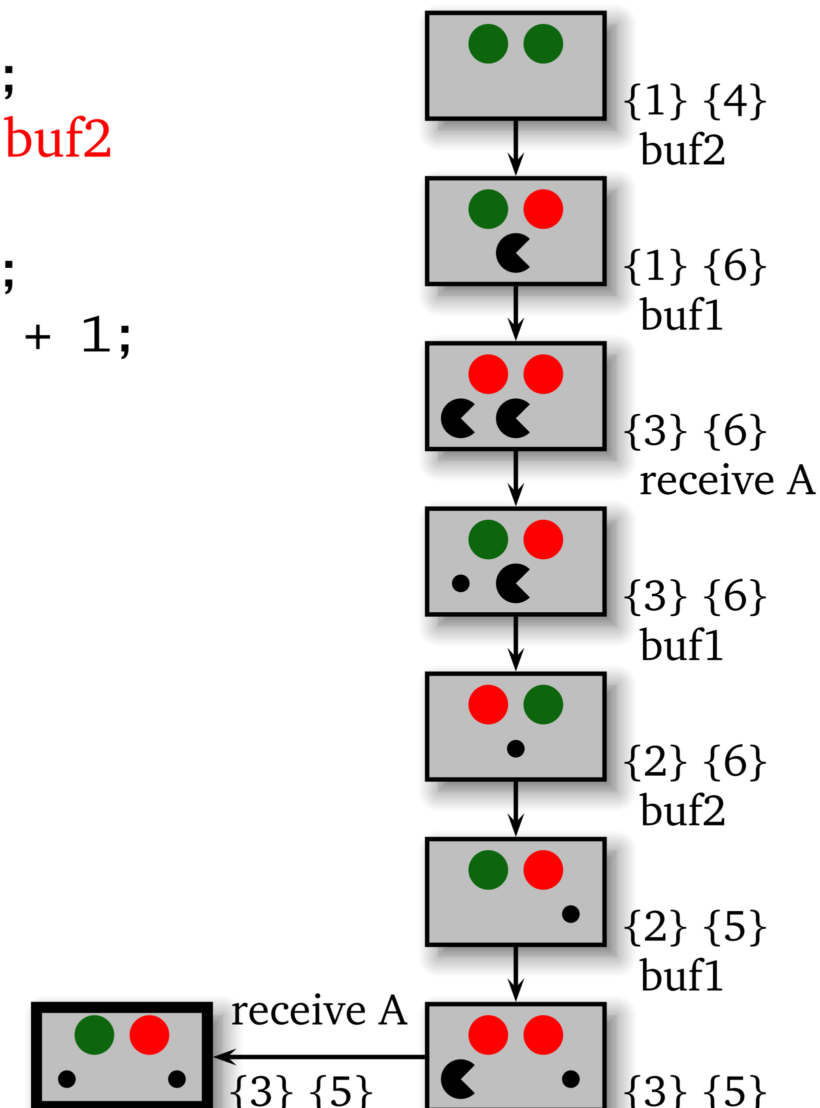




Abstract Simulation

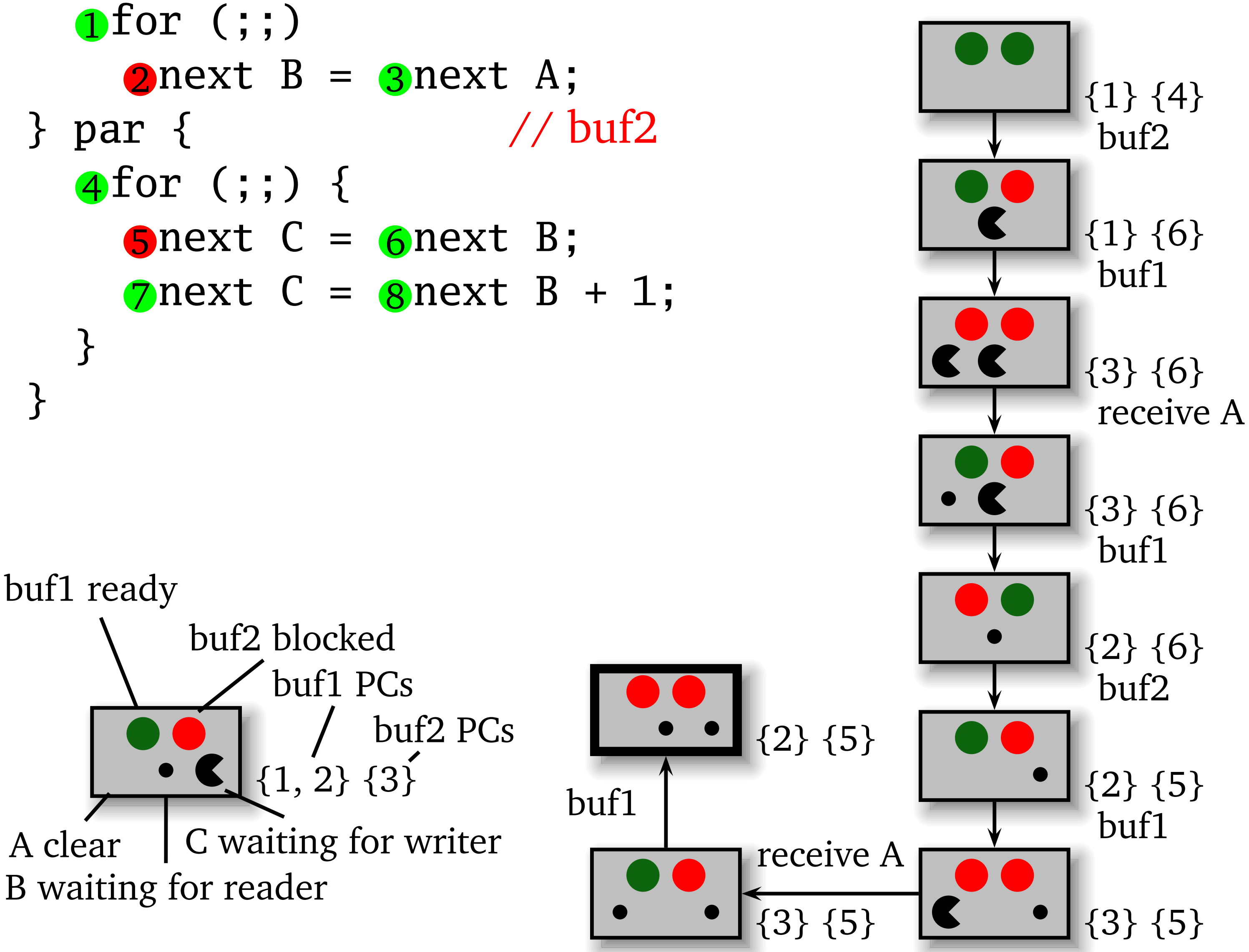
```
{                              // buf1
  1for (;;)
    2next B = 3next A;
} par {                        // buf2
  4for (;;) {
    5next C = 6next B;
    7next C = 8next B + 1;
  }
}
```





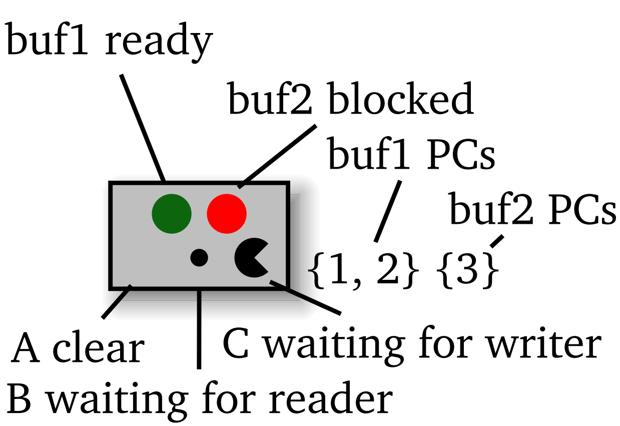
Abstract Simulation

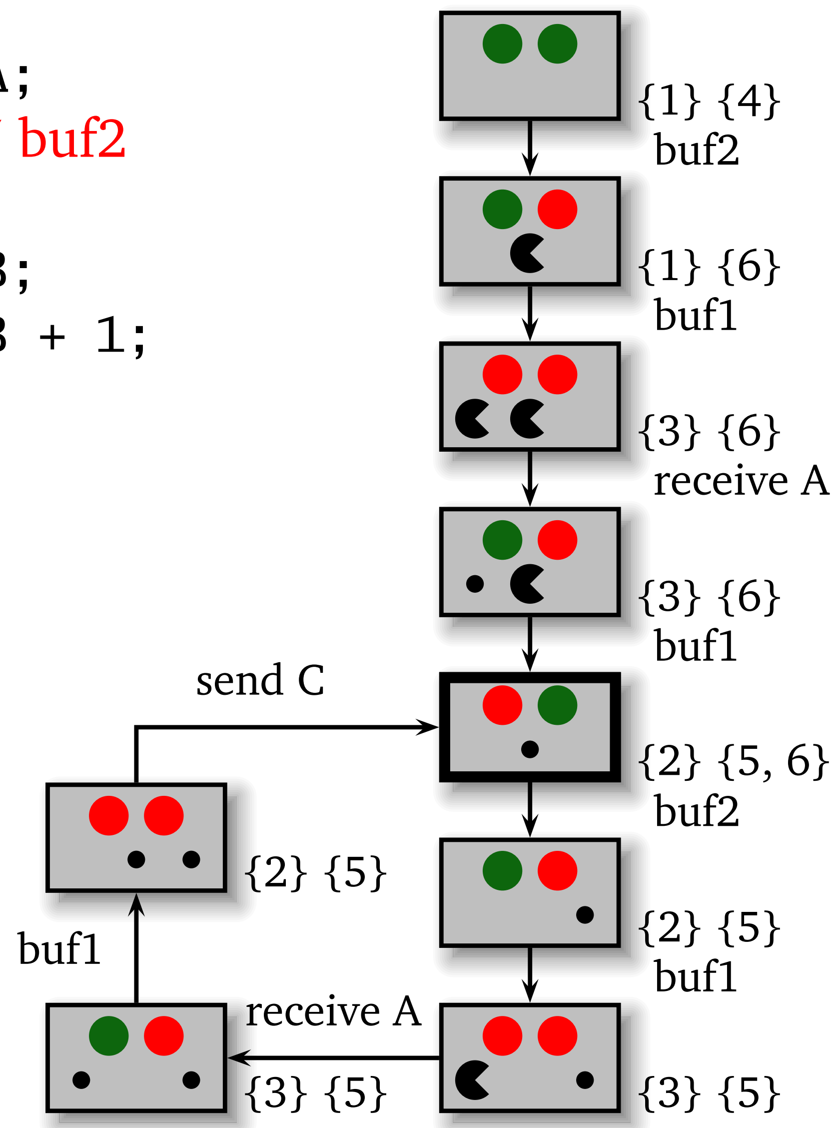
```
{                            // buf1
  1for (;;)
    2next B = 3next A;
} par {                      // buf2
  4for (;;) {
    5next C = 6next B;
    7next C = 8next B + 1;
  }
}
```



Abstract Simulation

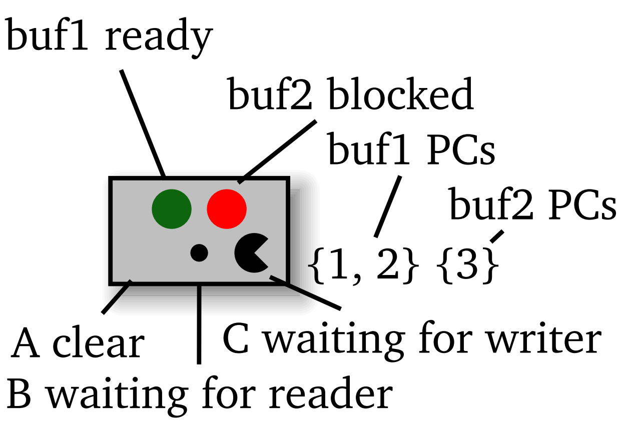
```
{                              // buf1
  1 for (;;)
      2 next B = 3 next A;
} par {                        // buf2
  4 for (;;) {
      5 next C = 6 next B;
      7 next C = 8 next B + 1;
  }
}
```

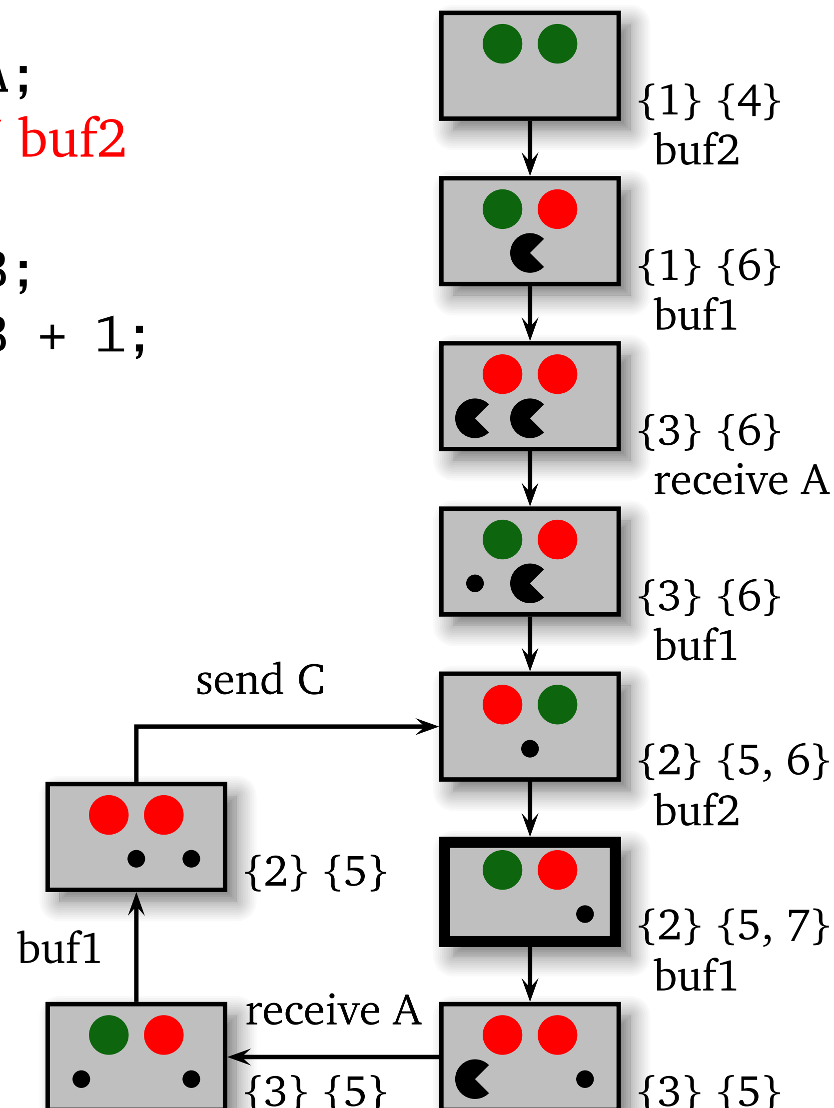




Abstract Simulation

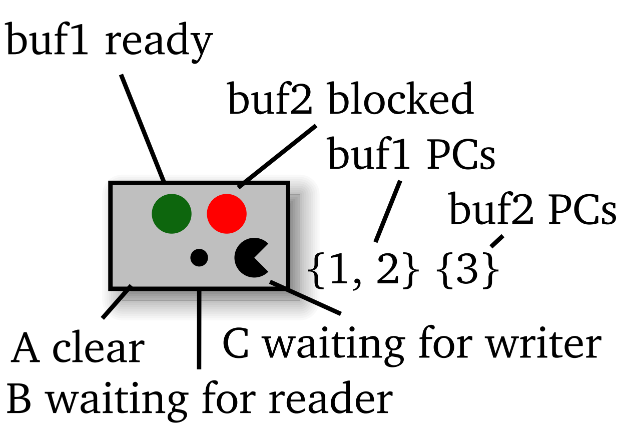
```
{                            // buf1
   1for (;;)
     2next B = 3next A;
} par {                      // buf2
   4for (;;) {
     5next C = 6next B;
     7next C = 8next B + 1;
   }
}
```

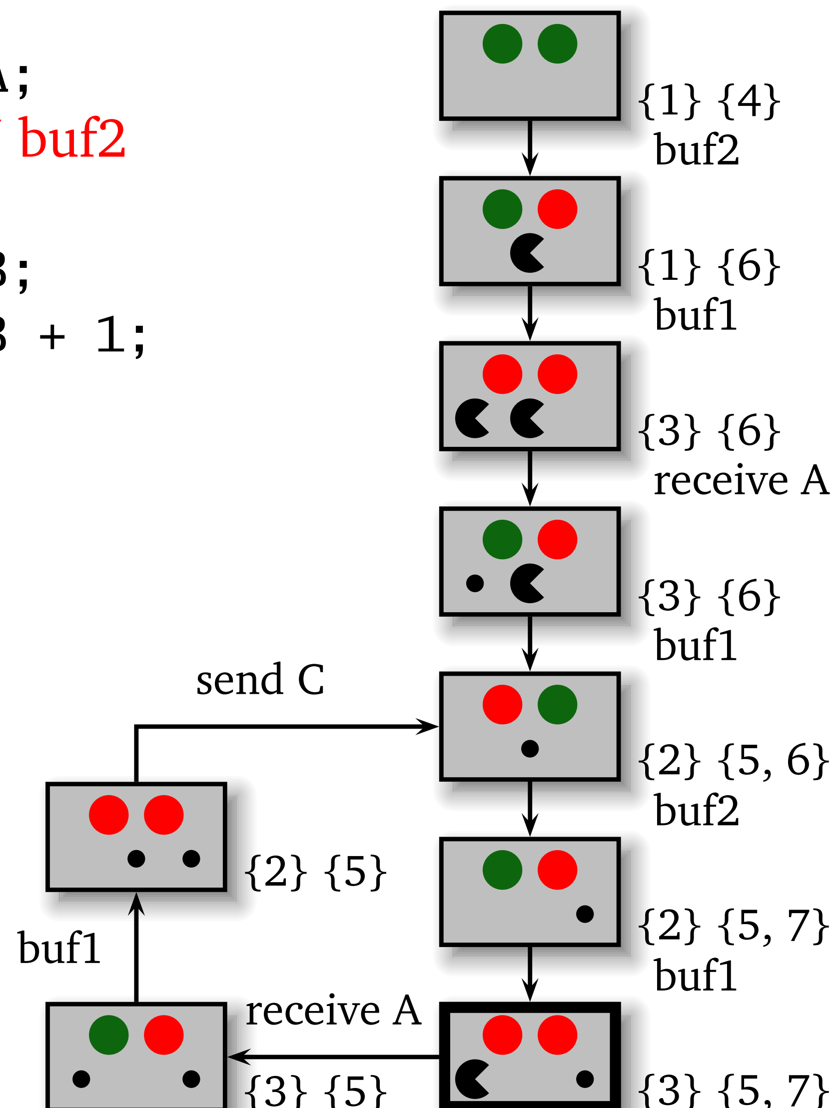




Abstract Simulation

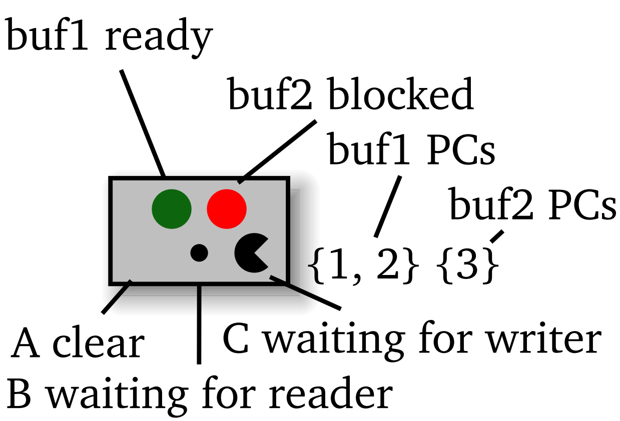
```
{                           // buf1
  1for (;;)
    2next B = 3next A;
} par {                     // buf2
  4for (;;) {
    5next C = 6next B;
    7next C = 8next B + 1;
  }
}
```

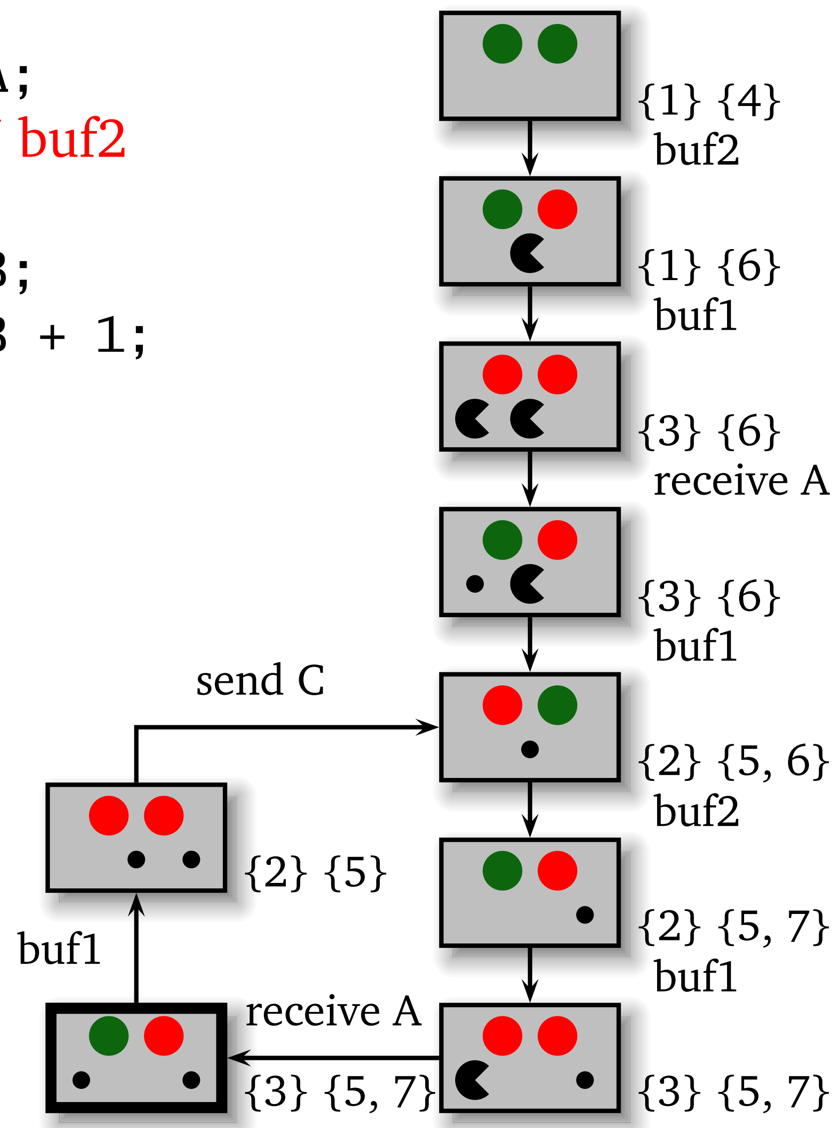




Abstract Simulation

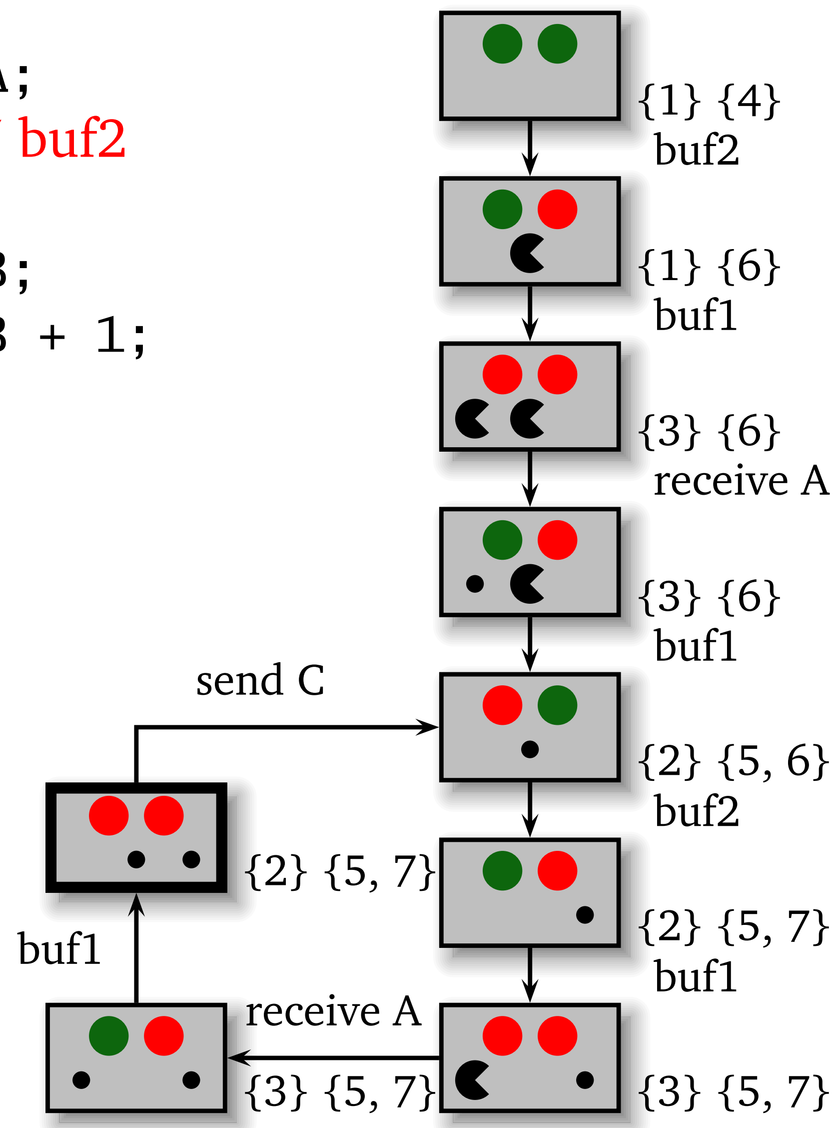
```
{                               // buf1
  ①for (;;)
    ②next B = ③next A;
} par {                         // buf2
  ④for (;;) {
    ⑤next C = ⑥next B;
    ⑦next C = ⑧next B + 1;
  }
}
```

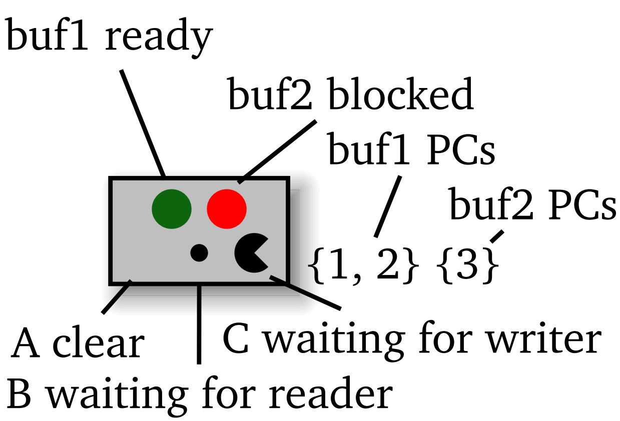




Abstract Simulation

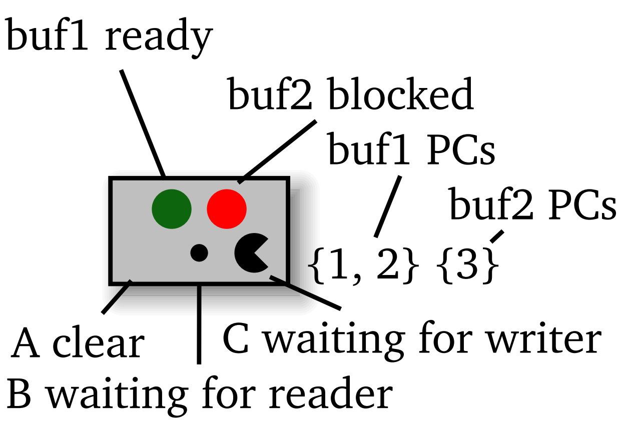
```
{                          // buf1
  1for (;;)
    2next B = 3next A;
} par {                    // buf2
  4for (;;) {
    5next C = 6next B;
    7next C = 8next B + 1;
  }
}
```

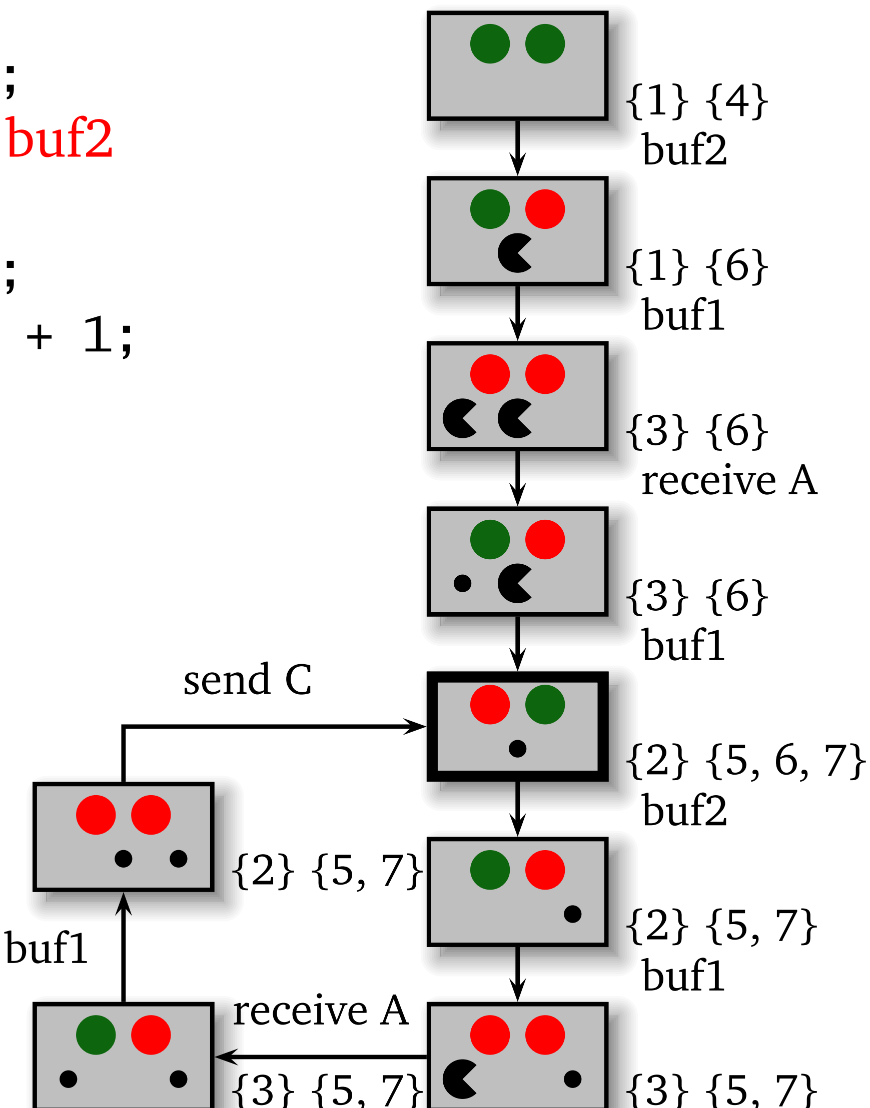




Abstract Simulation

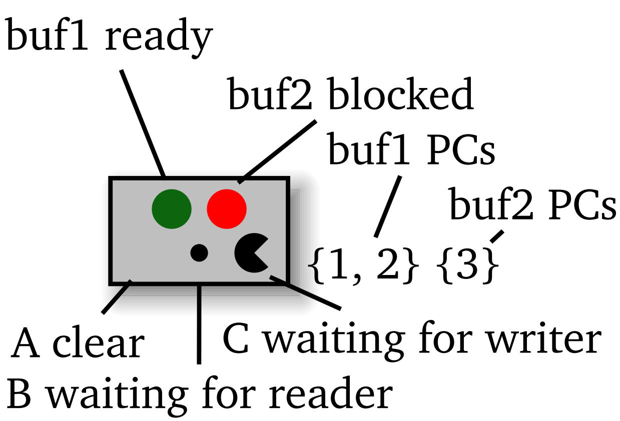
```
{                              // buf1
  1for (;;)
    2next B = 3next A;
} par {                        // buf2
  4for (;;) {
    5next C = 6next B;
    7next C = 8next B + 1;
  }
}
```

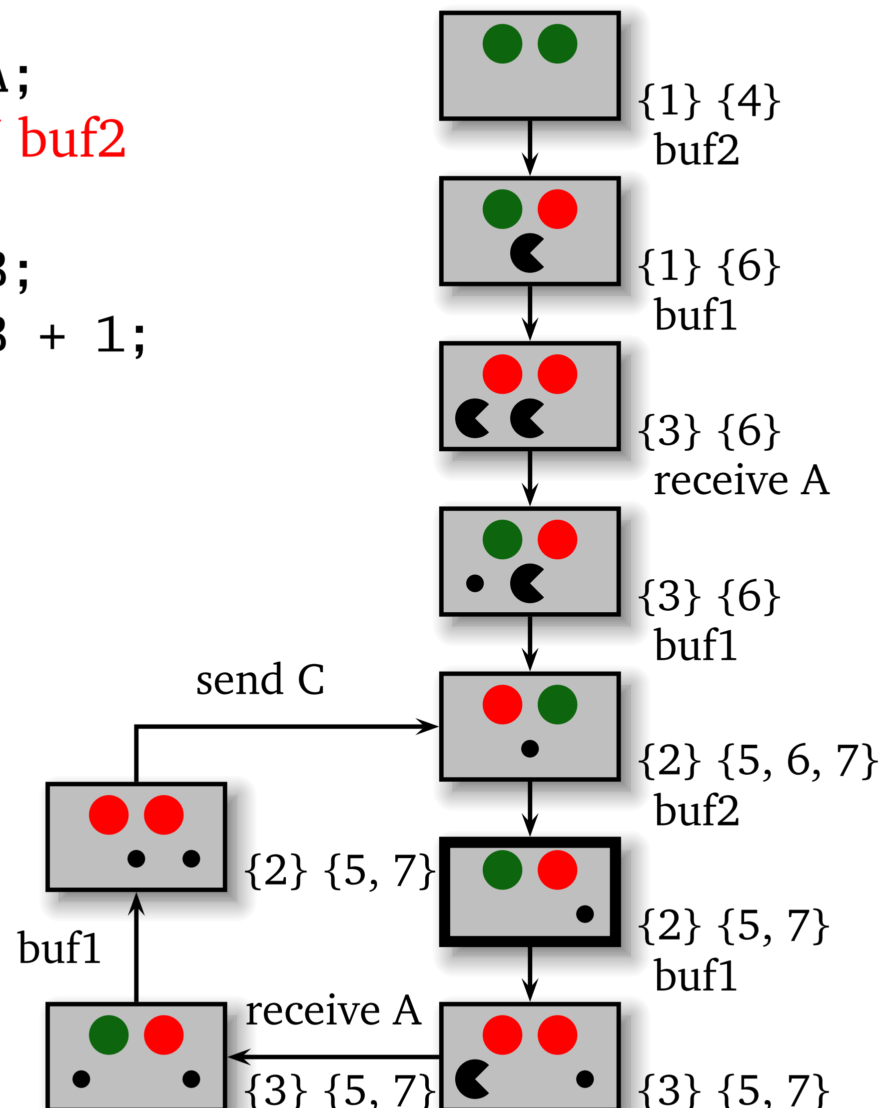




Abstract Simulation

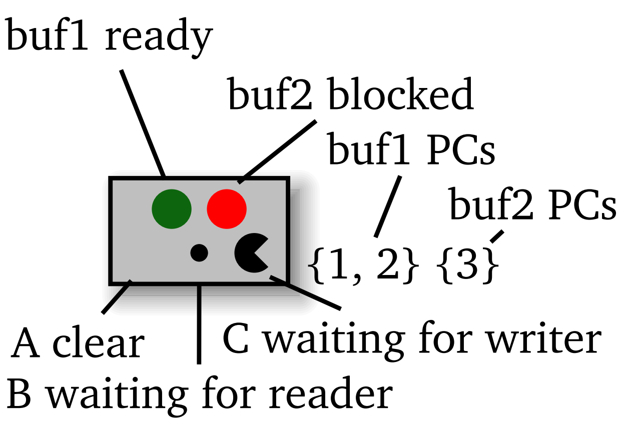
```
{                                  // buf1
   1for (;;)
      2next B = 3next A;
} par {                            // buf2
   4for (;;) {
      5next C = 6next B;
      7next C = 8next B + 1;
   }
}
```

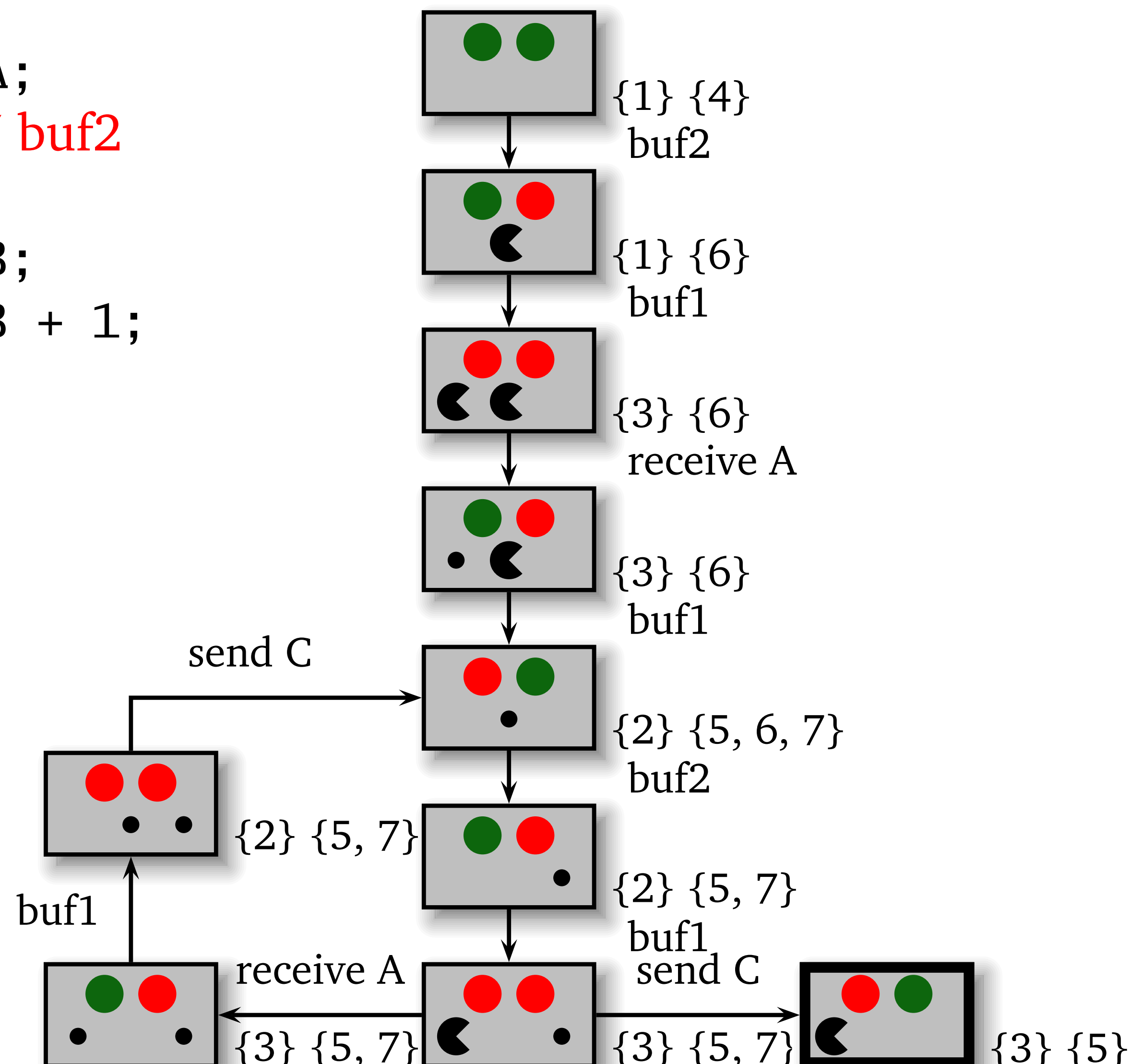




Abstract Simulation

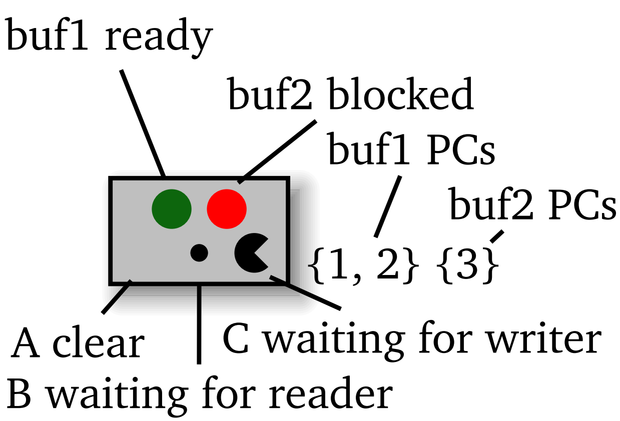
```
{                          // buf1
  1for (;;)
    2next B = 3next A;
} par {                    // buf2
  4for (;;) {
    5next C = 6next B;
    7next C = 8next B + 1;
  }
}
```

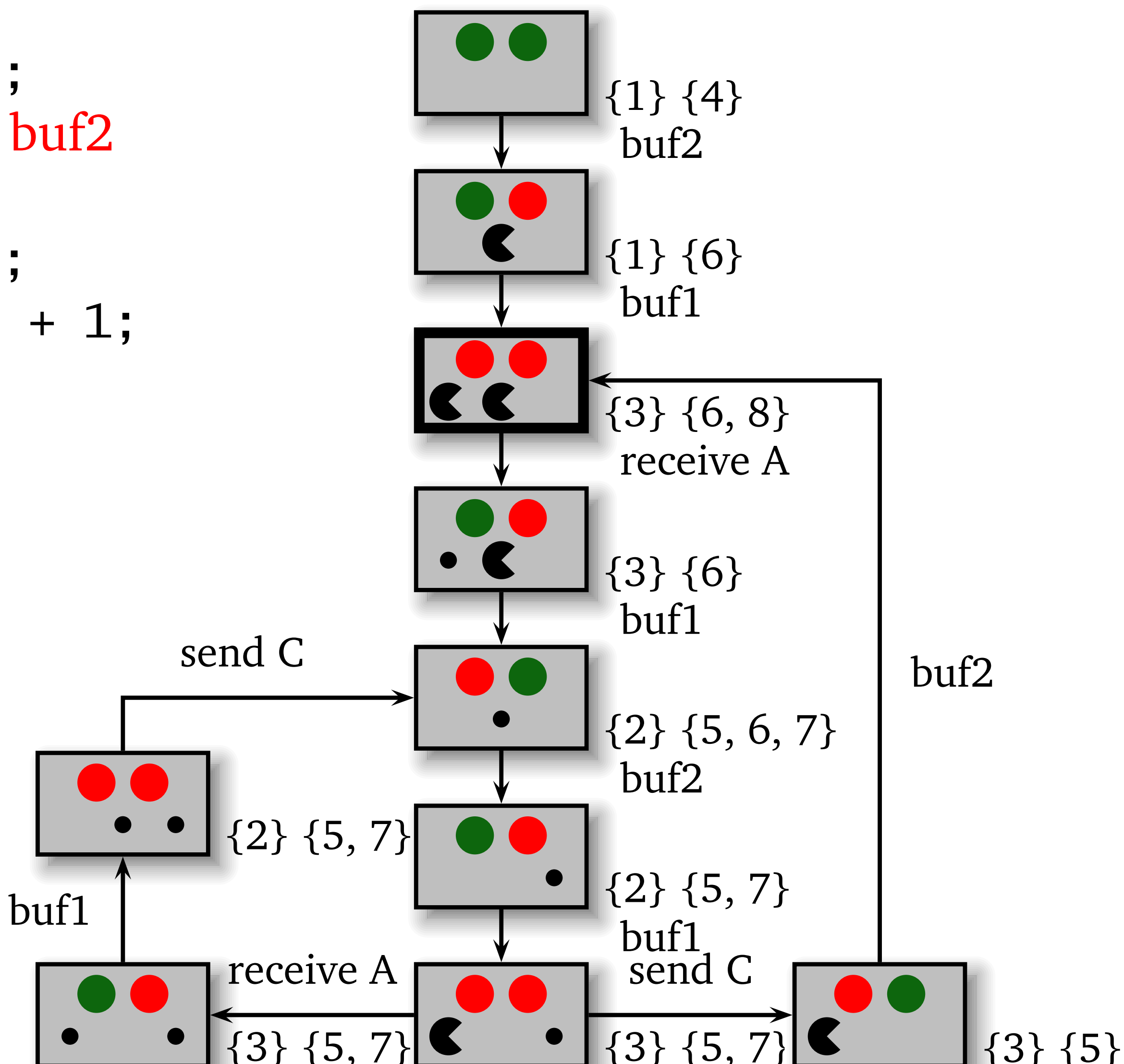




Abstract Simulation

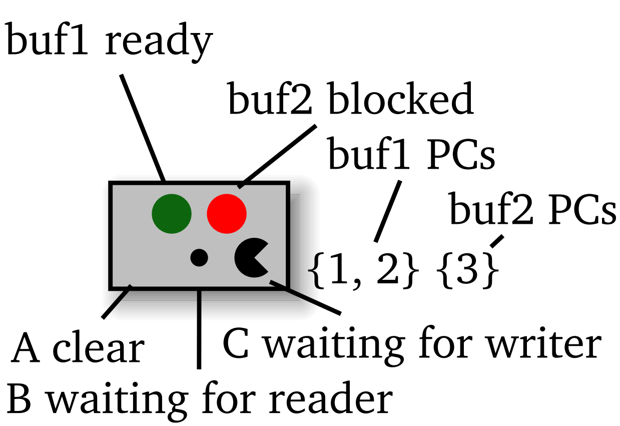
```
{                          // buf1
  1for (;;)
    2next B = 3next A;
} par {                    // buf2
  4for (;;) {
    5next C = 6next B;
    7next C = 8next B + 1;
  }
}
```

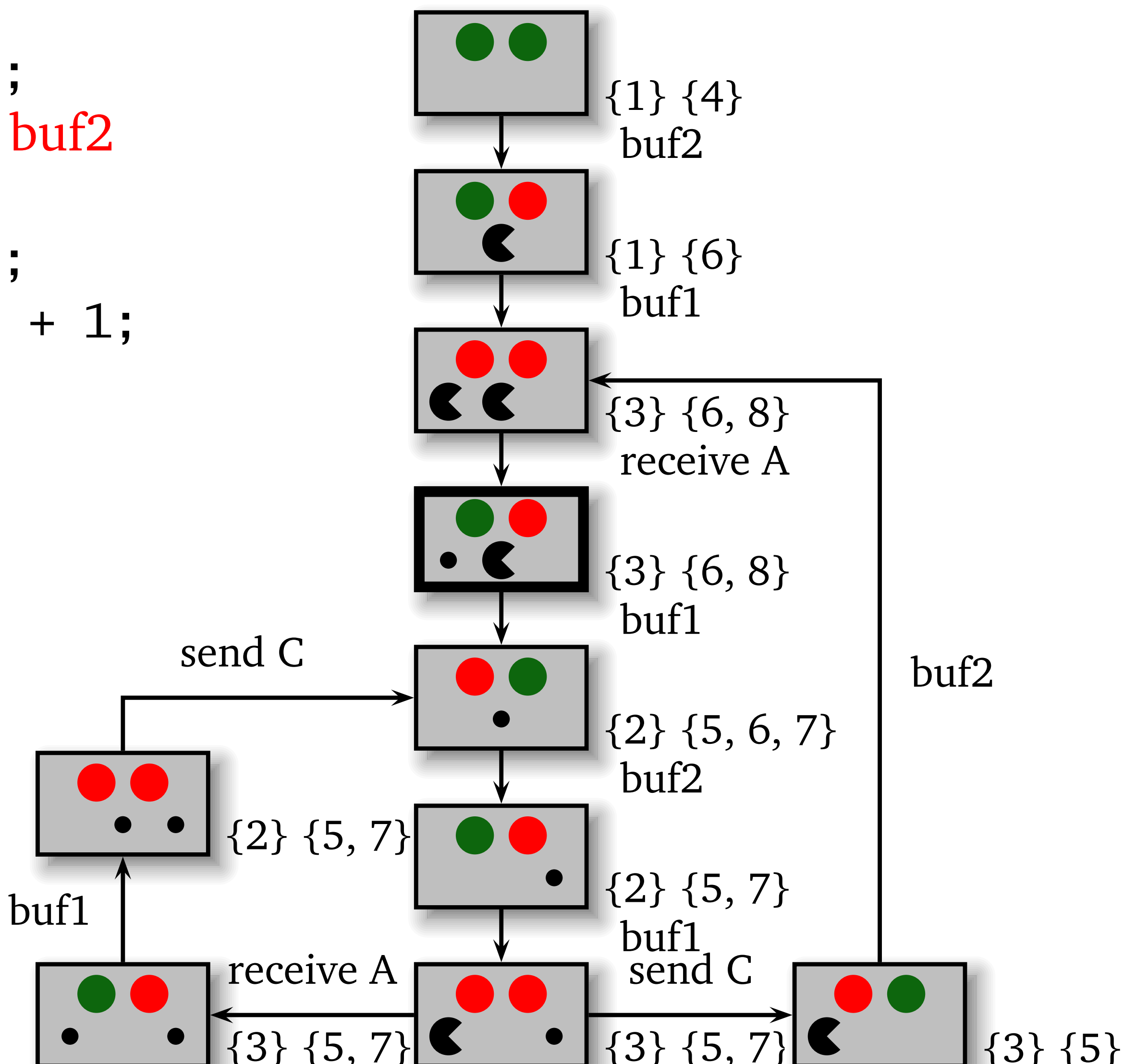




Abstract Simulation

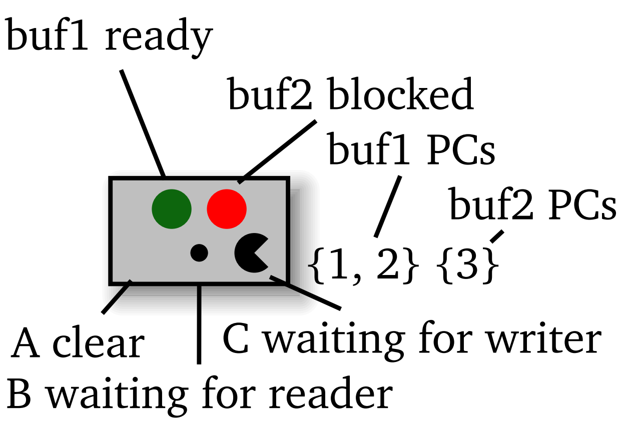
```
{                              // buf1
   1for (;;)
      2next B = 3next A;
} par {                        // buf2
   4for (;;) {
      5next C = 6next B;
      7next C = 8next B + 1;
   }
}
```

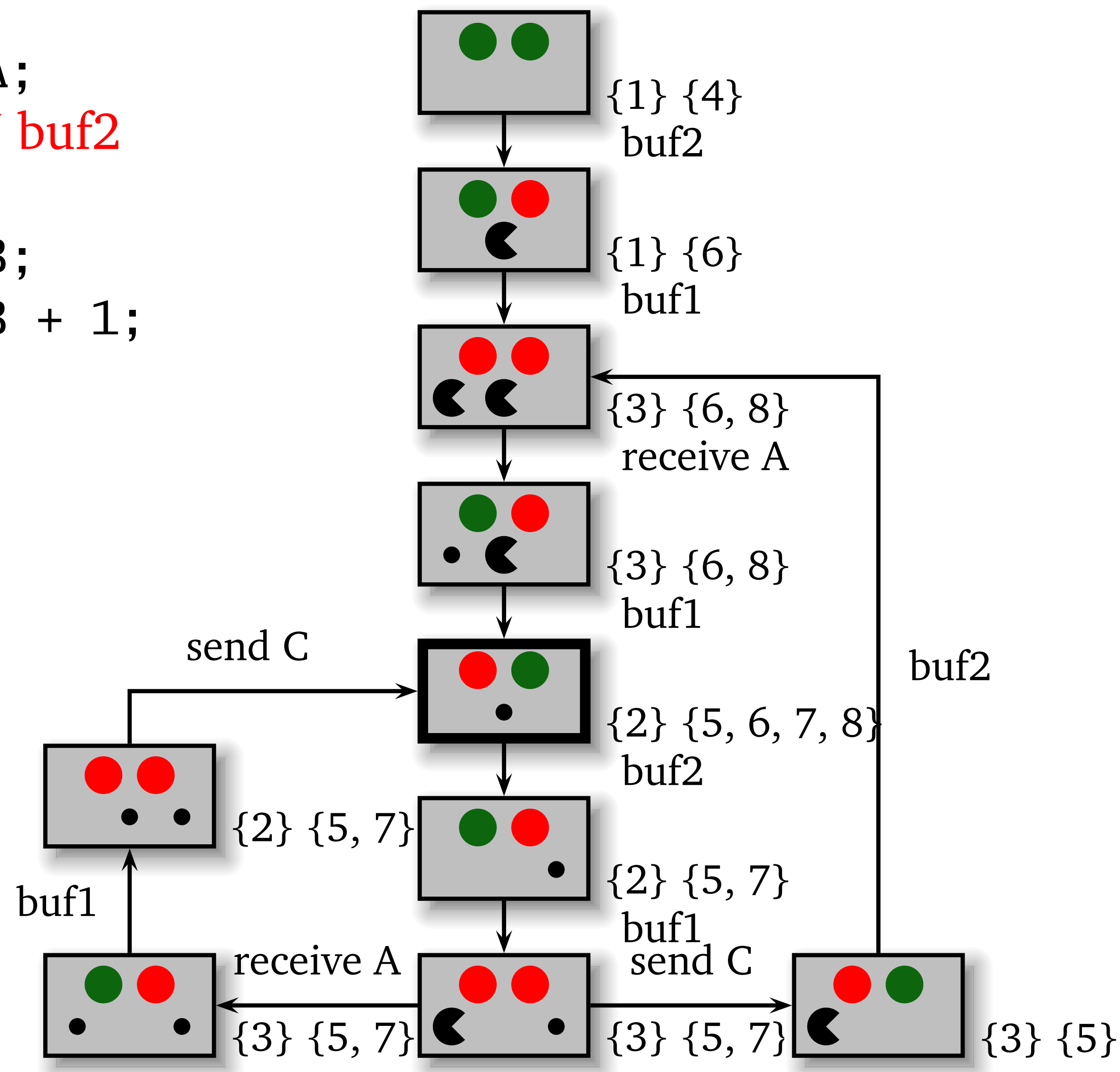




Abstract Simulation

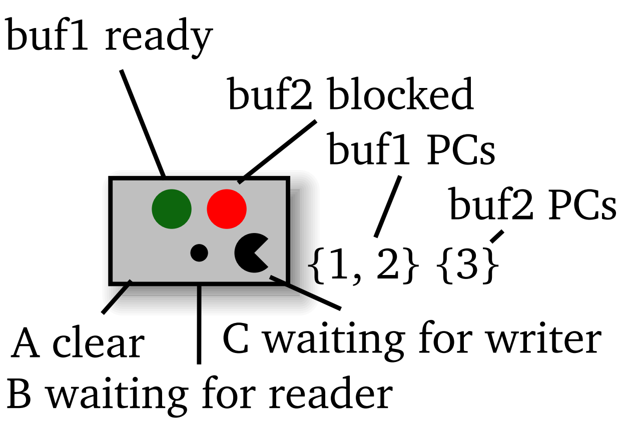
```
{                          // buf1
  1for (;;)
    2next B = 3next A;
} par {                    // buf2
  4for (;;) {
    5next C = 6next B;
    7next C = 8next B + 1;
  }
}
```

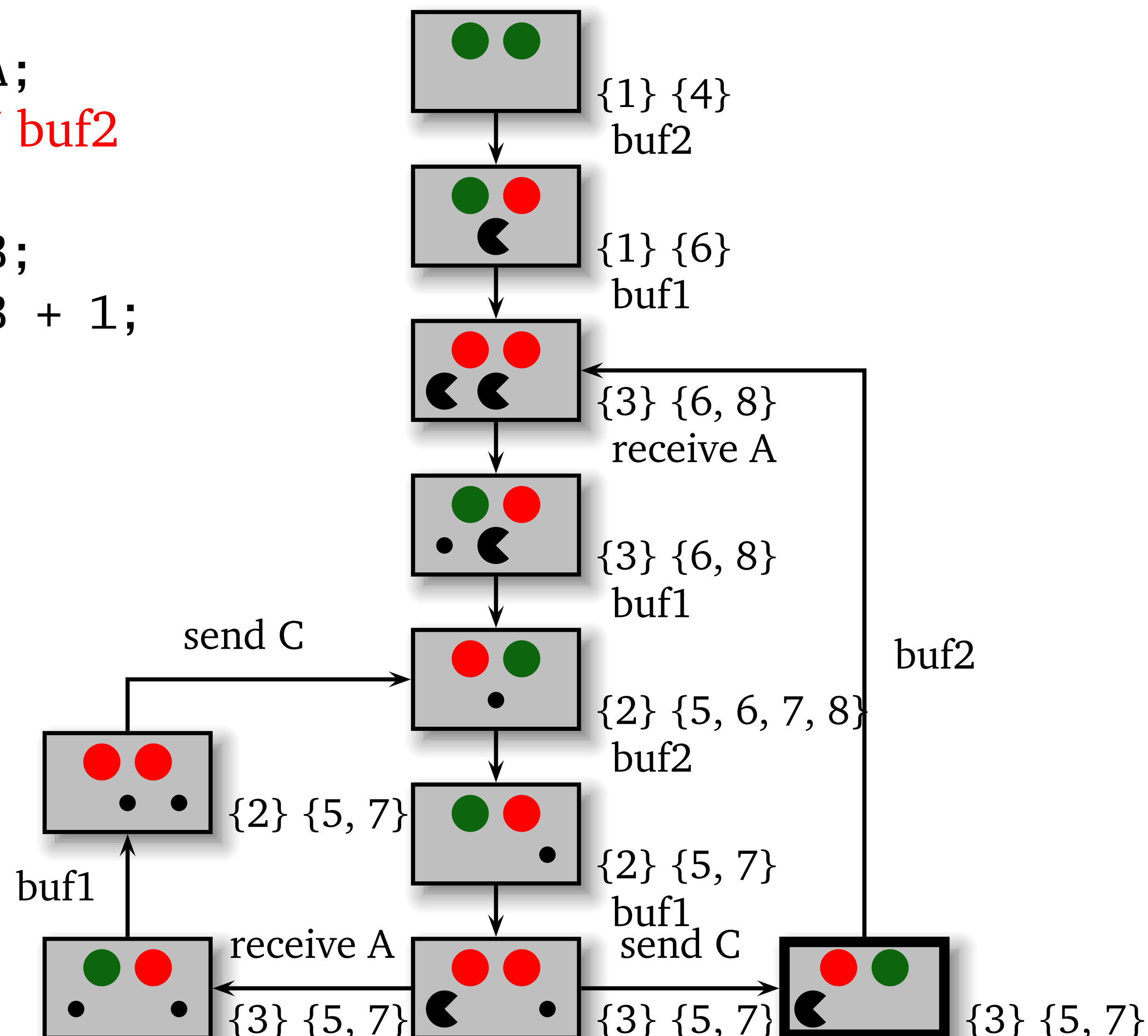




Abstract Simulation

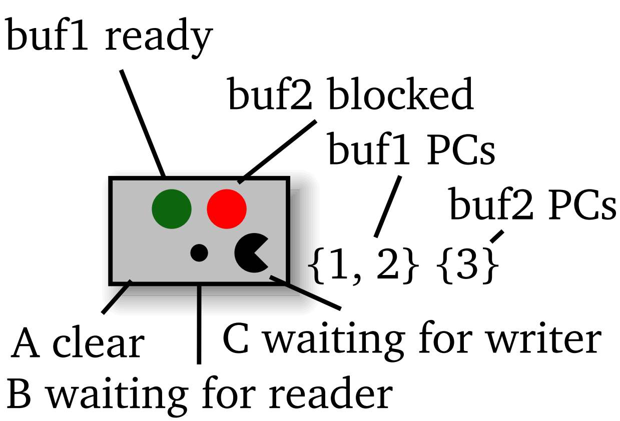
```
{                              // buf1
  1for (;;)
    2next B = 3next A;
} par {                        // buf2
  4for (;;) {
    5next C = 6next B;
    7next C = 8next B + 1;
  }
}
```

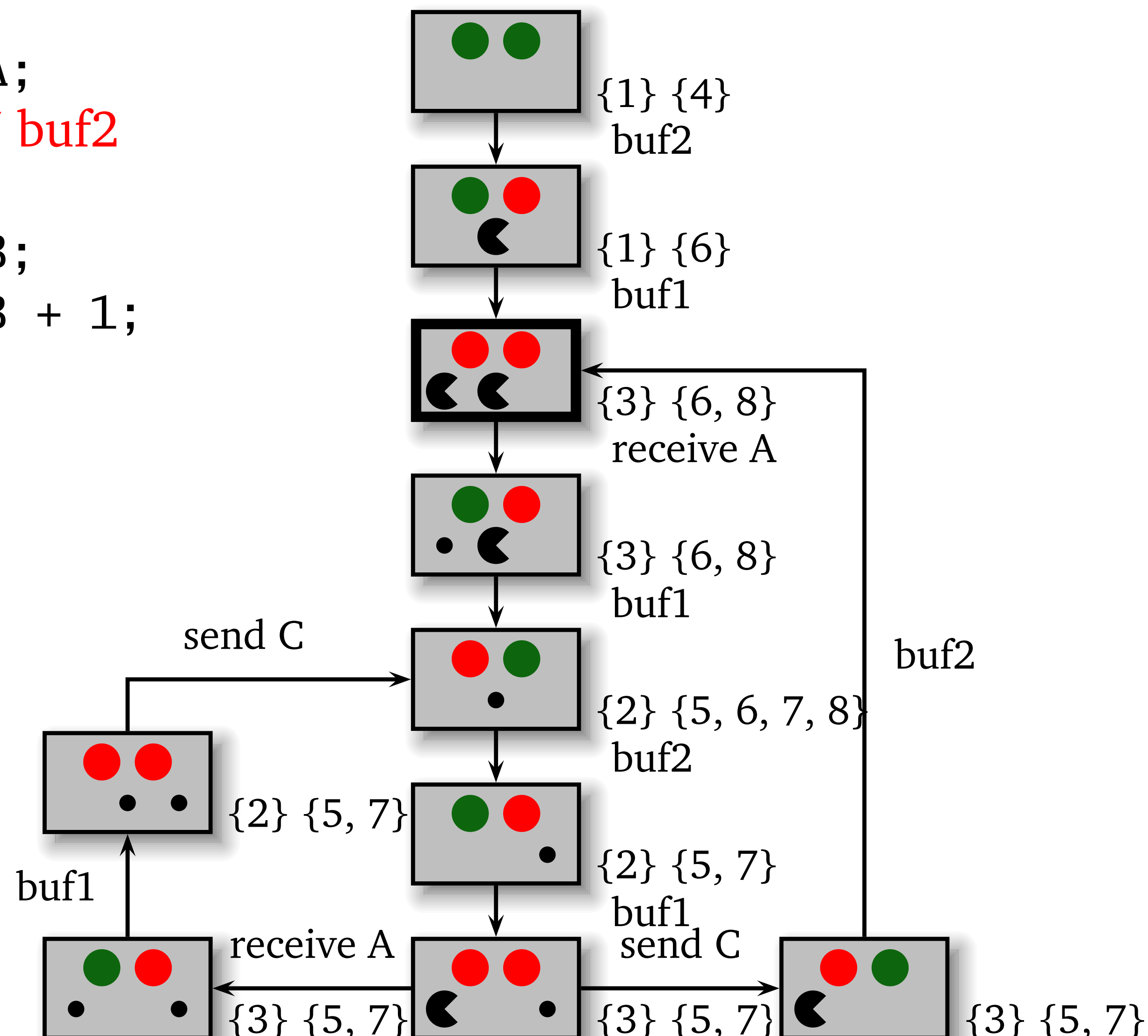




Abstract Simulation

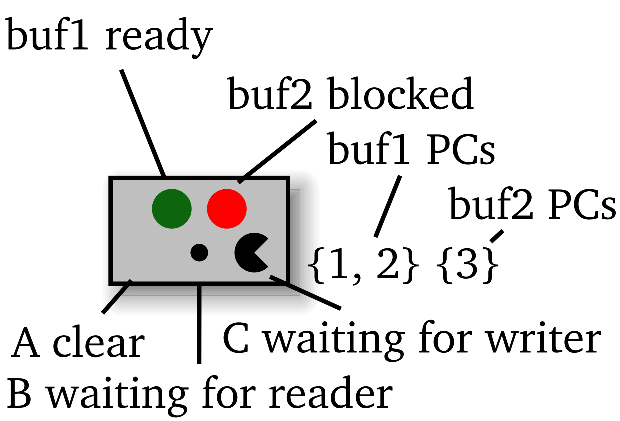
```
{                          // buf1
  1for (;;)
    2next B = 3next A;
} par {                    // buf2
  4for (;;) {
    5next C = 6next B;
    7next C = 8next B + 1;
  }
}
```

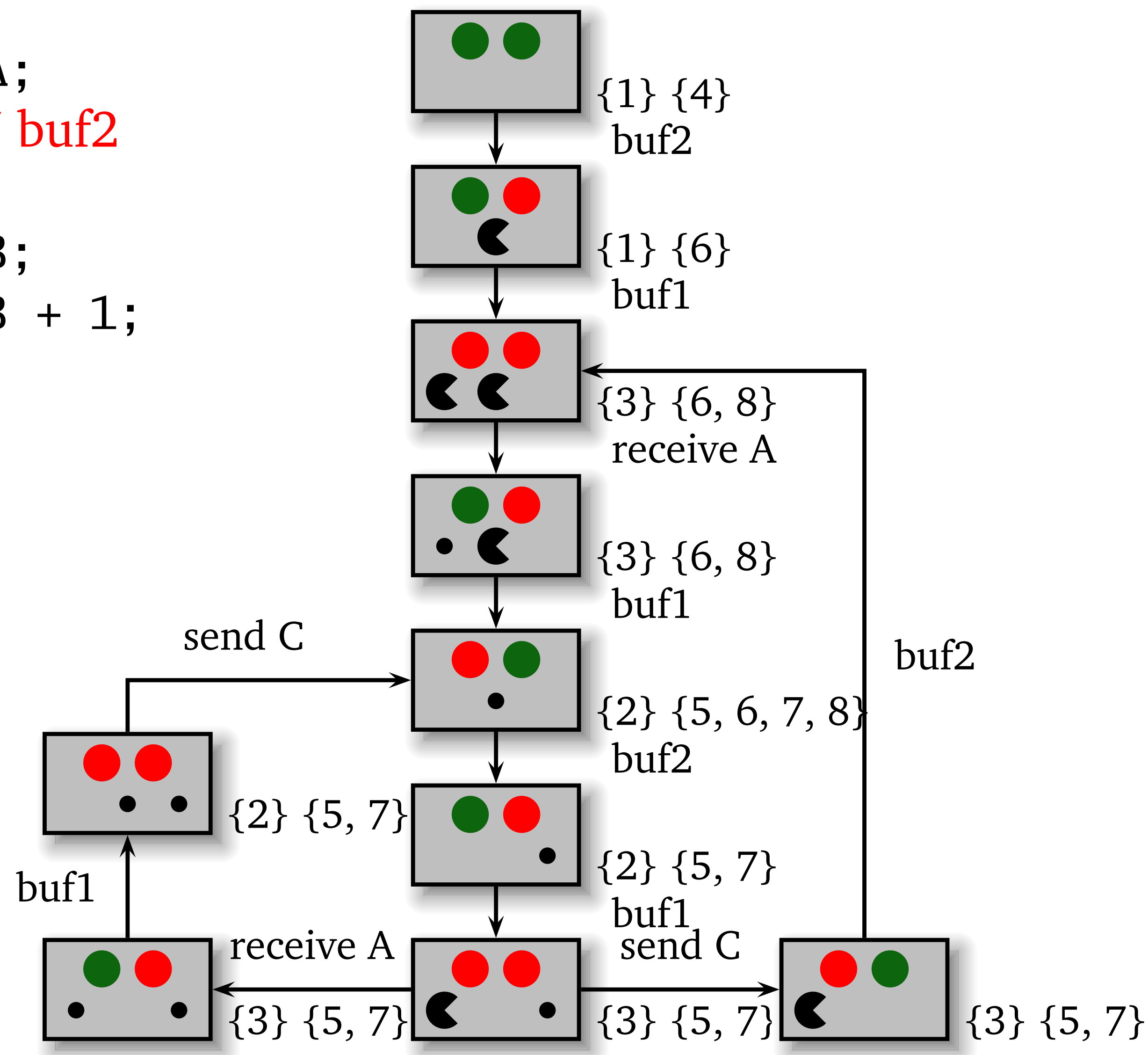




Abstract Simulation

```
{                                // buf1
  1for (;;)
    2next B = 3next A;
} par {                          // buf2
  4for (;;) {
    5next C = 6next B;
    7next C = 8next B + 1;
  }
}
```





Experiments

Example	Lines	Processes	States	
			Partial	Full
Berkeley	36	3	5	6
Buffer2	25	4	10	8
Buffer3	26	5	20	10
Buffer10	33	12	174	24
Esterel1	144	5	49	56
Esterel2	127	5	24	18
FIR5	78	19	229	79
FIR19	190	75	2819	372

Conclusions

- The SHIM Model: Sequential processes communicating through rendezvous
- Sequential language plus
  - concurrency,
  - communication, and
  - exceptions.
- Scheduling-independent
  - Kahn networks with rendezvous
  - Nondeterministic scheduler produces deterministic behavior

Future Work

- Automata abstract communication patterns
  Useful for deadlock detection, protocol violation

Future Work

- Automata abstract communication patterns
  Useful for deadlock detection, protocol violation
- Synthesis for multicore processors
  Compile together the processes on each core

Future Work

- Automata abstract communication patterns
  Useful for deadlock detection, protocol violation
- Synthesis for multicore processors
  Compile together the processes on each core
- Hardware/software cosynthesis
  Bounded subset has reasonable hardware semantics

Future Work

- Automata abstract communication patterns
  Useful for deadlock detection, protocol violation
- Synthesis for multicore processors
  Compile together the processes on each core
- Hardware/software cosynthesis
  Bounded subset has reasonable hardware semantics
- Richer data structures
  Shared arrays, Trees, etc.

Future Work

- Automata abstract communication patterns
  Useful for deadlock detection, protocol violation
- Synthesis for multicore processors
  Compile together the processes on each core
- Hardware/software cosynthesis
  Bounded subset has reasonable hardware semantics
- Richer data structures
  Shared arrays, Trees, etc.
- Convince world: scheduling-independent concurrency is good