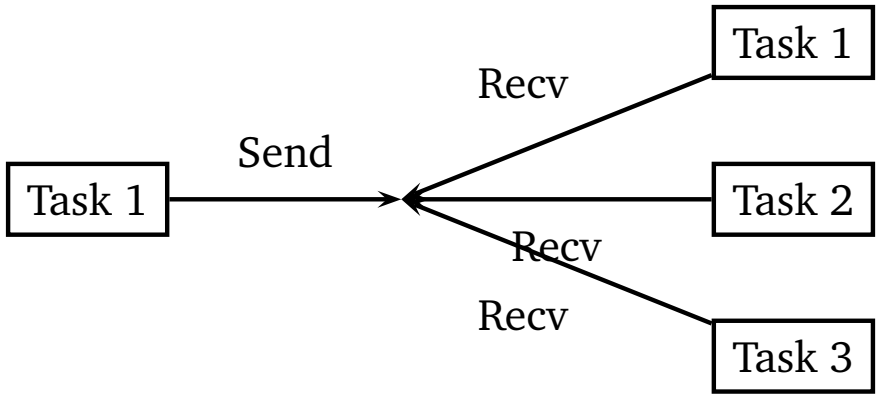
Static Deadlock Detection for the SHIM Concurrent Language

Nalini Vasudevan Stephen A. Edwards

Columbia University

The SHIM Model

- Stands for *Software Hardware Integration Medium*
- Race free, scheduling independent, concurrent model
- Blocking synchronous 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;
}
```

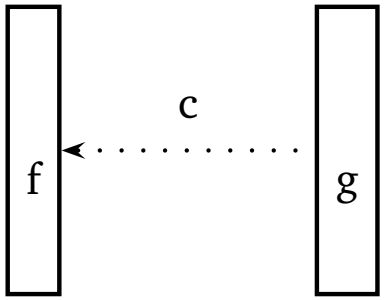
Additional Constructs

$stmt_1$ *par* $stmt_2$	Run $stmt_1$ and $stmt_2$ concurrently
send var	Send on channel *var*
recv var	Receive on channel *var*

Communication

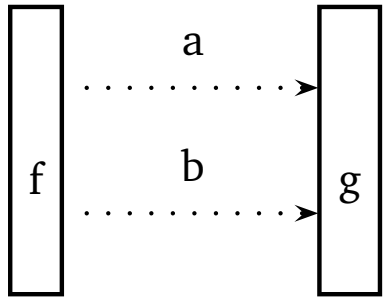
- Blocking: wait for all processes connected to *c*

```
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
   b = 5; // sets c
   send b; // send (wait for f)
  // b now 5
}
void main() {
  chan int c = 0;
  f(c); par g(c);
  c = c * 2;
}
```



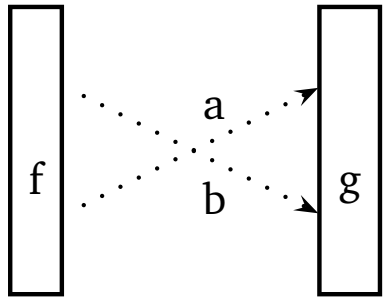
Another Example

```
void main() {
 chan int a, b;
 {
    // Task 1
    a = 15, b = 10;
    send a;
    send b;
 } par {
    // Task 2
    int c;
    recv a;
    recv b;
    c = a + b;
  }
}
```



The Problem

```
void main() {
 chan int a, b;
 {
    // Task 1
    a = 15, b = 10;
    send a;
    send b;
 } par {
    // Task 2
    int c;
    recv b;
    recv a;
    c = a + b;
  }
}
```

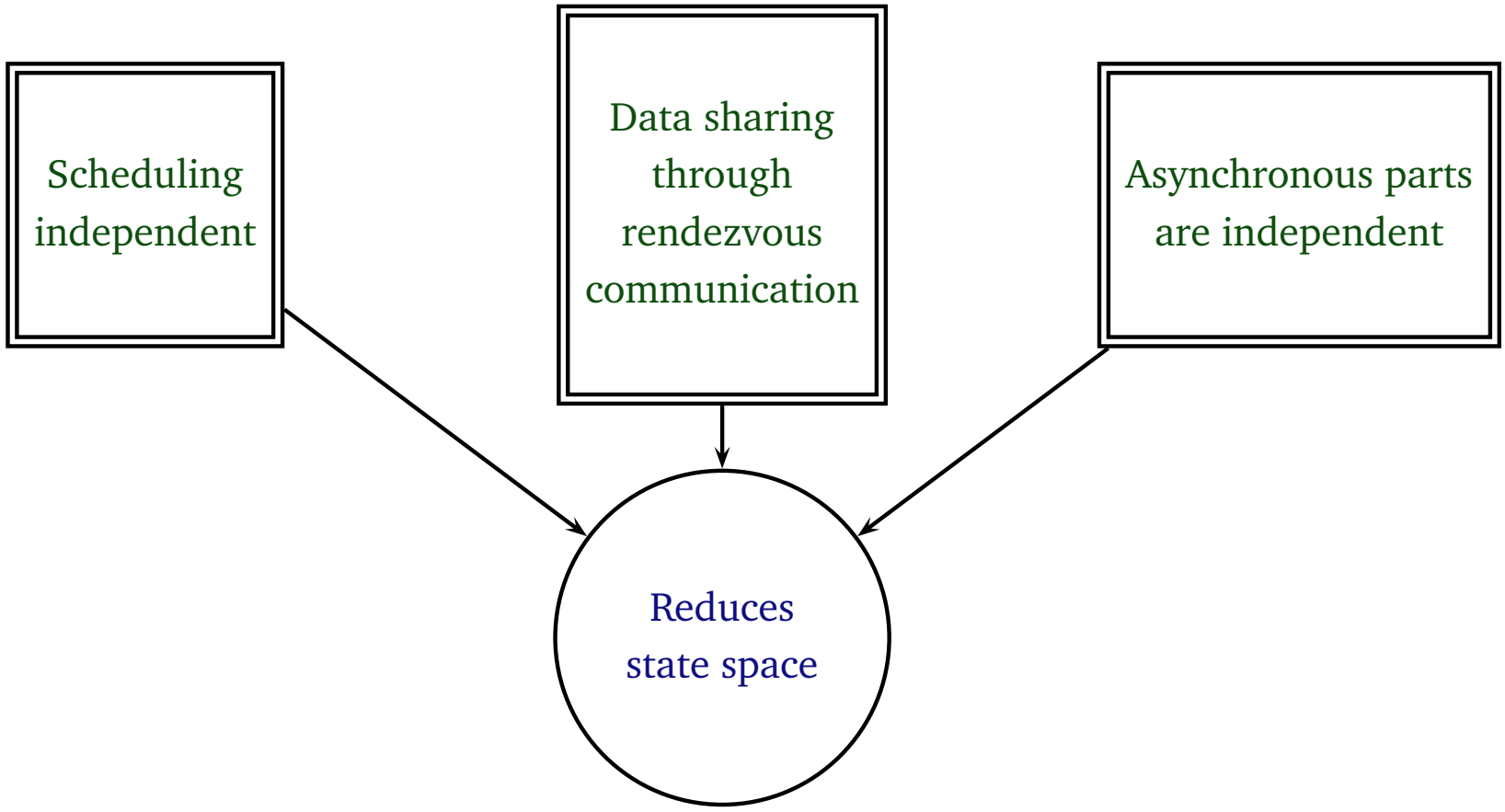


Motivation

- Why SHIM? No data races.
- Deadlocks in SHIM are deterministic (always reproducible).
- SHIM's philosophy: It prefers deadlocks to races. Can be detected at run-time.

Can we statically detect deadlocks in a program?

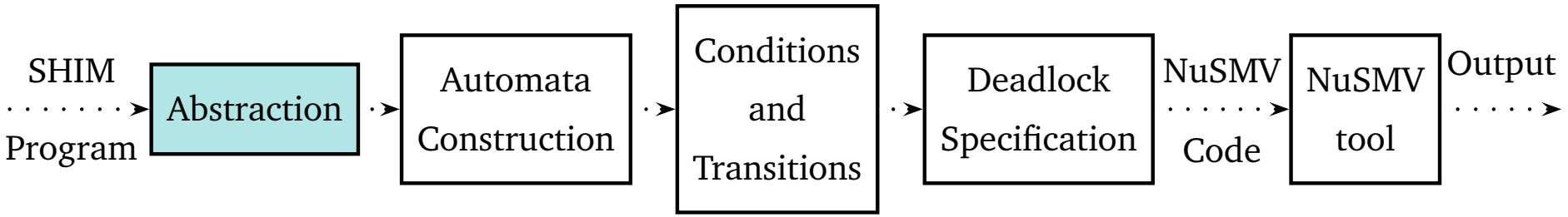
SHIM design for static analysis



Assumptions

- All functions are known at compile time.
- Recursion is statically bounded.
- Channel connections are known at static time.

The Algorithm



Abstraction

```
void main() {
 chan int a, b;
 {
     // Task 1
     a = 15, b = 10;
     send a;
     send b;
 } par {
     // Task 2
     int c;
     recv b;
     recv a;
     c = a + b;

   }
}
```

Abstraction

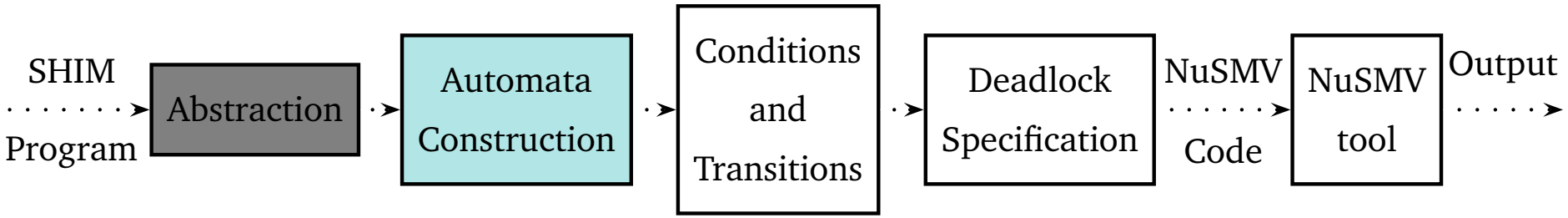
```
void main() {
 chan int a, b;
 {
     // Task 1
     a = 15, b = 10;
     send a;
     send b;
 } par {
     // Task 2
     int c;
     recv b;
     recv a;
     c = a + b;

  }
}
```

```
void main() {

 {
     // Task 1
     wait a;
     wait b;
 } par {
     // Task 2

     wait b;
     wait a;

  }
}
```

Abstraction

```
void main() {
 chan int a, b;
 {
     // Task 1
     a = 15, b = 10;
     send a;
     send b;
 } par {
     // Task 2
     int c;
     recv b;
     recv a;
     c = a + b;

   }
}
```

```
void main() {

 {
     // Task 1
     wait a; (1)
     wait b; (2)
 } par { (1)
     // Task 2

     wait b; (1)
     wait a; (2)

   }
}
```

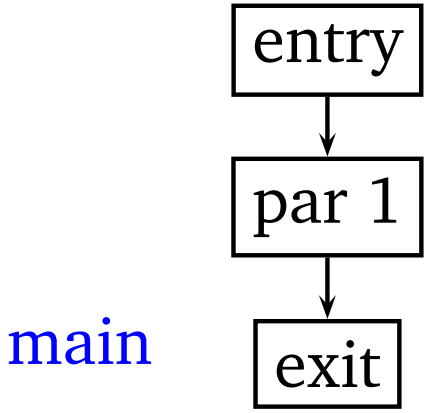

SHIM
Program
Abstraction
Automata
Construction
Conditions
and
Transitions
Deadlock
Specification
NuSMV
Code
NuSMV
tool
Output

Automata

```
void main() {
 {
     // Task 1
     wait a; (1)
     wait b; (2)
 } par { (1)
     // Task 2
     wait b; (1)
     wait a; (2)
  }
}
```

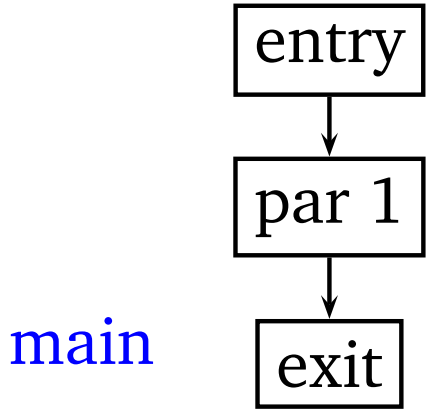
Automata

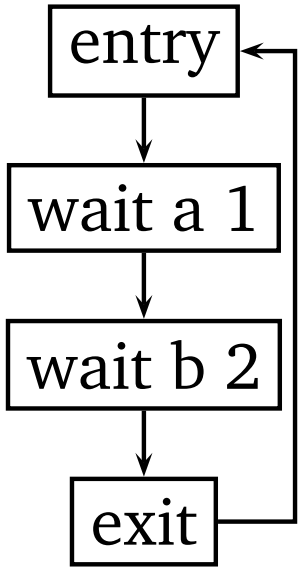
```
void main() {
 {
     // Task 1
     wait a; (1)
     wait b; (2)
 } par { (1)
     // Task 2
     wait b; (1)
     wait a; (2)
  }
}
```



Automata

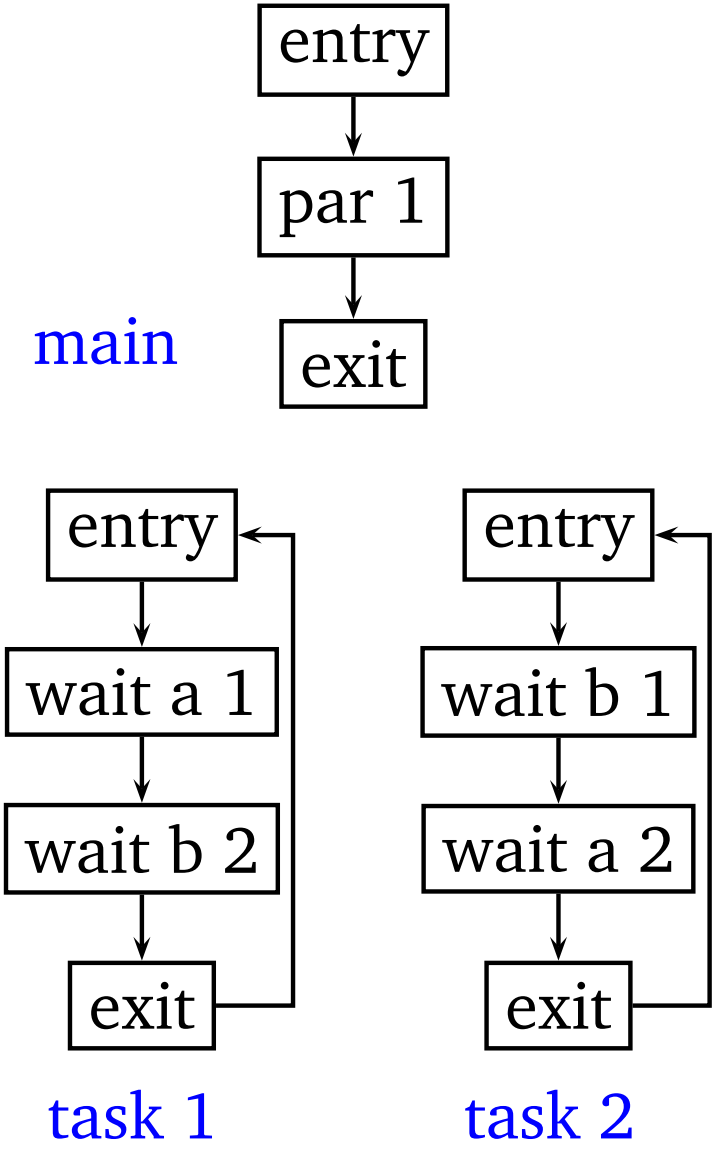
```
void main() {
 {
     // Task 1
     wait a; (1)
     wait b; (2)
 } par { (1)
     // Task 2
     wait b; (1)
     wait a; (2)
  }
}
```





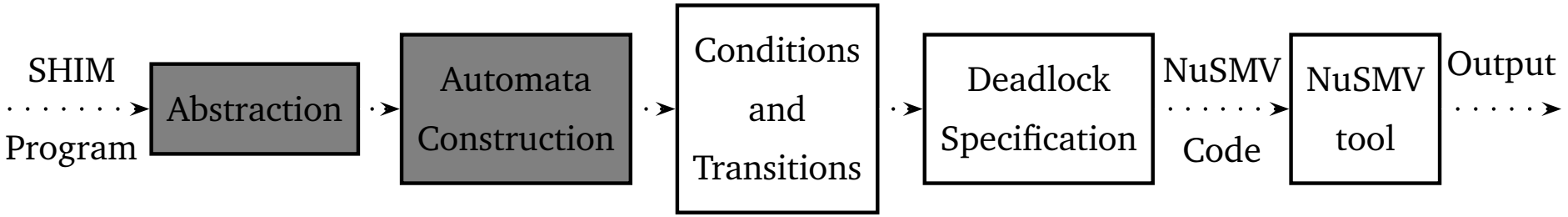
Automata

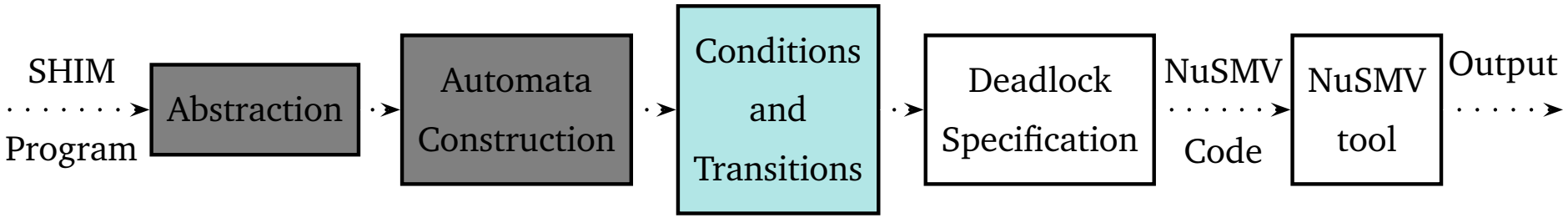
```
void main() {
 {
     // Task 1
     wait a; (1)
     wait b; (2)
 } par { (1)
     // Task 2
     wait b; (1)
     wait a; (2)
  }
}
```



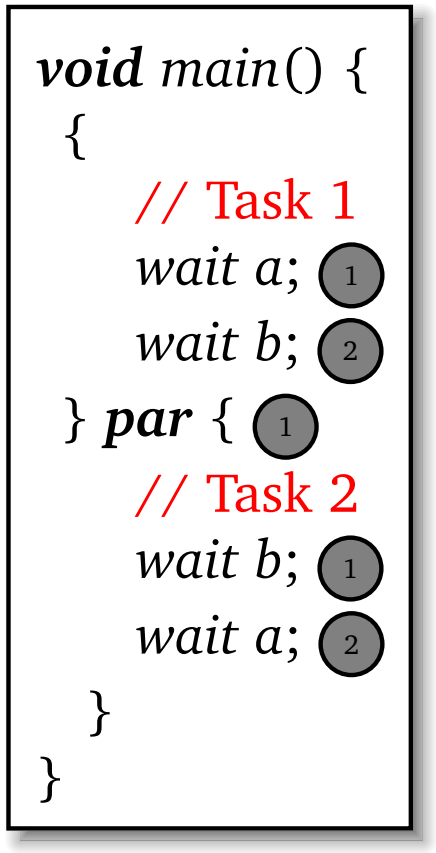
NuSMV

- A BDD and SAT based model checker.
- Conditions for transitions expressed as Boolean functions.
- Specifications can be expressed in Temporal Logic.
- We translate SHIM to NuSMV.




SHIM
Program
Abstraction
Automata
Construction
Conditions
and
Transitions
Deadlock
Specification
NuSMV
Code
NuSMV
tool
Output

Rendezvous Conditions



- *a* is connected to main, task_1, task_2;

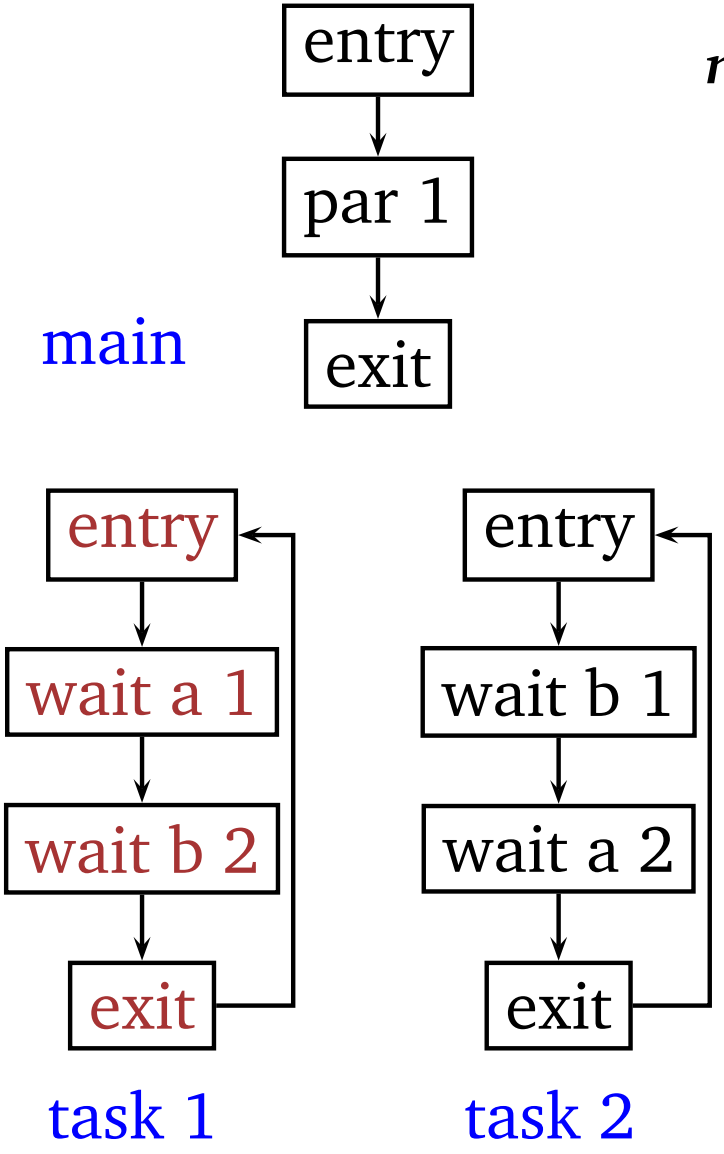
  ready_a :=

  main = *par_1* &

  task_1 = *wait_a_1* &

  task_2 = *wait_a_2*

Transitions



next(*task_1*) :=

case

(*task_1* = *entry*) & (*main* = *par_1*): *wait_a_1*;

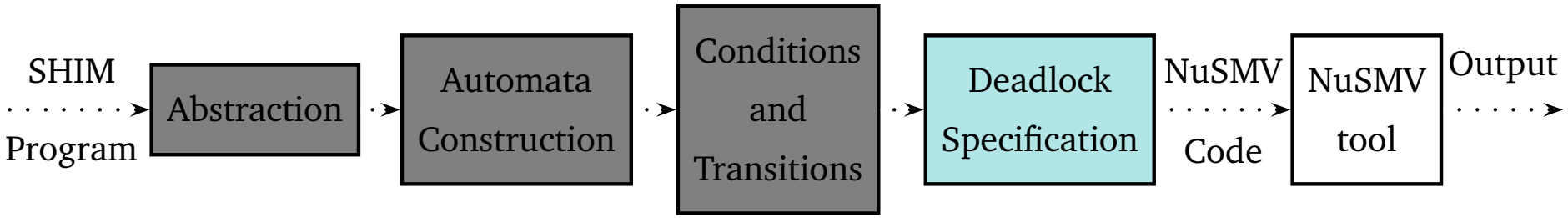
(*task_1* = *wait_a_1*) & *ready_a*: *wait_b_2*;

(*task_1* = *wait_b_2*) & *ready_b*: *exit*;

(*task_1* = *exit*) & (*task_2* = *exit*): *entry* ;

1: *task_1*;

esac;

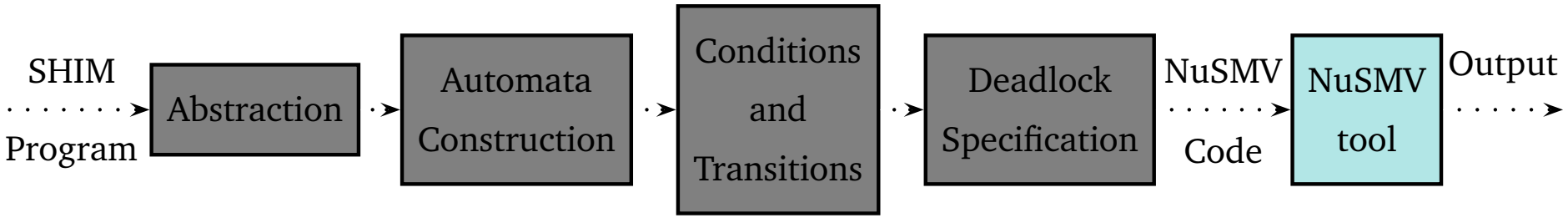

SHIM
Program
Abstraction
Automata
Construction
Conditions
and
Transitions
Deadlock
Specification
NuSMV
Code
NuSMV
tool
Output

Deadlock states

- Maintain a progress bit for each task.
- If no task makes any progress, then the program is in the deadlock state.

$$\textbf{\textit{SPEC AG}}((main != exit) \rightarrow (progress\_main = yes \mid progress\_task\_1 = yes \mid progress\_task\_2 = yes))$$

- Checking for absence of deadlock.

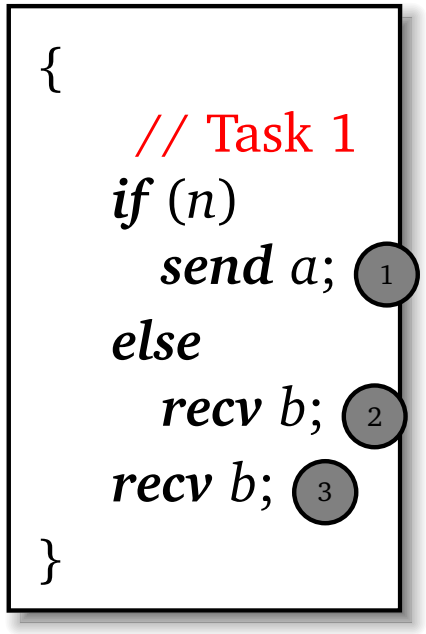

SHIM
Program
Abstraction
Automata
Construction
Conditions
and
Transitions
Deadlock
Specification
NuSMV
Code
NuSMV
tool
Output

NuSMV output

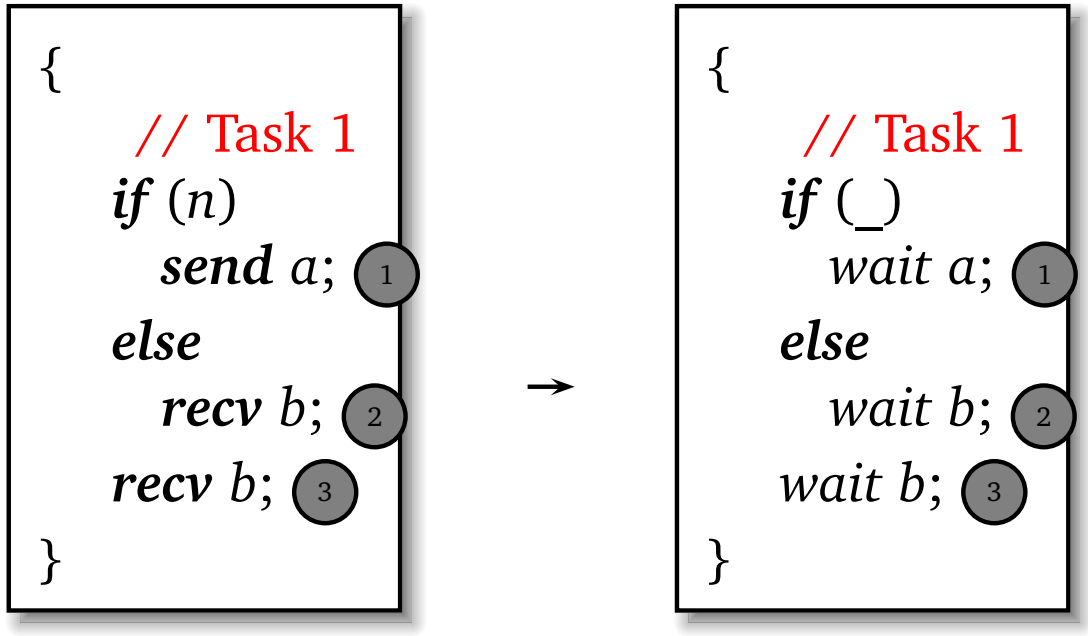
```
void main() {
 chan int a, b;
 {
     // Task 1
     a = 15, b = 10;
     send a;
     send b;
 } par {
     // Task 2
     int c;
     recv b;
     recv a;
     c = a + b;

   }
}
```

```
-- specification AG (main != exit ->
     ( progress_main = yes)
     | progress_task_1 = yes)
     | progress_task_2 = yes) is false
 ..
 ..
-> State: 1.2 <-
  progress_main = no
  task_2 = wait_b_1
  task_1 = wait_a_1
-> Input: 1.3 <-
-> State: 1.3 <-
  progress_task_2 = no
  progress_task_1 = no
```

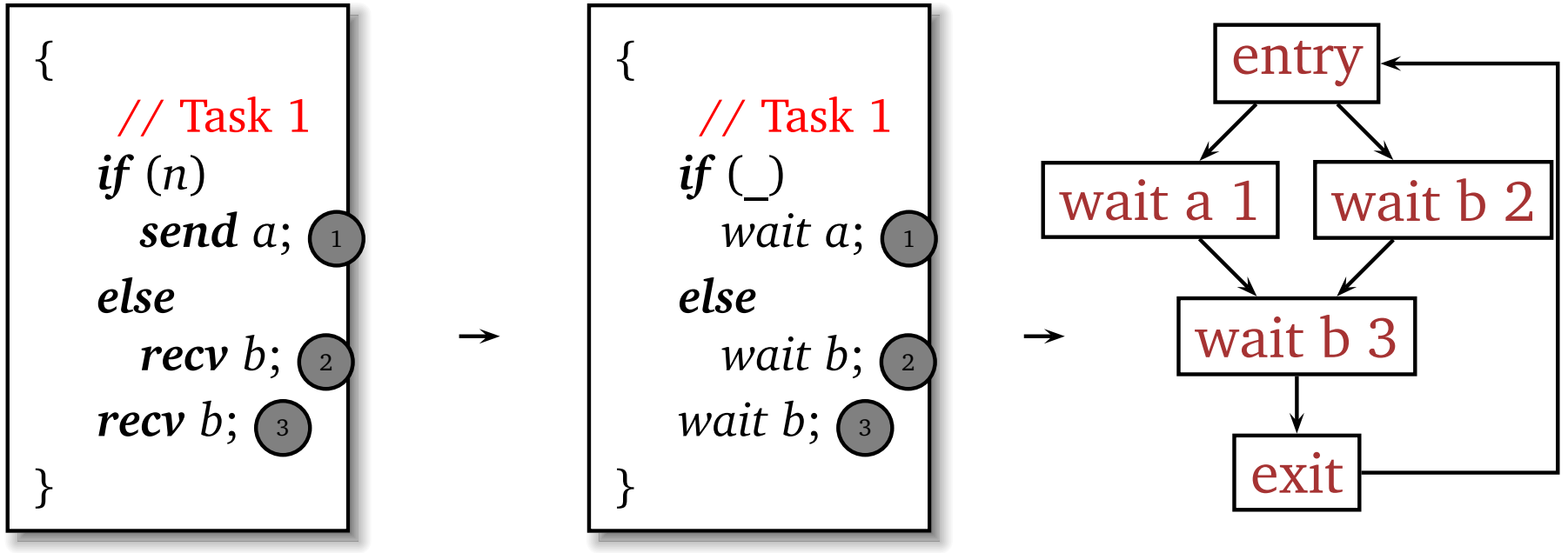
Conditional Statements



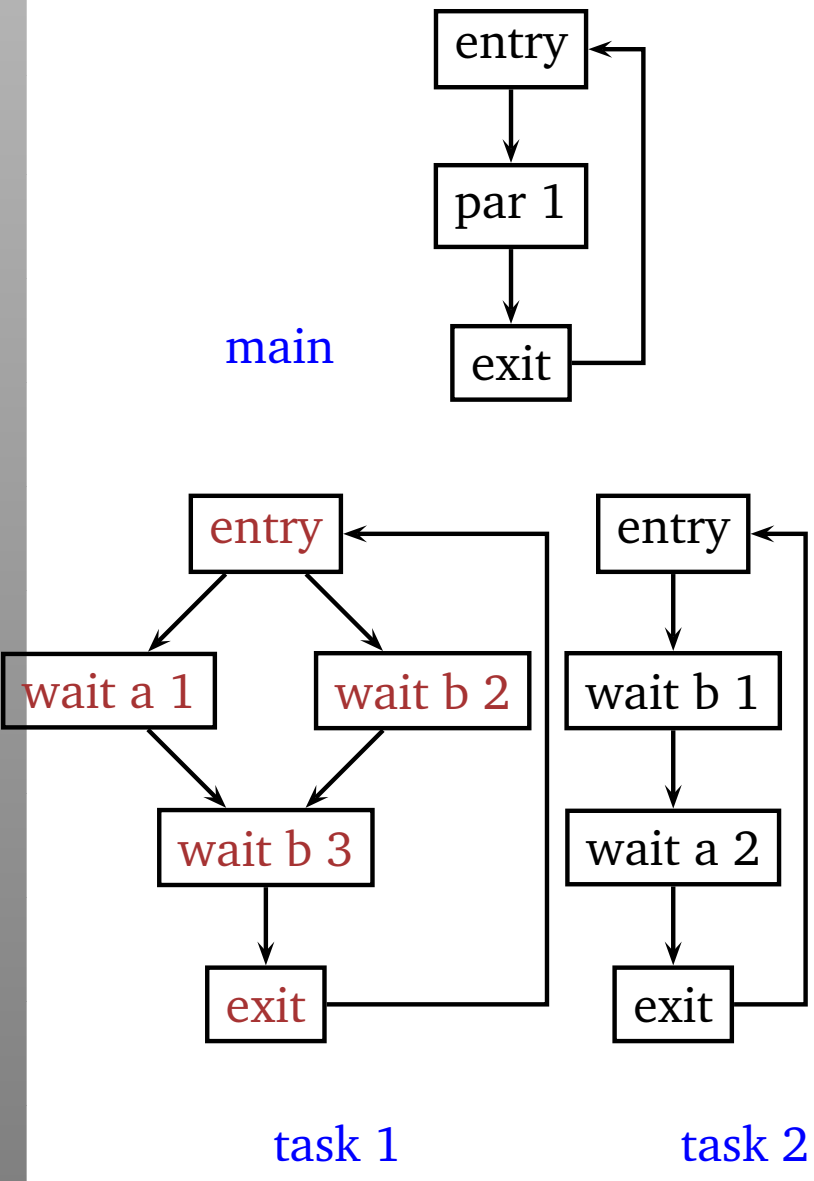
Conditional Statements



Conditional Statements



Conditional Statements



next(*task_1*) :=

case

(*task_1* = *entry*) & (*main* = *par_1*): {*wait_a_1*, *wait_b_2*};

(*task_1* = *wait_a_1*) & *ready_a*: *wait_b_3*;

(*task_1* = *wait_b_2*) & *ready_b*: *wait_b_3*;

..

..

esac;

Checking for deadlock

- Reports error if any path in the program deadlocks.

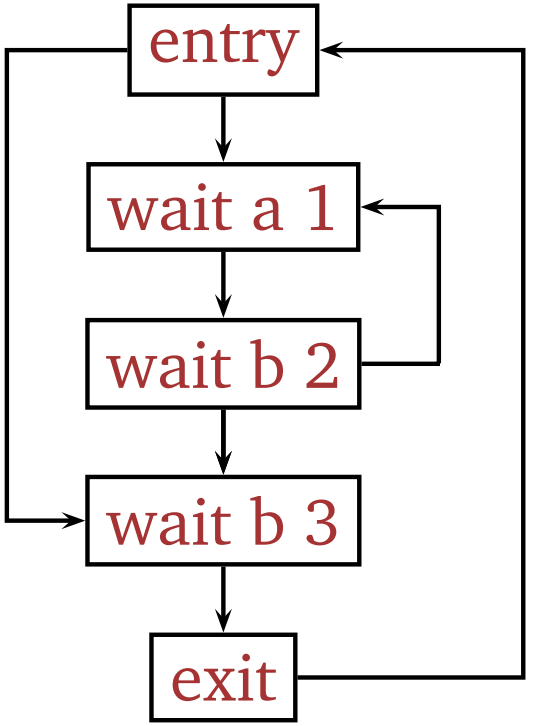
$$\textbf{\textit{SPEC AG}}((main != exit) \; -> \; (progress\_main = yes \; | \; progress\_task\_1 = yes \; | \; progress\_task\_2 = yes \;))$$

  Reports possibility of deadlock.

- May generate false-positives.

Loops

```
{
    // Task 1
   for(i = 0; i < n; i++ ) {
      send a; (1)
      recv b; (2)
   }
   recv b; (3)
}
```



Results

Example	Lines	Channels	Tasks	Result	Runtime	Memory
Source-Sink	35	2	11	No Deadlock	0.2 s	3.9 MB
Pipeline	30	7	13	No Deadlock	0.1	2.0
Prime Sieve	35	51	45	No Deadlock	1.7	25.4
Berkeley	40	3	11	No Deadlock	0.2	7.2
FIR Filter	100	28	28	No Deadlock	0.4	13.4
Bitonic Sort	130	65	167	No Deadlock	8.5	63.8
Framebuffer	220	11	12	No Deadlock	1.7	11.6
JPEG Decoder	1020	7	15	May Deadlock	0.9	85.6
JPEG Decoder Modified	1025	7	15	No Deadlock	0.9	85.6

Conclusions

- Using synchronous methodologies to verify asynchronous systems.
  - We do not need the power of SPIN.
- Modest sized programs verified in a few seconds.
- Future Work
  - Handle exceptions.
  - Convince the world: SHIM's philosophy- Deadlocks are better than data races.