Simulation-Based Verification of Hardware with Strategies

Michael Katelman and José Meseguer

University of Illinois at Urbana-Champaign

September 17, 2010

"Implied needs are in: (1) verification, which is a bottleneck that has now reached crisis proportions . . . "

". . . due to the growing complexity of silicon designs, functional verification is still an unresolved challenge, defeating the enormous effort put forth by armies of verification engineers and academic research efforts."

"Multiple sources report that in current development projects verification engineers outnumber designers, with this ratio reaching two to one for the most complex designs."

[ITRS 09]

one approach to mitigating the verification burden: help verification engineers be more productive by providing a better programming language to work in

- strategy paradigm

  programmatically coordinate multiple simulations in unison
- `vlogsl`

  an EDSL in Haskell supporting the strategy paradigm

1. language-level approach, justification
2. strategy paradigm, definition and motivation
3. `vlogsl` architecture
4. maze examples
5. concluding remarks

assembly

```
pushl %ebp
movl  %esp,%ebp
```

assembly  C, Fortran, etc.

```
pushl %ebp
movl  %esp,%ebp
```

```
int sum(int x, int y){
  printf("x+y=%d", x+y);
  return x+y;
}
```

assembly → C, Fortran, etc. → C++, Java, Haskell, etc.

```
pushl %ebp
movl  %esp,%ebp
```

```
int sum(int x, int y){
  printf("x+y=%d", x+y);
  return x+y;
}
```

```
sum = foldr (+) 0
```

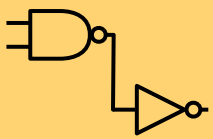
assembly → C, Fortran, etc. → C++, Java, Haskell, etc.

```
pushl %ebp
movl  %esp,%ebp
```

```
int sum(int x, int y){
  printf("x+y=%d", x+y);
  return x+y;
}
```

```
sum = foldr (+) 0
```

schematic capture



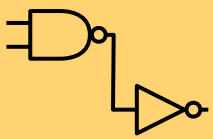
assembly → C, Fortran, etc. → C++, Java, Haskell, etc.

```
pushl %ebp
movl  %esp,%ebp
```

```
int sum(int x, int y){
  printf("x+y=%d", x+y);
  return x+y;
}
```

```
sum = foldr (+) 0
```

schematic capture → Verilog, VHDL



```
always @(posedge clk)
  ctr1 <= ctr1+1;
```

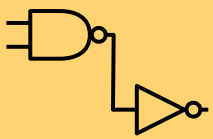
assembly → C, Fortran, etc. → C++, Java, Haskell, etc.

```
pushl %ebp
movl  %esp,%ebp
```

```
int sum(int x, int y){
  printf("x+y=%d", x+y);
  return x+y;
}
```

```
sum = foldr (+) 0
```

schematic capture → Verilog, VHDL → SystemC, Bluespec, etc.



```
always @(posedge clk)
  ctr1 <= ctr1+1;
```

```
FIFO#(Bit#(8)) q <- mkFIFO;

rule compute;
  let x = q.first();
```

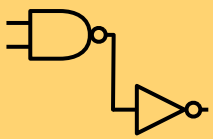
assembly → C, Fortran, etc. → C++, Java, Haskell, etc.

```
pushl %ebp
movl  %esp,%ebp
```

```
int sum(int x, int y){
  printf("x+y=%d", x+y);
  return x+y;
}
```

```
sum = foldr (+) 0
```

schematic capture → Verilog, VHDL → SystemC, Bluespec, etc.



```
always @(posedge clk)
  ctr1 <= ctr1+1;
```

```
FIFO#(Bit#(8)) q <- mkFIFO;

rule compute;
  let x = q.first();
```

Verilog, VHDL

```
adr = 7'h01;
dat = 8'hFF;
```

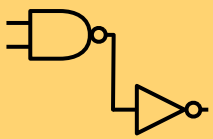
assembly → C, Fortran, etc. → C++, Java, Haskell, etc.

```
pushl %ebp
movl  %esp,%ebp
```

```
int sum(int x, int y){
  printf("x+y=%d", x+y);
  return x+y;
}
```

```
sum = foldr (+) 0
```

schematic capture → Verilog, VHDL → SystemC, Bluespec, etc.



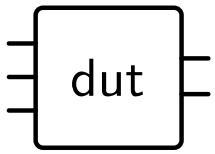
```
always @(posedge clk)
  ctr1 <= ctr1+1;
```

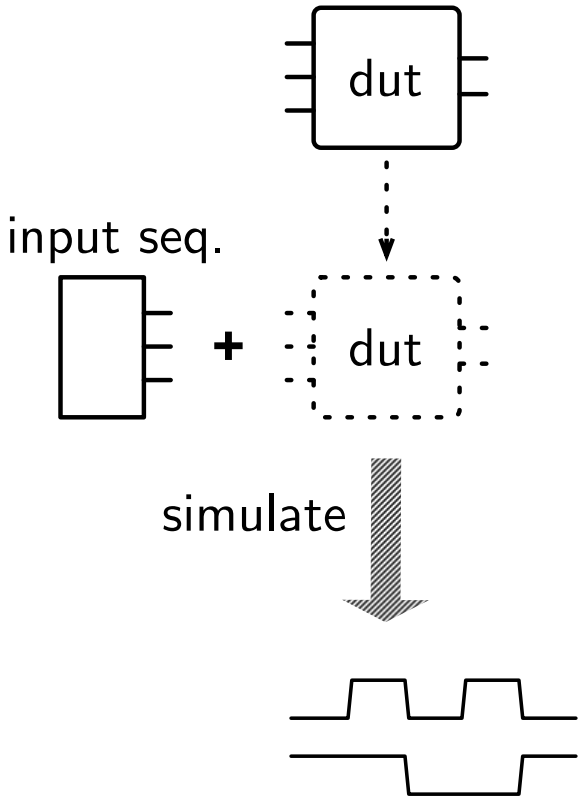
```
FIFO#(Bit#(8)) q <- mkFIFO;

rule compute;
  let x = q.first();
```

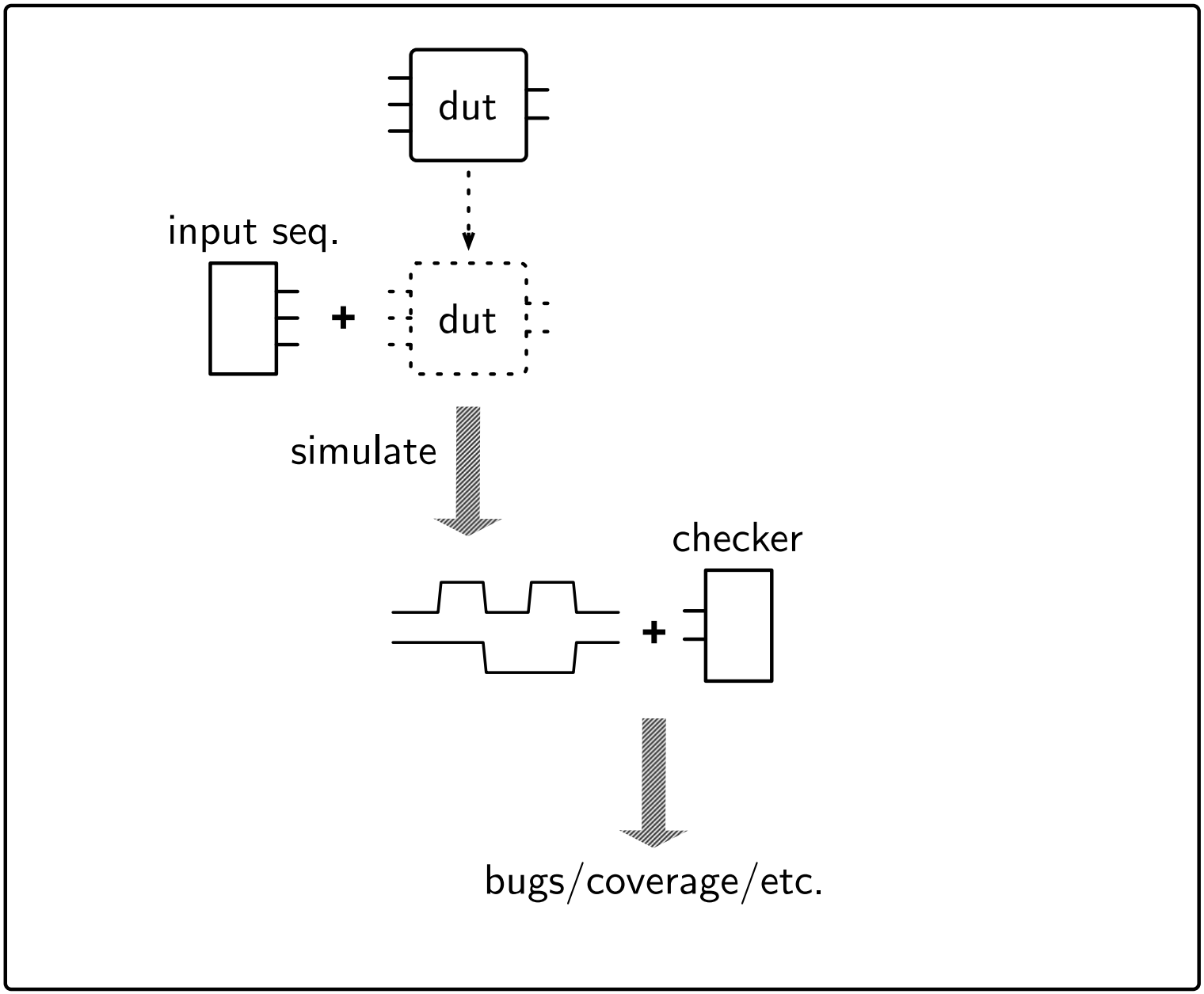
Verilog, VHDL → SystemVerilog, e, etc.

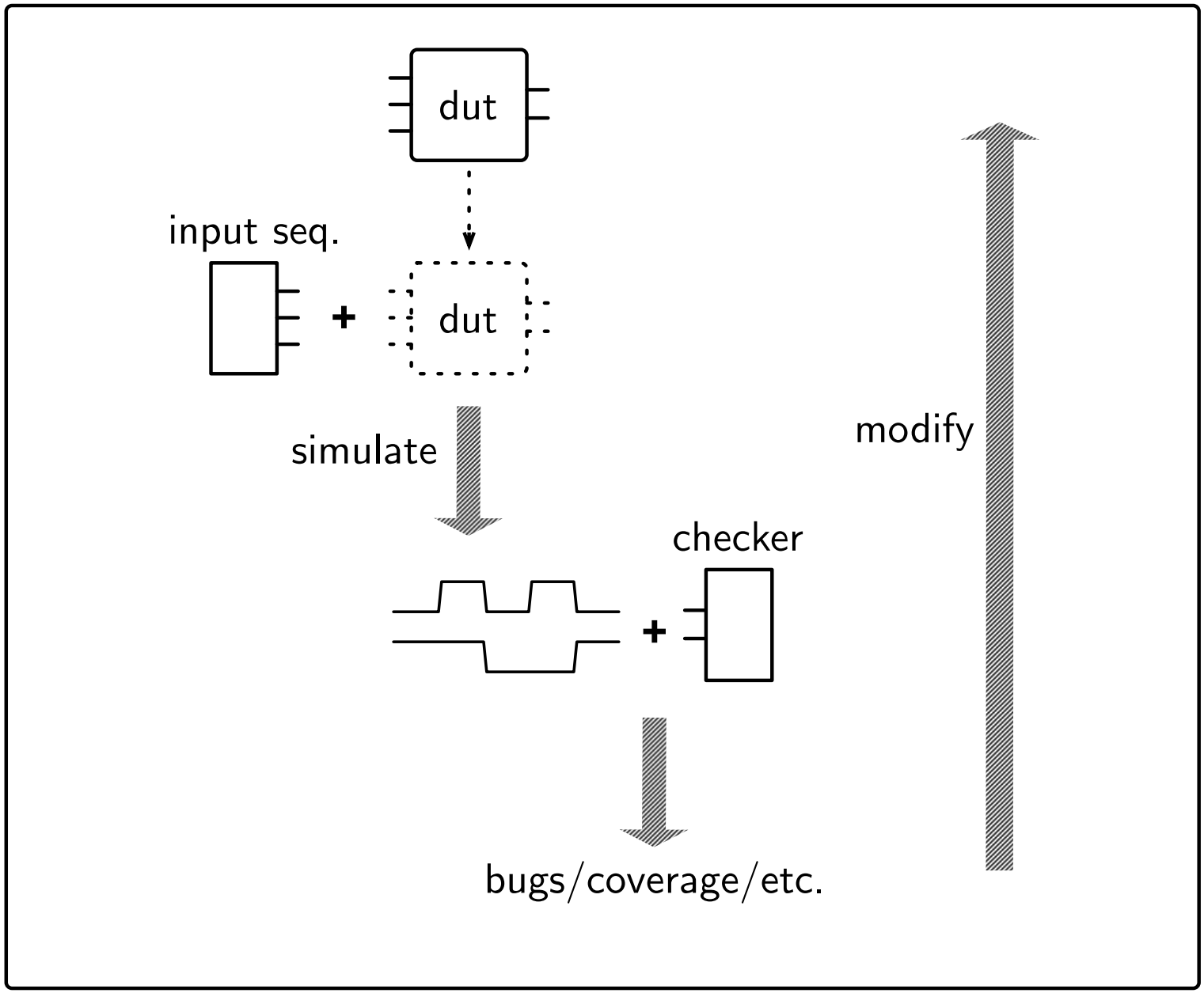
```
adr = 7'h01;
dat = 8'hFF;
```

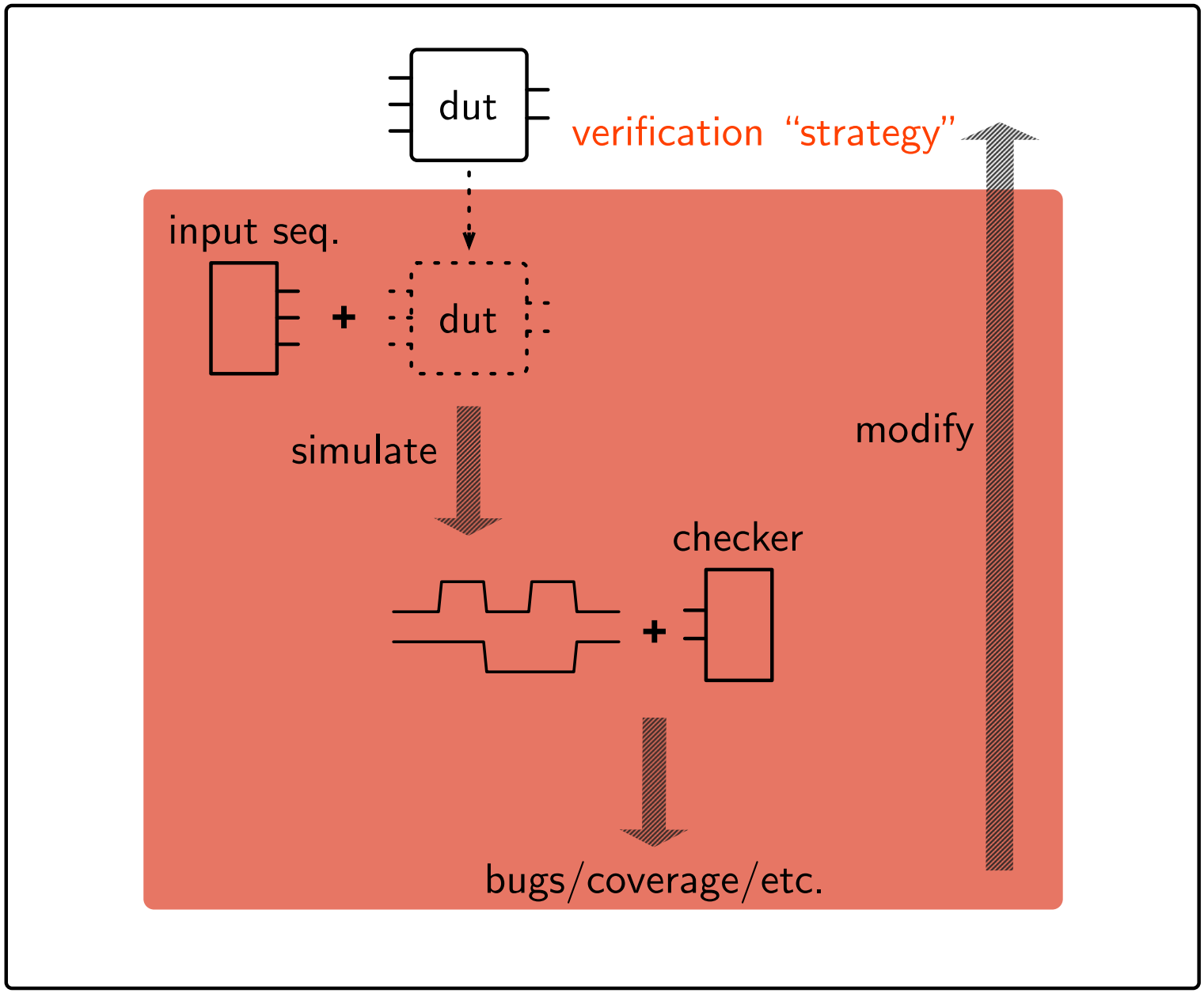
```
class BusWrite;
  rand bit [6:0] adr;
  rand bit [7:0] dat;
```

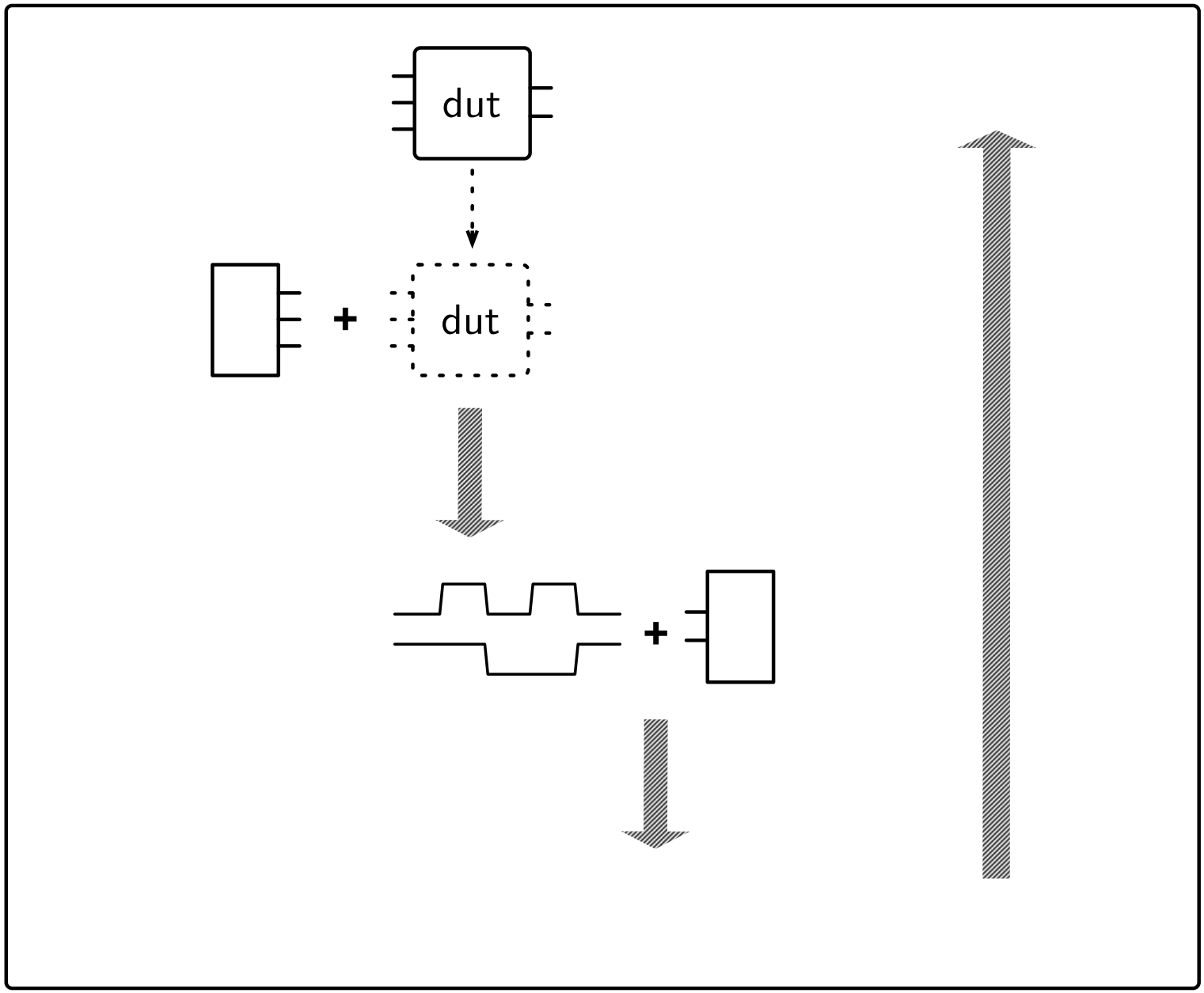

dut

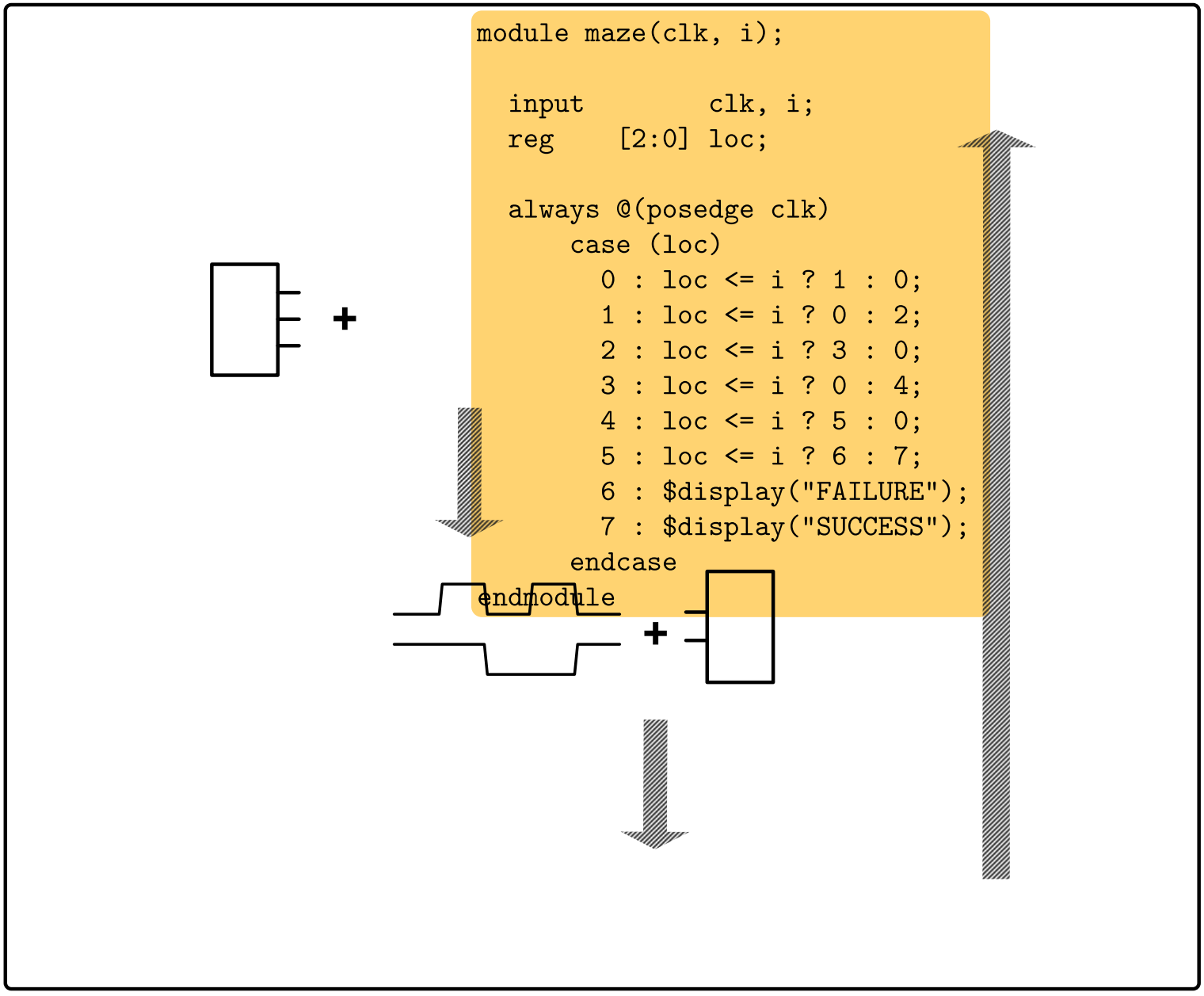

dut
input seq.
+
dut
simulate


dut
input seq.
+
dut
simulate
checker
+
bugs/coverage/etc.


dut
input seq.
+
dut
simulate
checker
+
modify
bugs/coverage/etc.

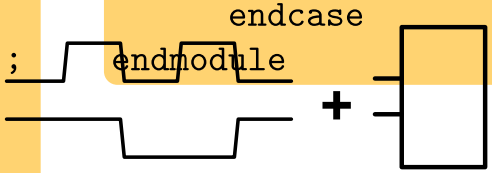

dut
verification "strategy"
input seq.
+
dut
simulate
modify
checker
+
bugs/coverage/etc.


dut
+
dut
+


module maze(clk, i);
input clk, i;
reg [2:0] loc;
always @(posedge clk)
case (loc)
0 : loc <= i ? 1 : 0;
1 : loc <= i ? 0 : 2;
2 : loc <= i ? 3 : 0;
3 : loc <= i ? 0 : 4;
4 : loc <= i ? 5 : 0;
5 : loc <= i ? 6 : 7;
6 : $display("FAILURE");
7 : $display("SUCCESS");
endcase
endmodule

```
class Bit;
  rand bit val;
endclass


  maze m(clk, i);

  initial
  begin
    clk = 0; i = 0;
    repeat (10)
    begin
      @(posedge clk);
      x.randomize;
      i = #1 x.val;
    end
    $finish;
  end
```

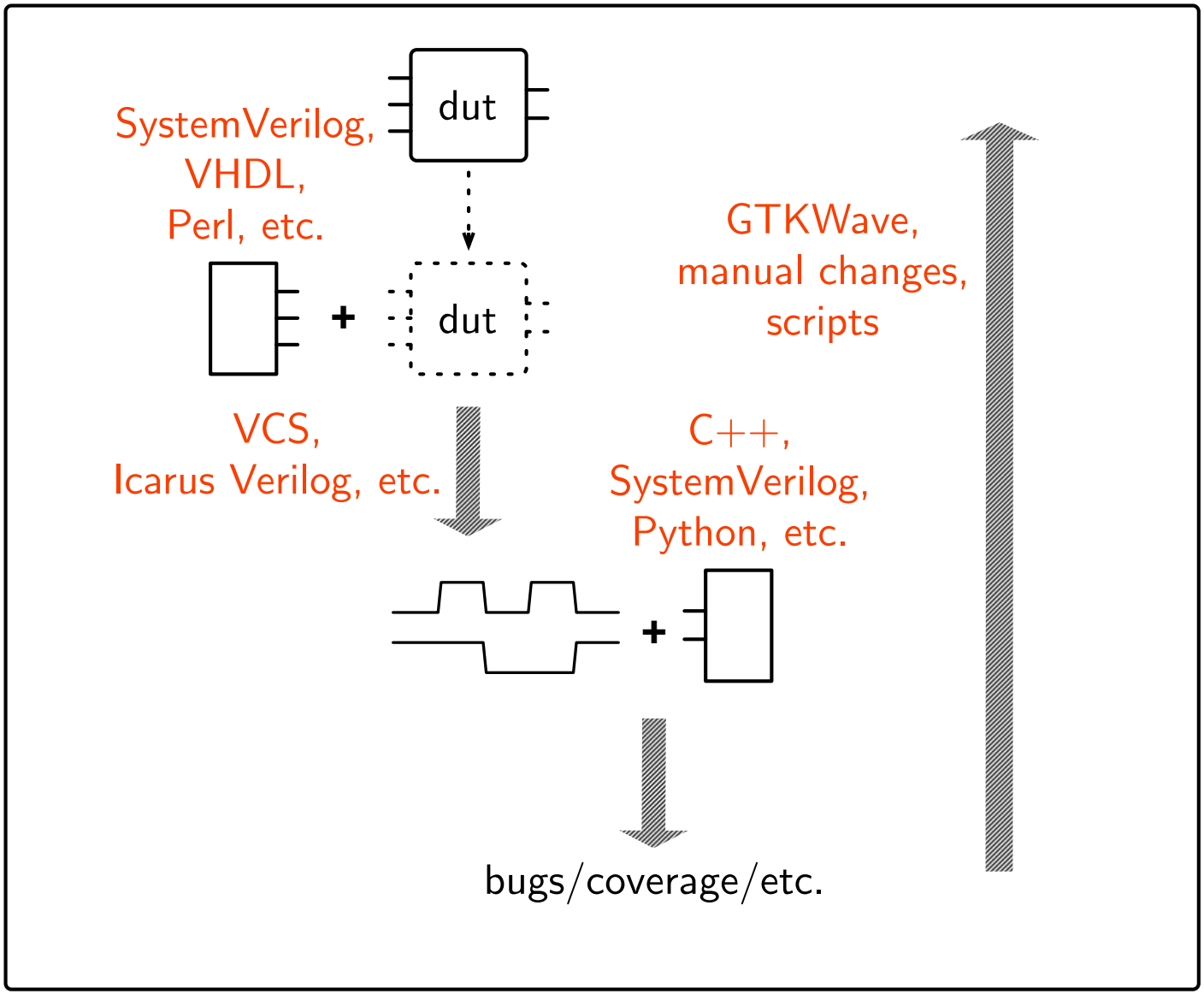
```
module maze(clk, i);

  input        clk, i;
  reg    [2:0] loc;

  always @(posedge clk)
      case (loc)
        0 : loc <= i ? 1 : 0;
        1 : loc <= i ? 0 : 2;
        2 : loc <= i ? 3 : 0;
        3 : loc <= i ? 0 : 4;
        4 : loc <= i ? 5 : 0;
        5 : loc <= i ? 6 : 7;
        6 : $display("FAILURE");
        7 : $display("SUCCESS");
      endcase
endmodule
```

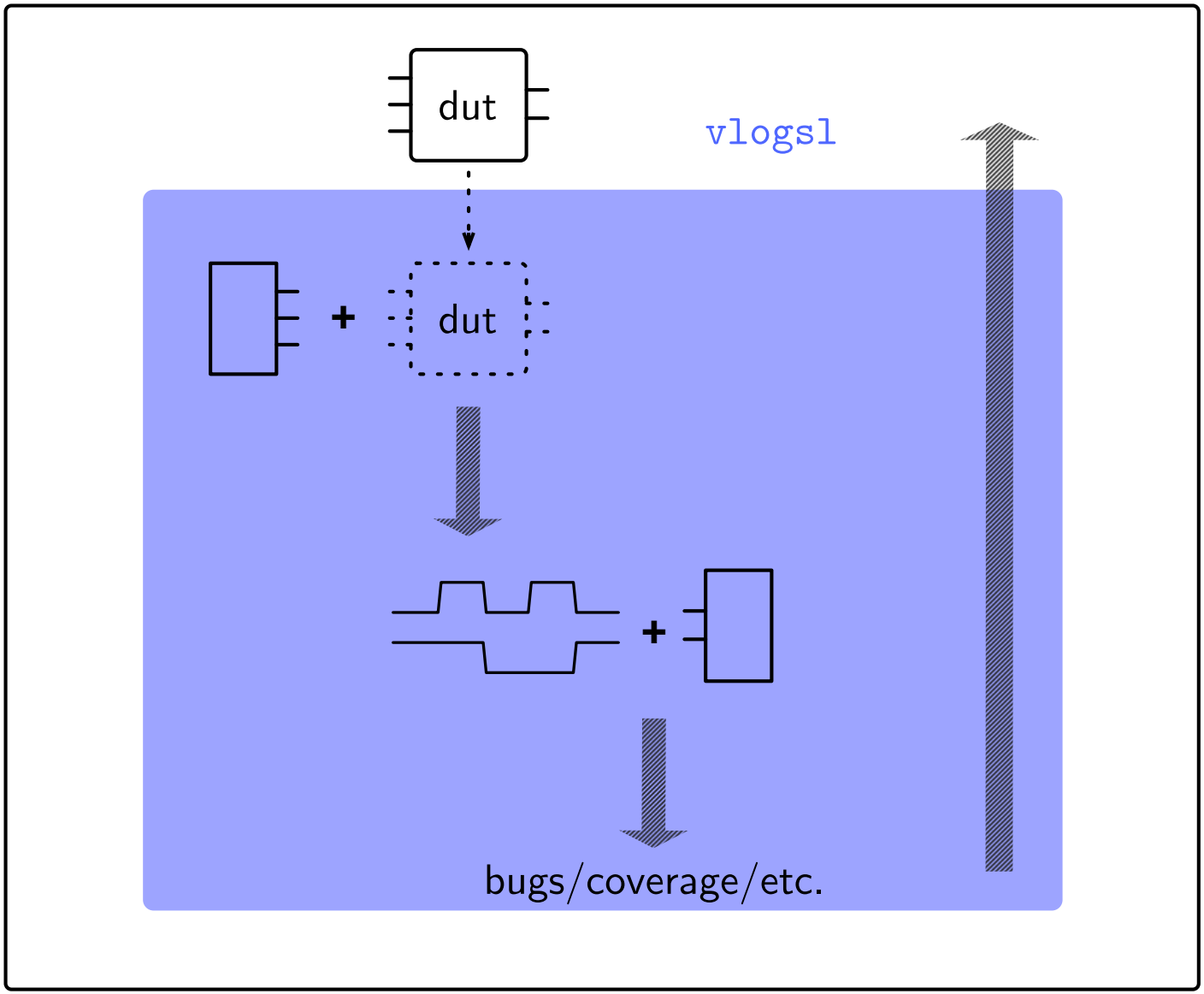


```
class Bit;
  rand bit val;
endclass


  maze m(clk, i);

  initial
  begin
    clk = 0; i = 0;
    repeat (10)
    begin
      @(posedge clk);
      x.randomize;
      i = #1 x.val;
    end
    $finish;
  end
```

```
module maze(clk, i);

  input        clk, i;
  reg    [2:0] loc;

  always @(posedge clk)
      case (loc)
        0 : loc <= i ? 1 : 0;
        1 : loc <= i ? 0 : 2;
        2 : loc <= i ? 3 : 0;
        3 : loc <= i ? 0 : 4;
        4 : loc <= i ? 5 : 0;
        5 : loc <= i ? 6 : 7;
        6 : $display("FAILURE");
        7 : $display("SUCCESS");
      endcase
endmodule
```

```
vcs -full64 -sverilog input.sv maze.v

for j in {1..100} ; do
  ./simv +ntb_random_seed=$j | grep -q SUCCESS
  if [ $? -eq 0 ] ; then
    echo "succeeded at $j."
    exit
  fi
done ; echo "failed all 100."
```


SystemVerilog,
VHDL,
Perl, etc.
dut
dut
VCS,
Icarus Verilog, etc.
C++,
SystemVerilog,
Python, etc.
GTKWave,
manual changes,
scripts
bugs/coverage/etc.

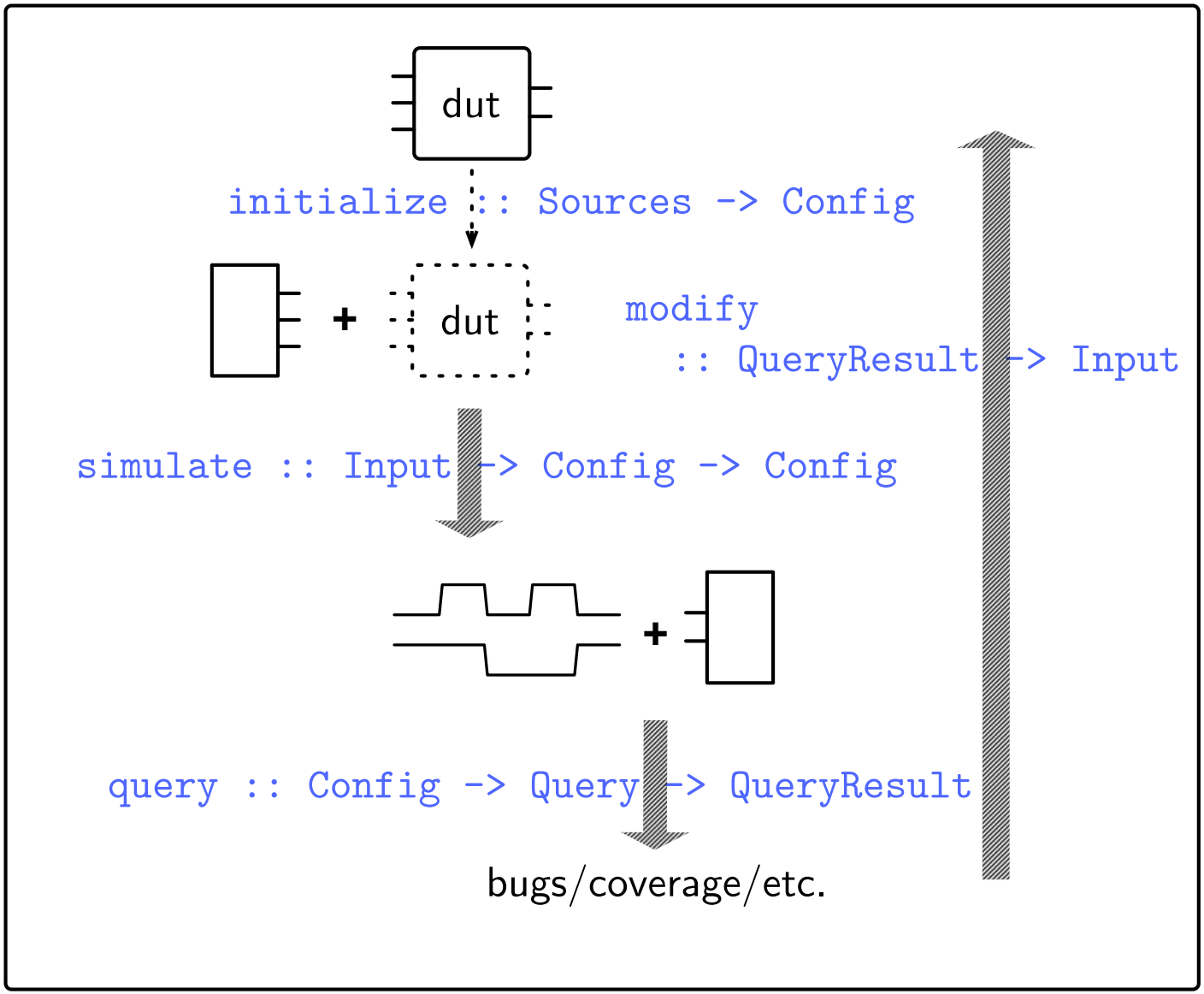

dut
vlogsl
+
dut
+
bugs/coverage/etc.

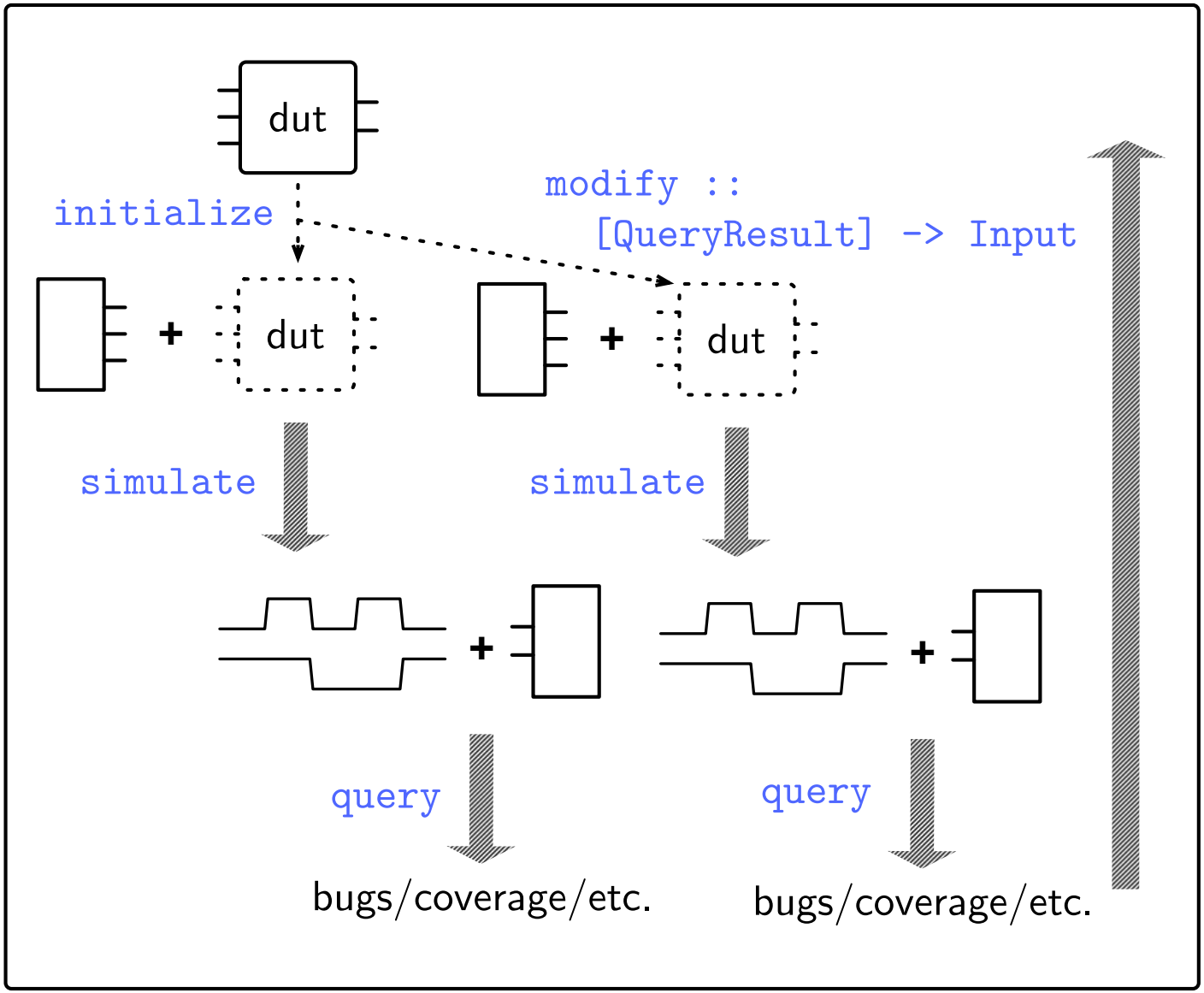
a data type called `Config` *is the linchpin of* `vlogsl`*; it provides a first-class representation of a Verilog device*

- event queues: active, non-blocking, future, etc.
- current and past values of all source-level nodes
- top-level input, clock, and reset names

`vlogsl` *has three basic classes of functions that rely on the configuration data type*

```
initialize :: Sources             -> Config
simulate   :: Input   -> Config -> Config
query      :: Query   -> Config -> QueryResult
```


dut
initialize :: Sources -> Config
+
dut
modify
:: QueryResult -> Input
simulate :: Input -> Config -> Config
+
query :: Config -> Query -> QueryResult
bugs/coverage/etc.


dut
initialize
modify ::
[QueryResult] -> Input
dut
dut
simulate
simulate
query
query
bugs/coverage/etc.
bugs/coverage/etc.

```
main = do
  dut :: Config <- initialize dev
```

.

```
main = do
  dut :: Config <- initialize dev




trial = do
  simulateR (unspecified 10)
  query ("loc" `eq` 7)




.
```

```
main = do
  dut :: Config <- initialize dev




trial = do
  simulateR (unspecified 10)
  query ("loc" `eq` 7)

aux 100 simv = return Nothing
aux j   simv = do
  result <- simv
  if result
    then return (Just j)
    else aux (j+1) simv



.
```

```
main = do
  dut :: Config <- initialize dev









trial = do
  simulateR (unspecified 10)
  query ("loc" `eq` 7)

aux 100 simv = return Nothing
aux j   simv = do
  result <- simv
  if result
    then return (Just j)
    else aux (j+1) simv

reportResult x =
  case x of
    Nothing -> putStrLn "failed all 100"
    Just j  -> putStrLn ("succeeded at " ++ show j)
```

```
main = do
  dut :: Config <- initialize dev
  res <- runStrat strategy dut
  reportResult res

strategy = do
  dut <- get
  aux 0 (runStrat' trial dut)

trial = do
  simulateR (unspecified 10)
  query ("loc" `eq` 7)

aux 100 simv = return Nothing
aux j   simv = do
  result <- simv
  if result
    then return (Just j)
    else aux (j+1) simv

reportResult x =
  case x of
    Nothing -> putStrLn "failed all 100"
    Just j  -> putStrLn ("succeeded at " ++ show j)
```

```
strategy :: Strat (Maybe Config)
strategy = do
  simulate (unspecified 10)
  x <- querySMT 5 success
  case x of
    Nothing    -> return Nothing
    Just subst -> do
          cnfg <- applyM subst
          return (Just cnfg)

success = "loc" `eq` 7
```

```
strategy :: Strat (Maybe Config)
strategy = aux 0

aux 100 = return Nothing
aux j   = do
  simulateR (unspecified 5)
  failCnfg <- get
  simulate  (unspecified 5)
  x <- querySMT 5 success
  case x of
    Nothing    -> do
      put failCnfg
      aux (j+1)
    Just subst -> do
      succCnfg <- applyM subst
      return (Just succCnfg)

success = "loc" `eq` 7
```

```
strategy :: Strat (Maybe Config)
strategy = do
  xs   <- runContT (callCC $ \exit -> btSearch [] exit) return
  cnfg <- get
  if 7 `elem` xs
    then return (Just cnfg)
    else return Nothing

btSearch xs exit = do
  x <- lift (intValOfId locId)
  when (x == 7) (exit (x:xs))
  if x `elem` xs
    then return xs
    else do
      lift (simulate i1)
      ys <- saveAndRestore (btSearch (x:xs) exit)
      lift (simulate i0)
      btSearch ys exit
```

Intel Xeon X5570 (2.93GHz, 8MB L3, Nehalem), 24 GB RAM, Linux kernel 2.6.18, 64-bit. ghc 6.10.4 with -O1. VCS-MX D-2009.12_Full64.

random	0.070s
symbolic	0.018s
mixed	0.015s
backtrack	0.013s
script	7.111s

summary and future work:

- strategy paradigm
- `vlogsl`
- novel use cases
- `vlogsl`
- large case study