A Meta-Language for Hardware Testbench

Michael Katelman and José Meseguer

University of Illinois at Urbana-Champaign

March 21, 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 2009)

some things that designers do:

- *code RTL*
- *optimization*
- *rudimentary testing*

some things that verification engineers do:

- *write testbenches*
- *write checkers*
- *measure coverage*
- *file bug reports*

some things that verification engineers do:

- *write testbenches*
- *write checkers*
- *measure coverage*
- *file bug reports*

about 95% *of bugs are found through simulation* (ITRS 2009)

goal: make the life of a verification engineer better.

some possibilities:

- *smarter testing algorithms* (Magellan, DeNibulator)
- *language-level improvements* (OOP, temporal assertions)
- *management tools* (coverage statistics)

our approach: a new language for building testbenches

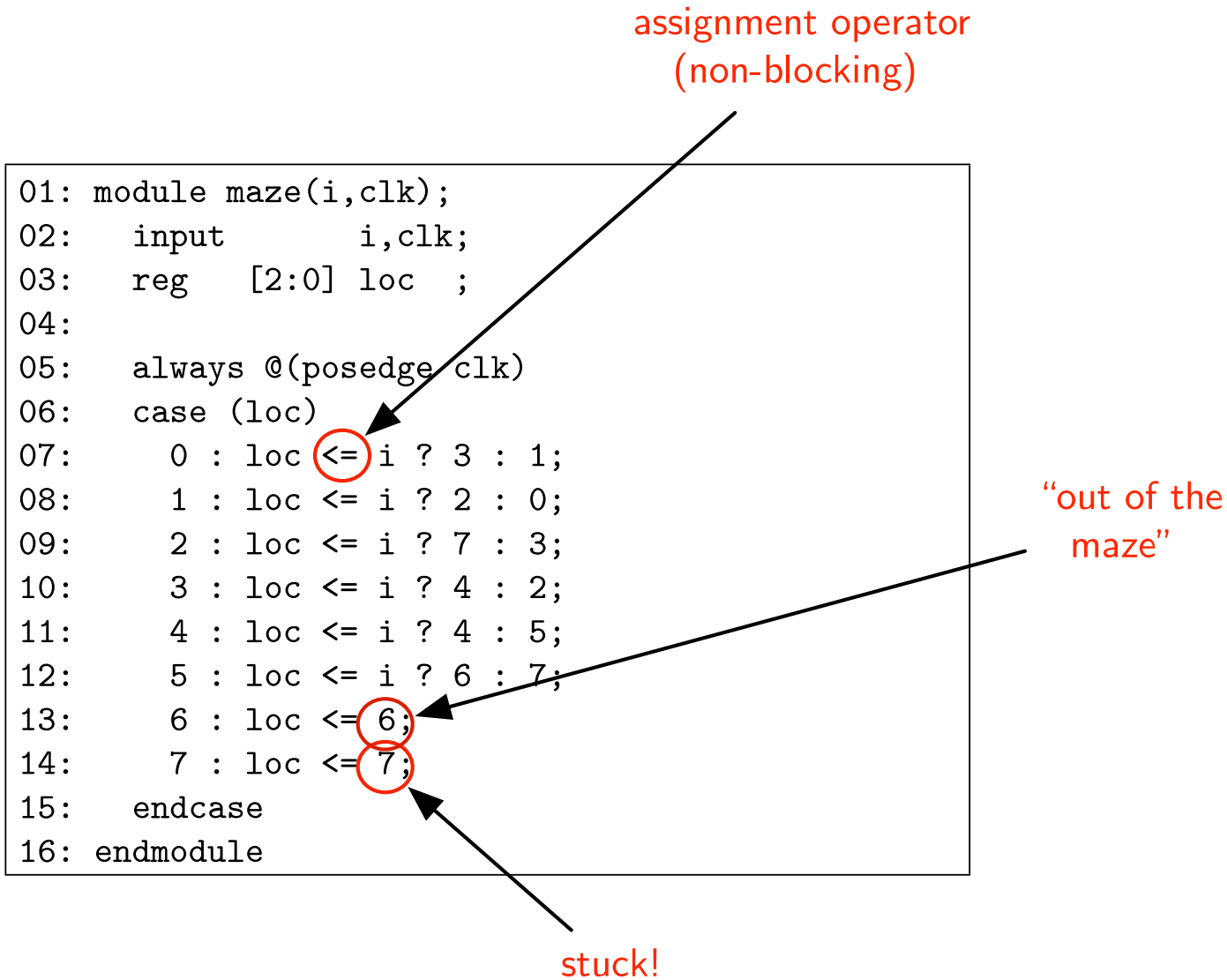
our approach: a new language for building testbenches

- *currently: testbench is an environment for DUT*

our approach: a new language for building testbenches

- *currently: testbench is an environment for DUT*
- *our language: testbench operates at a higher-level*
  - *testbench = meta-program analyzing DUT via simulation*
  - *DUT-level simulation = programming facility*

a Verilog module:

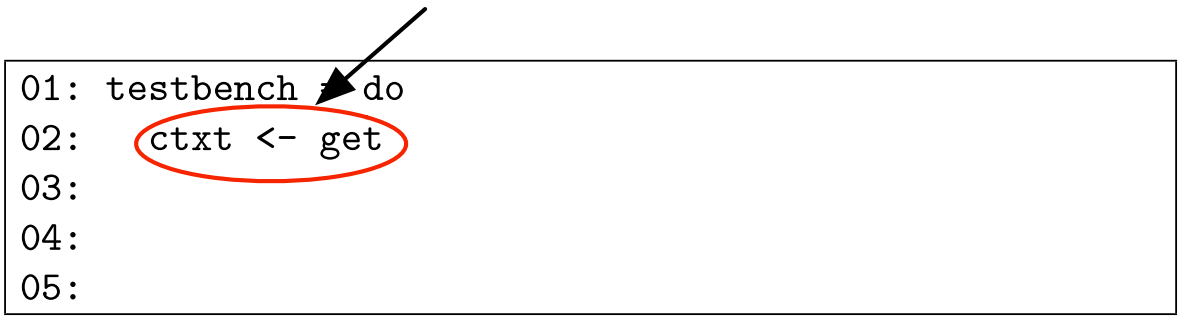


a testbench in our meta-language:

```
01: testbench = do
02:
03:
04:
05:
```

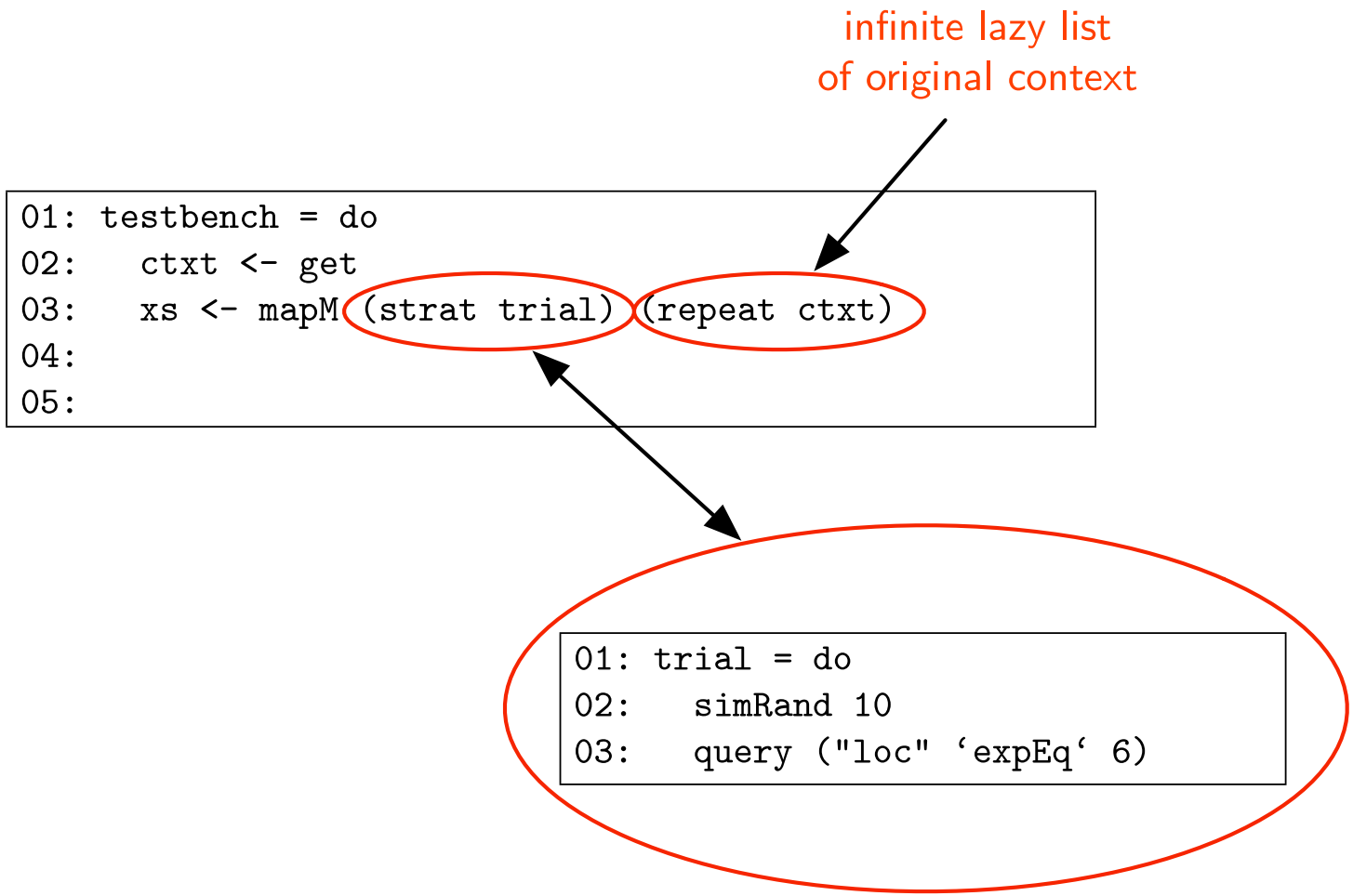
a testbench in our meta-language:

the current simulation context
as a (first-class) data value

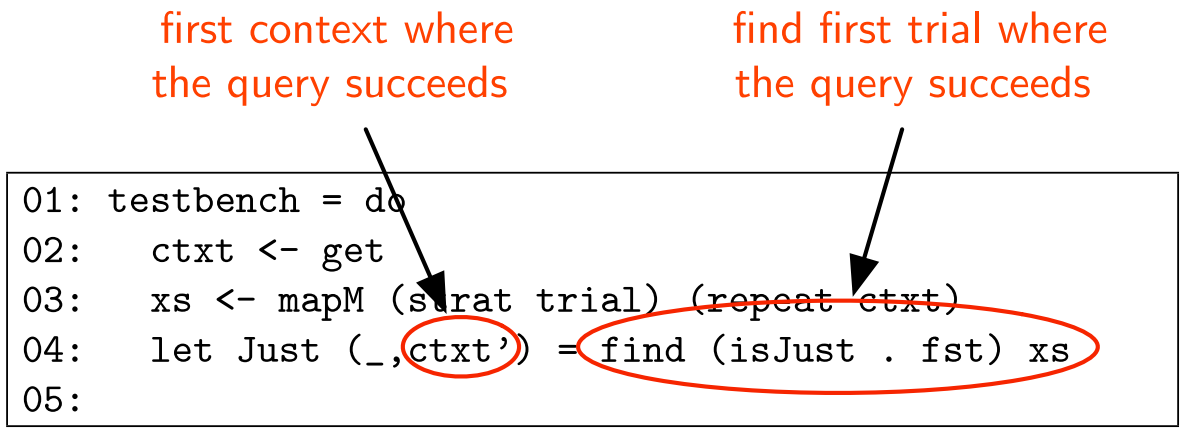


```
01: testbench s do
02:   ctxt <- get
03:
04:
05:
```

a testbench in our meta-language:



a testbench in our meta-language:




```
01: trial = do
02:   simRand 10
03:   query ("loc" `expEq` 6)
```

a testbench in our meta-language:

proceed with the context found

```
01: testbench = do
02:   ctxt <- get
03:   xs <- mapM (strat trial) (repeat ctxt)
04:   let Just (_,ctxt') = find (isJust . fst) xs
05:   put ctxt'
```

```
01: trial = do
02:   simRand 10
03:   query ("loc" `expEq` 6)
```

our motivation, by way of analogy: from *Edinburgh LCF* (Gordon, Milner, and Wadsworth; 1979):

> *"Two extreme styles of doing proofs on a computer have been explored rather thoroughly in the past."*

- > *"The first is 'automatic theorem proving'; typically a general proof-finding strategy is programmed, and the user's part is confined to first submitting some axioms and a formula to be proved, secondly (perhaps) adjusting some parameters of the strategy to control its method of search, and thirdly (perhaps) responding to requests for help from the system during its search for a proof."*

substitute constrained randoms for "automatic theorem proving"

- *"The second style is 'proof checking'; here the user provides an alleged proof, step by step, and the machine checks each step. In the most extreme form of proof checking each step consists in the application of a primitive rule of inference, though many proof checking systems allow complex inferences (e.g. simplification of logical formulae) to occur at one step. One feature of this style is that the proof is conducted forwards, from axioms to theorem . . . "*

substitute directed testing for "proof checking"

"There are no doubt many ways of compromising between these two styles, in an attempt to eliminate the worst features of each - e.g. the inefficient general search strategies of automatic theorem provers, and the tedious and repetitive nature of straight proof checking."

just as was the case for LCF, we want to find some middle ground

"There are no doubt many ways of compromising between these two styles, in an attempt to eliminate the worst features of each - e.g. the inefficient general search strategies of automatic theorem provers, and the tedious and repetitive nature of straight proof checking."

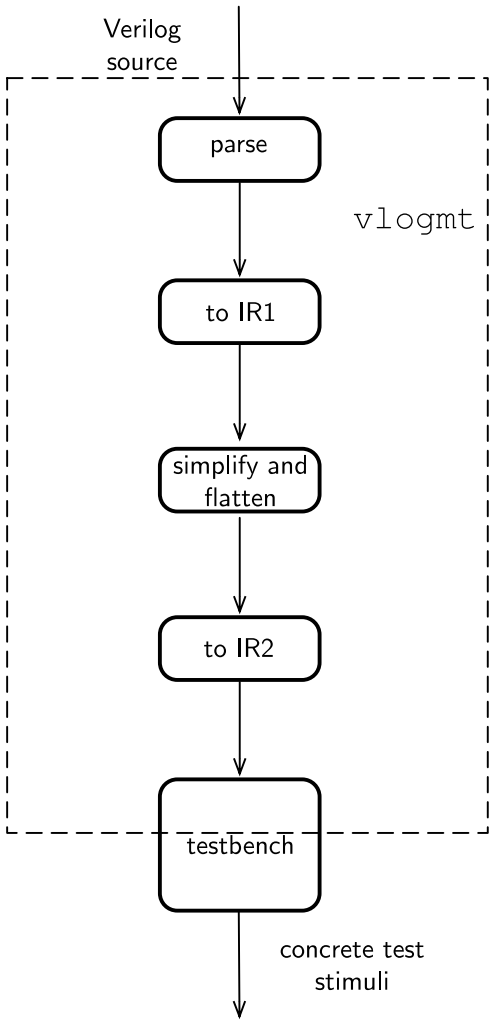
just as was the case for LCF, we want to find some middle ground

solution for LCF: ML

main features of our meta-language:

- *simulation context of design under test is a first-class object*
- *simulation context is symbolic*
- *simulation = function from sim. context to sim. context*
- *integration with very general, efficient bit-level SMT solver*
- *embedded in a high-level declarative language*

our tool: `vlogmt`, a work in progress



the primary data-type is a "state-monad"

```
type (VSI a) = (StateT Context IO a)
```

just (essentially) a *function from contexts to contexts*

```
Context -> IO (a,Context)
```

configuration: information needed to carry out Verilog simulation

```
data Context = Ctxt {
        clk :: identifier,
      guard :: symbolic expression,
      state :: identifier → symbolic expression,
    activeQ :: list of processes,
  waitingQ :: list of processes,
     inputs :: list of identifiers,
     ...
}
```

key operation: symbolic simulation

```
delta :: VSI Int
delta = do
  done <- epsilon
  if not done
    then delta
    else tickClock
```

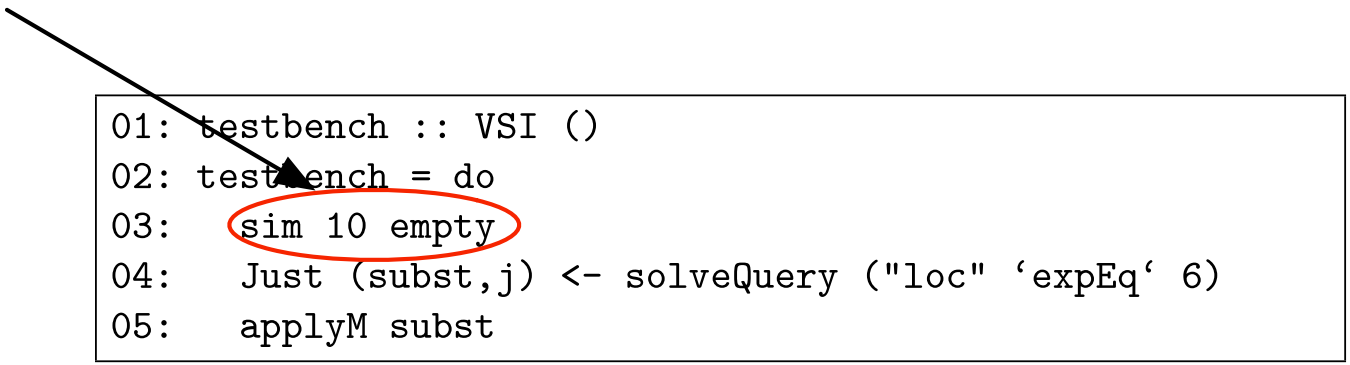
```
epsilon :: VSI Bool
epsilon = do
  xs <- gets activeQ
  if (not . null) xs
    then do
      mapM evalP xs
      return False
    else do
      xs <- gets inactiveQ
      ...
      ...
        else return True
```

our maze example again:

```
01: module maze(i,clk);
02:   input        i,clk;
03:   reg   [2:0] loc ;
04:
05:   always @(posedge clk)
06:   case (loc)
07:     0 : loc <= i ? 3 : 1;
08:     1 : loc <= i ? 2 : 0;
09:     2 : loc <= i ? 7 : 3;
10:     3 : loc <= i ? 4 : 2;
11:     4 : loc <= i ? 4 : 5;
12:     5 : loc <= i ? 6 : 7;
13:     6 : loc <= 6;
14:     7 : loc <= 7;
15:   endcase
16: endmodule
```

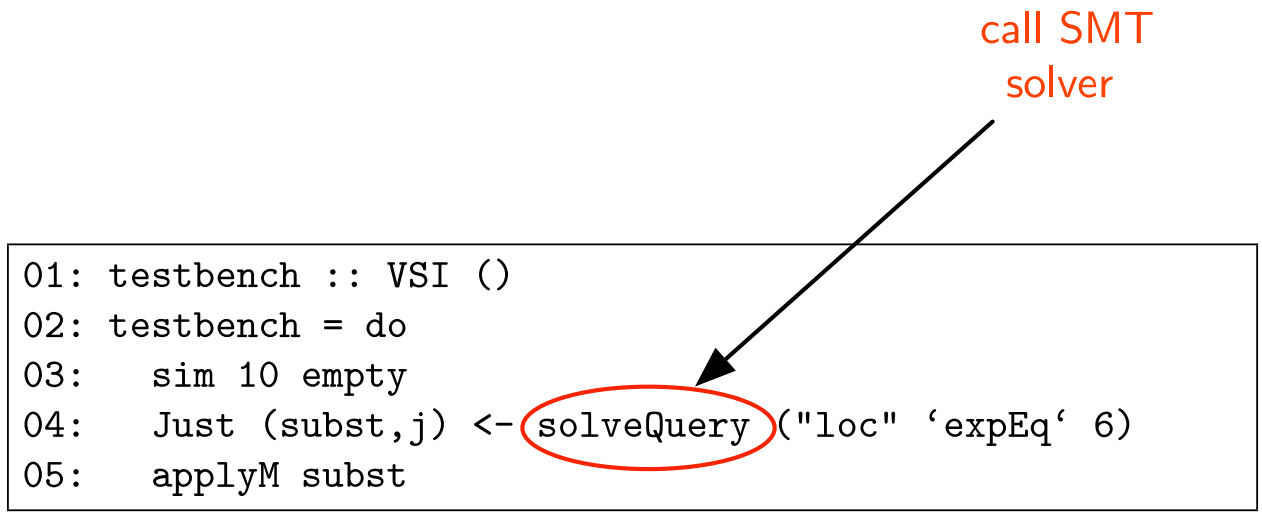
fully automatic solution:

symbolic simulation
for 10 cycles



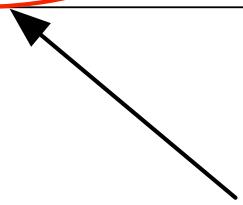
```
01: testbench :: VSI ()
02: testbench = do
03:   sim 10 empty
04:   Just (subst,j) <- solveQuery ("loc" `expEq` 6)
05:   applyM subst
```

fully automatic solution:



fully automatic solution:

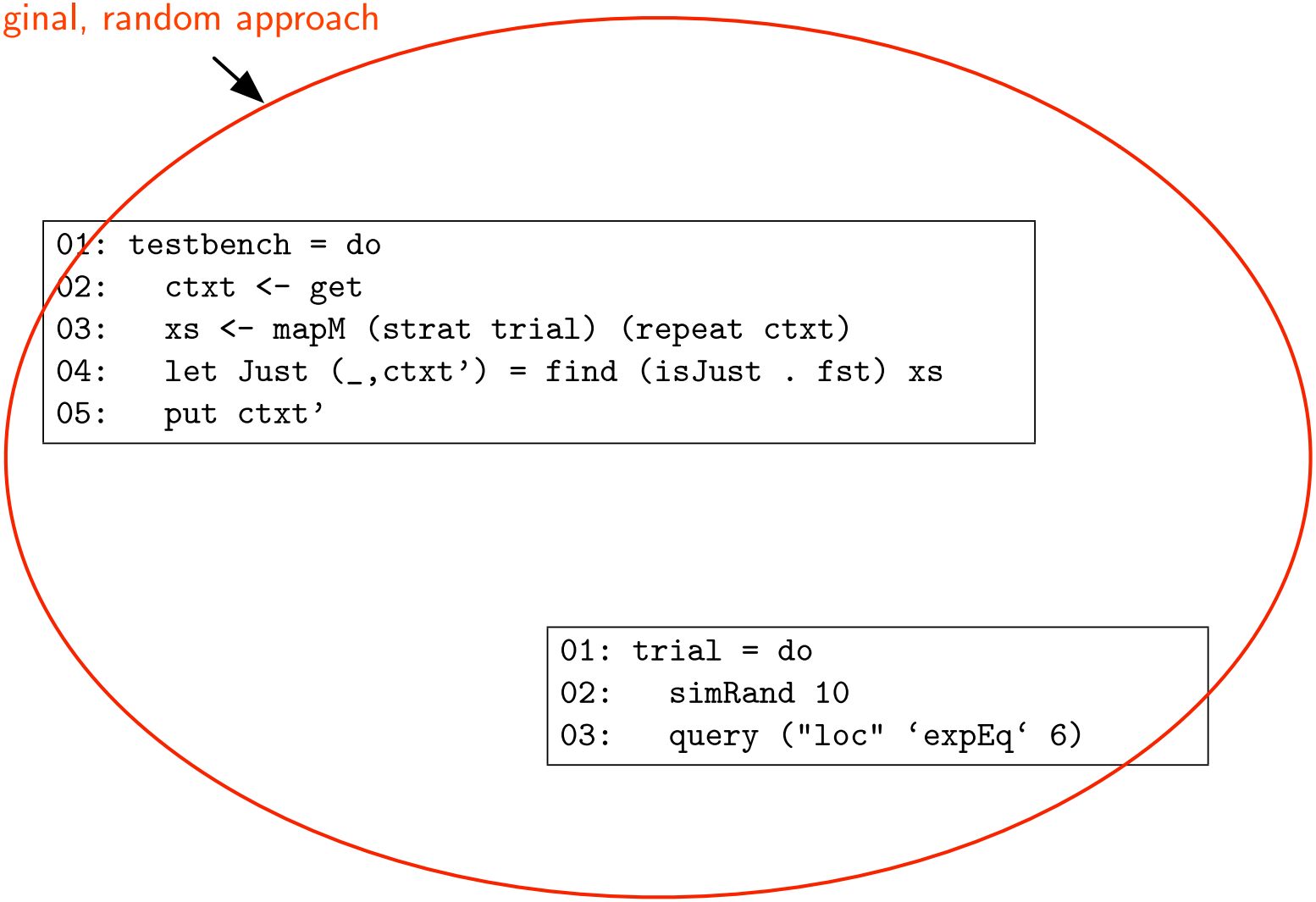
```
01: testbench :: VSI ()
02: testbench = do
03:   sim 10 empty
04:   Just (subst,j) <- solveQuery ("loc" `expEq` 6)
05:   applyM subst
```



resolve symbolic values
to concrete values

mixture of concrete and symbolic (like Magellan):

original, random approach

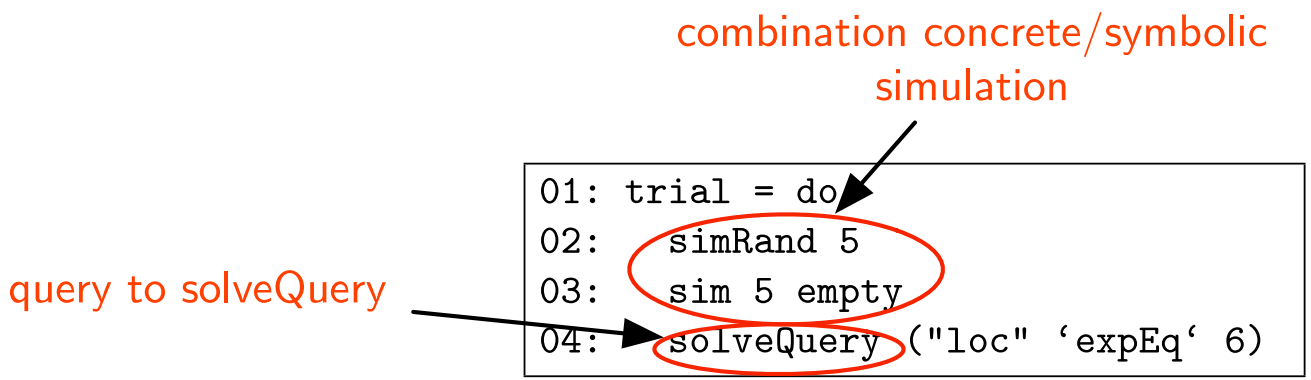


```
01: testbench = do
02:   ctxt <- get
03:   xs <- mapM (strat trial) (repeat ctxt)
04:   let Just (_,ctxt') = find (isJust . fst) xs
05:   put ctxt'
```

```
01: trial = do
02:   simRand 10
03:   query ("loc" `expEq` 6)
```

mixture of concrete and symbolic (like Magellan):

```
01: testbench = do
02:   ctxt <- get
03:   xs <- mapM (strat trial) (repeat ctxt)
04:   let Just (_,ctxt') = find (isJust . fst) xs
05:   put ctxt'
```



mixture of concrete and symbolic (like Magellan):

need to apply substitution
from SMT solver

```
01: testbench = do
02:   ctxt <- get
03:   xs <- mapM (strat trial) (repeat ctxt)
04:   let Just (y,ctxt') = find (isJust . fst) xs
05:       Just (subst,j) = y
06:   put (apply subst ctxt')
```

```
01: trial = do
02:   simRand 5
03:   sim 5 empty
04:   solveQuery ("loc" `expEq` 6)
```

backtracking (very complex control):

```
testbench = runContT (callCC $ \exit -> g [] exit) return
  where g xs exit = do
          p <- lift (valOfId "loc")
          when (p == expConst 6) $ exit []
          if p `elem` xs
            then return xs
            else do
              ctxt <- lift get
              lift (sim 1 $ fromList [mkI 0 ("i",exp1)])
              xs'  <- g (p:xs) exit
              lift (put ctxt)
              lift (sim 1 $ fromList [mkI 0 ("i",exp0)])
              g xs' exit
```

does any of this make the life of a verification engineer better?

-

does any of this make the life of a verification engineer better?

- can *program* testing strategies, don't have to do it *by-hand*

does any of this make the life of a verification engineer better?

- more information available to make decisions
  - know future state now, other worlds, etc.

does any of this make the life of a verification engineer better?

- measured approach to automatic input resolution with SMT

does any of this make the life of a verification engineer better?

- benefits of modern declarative languages
  - type checking, algebraic data types, pattern matching, etc.

summary:

- *empower verification engineers when building testbenches*
- *testbench = meta-program to analyze DUT via simulation*
- *many novel strategies enabled under this paradigm*
- *integrate with SMT and embed in declarative language*
- *working on a tool,* `vlogmt`

future work:

- *continue to build up* `vlogmt`*!*
- *substantial case studies*
- *explore the idea of interaction*

Thanks!