interactive code checking with Cobra

a Tutorial

Gerard Holzmann
Nimble Research
gholzmann@acm.org

topics covered

1. *background* and principle of operation
   - installation and configuration
   - guide to online documentation
2. *pattern* queries and regular expressions
   - exercises
3. *interactive* queries
   - token attributes
   - sets and ranges
   - functions
   - reading files, libraries
   - exercises
4. *scripted* queries
   - recursive functions
   - associative arrays
   - the query libraries
   - using concurrency
   - exercises
5. *standalone* checkers
   - using concurrency: multi-threaded checkers
6. use of Cobra for *runtime verification*
   - using live data or event-logs

pattern searches

what’s wrong with using “grep”?

```
$ grep  -e x *.c | wc
   1136    7251   57700
```

sample match: `prefix = s;`

```
$ cobra -pat x *.c | wc
     96      549    3647
```

matches *tokens* named **x**,

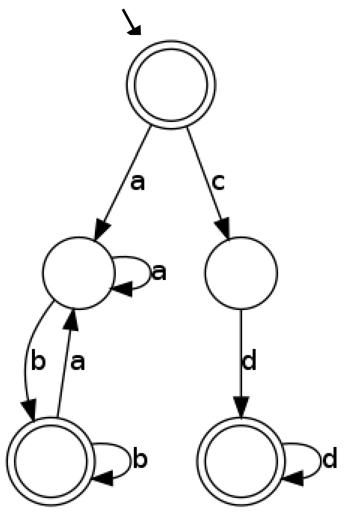
sample match: `strcmp(x->txt, “x"))`

note: the pattern search does not match either the word prefix or the string “x”

pattern searches

pattern expressions are a simplified form of *regular expressions*

- Regular expressions are used in many tools and applications for pattern matching *text strings*
- Examples include the Unix™ tools:
  - grep, sed, awk, lex, ed, sam, etc.
  - Google search patterns can also contain regular expressions
- Regular expressions define *finite state automata*
  - the automata accept precisely those text strings that match the regular expression
  - example: "(a+ b+)* | c d+“ defines the finite state automaton (FSA) shown on the right.
    - the FSA *accepts* input if it terminates in an accepting state (indicated by the double circles)



(,), +, *, and | are regular expression meta-symbols)

pattern searches

pattern expressions are defined over *lexical tokens* instead of *text-symbols*

```
$ cobra -pat x *.[ch]                                   # a trivial 'pattern'

$ cobra -pat '{ .* malloc ^free* }' *.c                 # don't-care: .*
                                                        # negation: ^, repetition: *
```

cobra guarantees that in all these patterns the nesting level of all brace pairs matches

```
$ cobra -pat '{ .* [static STATIC] .* }' *.c            # choice: [...]

$ cobra -pat '{ .* @type x:@ident ^:x* }' *.c           # types and name-binding :

$ cobra -pat '{ .* x:@ident -> .* if ( :x /= NULL ) .* }' *.c
```

embedded regular expressions

to match the *token* / itself, use an escape: \/

pattern search examples

matching for-loops not followed by a compound statement

```
$ cobra -pat 'for ( .* ) ^{'  *.c                      # first try
  5814      for (n = v_names[ix]; n; lastn = n, n = n->nxt) // mk_var
  5815      {         if (n->h2 > h2)

$ cobra -pat 'for ( .* ) ^[{ @cmnt]*' *.c              # second try
  2834      for (i = 0; i < Ncore; i++)
  2835      for (n = a_tbl[i].n[h1]; n; n = n->nxt) // sum_array

$ cobra -pat 'for ( .* ) ^[{ @cmnt for switch if]*' *.c  # third try
   793      for (yylen = 0; yystr[yylen]; yylen++)
   794        continue;
```

because the searches are so fast, it is easy to iterate them
to home in on the right set of matches

pattern searches

adding preprocessing with -cpp

```
$ cobra -cpp -pat 'for ( .* ) ^{' *.c
```

pattern searches

adding preprocessing with -cpp

```
$ cobra -cpp -pat 'for ( .* ) ^{' *.c
cobra_ctok.c:
  1:    2483    for ( i = 0; i < nr; ++i )
  2:    2484         *(dst++) = *(src++);

  3:    2538    YY_INPUT((& ... ), ...
...
```

? (a macro)

pattern searches

adding preprocessing with -cpp

note: in pattern expressions (and) are normal symbols, not meta-symbols as in classic regular expressions

```
$ cobra -cpp -pat 'for ( .* ) ^{' *.c
cobra_ctok.c:
  1:   2483   for ( i = 0; i < nr; ++i )
  2:   2484          *(dst++) = *(src++);

  3:   2538   YY_INPUT((& ... ), ...
...
```

```
...
for (n=0; n < max_size && \
    (c = getc(yyin))!= EOF && c != '\n'; ++n ) \
    buf[n] = (char) c; \
...
```

traditional regular expressions are also supported,

but require a lot of escape symbols when searching code

note the required spaces to separate tokens

- example

  $ cobra –regex ‘switch \(. \) { (case . : .* break ;)* }’ *.c

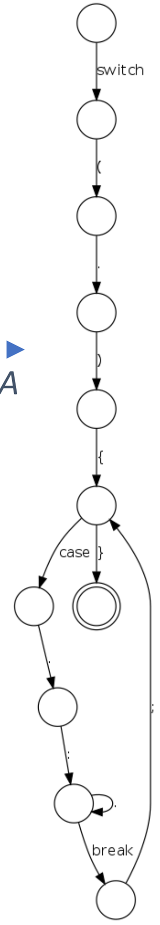
  FSA

  (and) are now *meta-symbols* (used for grouping)

  as are +, ?, and |

a plain (or) must now be written \(and \) to distinguish

them from the *meta-symbols* (which hampers readability)



regular expressions vs pattern expressions

overview of the differences

(and)	grouping
|	choice, e.g. "(a | b)" matches a *or* b
+	one or more repetitions
?	zero or *one* repetition

not meta-symbols in pattern expressions

*	zero or more repetitions
.	match any token
@type	match a particular token class, e.g., @ident
x:@type	bind the variable-name x to a specific token *name*
:x	refer to a previously bound name
[and]	define a set of options, e.g., [a b c] matches one of a b or c

* and]	when *preceded* by a space is a regular symbol
[	when *followed* by a space is a regular symbol
/re	match token if the *token-text* matches re

in pattern expressions

interactive pattern searches

exploring code with structural patterns

$ cobra –N8 \`cat thousands_of_filenames\` # e.g., linux-4.3

```
8 cores 39133 files 84,111,645 tokens
: # if/else/if chains must end with else
: pat else if ( .* ) { .* } ^else
```

matched braces

```
: # every non-void fct must have a return stmnt
: pat ^void @ident ( .* ) { ^return* }
```

no quotes around the pattern are required but now a “;” token match must be protected with “\;” to prevent interpretation as a query-command separator

name binding in pattern searches

some examples

find assignments to the control variable of a for-loop, inside the loop body

```
$ cobra –pat "for ( x:@ident .* ) { .* :x = .* }" *.c
```

matching braces *matching braces*

find local variable declarations that aren't used in the function body

```
$ cobra –pat ") { .* @type x:@ident ^:x* }" *.c
```

to avoid matching on structure declarations

note: *all* individual *tokens* in the pattern must be separated by *spaces*

exercises

your turn

```
: pat @type /restore ( .* ) { .* }            # find function names containing “restore”

: pat x:@ident += @ident ( ^,* :x  .* )       # using return value of a fct in its first parameter

: pat                                         # in macro defs, arguments must be enclosed in braces

: pat                                         # there can be no _ in typedef names

: pat                                         # typedefs for pointers must have a _ptr suffix

: pat                                         # the body of an if-statement is not enclosed in { … }
```