Cobra: Fast Structural Code Checking (Keynote)

Gerard J. Holzmann
Nimble Research
Monrovia, California 91016, USA
gholzmann@acm.org

ABSTRACT

In software analysis most research has traditionally been focused on the development of tools and techniques that can be used to formally prove key correctness properties of a software design. Design errors can be hard to catch without the right tools, and even after decades of development, the right tools can still be frustratingly hard to use.

Most software developers know, though, that the number of coding errors is usually much larger than the number of design errors. If something is wrong, the problem can usually be traced back to a coding mistake, and less frequently to an algorithmic error. Tools for chasing down those types of errors are indeed widely used. They are simple to use, and considered effective, although they do not offer much more than sophisticated types of trial and error.

The first commercial static source code analyzers hit the market a little over a decade ago, aiming to fill the gap between ease of use and more meaningful types of code analysis. Today these tools are mature, and their use is widespread in commercial software development, but they tend to be big and slow. There does not seem to be any tool of this type yet that can be used for interactive code analysis.

We'll describe the design and use of a different type of static analysis tool, that is designed to be simple, small, and fast enough that it can effectively be used for interactive code analysis even for large code bases.

CCS CONCEPTS

•**Software and its engineering → Automated static analysis;** *Software verification; Dynamic analysis; Software development techniques; Search-based software engineering;*

KEYWORDS

static source code analysis, interactive analysis, lexical analysis, structural code analysis, regular expressions, token expressions

ACM Reference format:
Gerard J. Holzmann. 2017. Cobra: Fast Structural Code Checking (Keynote). In *Proceedings of International SPIN Symposium on Model Checking of Software , Santa Barbara, CA, USA, July 2017 (SPIN'17-KEY),* 8 pages.
DOI: 10.1145/3092282.3092313

SPIN'17-KEY, Santa Barbara, CA, USA

DOI: 10.1145/3092282.3092313

1 INTRODUCTION

When analyzing, or exploring, code we often look for patterns. The patterns can be very simple, for instance we may want to check where a particular function is being called or where a specific identifier is used. Let's consider that last example in a little more detail. Suppose we want to find every occurrence of a variable named `f` in one or more source files. I'll use the source code for the Cobra tool itself for this example [3]. The standard Unix tool `grep` isn't very good at handling this task, matching any line that has the character `f` on it. Here just a count of all the matches it finds:

```
$ grep -e f *.[ch] | wc -l
4906
```

The command matches any line that happens to contain the letter `f`, not just the matching identifiers. Here is how the Cobra tool handles the same search, narrowing the results down to exact matches of the name itself. The first few of these are, for instance:

```
$ cobra -e f cobra.h | sed 7q
259 matches
cobra_te.c:
  1:   1075  add_match(Prim *f, Prim *t, Store *bd)
  2:   1087   m->from = f;
cobra.h:
  3:     36   Function *f; // set for fct calls
  4:     81   void (*f)(char *, char *);
```

We could narrow this down further with just a few extra keystrokes to matches of only global, or only local variables, but we'll return to the details later.

Suppose we want to check if there are any empty `else` statement blocks in say the Unix 10th Research Edition source code. This requires us to check for a combination of two symbols: the keyword `else` followed by a semicolon `;`. The `grep` tool will be of even less use now, because we do not care if there are newlines in between the keyword and the semi-colon.

To do this check we first collect all source file names we want to search in a file called `ch_files`, and then use the Cobra tool to look for the pattern, in this case without attempting to preprocess the code. The following command searches the 1.3 Million lines of source code from the Unix `cmd` subdirectory and reports the following matches:

```
$ cobra -e "else ;" `cat ch_files`
tc.c:406:   #define oput(c) \
             if (output) putchar(c); else;
jc.c:531:   #define oput(c) \
             if (output) xychar(xx, yy, c); else;
dcan.c:540: #define oput(c) \
             if (output) xychar(c); else;
print.c:143:      else;
```

```
tree.c:61:        else;
tabs.c:156:       else;
putpcc.c:1348:    else ; /* fall through to update */
```

That is still a rather simple type of check, but it gets more interesting if we now want to look for not just empty `else` statements but also empty `if` statements. To express that type of pattern, we will need to move up to regular expressions.

1.1 More Complex Patterns

Cobra regular expressions support the usual meta-symbols: a dot (`.`) represents a match of any token, the Kleene star (`*`) represents a repetition of zero or more copies of the pattern that precedes it, and parentheses are used for grouping. If we want to use a literal match of a meta-symbol, we can escape the special meaning of the symbol by prefixing it with a backslash (\).

Let's check for empty `if` statements in the `src/cmd` subdirectory of the Plan9 operating system, which has about 340 KLOC of code. We six matches for the pattern:

```
$ cobra -e "if \( .* \) \;" `cat ch_files`
unixcrap.c:97:
  1:     if(flag['S'] && flag['H'])
  2:          ;
print.c:56:
  3:     if (p->o_type == BLOCK)
  4:          ;  /* nothing at all */
  5:     else if (invis && !fill)
  6:          ;  /* nothing at all */
sx.c:108:
  7:     if(Bread(b, x->data, len) != len)
  8:          ;  /* oops! */
print.c:38:
  9:     if (p->o_attr & INVIS || p->o_type \
                                    == BLOCK)
 10:          ;  /* nothing at all */
writepng.c:92:
 11:     if(a == 255 || a == 0)
 12:          ;
```

All these matches span more than one line, so they cannot be caught by a string pattern matching tool like `grep`, and of course the details of the conditional of the `if` statements are different in each case.

In the above expression we rely on the Cobra tool matching the opening and closing braces of the conditional expression, so that the expression will still give the right result even if a function call were to appear as part of the condition.

There are a vast number of matches of empty if statements to be found in the Unix sources, including suspicious looking matches like these (in `lcc` compiler code from 1992):

```
lcc/ph/andif.c:
  if (K.b1 && ppH->pH->c); else complain(__LINE__);
  if (ppH->pH->c && K.b1); else complain(__LINE__);
  if (K.b1 && ppH->pH->sc); else complain(__LINE__);
  if (ppH->pH->sc && K.b1); else complain(__LINE__);
  if (K.b1 && ppH->pH->s); else complain(__LINE__);
```

It's not hard to define other types of useful patterns. To find `while` statements that are not enclosed in curly braces, for instance, we can use this pattern:

```
$ cobra -e "while \( .* \) ^[{;]" *.[ch]
```

Note that the keyword `while` can also be used in a `do...while` sequence, so we have to check for the absence of both a curly brace and a semi-colon after the condition.

One limitation of this approach to code analysis is that the patterns we can use are not flow-sensitive. We'll see shortly we can use the builtin Cobra query languages to extract also control-flow graphs, but this of course requires more than just regular expressions. It is remarkable, though, how many types of quick source code checks the use of simple regular expressions over lexical tokens can support.

Here is, for example, a simple check to see if a code base contains any `switch` statements that do not contain a `default` clause. In the following expression we again use the ^ operator for negation, i.e., to indicate the absence of a specific keyword:

```
$ cobra -e "switch \( .* \) { ^default* }" *.[ch]
```

When applied to the Spin source code (about 42 KLOC), this locates and prints 38 matches in the code.

1.2 Variable Binding

Suppose we want to check for cases where the control variable from a `for` loop is assigned values in the body of the loop. To do this it should be possible to remember the name of likely candidates for the loop control variable, and then match against that name in the body of the loop. We can do so by using a variable binding mechanism, as follows:

```
$ cobra -e "for \( x:@ident .* \) { .* :x = .* }" *.c
```

For simplicity, we assume here that the first identifier name that appears in the initial segment of a the `for` loop is the loop control variable we're interested in. This is not always the case, but it helps us get started. This identifier name is bound to the variable name `x` here. Next, the expression scans the body of the loop and checks if it contains any direct assignment to that variable. If so, the loop is matched. If not, it passes the test.

The prefix `@` allows us to refer to the type, rather than the text, that is associated with a token.

There are of course other ways to modify the loop variable, for instance with a pre- or post-increment or decrement operation, or by taking its address and passing it to a function. Each of these checks can be expressed by further extending the expression.

We can use the same principle of variable binding to define quick checks of other types of potential problems, for instance to see if we can find the declaration of a variable that isn't used anywhere in the remainder of the function body:

```
$ cobra -e "\) { .* @type x:@ident ^:x* }" *.c
```

Here we use an initial closing round brace to make sure we're scanning function bodies, and not structure declarations or array initializations.

2 INTERACTIVE CODE EXPLORATION

There are of course also other ways to mine source code for patterns of interest. For the empty `if` statement such an exploration could consist of first looking for all `if` statements, then skipping past the condition that follows it, and checking for the immediately following lexical token. We can do that interactively, by typing query

commands at the Cobra tool, which it responds to instantaneously. Here we'll look for `while` statements that are not enclosed in curly braces, in the Spin sources.

```
$ cobra *.[ch]
1: mark while \(
2: next
3: jump
4: next
5: mark no {
6: mark no \;
7: display
pangen1.c:534:
  while (i > 0 && buf[--i] == ' ') buf[i] = '\0';
  ... (13 more matches)
```

In this session, we first mark all lexical tokens named `while` followed by a open round brace. Then, we move the mark point forward one step, so that we are now pointing at the brace. That brace defines a range in the source text, and we can now move the mark from either end to the other with a jump command. (The same holds for curly and square braces.) One more move forward puts us right after the condition, which should either be a curly brace or a semi-colon, in case it is a `do..while` statement.

We unmark all matches of those two symbols and are now left with all proper `while` statements that have a body that is not enclosed in curly braces, and we can display the matches. In the case of the Spin code, this check finds 14 matches, out of 70 `while` statements that appear in the code for version 6.4.6.

The above may look a little laborious, so the tool defines shorthands for all frequently used commands of this type. These shorthands allow us to abbreviate the entire command sequence to one line:

```
$ cobra *.[ch]
1: m while \(; n;j;n; m no /[{\;]; d
```

Here we also combined the two unmarking operations for removing matches of curly braces or semi-colons into a single command, by using a regular expression.

To move this up one more step, we can also pass this command sequence to the Cobra tool on the command line, so that we can type:

```
$ cobra -c "m while \(; n;j;n; m no /[{\;]; d" *.[ch]
...
```

and find the same matches with an off-line check.

If we now look back at the earlier checks that we did with regular expressions, we can see that they are also simple to formulate in this command language, but the command language allows us to do a little more as well. First, consider finding empty `else` statements. That's simple to check with one or two commands. The mark command allows us to match one either one or two tokens in sequence, for instance as follows:

```
$ cobra -c "m else \;; d" *.[ch]
```

Note that the first semi-colon is should be a literal match of a semi-colon, not a command separator, so it is escaped with a backslash.

Let's take another look at the earlier check we defined finding for `switch` statements without a `default` clause.

That earlier check had an flaw. If, for instance, two `switch` statements are nested, and the inner statement has a `default` clause, but the outer statement does not, then this check would fail to report a match for the outer statement. This can be handled in the command language, as follows:

```
$ cobra -c "m switch; n {; c top no default; d" *.[ch]
```

The `n` (`next`) command with an argument allows us to move quickly to the first following match of a token matched by that argument. Then, the `c` is the `contains` command that can be applied to ranges, like the one started by the opening curly brace of the switch statement. This `contains` command, like most other commands, can take qualifiers that can slightly change its semantics. Normally, the command checks if a specified token appears inside a previously marked range. If not, the range will be unmarked as no longer interesting. Two of the qualifiers that can be used are `top` and `no`. The `no` qualifier simply inverts the match: now for a match the range should not contain the specified token. The `top` qualifier restricts the match, or the absence of a match, to the top level of nesting in the range. That is: it will not look inside sub-blocks, as would inevitably be used for any nested `switch` statements.

If we measure in terms of mere keystrokes, the regular expression that wasn't quite precise took 29 keystrokes, and the command sequence that is exact takes 34, so there's not a great difference there. Both methods of pattern matching can be used interactively, which is the target use of the Cobra tool.

2.1 Using Sets and Functions

Command scripts can be defined as named functions, which can be stored in libraries and imported when needed. An additional capability that can be very useful is the ability to define and manipulate sets of matches. We can illustrate both these features with the following example.

Consider the case where we want to check if dynamic memory allocation can take place after we reach a particular target function `fct()`.

To check this, we can first find all functions that contain calls to memory allocation functions like `new` or `malloc`. We put all matching functions in a first set. Next, we can define a second set of all functions that are reachable from `fct()`, and then check if this second set overlaps with the first. Here's how we can define a Cobra function that accomplishes this.

We use two predefined functions in this command script. The first, `fcts`, marks the function name in all function definitions that can be found in the code. The second, `fcg`, computes the function call graph by default starting from the function `main`, or when given an argument, starting from the function named in that argument. The command script can now be written as follows:

```
def find(fct, bad)
   r        # reset: clear all marks
   fcts     # mark function definitions
   n {      # move to start of fct body
   c bad    # does it contain argument bad?
   b \(; b  # move mark back to function name
   >1       # save names of these fcts in set 1

   r        # clear all marks
```

```
    fcg fct  # mark functions reachable from fct
    <&1      # intersect marks with set 1
end
find(work, malloc)
d            # display the results
```

There are five operations that can be performed on Cobra sets:

```
>n    # save current marks in set n
<n    # assign: replace current marks with set n
<|n   # union: add marks from set n
<&n   # intersection: keep only marks also in set n
<^n   # subtract: keep only marks not in set n
```

For the `find` script from above, the intersect operation is what we need to identify the functions that are reachable from `work()` and that call `malloc`.

Cobra also uses sets internally to implement an undo operation, that restores the session to the state it was in just before the last operation performed.

2.2 Creating Dashboards

The approach to static analysis sketched here is fast enough that standard checks can be run each time code is checked into a build, or each time code is compiled. It is also relatively straightforward to collect metrics or statistics on a code base, display the results on a quickly extracted html dashboard (see Figure 1), or use these metrics to track the evolution of a code base from one release to the next.

3 INLINE PROGRAMS

Although the command language allows us to explore many more code features than is possible with regular expression pattern matching, it too still puts some limits on what we can do with code that is preprocessed into a sequence of lexical tokens. There is, for instance, no convenient way to define variables, or to navigate the code in more interesting ways.

As a simple example of this, consider locating cases where a file descriptor is opened in a given function, but not closed within the same function. Note that multiple file descriptors could be opened, and only some of them closed, so we would need some way to bind multiple file descriptor names to variables.

Technically, we could express the desired pattern in a regular expression, if we could use negation not just on token values but also on sequences of token values, using the expression:

```
"\) { .* x:@ident = fopen ^(fclose \( :x \) )* }"
```

Here we use variable binding to remember the file descriptor variable that appears in an `fopen` call, and then check if a sequence of four tokens is absent from the remainder of the function body, using round braces to group the sequence.

Regular expressions usually do not support any type of negation, and Cobra supports only a limited form for just single tokens or ranges of tokens specified with a range definition (e.g., `[^a b c]`). So a sequence of tokens cannot be negated.

For cases like these Cobra support a rich inline scripting language, that includes support for conditional branching, iteration, and both scalar variables and associative arrays that can store values, strings, and token references.

An example of the use of this language to solve the file descriptor example problem is as follows (line numbers added).

```
 1 r
 2 fcts
 3 %{
 4     if (!.mark)
 5     {    Next;
 6     }
 7     .mark = 0;
 8     f = .;    # function name
 9     while (.txt != "{")
10     {    . = .nxt;
11     }
12     s = .; t = .jmp;   # function body
13     unset Open;
14     unset Close;
15     while (.lnr <= t.lnr && . != t)
16     {   if (.txt == "fopen")
17         {   . = .prv;
18             if (.txt == "=")
19             {   . = .prv;
20                 if (@ident)
21                 {    Open[.txt] = .;
22                 }
23                 . = .nxt;
24             }
25             . = .nxt;
26         }
27         if (.txt == "fclose")
28         {   . = .nxt;
29             if (.txt == "(")
30             {   . = .nxt;
31                 Close[.txt] = 1;
32         }   }
33         . = .nxt;
34     }
35     for (i in Open)
36     {   if (!Close[i.txt])
37         {   g = Open[i.txt];
38             print g.fnm ":" g.lnr ": file-desctr";
39             print i.txt " opened but not closed\n";
40     }   }
41 %}
```

The script starts by checking if the currently examined token is a function name that was marked in the earlier call to predefined function `fcts`. If no mark was set, the script exits and moves on to the next token, using meta-command `Next`. We now clear the mark, and set a variable `f` to point to the current token location (line 8).

Next, we move forward in the token sequence to the start of the function body, using the `while` loop on lines 9–11. On line 12 we mark the start of the function body, which defines a range from the opening to the closing curly brace. The location of the closing brace is available in the `.jmp` field of the token. Two associative arrays that we use later to collect the relevant information from the

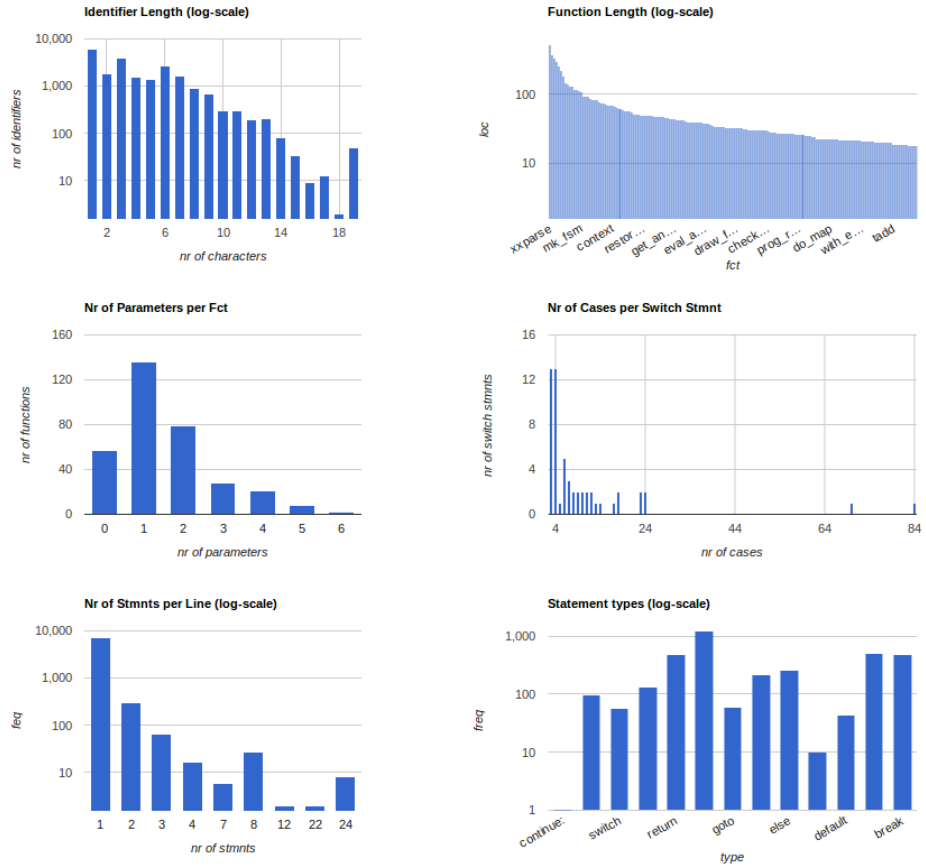

Figure 1: Example of an automatically generated html dashboard, collecting metrics of a code base.

function body are now cleared (lines 13,14), and we start searching the body of the function, again using a while loop (lines 15–34).

We're interested in two specific function calls: `fopen` and `fclose`. On a match of `fopen` (line 16) we search backwards to find the name of the identifier that is used to store the file descriptor. The location of that identifier is stored in associative array `Open` that is indexed with the file descriptor name, a string.

On a match of `fclose` (line 27) we search forwards to find the argument of the call, and store that string in another associative array `Close`, this time just using an integer count to indicate that we saw the call. As is common in scripting languages, variables do not need to be declared. Their type is deduced from context, and is allowed to change.

Once the whole function body has been searched, and we reach line 35, we check all elements in the `Open` array, and for each element we check if the corresponding `Close` array element has a non-zero value (line 36). If not, we retrieve the token location of the `fopen` call from the `Open` array, and report the missing call.

This script searches 9,370 functions defined in the 1.3 million lines of code for the Plan9 command directory in 2 seconds, and reports 21 matches. An example of a match appears in the `main` function of `awk`, where a file descriptor is opened and indeed never closed.

```
./awk/maketab.c:127: \
    file descriptor fp opened but not closed
```

3.1 Functions

The scripting language supports functions that can be recursive and return values. A not particularly meaningful example that illustrates these features is the following definition of a function to calculate factorials:

```
%{
    function fact(n)
    {   print "call " n "\n";
        if (n <= 1)
        {     return 1;
        }
        return n*fact(n-1);
    }
    print "10! = " fact(10) "\n";
    Stop;
%}
```

Note that the Cobra tool will attempt to execute this inline program once for every token in the input stream, so the program will not be invoked at all if there is no input stream (e.g., if we start the tool without giving it a source program to process). It also explains the need for the `Stop` statement at the end of the program. Without it, the factorial would be computed as many times as there are lexical tokens in the input.

A more useful example of a function that can help navigate the input is the following definition of a support function called `expect` that looks ahead one token to a match of an expected pattern. If there is no match, the function will call the `Next` statement to stop the execution for the current token and move on to the next. Note that the look-ahead token is stored in a variable `q` and that therefore the current position of the token does not change unless there is a match. In that case, the position is moved forward passed the look-ahead token.

```
%{
  function expect(a)
  {   q = .nxt;
      if (q.txt != a)
      {    Next;
      }
      . = q.nxt;
  }

  if (@ident)
  {   expect(":");
       ...
  }
%}
```

3.2 Concurrency

The processing that is performed for query handling is in principle easily parallelized. The preparation of the internal data structures is mostly independent for all source files, so this processing can benefit from the use of multiple threads of execution on multi-core systems. For handling most queries, the token stream can trivially be divided in N equal size fragments, with each fragment searched separately by a separate thread.

When executing on a multi-core Linux-like system with four or more cores, the tool by default uses four parallel threads of execution both for the preprocessing of source files and for processing queries. The user can override this default choice with a command line argument, or interactively, and set the number of threads to use to any other number.

It is, of course, rarely useful to define the use of more parallel threads than the hardware can support, and on some systems the effects of non-uniform memory access (NUMA) and caching can limit the benefits of using the maximum number of cores. We'll return to this in Section 4.

Within inline Cobra programs, the user can control the use of concurrency with a number of primitives. By default, if four cores are defined, each inline program will separately process a quarter of the token sequence in parallel. Data structures are maintained separately for all cores, but can be unified when needed. If there is a critical part of an inline program that must be handled sequentially by all cores, there are several ways to handle that. The first, and most obvious, method is to use locks, as in:

```
...
lock();
print cpu ": my count: " Count "\n";
unlock();
...
```

where predefined variable `cpu` gives the number of the thread that is executing. Another method is to schedule the threads to let them pass through a gate one by one, using a single global token reference that all threads can access to communicate:

```
...
# await turn:
while (first_t.mark != cpu)
{   print "";
}

print cpu ": my count: " Count "\n";
first_t.mark++;


...
# wait for all threads to be done:
while (first_t.mark != ncpu)
{   print "";
}
...
```

Here we make each thread wait until its turn comes up, while printing empty strings. First the thread with id zero will be able to pass the gate. After it has completed its processing it increments the global variable that is used for this purpose, and moves on. If we need for the threads to wait at some point for all to have completed the critical part of the code, we can do so as shown in the second code fragment.

3.3 Data Sharing

By default, the threads do not share data. This is to make sure that all threads can proceed with their processing of the token sequence in parallel without the risk of any conflicts. This means that each thread has a separate copy of each variable and associative array that is used. It is possible to interrogate the value of a specific variable or associative array element across threads, or to unify the contents completely so that, for instance, all threads can see all indices of an associate array, but this unification can be costly and can easily end up costing more time than is saved by the parallel processing itself.

The recommended method of sharing is therefore for each thread to store only information in the token sequence itself. Since each thread works on a separate part of that sequence, there is again no possibility of a conflict, and the data is trivially available to the user after the inline program completes. For instance, when looking for a complex pattern, the location of the pattern itself can be recorded simply by setting the `.mark` field of the tokens that are included in the match. The matches can then be displayed in the standard way later.

If it is necessary to store more complex data in a token, the scripting language allows the creation of additional lexical tokens that

can be bound to any given token in the sequence. In this way additional values, strings, and token references can be associated with matched tokens, without risking data conflicts between threads. The creation of a new token (with a call to the function `newtok()` will of course have to be protected with a lock.

3.4 Predefined Checks

The Cobra distribution comes with about 80 predefined checks and checker libraries, e.g., encoding rules from the Misra 1997, 2004, and 2012 guidelines [6], the JPL Coding Standard [5] and the Power of Ten Rule set [4].

As an example, an overview of rule violations with the Misra2012 guidelines for the Cobra code itself completes in about 5 seconds, checking 53 of the rules in `terse` mode, and finding deviations for 23 of the rules.

To avoid a possible copyright violation, we've omitted the textual description of each rule for which violations are reported below, and just give the counts for each deviation that is reported. The Misra guidelines target embedded systems software and rule out, for instance, the use of dynamic memory allocation, recursion, multiple break statements in a loop, or the direct use of any of the predefined data types.

```
$ cobra -terse -f misra2012 *.[ch]
=== R3.1:    (Required) ...:    6
=== Dir4.4:  (Advisory) ...:  113
=== R20.5:   (Advisory) ...:   13
=== Dir4.6:  (Advisory) ...: 1634
=== Dir4.12: (Required) ...:  163
=== R2.6:    (Advisory) ...:   16
=== R2.7:    (Advisory) ...:    3
=== R13.4:   (Advisory) ...:   19
=== R13.5:   (Required) ...:    6
=== R15.1:   (Advisory) ...:   64
=== R15.2:   (Required) ...:   10
=== R15.4:   (Advisory) ...:   11
=== R15.5:   (Advisory) ...:   54
=== R15.6:   (Required) ...:   16
=== R15.7:   (Required) ...:   38
=== R16.3:   (Required) ...:  281
=== R16.4:   (Required) ...:   10
=== R17.2:   (Required) ...:  101
=== R17.8:   (Advisory) ...:   15
=== R19.2:   (Advisory) ...:   12
=== R21.3:   (Required) ...:  163
=== R21.7:   (Required) ...:   16
=== R21.8:   (Required) ...:   23
```

A list of predefined checkers is printed with command-line option `cobra -lib`.

4 PERFORMANCE

For most applications, Cobra queries are processed fast enough for truly interactive use. For large to very large code bases, with millions of lines of code, there is a startup cost related to the basic lexical analysis that is used to prepare the input token sequence. Notably though, any code base can be handled even when there is no reasonable way to compile it for instance when it is incomplete, or when the required compilation directives aren't known.

Table 1: Performance in seconds, for analyzing all Linux 4.3 sources (cf. Figure 2)

Nr. Cores	*Startup*	*Empty else*	*Switch w/o default*
1	153.3	2.62	2.35
2	93.1	2.10	1.82
4	58.2	1.01	1.04
8	46.7	0.60	0.57
16	47.6	0.46	0.43
32	24.3	0.35	0.40

When using command-line queries (i.e., in non-interactive use), the cost of this startup has to be paid for every query that is submitted. This makes it more attractive to prepare scripts containing multiple queries that can be executed in sequence without restarting the tool.

For interactive use, the startup cost is similarly paid only once, and not repeated for any of the subsequent queries, for instance when more complex scripted queries need to be run multiple time while they are developed.

To start an interactive session for code bases up to about 30 KLOC, the tool initialization takes under a second.

Reading 1.3 million lines of source code in the `cmd` directory of 10th Edition Unix with 5,293 source files creates the sequence of seven million tokens in about 4.5 seconds, with the default use of four cores on a 1.4 MHz multi-core Ubuntu Linux system.

For a more extreme case, if we read in all 18.6 million lines of source code from the Linux 4.3 distribution, corresponding to about half a GigaByte of code, again using four cores, processing all 39,144 source files, now takes about 60 seconds.

The startup time can be reduced by using more cores in parallel, but access to disk to read all the files can limit the speedups.

If we start a Cobra session for the 18.6 million lines of Linux 4.3 code using 32 cores, the startup time drops to 24.3 seconds. At this point, the query processing returns to being interactive, even for this large of a code base.

For comparison, just doing a simple wordcount of the 537 million bytes with `wc`, without any further processing, already takes 13.5 seconds on the same machine.

Locating 17 occurrences of the keyword `else` followed immediately by a semi-colon in the 18.6 Million lines of code takes about 0.35 seconds for the last of the 32 parallel threads to finish its search, as does finding all 43 empty `if` statements with the command sequence:

```
m if \(; n; j; n; m \& \; d
```

Similarly, the search for `switch` statements lacking a `default` clause reports 9,627 matches in the code in about 0.4 seconds for the longest running thread. Table 1 lists the response times measured when varying numbers of cores are used, as also illustrated in more detail in Figure 2.

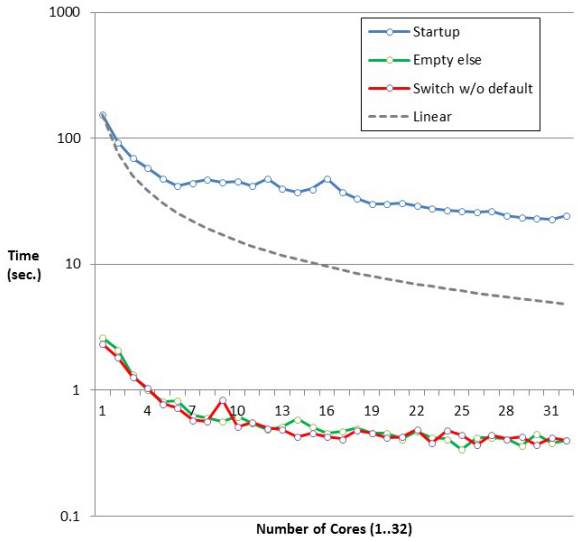

Figure 2: Multicore scaling, processing 18,633,817 lines of code from the Linux 4.3 distribution, read from 39,144 separate .c and .h files

4.1 Some Limitations

In Figure 2, we indicated what linear scaling would look like for the startup phase. Even though all threads, in both phases, are running independently, without accessing any shared data, the scaling behavior is clearly not linear.

There are small hickups in the data for 12 and 16 cores, which likely has something to do with the specific architecture of the Ubuntu machine that we used for these measurements. The machine has 32 cores, configured as four eight-core 64-bit AMD Opteron 6272 chips, each core running at 1.4 MHz, with a cache of 2 MB.

In these measurements there seems to be a sweet spot when using four parallel threads for both startup and query processing, which is why this was chosen as the default on multi-core systems. As noted, the scaling behavior could well be different on future systems, with memory architectures that are friendlier to multi-threaded code.

The use of multi-core processing can be very efficient if we are collecting data that need not be shared between cores. This is different if the cores have to communicate and coordinate their results. If we run inline Cobra programs in multi-core mode, all cores will by default collect their data (e.g., the values of variables and associative arrays) in separate data structures, to avoid possible race conditions and data corruption.

When the data needs to be combined, we can use predefined functions that unify the data and make it available to one or more cores for final processing, for instance to prepare a summary of findings. This unification phase can become a bottleneck, in some cases more than nullifying any time saved by the parallelization of the data collection phase.

It can also be difficult to predict what number of cores can speed up a search the most, given the potential caching conflicts on larger machines with a non-uniform memory access architecture.

5 SYNOPSIS

We've sketched the design and possible uses of a different type of static source code analysis that requires only a basic lexical analysis of source code, that can be performed quickly. Once initialized, user queries can be processed interactively, which in most cases means in under a second.

The method is readily parallelized, which allows the approach to scale to the exploration of even very large code bases. The code that is processed does not need to be compileable, and in fact it does not need to be source code either. In principle, the input could be English prose, where the user can provide a map file for categorizing words as nouns, pronouns, verbs, adjectives, prepositions, etc. It is then possible, for instance, to query the prose for specific patterns based on this user-defined categorization. The query "find all sentences that end in three subsequent verbs" would be quickly processed. We also applied the tool to event-streams generated by run-time verification tools to efficiently answer queries about the properties of those streams.

The regular expression matching algorithm used is based on the remarkably fast algorithm that was first described by Ken Thompson in 1968 [8]. An excellent discussion of the performance of many regular expression matching algorithms is also available from Russ Cox [1].

The most recent version of the tool (Version 2.0 at the time of writing [2]) is a significant extension of the first version that was developed by the author at JPL. Tool performance was improved with builtin support for the parallel preprocessing of source code, and the inline scripting language was extended with recursive functions with associative arrays, array iterators, and primitives for concurrency control.

To achieve interactive response times, the tool does limit the types of queries that can be processed. Queries that require deep flow or pointer alias analysis are clearly not the primary target for a tool of this type. For those applications the larger commercial static source code analyzers will remain an essential resource, see for instance [7]. The latter though are decidedly not interactive, nor are they likely to reach that point in the near future.

REFERENCES

[1] Russ Cox, "Regular expression matching can be simple and fast," https://swtch.com/~rsc/regexp/regexp1.html.
[2] Cobra documentation and distribution, http://spinroot.com/cobra.
[3] Gerard J. Holzmann, "Cobra: a light-weight tool for static and dynamic analysis," Innovations in Systems and Software Engineering, a NASA Journal, Vol. 13, Issue 1, 1–15, March 2017,
[4] Gerard J. Holzmann "The Power of Ten – Rules for Developing Safety Critical Code," IEEE Computer, June 2007, pp. 93–95, http://spinroot.com/p10/.
[5] JPL Institutional Coding Standard for the C Programming Language, March 2009, http://lars-lab.jpl.nasa.gov/JPL_Coding_Standard_C.pdf.
[6] MISRA, "Guidelines for the use of the C language in critical systems," Motor Industry Software Reliability Association, 1997, 2004, 2012, https://www.misra.org.uk/.
[7] Overview of static source code analyzers, htpp://spinroot.com/static/.
[8] Ken Thompson, "Programming Techniques: Regular expression search algorithm". Communications of the ACM. 11 (6), June 1968, 419–422.