Cobra: a light-weight tool for static and dynamic program analysis

Gerard J. Holzmann*

NASA Jet Propulsion Laboratory,
California Institute of Technology, Pasadena, CA, USA
gholzmann@acm.org

Abstract. Static source code analysis tools have become indispensable for the development of reliable software applications. The best analyzers can reveal subtle flaws in a code base, but they can also be slow. In part this is due to the collection of detailed information about the possible data and control flow of an application to support the broadest possible range of analyses. For larger code bases it is not unusual that even the best of breed static analyzers can take an hour or more to complete an analysis.
In this paper we describe a framework for a much faster, but more light-weight type of static analysis that can support *interactive* use for standard types of queries. The Cobra tool we designed for this purpose can scale to explore millions of lines of code interactively. The tool is mostly language agnostic, and can therefore easily be configured to resolve even dynamic program analysis queries.

Keywords: Static analysis · Source code analysis · Lightweight tools · Token expression matching · Dynamic analysis · Runtime verification

1 Introduction

The Cobra[1] tool was designed to support fast, interactive, resolution of relatively lightweight code queries, even on large code bases. To allow the tool to startup quickly it avoids pre-computing detailed information that could support more sophisticated types of analysis. Instead, the tool provide fast access to an extremely simple data structure that captures all the available information from an application in such a way that all other data may be deduced separately, in response to specific user-defined queries. The data structure is also chosen to be directly amenable to the parallelization of query resolutions on standard multi-core machines.

Cobra supports a range of formalisms for expressing queries. One method, for instance, is to formulate queries as regular token expressions on code patterns;

* This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

[1] An acronym for **Co**de **Br**owser and **A**nalyzer.

another is to use in-line command scripts, or sequences of query commands to mark and modify items of interest in the input.

The tool is not meant to be competitive with more sophisticated, but slower, mainstream static analyzers for the resolution of detailed code analysis queries. The tool could be extended to support those types of queries, but doing so may cause it to lose its ability to support interactive use. Although the scope of queries is limited, the Cobra tool tends to be surprisingly effective in handling common types of queries that range from smart code browsing, to statistics gathering, and the resolution of a broad range of standard program analysis problems.

The Cobra tool targets the analysis of C, C++, and Java code, but it is not restricted to those languages. Beyond the recognition of the normal language specific keywords and tokens, that can be configured separately, the tool is mostly language agnostic. This means that it is relatively easy to retarget the tool to the analysis of other programming languages, and even to input streams that do not correspond to programming languages at all. As an example of this we will show how the Cobra tool can be used for runtime analysis, where the input "language" is a stream of events, instead of program source text.

Section 2 summarizes the principle of operation of the Cobra tool, and describes the data structure that is built when it starts up. Section 3 describes the formalisms that are provided for writing queries, and illustrates their use with examples. Section 3.4 also covers a more atypical application of the tool to the runtime analysis of event logs. Section 4 discusses performance in more detail, and shows how the tool's operation can be parallelized on multi-core systems. Section 5 covers some related work.

Section 6 summarizes the main benefits and limitations of the chosen approach, and Section 7 concludes the paper.

2 Principle of Operation

Cobra uses a lexical analyzer, by default configured for recognizing C, C++, or Java sources, to process the code from all files that are specified on the command-line. By default, the tool runs this code through a standard C preprocessor to expand macros and include files, but the preprocessing phase can also be disabled with a command-line option, for instance when the files being read were preprocessed before, or where the code is too platform dependent. The tool then builds a data structure that can be used for querying that source code interactively, or with predefined scripts or programs.

The data structure that is constructed is a simple linked list of lexical tokens, enhanced with only minimal annotations and links, for instance to identify matching pairs of parentheses, brackets and braces in the code. Despite its simplicity, the tool can be remarkably powerful in locating complex patterns in a code base to assist in peer code review, development, or basic code analysis.

The tool supports four main formalisms for writing queries. In order of increasing capabilities, the user can explore a code base by writing (a) regular ex-

pressions on code patterns (so-called *token expressions*), (b) interactive queries, (c) inline programs, and (d) standalone checkers.

Token expressions can be used when a specific coding pattern is looked for, but they cannot do much more than that. Unlike *grep*, which works on character sequences on a line by line basis, Cobra expressions are formulated on sequences of lexical tokens, ignoring white space. A simple example of a Cobra token expression query is:

```
$ cobra -re '{ FILE \* x:@ident .* :x = fopen ^fclose* }' *.c
```

which prints all code blocks that contain a local declaration of a file descriptor (an identifier that is bound to *x*), an assignment to that same descriptor in an *fopen* call (referred to as *:x*), that is not followed by any calls to *fclose*. The tool makes sure that the curly braces match only if they appear at the same nesting level in the code.

Internally, Cobra converts the token expressions into inline Cobra programs, which provide a much richer context for code exploration. We will therefore discuss these regular token expressions, and the details of variable binding, after we have covered interactive queries and inline programs, in Section 3.3.

Interactive queries, discussed in Section 3.1, can resolve the most frequent types of issues that may come up in an initial exploration of a possibly large code base, for instance during code development, or in code review sessions. These types of queries can be used to quickly find, mark, and list interesting patterns of code, e.g., corresponding to deviations from a coding standard.

When more complex query needs to be handled, that may require iteration, conditional branching, or the use of additional data structures, then an inline Cobra program can be used. This is discussed in Section 3.2.

For still more complex types of queries, that can more easily be resolved with the full power of a C program, we can use the basic infrastructure and data-structures that are provided by the Cobra front-end to construct standalone checkers. We will, however, not discuss this application of the Cobra tool further in this paper.

Internal Data Structure When analyzing C or C++ code, the user can choose to explore the original sources, or the preprocessed version of the code. Preprocessing with *gcc* is enabled by default. It can be replaced with any other conforming compiler (e.g. *clang*).

To enable interactive use, Cobra performs only limited additional processing of the input stream, instead choosing to provide the available data directly to the user in a form that supports fast exploration. In a typical use, when Cobra is used to explore the source code of a C program, Cobra uses a lexical analyzer to separate the input stream into lexical tokens. The lexical analyzer, though, is not used as a front-end for a parser, but as a generic *classifier* that recognizes and categorizes key elements of the input data. The general context is illustrated in Figure 1.

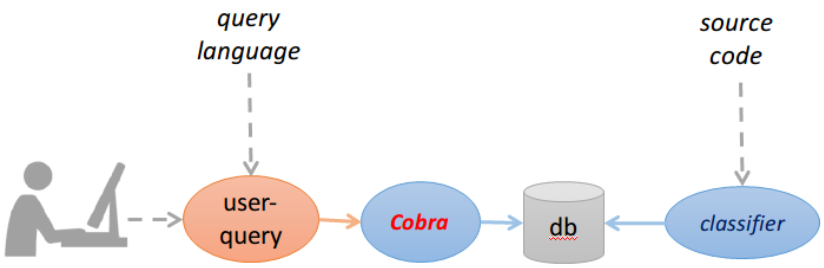

Fig. 1. Cobra context.

The lexical analyzer tags items in the usual categories, such as *identifier*, *constant integer*, *keyword*, *qualifier*, *preprocessing directive*, *comment*, *operator*, *type*, etc. Tokens that do not fall into one of these categories, such as parentheses, braces, brackets, semi-colons, etc., remain uncategorized as plain text items.

Each subsequent token in the input stream is stored as an element in a single linked list that contains all the data, in the order in which it is read. Cobra is generally applied to sets of files, which means that the token sequence that is created contains the contents of all files in the order in which they are read.

The key to the working of Cobra is the data that is available in the otherwise extremely simple data structure (the linked list). Each element (i.e., token) in the linked list carries only the information that is needed to determine its context in the data-stream, plus a small amount of information that speeds up navigation across tokens.

Each token structure records the text of the item, the filename and linenumber on which it is located in the source, the level of nesting at which the token appears in curly, round, and square braces, the length of the corresponding source text, the category to which it was assigned (e.g., *identifier*), the sequence number of the token in the input stream, pointers to the next and the preceding token, and most importantly a *mark* field that identifies if the token was selected in a search.

The marks define the set of currently selected tokens. The user can use query commands to store, modify, restore, or combine these token sets.

In addition, for all curly, round, and square braces, each token structure also contains a link to its matching token at the same level of nesting (if present). That is, an open curly brace character { contains a forward link to the matching close character }, and similarly the close character contains a backward link to the matching open character at the same level of nesting. Figure 2 illustrates this structure for the following fragment of a C program.

```
int
main(void)
{
  ...
```

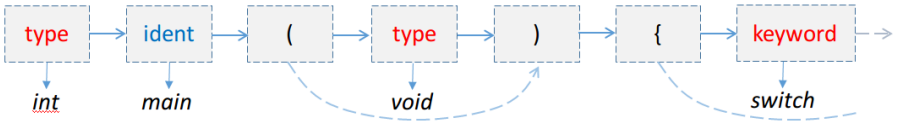

Fig. 2. Fragment of the linked list data structure constructed for a small C program.

```
}
```

Note that the text string *main* is recognized as an identifier, with a pointer to the text itself. Further, *int* and *void* are recognized as data types, and *switch* is recognized as a keyword. The otherwise uncategorized tokens (and) are associated with forward and backward links, as are the tokens for the curly braces { and } (not shown in Figure 2).

The advantage of the linked list data structure is that it can be scanned very quickly. Because the data structure is laid out in memory sequentially, it is cache-friendly. Further, as we will discuss in more detail below, any search of the data structure for specific token sequences can be trivially split across multiple cores, to obtain a further speedup when needed.

In the following sections we briefly discuss the main query types that are supported by the tool, starting with the most basic type of interactive queries. After this we discuss inline programs in Section 3.1, followed by regular token expressions (which are converted into inline programs internally) in Section 3.3. As noted earlier, we will not discuss yet another way to use the Cobra framework, which is to write standalone checkers that are linked with the Cobra front-end.

3 Query Languages

3.1 Interactive Queries

In interactive mode Cobra accepts two types of queries: (a) global commands that provide general information about the code that is explored, or the state of the search, but that do not modify the state of the system, and (b) query commands that are used to add, delete, move, or extend marks to identify patterns of interest in the code.

Every query command is always applied to *all* tokens in the input stream. To do so, the Cobra engine visits every token in the data structure that was built and applies the command. In some cases this means that it will need to search the context of a given token, e.g., to find the closest neighboring token that matches a user-defined pattern. The essence is though that each token is visited in turn, and the same operation is applied to every single one. The key to Cobra's operation is that this can be done extremely fast, even for large code bases. We will see later how we can also use multi-threading to divide query resolutions over multiple threads to gain a further speedups.

Queries can be used to set, move, remove, extend, stretch, or interrogate user-defined mark points on tokens, and they can be used to define and inspect ranges of marked tokens. Most types of query commands support a number of optional qualifiers that can be used to fine-tune their operation.

In total, Cobra supports approximately thirty different global and query commands. Most common queries require only four of five of these commands though. We will cover the most important query types with the help of some examples.

The simplest type of Cobra query is to identify and mark specific types of tokens in the input stream, which normally corresponds to the lexical tokens derived from the preprocessed source code of an application. As a first example, we may want to mark all uses of the library call *malloc*, and list the matches in various forms.

```
$ cobra *.[ch] # use cobra on its own sources
30209 # the number of tokens in the input
: mark malloc
2 matches
: list
cobra_prep.c:
  1:     231  'malloc'
cobra_re.c:
  2:      78  'malloc'
: display
cobra_prep.c:
  1:    231  {  void *n = malloc(size);
cobra_re.c:
  2:     78     s = (void *) malloc(n);
: pre
cobra_prep.c:
  1:   231  { void * n = malloc ( size ) ;
  1:                     ^^^^^^
cobra_re.c:
  2:    78  s = ( void * ) malloc ( n ) ;
  2:                       ^^^^^^
: q # end the session
$
```

In this example, Cobra reads in about 16K lines of C sources and header files, and creates a linked list data structure with about 30K lexical tokens. The Cobra command prompt is the single colon ":". In the session shown above we give a *mark* command (which can also be abbreviated to a single *m*) followed by the token text we are looking for: *malloc*. Cobra locates two matches in the code. We next issue a global command named *list* (or its abbreviation *l*), which prints the location of the two matches.

The first match is found in the source file named *cobra_prep.c* on line 231, and the second is found in file *cobra_re.c* on line 78.

The *list* command only shows the matching tokens, but not their context. To see the source line where each token is found we can use the *display* (or *d*) command. If we need still more context, we can also ask Cobra to display the preprocessed version of the code, with the matched tokens marked explicitly. This can be done with the *pre* (or *p*) command.

Using abbreviations, which will quickly become the preferred way of working with the tool, the entire query sequence above can be written in a single line as follow, using a semi-colon as a command separator:

```
: m malloc; l; d; p
```

or when executed from the command-line:

```
$ cobra -e 'm malloc; l; d; p' *.[ch]
```

Even with the minimal functionality we discussed so far it is already possible to obtain quite useful behavior, for instance to quickly generate a broad range of code statistics for a large application and display it on an web dashboard page. Figure 3 shows an example of a graph that was generated in a fraction of a second for an early version of the Cobra code itself, plotting the frequency of use of different data-types and qualifiers. The data is generated with a sequence of interactive Cobra query commands like this:

```
: r; m @const_int; = int_num
: r; m void; = void
...
```

which are postprocessed with a small shell script and imported as data into a Google online spreadsheet. The *r* is the abbreviated notation for the Cobra *reset* command, which clears all previous matches before a new mark operation is performed. The = command prints the number of matches, optionally preceded by the text that is given as an argument.

We can also display the context of each match in more detail. For instance, to display the three lines before and after the first match, we can issue a *d* command with some additional arguments:

```
: r; m malloc; d 1 3
cobra_prep.c:
  1:    228
  1:    229  void *
  1:    230  emalloc(size_t size)
  1: >  231  {  void *n = malloc(size);
  1:    232
  1:    233     if (!n)
  1:    234     {    fprintf(stderr, "out of memory\n");
:
```

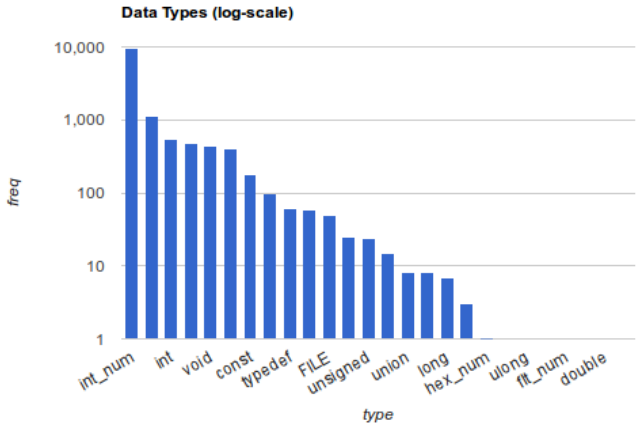

Fig. 3. Example of a Cobra generated dashboard.

Note that the call to *malloc* appears in a function called *emalloc*, which itself is not matched by the *mark* command we used because that name does not fully match the string we provided. It is possible to match on a fragment of a token text, using standard regular expressions in mark operations, for instance by replacing the query sequence with:

```
: r; m /malloc; d 1 3
```

To mark a token type (or category), rather than the literal text that is associated with each token, we can use a type query by preceding the reference with the @ symbol:

```
: reset # remove all current marks
: mark @qualifier
100 matches
: p 1
  1:    87  const Prim * q ;
  1:        ^^^^^
```

Capturing Context So far we have only talked about locating specific terms or tokens in a code base. For most queries of interest, though, we should be able to capture things in a specific context. We discuss this next.

Assume that we want to check that every switch statement in a C program has a default clause. This is Rule 16.4 in the well-known MISRA 2012 guidelines for safety critical software development. We could do a rough check with the Unix tool *grep*, for instance as follows:

```
$ cat *.[ch] | grep -c -e switch
```

```
182
$ cat *.[ch] | grep -c -e default
179
```

Clearly, the two numbers we get are not the same, but this does not mean that the rule is necessarily violated, because grep will report also occurrences of the keywords that appear inside strings or comments. We can use Cobra to perform a more accurate check.

We start the tool, and use the default preprocessing step to make sure that even switch or default clauses that appear in macros will be interpreted correctly.

```
$ cobra *.[ch]
305232
: m switch
111 matches
: r; m default
73 matches
```

This already looks very different, because we now accurately match the keywords in the preprocessed code, and not also in comments or strings.

The numbers do not match, so how do we find the switch statements that do not have a default clause? Three Cobra commands suffice to locate them (after a *reset* to clear the earlier matches):

```
: r; m switch
111 matches
: next {
111 matches
: contains top no default
38 matches
```

The *mark* command, abbreviated to *m*, again marks all keywords that match the string *switch*. The *next* command that follows it moves the mark point for each of those matches to the next occurrence of a curly brace token, which is part of any syntactically valid switch statement.

The *next* command is one of a small group of navigation commands that can be used to move mark points. If used without an argument, the command simply moves each existing mark forward to the immediately following token. A related command is *back*, which can be used to move each existing mark back one position, or to a specific token if given an argument. But back to our command script.

If, when executing the command *next* {, there is no curly brace token anywhere after an existing mark, (e.g., when the program is not syntactically valid) the mark point is deleted. We can see that this was not the case here, since the number of matches that is reported after the execution of the *next* command remained unchanged.

Next, the task is to identify those bodies of switch statements that do *not* contain the keyword *default* at the nesting level of the *switch* statement that is being inspected. That is: we don't want to match on default keywords that may appear in another switch statement that may be nested inside the current statement. This is achieved by using a *contains* query command with the qualifiers *no* and *top*. The *contains* command is defined to work over ranges that are defined by default for all matching pairs of braces, parentheses, or brackets.

Since the current mark points have been positioned on an opening curly brace, the *contains* command will restrict the search for matches to the default range that is defined for that symbol.

The result shows that 38 of the 111 *switch* statements do not contain a *default* clause. At this point we can again display the line numbers for all the matches with a *list* or *display* command.

If we make use of the shorthands that Cobra provides for the key commands, we can write the complete query as a single command, including the display of the matching lines at the end:

```
$ cobra -e 'm switch; n {; c top no default; d' *.c
```

We can define this sequence also as a script, and place it into a library file, so that we don't have to remember this particular sequence of queries. We do that so by writing the definition into a file, say *badswitch.cobra*, followed by a call to that script:

```
$ cat badswitch.cobra
def badswitch
      r; m switch; n {; c top no default; d
end
badswitch
```

Cobra can read and execute that script file in interactive mode with a dot command, as follows:

```
$ cobra *.c
: . badswitch.cobra
```

Finally, we can also execute the script file directly from the command line, as follows:

```
$ cobra -f badswitch.cobra *.c
```

Using Sets A more interesting use of the Cobra command set is to use sets. We will illustrate this with the help of an example, which is to find all cases of backward goto jumps in a program. Backward goto jumps are a violation of Misra Rule 15.2. Notably, the earlier Misra Rule 15.1 strongly discourages the use of all goto statements, but does not require it.

First we find the goto statements, with a simple *mark* command (we will assume that no other tokens were marked earlier).

```
: mark goto
115 matches
```

We now advance the mark point from the goto to the label name. That is easy, since the name must follow the keyword directly:

```
: next
115 matches
```

We now save the current collection of marks in a predefined set as follows:

```
: >1
```

By default there are four sets available to store matches, but this can be changed where needed with a command-line option.

Set one now contains the set of all goto statements that appear in the code, with a mark placed on the corresponding label names. Some of these label names will appear in the code before the goto, and some will appear after it, and our task is now to distinguish these two cases.

We can do so by searching forwards from each label name to a possible point in the code where the same name appears again. That will be true only for forward jumps. The command we can use for this is the Cobra *stretch* command:

```
: stretch $$
102 matches
```

What the *stretch* command tries to do is to define a new range for the matched tokens that starts at a current mark and extends to the pattern that is specified as its argument. In this case the argument we need is a reference to the (text of) the currently matched token, which is available in the symbolic reference $$.

For our current purpose we do not actually need the ranges themselves, but only the identification of those goto statements in our set for which it is possible to create the range.

The number of matches went down to 102, which means that for 13 goto statements the range could not be created. Those should correspond to the backward jumps that we are interested in.

At this point, though, we have only the forward jumps matched. So we have to do some more work.

We first store the new subset of matches in a second set:

```
: >2
```

We have now defined two sets: one with all jumps, and a second with only the forward jumps. What we need is the difference between those two sets. To obtain it, we first we clear the current marks and reload the marks from the first set.

```
: reset
: <1
115 matches
```

Now we use the second set to find only those marks that are not shared between the two sets.

```
: <^2
13 matches
```

This uses the restore operator `<` in combination with the modifier ^ (caret) to compute the exclusive-or of two sets (the current set and set 2). Other available combinations include `<&` for computing set intersection, and `<|` for computing a set union.

After the restore operation we have identified the 13 backward jumps, which we can now list in any of the available forms we discussed before. Note that this check identifies the backward jumps even if a given label name is used in both forward and backward gotos.

We can summarize the complete check again in a single line, using shorthands as before:

```
: r; m goto; n; >1; s $$; >2; r; <1; <^2
```

and we can store the sequence in a script library for later use.

Recursion As another example of a Cobra query, consider the problem of identifying all recursive function declarations in a C program.

We can do so in Cobra with a script of nine commands. For clarity we give the script here without abbreviations, and without concatenating short commands on a single line. The # symbol is the Cobra comment delimiter.

```
: mark @ident (   # mark identifiers followed by a (
: next            # move the marks to that opening (
: jump            # move the marks to the matching )
: extend {        # retain the marks only if the next
                  # token is a { (the start of the body)
: jump            # move the marks back to the opening
                  # braces of the parameter lists
: back            # move the marks back to the function name
: stretch top }   # define new ranges starting at those
                  # names to the end of the function bodies
: mark ir $$      # retain the marks only of marked function
                  # name appears inside each range
                  # and move the remaining marks to those
                  # names (i.e., likely recursive calls)
: extend (        # retain the resulting marks only if
                  # they are followed by a parameter list
```

The identification of all function definitions and function calls has been predefined in Cobra itself. The command *fcg* can be used to generate the function call graph in dot-format. When more local information is needed, for instance to find just all callers and callees of a function, the predefined command *context* can also be useful. Finally, the predefined command *ff* can be used to find a specific function definition by name, and the command *cfg* can be used to display the control-flow graph of a named function.

3.2 Inline Programs

The most powerful mechanism that the Cobra tool has for expressing queries is an inline program. Cobra inline programs are enclosed between character delimiters:

```
%{
 ...
%}
```

Since an inline program typically consists of multiple lines of text, and will generally be edited until it performs as intended, it is usually not typed in interactively, but stored in a file and then called from the command line, as in:

```
$ cobra -f file_with_cobra_program *.[ch]
```

This call executes the inline program, returns the result, and stops. The program can of course also be written interactively. In that case it will run immediately once the closing delimiter is seen.

Cobra inline programs can be used to write more complex queries than interactive commands allow. This includes queries that require the use of conditional selection and scanning larger parts of the code to identify patterns of interest. Cobra programs can store data, add, delete, or move token markings, and print more information than is possible with interactive queries. Cobra programs, though, have their own syntax and grammar specific to their typical use, and this syntax is different from the interactive query commands we discussed earlier. As can be expected, there is a way to express any interactive query also in inline programs, but the syntax of inline programs is different, and significantly more expressive, as we will see shortly. A sample collection of both Cobra scripts and inline programs is part of the standard tool distribution.

As a first example of an inline Cobra program, consider this definition:

```
%{
 print .fnm ":" .lnr ": " .txt "\n";
%}
```

This program contains just a single print statement that is executed once for each token in the data structure that was built for the program being analyzed. The print command takes any number of arguments, and it will print output

for each argument that matches its type: numbers are printed as numbers, and strings as strings, with the special characters `\n` and `\t` interpreted as newline and tab respectively. A sample single output line from the above program can be:

```
cobra_lib.c:1237: if
...
```

A line of source code will typically hold several tokens, so a single filename:line-number combination does not uniquely identify the tokens. When needed, we can identify individual tokens by adding also a sequence number to the output, which is available in the predefined integer token attribute *.seq*. A short example program that prints this sequence number and text of each token is:

```
%{
 print .seq " " .txt "\n";
%}
```

Just like in C, statements in Cobra programs are terminated by a semi-colon.

There are three types of token attributes that can be referred to in inline programs. Grouped by the type that each of these return, they are:

- Strings: *.fct*, *.fnm*, *.txt*, *.typ*
- Numbers: *.round*, *.bracket*, *.curly*, *.len*, *.lnr*, *.mark*, *.seq*, *.tag*
- Token references: *.back*, *.bound*, *.forw*, *.nxt*, *.prv*

The dot that precedes each attribute refers to the current token being processed. The language also allows us to define, set, and query token variables and refer to their attributes, e.g. as *q.len*, with commands that we will discuss shortly. If a literal version of each keyword is needed it can be enclosed in quotes.

In addition to token attributes, there are a few additional keywords in the language. They inline language defines the following control-flow keywords with the usual semantics.

- Keywords: *if*, *else*, *while*, *break*, *continue*, and *goto*.

The following four additional terms are predefined:

- Predefined: *CobraArg*, *Next*, *Stop*, and *print*.

CobraArg is a string that can be specified by providing Cobra with a *-e* argument on the command line.

Next is a meta-command that when executed will to end the processing of the current token and advance to the processing of the next token, starting the inline program anew.

Stop is a meta-command that when executed will abort the processing of tokens and end program execution.

Print, finally, is the predefined print function, accepting one or more number or string arguments, discussed above.

A complete Cobra program is a sequence of statements, each terminated by a semi-colon. The keywords *break* and *continue* can only be used inside *while* statements, with the usual effect on loop control.

If and *while* statements are followed a sequence of one or more statements that must be enclosed in curly braces. The syntax of an *if* statement is:

```
if ( expr ) { ... } [ else { ... } ]
```

where the *else*-part is optional, as usual. Similarly, the syntax of a *while* statement is:

```
while ( expr ) { ... }
```

There is also a range of operators in the language, the most important of which is assignment `=`. Assignments are written the conventional C-like way:

```
lhs = rhs;
```

Where the left-hand side (lhs) is either the token attribute *.mark*, a token location . (dot), or a token reference variable (to be discussed). The attribute *.mark* is the only token attribute that can be modified by the user. Some examples of Cobra inline program statements are:

```
.mark = 5;    # set the value of .mark to 5
.mark--;      # decrement the value of .mark
.mark++;      # increment the value of .mark
. = .nxt;     # move forward one position
. = .prv;     # move backward one position
. = .forw;    # move to end of range associated with .
. = .back;    # move to start of range associated with .
q = .;        # assign a token reference variable q
q = .nxt;
. = q;        # move to the location pointed to by q
. = q.forw;
```

A single name is always interpreted to be a token reference variable. Such reference variables do not need to be declared.

Another type of data object supported in the language is an associative array. Associative arrays also need not be declared before they are used. They are identified by a basename and an index, for instance as follows:

```
basename[index] = value;
```

The index can be either a text string or a number, and the value stored can be a value, a text string, or a token reference. Some examples of the use of associative arrays are:

```
X[.txt] = .mark;
X[.txt]++;
Y[.mark] = .fnm;
Z[.fnm] = .;
```

There is one restriction to the use of the values stored in associate arrays, and that is that when a token reference is stored, its attributes cannot be referenced directly. So if we use the definition of `Z` from above, we cannot refer directly to `Z[.fnm].mark`, but must do so indirectly, for instance as:

```
q = Z[.fnm]; q.mark++;
```

Expressions in the inline language include the usual set of unary and binary operators. The binary operators are:

```
arithmetic:  +, -, *, /, %
boolean:     >, >=, <, <=, ==, !=, ||, &&
```

The unary prefix operators are:

```
!      (logical negation)
-      (unary minus)
~      (true iff .txt contains pattern, eg ~yy)
^      (true iff .txt starts with pattern, e.g. ^yy)
#      (true iff .txt equals pattern, e.g. #yy)
@      (true iff .typ matches the given type, e.g. @ident)
```

Note that the use of the # operator requires some caution because it can also be used as a comment delimiter. The rule is that if the # symbol is followed by a space or another # symbol, it is interpreted as a comment, and if it is followed by text it is interpreted as the unary operator.

In the remainder of this section we discuss three examples of relatively simple Cobra inline programs.

Example 1 The following example shows how we can match on a text string that is specified on the command-line argument to Cobra itself.

```
$ cat igrep.cobra
%{
 if (@ident && .txt == CobraArg)
 {     print .fnm ":" .lnr ": " .txt "\n";
 }
%}
$ cobra -f igrep.cobra -e j *.c # CobraArg is set to j
cobra_lib.c:1824: j
...
cobra_lib.c:2041: j
cobra_lib.c:2041: j
cobra_lib.c:2041: j
```

Note that this matches on tokens, and that there can be more than one match per line of source code. Line cobra_lib.c:2041 above, for instance, has three occurrences of an identifier that matches of the token text:

```
cobra_lib.c:2041: for (j = 0; x && j < span; j++)
```

In this case, a simpler mechanism exists, using the interactive query language, to achieve the same effect with the command:

```
$ cobra -e 'mark j; display' *.c
$ cobra -e 'm j; d' *.c     # using abbreviations
$ cobra -e 'j' *.c          # even shorter
```

Example 2 This example illustrates the use of a while loop and token references. The program counts the number of cases in a switch statement, taking into account that switch statements may be nested.

```
$ num cobra_progs/nr_cases.cobra
 1   def nr_cases
 2   %{
 3      if (.curly > 0 && #switch)
 4      {  # introduce token variable q:
 5         q = .;
 6         . = .nxt;
 7         if (.txt != "(" )
 8         {  . = q;
 9            Next;
10         }
11         . = .forw;
12         . = .nxt;
13         if (.txt != "{")
14         {  . = q;
15            Next;
16         }
17
18         q.mark = 0;
19         while (.curly >= q.curly)
20         {   if (.curly == q.curly + 1
21             &&  (#case || #default))
22             {   q.mark++;
23             }
24             . = .nxt;
25         }
26         print q.mark " " .fnm ":" q.lnr "\n";
27         . = q;
28      }
```

```
29   %}
30   end
31   nr_cases
```

Running this program produces output like this, reporting the size of switch statements:

```
$ cobra -f cobra_progs/nr_cases.cobra cobra_lib.c| sort -n
3 cobra_lib.c:1129
...
10 cobra_lib.c:963
22 cobra_lib.c:920
```

We will discuss this program, which covers almost all features of the inline language, line by line.

- Line 1 defines the code as a script, so that we can call it by name on line 31.
- The condition on Line 3 matches only tokens that have a nesting level for curly braces greater than zero (meaning that it only looks inside function definitions), and tokens that contain the text *switch*. Because switch is not a Cobra keyword (it is a C keyword) we can match it with a # operator. If we needed to match on text that happens to be a Cobra keyword such as `while`, we can match the text as a literal string: `.txt == "while"`.
- Line 5 contains an assignment to `q`, thus introducing a new variable name to point to the current token location, represented by the dot. Since `q` is a general token variable, we can use it to refer to all the predefined attributes of a token, even when it is not associated with a specific token location yet.
- Line 6 advances the token position forward one step. Note that this means that the implicit outer loop over all token values will also advance, unless we take care to reset the location like is done on lines 8, 14, and 27.
- Line 7 checks if the current token match an open round brace. If it does not, the code jumps back on Line 8 to the location we stored in `q` (line 5), and yields control back to Cobra on line 9, so that it can advance to the next token position and repeat the program.
- If we reach line 11, we know that the current token is `(` so we can move forward to the matching closing brace. At this point we will be at a token matching `)`, and on line 12 we move past it to the token position that follows.
- Line 13 checks if the token we see now is an opening curly brace. If not, we probably have a syntactically invalid C program, so we abandon the processing again, jump back to the original position remembered in variable q before yielding back control to cobra.
- Line 18 sets the value of the q.mark to zero. The while loop on lines 19-25 now counts all tokens matching the string *case*, within the body of the switch, but only counting such tokens that appear at the top level of nesting. This is done by checking the nesting level that is recorded in the attribute `.curly` from the current token, and comparing it to the value that is remembered in variable `q` for the switch statement that is currently being processed. Note

that `q` points to a location just outside the body of the switch statement, so we have to add one to the value of `q.curly` to get the top nesting level inside the switch body.
- Line 24 advances us through the code of the switch statement until the condition of the loop on line 19 no longer holds, when we exit the body of the switch statement.
- Line 26 prints the number of case clauses seen, together with the filename and line number of the switch statement itself.
- Line 27 restores the token position to the one recorded in `q`, so that when Cobra advances the token position for the next round of processing it will point to the statement following the token matching *switch* that we just processed. In this way we will be able to count also switch statements that appear in code nested inside the body of the statement we just processed, and get a count for these as well.

Example 3 The only concept that we have not yet discussed are associative arrays, which can be used to associate a value, string, or token reference with either a text string or a value in a named array. The following somewhat naive example will suffice to illustrate the basic concept:

```
%{
   if (#float)
   {   . = .nxt;
       if (@ident)
       {   .mark = 1;
           X[.txt] = .mark;
           print .fnm ":" .lnr ": declares '" .txt "'\n";
       }
       Next;
   }
   if (@ident && X[.txt])
   {   print .fnm ":" .lnr ": uses '" .txt "'\n";
   }
%}
```

We use an array named `X` (the name is arbitrary) to remember the current value of `.txt`, and associate it with the value of attribute `.mark` of the current token. In this case the right-hand side of the assignment is a value, but it can also be a string, or a token reference. The value stored in `X` is retrieved in the condition of the second *if* statement. If there turns out to be no value stored for the string specified, the value returned will be zero. The second *if* statement checks for every identifier whether the corresponding text string from `.txt` was recorded before. If so, we know that this identifier first appeared following the C keyword *float*, and must therefore be a floating point variable.

For simplicity, this version ignores that a type name like *float* can be followed by a comma-separated list of identifier names, and can include initializers.

The following version of this program preserves a little more detail by storing the location of the declarations as well, by using a token reference. We can now check that the retrieved value corresponds to an actual location by checking that the line number field of the reference is non-zero.

```
%{
   if (#float)
   {   . = .nxt;
       if (@ident)
       {   Store[.txt] = .;   # store current location
           print .fnm ":" .lnr ": declares '" .txt "'\n";
       }
       Next;
   }
   if (@ident)
   {   q = Store[.txt];
       if (q.lnr != 0)
       {   print .fnm ":" .lnr ": uses '" .txt "' ";
           print "declared at " q.fnm ":" q.lnr "\n";
   }   }
%}
```

Variable and array references are preserved across multiple invocations of Cobra programs, which helps to make the following example work. The second inline program below stops all processing once the *print* statement has been executed.

```
%{
  # check the identifier length for all tokens
 if (@ident && .len > q.len)
 {   q = .;
 }
%}
%{
 print "longest identifier: " q.txt " = " q.len " chars\n";
 Stop;
%}
```

3.3 Regular Token Expressions

It is relatively straightforward to add support for alternative query definition formalisms to the toolset, by converting each query from the new formalism into an inline Cobra program.

Cobra includes such a converter for basic regular expressions over lexical tokens. As an example, consider the following token expression for matching *switch* statements that do not contain a default clause, where for convenience we will ignore the special case of potentially nested *switch* statements:

```
$ cobra -re 'switch \( @ident \) { ^default+ }' *.c
```

The converter recognizes the usual meta-symbols for regular expression: `.` (any token), `*` (zero or more), `+` (one or more), `?` (zero or one) `|` (choice), and the round braces `(` and `)` for grouping. It also recognizes `^` for negation (note that using `^` to match the beginning of a line of text is not useful in this context).

The `@` symbol is again supported for matching on the type, rather than the text, of a token. We used that here to match an identifier.

In the example pattern we match on the lexical token *switch*, which is directly followed by an open round brace (which is escaped because it is not a meta-symbol here), followed by an arbitrary identifier, which is followed by a closing round brace. After this we match on any block enclosed in curly braces which does not contain the keyword *default* anywhere.

The converter makes sure that the nesting level of the closing curly brace in the regular expression matches that of the open curly brace that preceded it. Similarly, it will do the same for round and square braces. This means that we could also replace the token reference `@ident` with the meta-symbols `.*` without changing the meaning of the pattern.

The regular expression is converted into a deterministic finite state automaton with standard algorithms, which is then converted into an inline Cobra program to perform the matching. The conversion process takes a fraction of a second.

Variable Binding The following example illustrates the mechanism we defined for variable binding. Consider the case where we want to find any use of the control variable from a *for* loop that appears inside the body of the loop. We can express this as follows:

```
$ cobra -re 'for \( x:@ident .* \) { .* :x .* }' *.c
```

The notation `x:@ident` will bind the value of the matching identifier to the variable `x` (any name will do of course). Inside the body of the loop, the bound variable name is now referenced with the notation `:x`. Note again that the converter makes sure that the nesting level of the closing curly brace will match that of the open curly brace that preceded it. The presence of that closing curly brace makes sure that the match of bound variable `x` can only occur inside the body of the loop, and not beyond it.

3.4 Dynamic Analysis

Havelund, in [2] compared the performance of seven different tools for the runtime analysis of event logs. The length of each log varied from close to 30 thousand events to a little over two million events. Each event was recorded as a single line with the three fields: *event*, *task*, and *resource*, where *event* is either *grant* or *release*, and *task* and *resource* are numbers between 1 and 5000.

The tools Havelund evaluated, using different theories and implementation strategies, were compared on the speed with which they could check three requirements:

- R1: Every resource granted should eventually be released.
- R2: A resource and only be released by task t if it was earlier granted to task t and not yet released.
- R3: As long as a resource is granted to a task, it cannot be granted again, neither to the same task nor to any other task.

We can express these requirements as follows:

```
%{
    q = .; Stop;       # set q to a non-zero value
%}
%{
    if (#grant)
    {   . = .nxt;      # move to task field
        . = .nxt;      # move to resource field
        if (Held[.txt] == 1)
        {      print .fnm ":" .lnr " violates r3\n";
        } else
        {      q.mark++;
        }
        Held[.txt] = 1;
        Next;
    }
    if (#release)
    {   . = .nxt;     # skip to task
        . = .nxt;      # skip to resource
        if (Held[.txt] != 1)
        {      print .fnm ":" .lnr " violates r2\n";
        } else
        {      q.mark--;
        }
        Held[.txt] = 0;
    }
%}
%{
    if (q.mark != 0)
    {    print .fnm " " q.mark " resources still held: ";
         print "violates r1\n";
    }
    Stop;
%}
```

We used a prelude to initialize a token variable here, executing just a single statement before stopping, and a postlude to check that the *mark* field of that

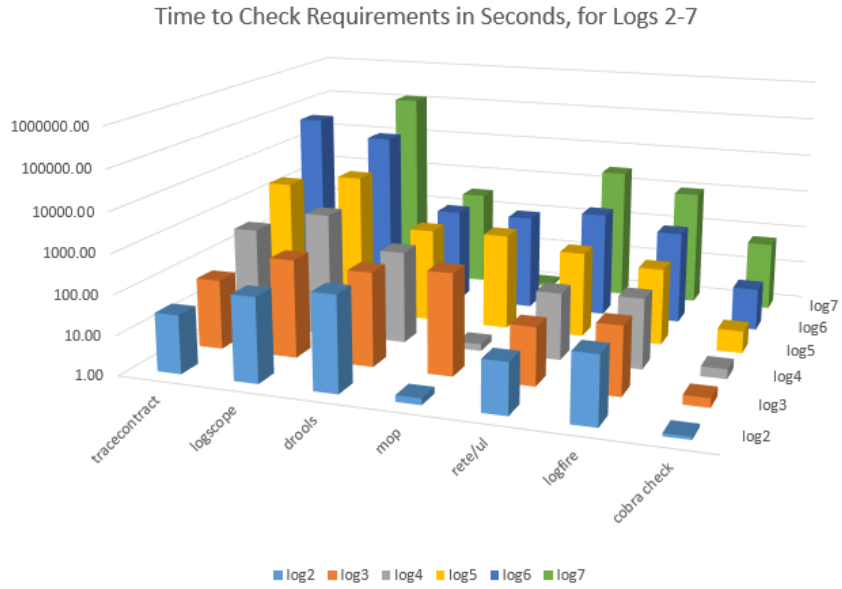

Fig. 4. Time to Check the Requirements, excluding Parsing for all Logs

variable is zero, to indicate that all resources that were granted have also been released.

The match for a *grant* event checks that the resource referenced is not already held, and then records the grant in associative array *Held*. If there is no violation, it also increases a count of the number of resources currently being held in the *mark* field of *q*.

The match for a *release* event checks similarly that the resource was indeed being held, and then clears the corresponding entry in the array. On success, the count maintained in *q.mark* field is decremented.

The program is relatively straightforward, and when useful it would of course be possible to add a converter from a more specialized formalism into the more detailed code of an inline program. The main question we want to address here is if the performance of Cobra could be competitive with the specialized tools for dynamic analysis that were described in Havelund's overview paperc̃iteHavelund.

In Table 1 of [2] performance is reported in two parts. The first part is the time that each tool takes to read and parse the logs, the second is the time taken to check the three requirements. In Havelund's measurements, the time to parse the log and build an internal Java data-structure was the same for all tools at approximately 23 seconds per one million events. Given the much simpler data structure that Cobra builds, we can expect shorter times needed to build this data structure for Cobra. On average Cobra spends about 2.5 seconds per one million events to parse the log, or about ten times faster.

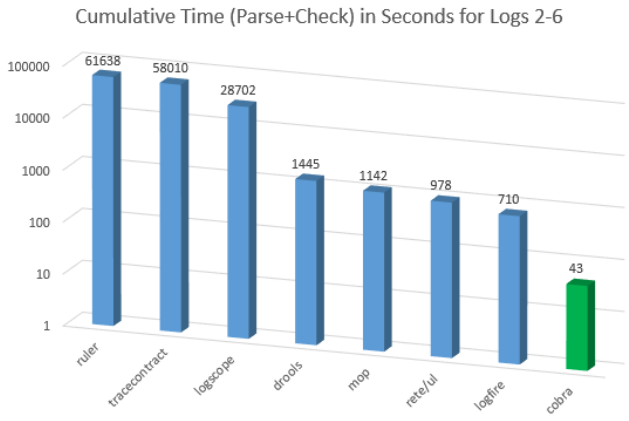

Fig. 5. Cumulative time to parse and check all requirements for Logs 2-6, with the tools sorted by time from high to low.

For the performance comparison, we first measured the time needed to check the requirements for each log, excluding the time needed to parse each log. Those results are shown in Figure 3. For clarity, the y-axis in this and the next two figures is plotted on a log-scale.

We also measured the total cumulative time for all logs that is used by each tool to read in each log and perform the checks. Not all tools were able to handle all logs, so we separated the results in two graphs. Figure 4 shows the cumulative times for all tools for the first five logs, which all tools were able to process. Separately, Figure 5 shows the cumulative times for the tools that could handle all six logs.

In a few cases the Mop tool can be seen to outperform Cobra in these checks, but in the large majority of checks, and especially for the cumulative end-to-end times, we see the surprising result that the Cobra checks turned out to be considerably faster than the specialized dynamic analysis tools. The main reason for this is likely that the specialized tools include more sophisticated support for specification formalisms with the matching data structures, where the Cobra specification style for these checks is much more rudimentary.

4 Performance

The processing of basic queries for large applications can be sped up by using multi-threading. The Cobra command-line argument -N can be used to define the use of multiple cores, to process equal parts of the token sequence in parallel, whenever possible.

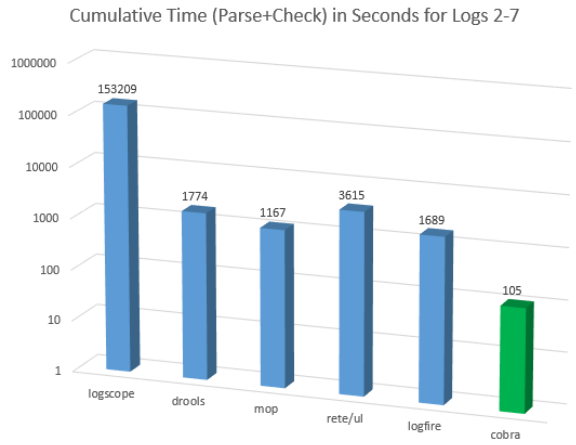

Fig. 6. Cumulative time to parse and check the requirements, for Logs 2-7, listing the tools in the same order as in Figure 4. This chart excludes the tools that fail to handle Log 7.

For instance, `-N4` tells Cobra to use four cpu-cores for processing, with potentially the corresponding speedup, barring the usual effects of caching and non-uniform memory access in large RAM memory sizes.

The definition of multiple cores does not change how inline programs are processed though, because each run of these programs generally stores information in variables or arrays that should be accessible to all cores at the appropriate point in the processing.

The slowest step in the use of Cobra on applications of a million lines or more is the standard preprocessing phase, if enabled. This preprocessing is done with the standard C preprocessor (e.g., gcc -E), outside Cobra itself. To avoid having to repeat the preprocessing every time Cobra is started it can be a substantial speedup to first preprocess all source files outside Cobra, with the right preprocessor directives, and then store the result in a single file that is given to Cobra, while disabling further preprocessing in that tool with command-line argument `-n`.

Another method to speedup the starting phase of a Cobra session for very large applications is to suppress the processing of header-files (i.e., to exclude them from the token sequence). This can be done with the Cobra command-line argument `-z`.

An example can illustrate the effect of these options.

We measured performance of Cobra for an application of three Million lines of C source code and header files combined. The application contains 1,725 source files, defining a total of 50,986 C functions.

The default startup of a session, with full preprocessing and including all header-files, takes about 1,800 seconds and produces a data-structure of 104 Million tokens.

Using eight cores, standard queries are processed over this data structure in about three seconds per query, which is fast enough to be useful for interactive use.

If, however, we can skip the preprocessing step and the inclusion of header files, by using `-n -z`, the startup reduces to 15 seconds; but we then of course do not get the benefit of seeing the preprocessed code. We can, however, preprocess the code separately and let Cobra read in a single file with all the preprocessed code, again using command-line options `-n -z`. The file with all preprocessed sources now contains 36 Million lines.

This reduces the Cobra startup time to 90 seconds and produces a data structure of about 13 Million tokens. If we prune the preprocessed file a bit more by removing redundancies that do not affect the analysis results (e.g., removing repeated includes of the same headerfiles), we can reduce this further to about 20 seconds.

Using eight cores the processing of interactive queries now reduces to under one second per query, which allows for real-time query processing even on this size of a code base.

5 Related Work

Some, though not all, of the capabilities of Cobra can also be found in mainstream IDEs like Eclipse or MS Visual Studio. It would be possible to import a Cobra-like infrastructure into such IDEs, though we have not attempted to do so.

A closely related tool we described elsewhere is the *Ctok* program [3]. This tool supports only a small subset of the capabilities of the Cobra tool though.

The best known standalone tool that is comparable to Cobra for exploring the source code of C programs is Cscope [1]. We illustrate in Table 1 how the standard types of queries that are supported by the Cscope tool can be handled by Cobra, requiring just a small subset of Cobra's capabilities.

Table 1 lists each command, assuming that we start with no current matches, and we omit the standard commands to display results.

6 Limitations

The central data structure that Cobra provides for lightweight code browsing and analysis consists only of a simple sequence of tokens. As we have shown, this allows for the interactive resolution of surprisingly many types of queries, but it also means that there are limitations to the types of queries that can be handled. Some more complex queries could be handled, by providing options to compute additional data structures on control and data-flow, but this would start to challenge the premise that Cobra is designed primarily for fast query resolution in interactive sessions. The types of queries that cannot be handled by Cobra will remain the domain, at least for now, of the more standard state-of-the-art commercial static source code analyzers, see e.g. [4].

Table 1. Resolving Cscope-like Queries with Cobra

Cscope Query	Cobra Version (m is short for *mark*)
Find a C keyword or symbol named *foo*	m *foo*
Find a global definition of *foo*	m *foo*; eval (.curly==0)
Find functions called by *foo*()	context *foo*
Find the word *foo* in a string	m @str; m & /*foo*
Find a match for the regular expression *foo* in the code	m /*foo*
Find all code in file *foo*.c	m /.; eval (.fnm == *foo*.c)
Find all code in files with names that include regular expression *foo*	m /.; eval (.fnm == /*foo*)
Find files that include the file *foo*.h (starting Cobra without preprocessing)	m @cpp; m & /*foo*.h
Find all assignments to variable *foo*	m *foo* =

7 Conclusion

The Cobra tool was designed to provide a fast, lightweight alternative to mainstream static source code analyzers. It can be used for interactive resolution of an unexpectedly broad class of relatively simple source code queries in code review sessions, for code analysis, or for code exploration during the code development process itself. The basic data structure that is built by the Cobra tools is simple enough that it supports a quick startup and easy parallelization of most queries.

Despite its simplicity, the query languages that Cobra supports allow us to construct function call graphs, and control-flow graphs on demand and to check compliance with common coding rules.

When needed, the tool can be extended in a relatively straightfoward way with additional functionality to support more complex types of queries. Examples of such extensions could be to add a symbol table, deduce type information, to perform alias analysis. We have shown how the tool can also support custom query languages, such a regular token expressions, which are translated internally into Cobra inline programs.

The approach to restrict basic preprocessing to just lexical analysis was originally intended as a compromise between performance and functionality. The performance gain over mainstream static source code analyzers should be clear. Surprising is perhaps how rich the functionality of the resulting tool has proven to be.

Acknowledgment This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

1. The history of Cscope: `http://cscope.sourceforge.net/history.html`
2. K. Havelund, Rule-based Runtime Verification Revisited. *Int. Journal on Software Tools for Technology Transfer*, Vol. 17, No. 2, April 2015, pp. 143-170.
3. G.J. Holzmann, Tiny Tools, *IEEE Software*, Jan/Feb. 2016, pp. 24-28.
4. Overview of static analyzers: `http://spinroot.com/static`