Software That Checks Software: The Impact of PC-lint

James Gimpel

In this column, James Gimpel gives some fascinating insights into the growth, technology, and impact of the very widely used static analyzer PC-lint in the never-ending battle against the bug.
— *Editors: Michiel van Genuchten and Les Hatton*

It all started with a question from Max Yaffee: "Why don't you do a lint?" This was 1985, in the early days of the IBM PC, when K&R (Kernighan and Ritchie) C reigned. C tools were limited, and function prototypes were discussed but not widely implemented. It so happened that our firm had just released a C interpreter for MS-DOS (C-terp), and, as it turns out, there's much in common between an interpreter and a static analyzer. For example, in addition to the usual components such as a preprocessor, symbol tables, and lexical and syntactic analyzers, both need the ability to process multiple compilation units (.c files). It seemed like a good fit.

Lint was a well-known utility in the Unix world. It was written by Steve Johnson, who also wrote the portable C compiler and invented YACC (Yet Another Compiler Compiler). Some programmers don't like the concept of a separate lint tool, having the mind-set that the compiler should do that task. But I remember when we were working at Bell Labs together, Johnson said that he wrote lint partially because he didn't want to "tip-toe" through the compiler, possibly breaking things while injecting lint-like messages.

Early Stages

So why not write a lint for DOS? For one thing, a static analyzer was regarded as theoretically impossible. This is because of Turing's famous halting problem, which showed that a Turing machine can't necessarily tell if another Turing machine will halt even if it has access to all of that machine's input. The problem with this statement is that many, if not most, people overlook the word "necessarily." Clearly, it's easy to predict that some machines will halt (such as a machine that contains only the **`halt`** instruction). So it's possible for a Turing machine to determine if some other Turing machines can halt. It's also possible for it to determine that some machines will loop forever. The question becomes how likely would it be to determine whether the machine stops. Hence, in extrapolating Turing machines to programs, we need to ask how effective can a static analyzer be in ferreting out bugs in a computer program? We know only that no static analyzer will be able to ferret out all bugs in all programs.

But we didn't have to find all bugs in all programs; we just had to check function call arguments against their corresponding function parameters. I decided this was a fine idea, and within a few months, our ad appeared in *Dr. Dobb's Journal*.

My previous experience linting my own programs left me with reservations. Generally, there were too many false positives and too few ways of suppressing messages. To avoid messages, programs were filled with casts. But because casts can't check the type from which they're casting, the net result can be less reliability than before. Hence, a significant effort went in to making sure, for example, that in the code

```
int n = 12;
unsigned u = n;
```

we wouldn't complain about the lack of a cast.

This article was published in *IEEE Software, vol. 31, no. 1, pp. 15-19, Jan.-Feb. 2014 , doi:10.1109/MS.2014.13*

Message Suppression

Early on, we realized that without message suppression, the programmer was on the horns of a dilemma: either every fragment of code generating a message had to be fixed, or the programmer was doomed to seeing the same message each time he or she ran our product (a form of "eat your supper now or you'll have it for breakfast"). Most compilers allow the programmer to suppress messages by warning level, but this was a rather crude tool. In our first version, you could suppress by individual error message such as **`-e473`** to suppress message **`473`**.

We soon realized that this was far from adequate. By our third version (1988), we supported suppression by the combination of message number and symbol (if the symbol's name appeared in the message). By the mid-1990s, we had grown to include a dozen new suppression options including **`-emacro(#,name),`** which had the effect of suppressing a particular message during the named macro's expansion. This was important because a macro needs to be more general than a specific expression, and the resulting expansion could be suspicious if it appeared in ordinary code. One example of this is the **`offsetof`** macro, which is sometimes implemented as

```
#define offsetof(st,mem) (&((st*)0)->mem)
```

A static analyzer might have conniptions when it sees the expansion of this macro, but a single **`-emacro`** option can quiet things down for all such expansions. As of today, our message suppression has grown to accommodate some 20 different message inhibition options.

Strong Typing

In C and C++, any type can be given a name using the **`typedef`** construction. For example,

```
typedef float Meters;
```

In this case, **`Meters`** can be used in place of **`float`** to obtain a more descriptive declaration—for example,

```
Meters length;
```

But C compilers don't make use of the distinction that the programmer is making. Consider this example:

```
typedef float Seconds;
Seconds time;
time = length; // whoops
```

Although clearly a mistake has been made (**`Seconds`** should never be compatible with **`Meters`**), the compiler issues no warning message.

We introduced strong type checking in version 5.00 (1991) and included the notion of type hierarchies, whereby data could flow between parent and child type but not between child types (by default).

By version 9 (2008), a complete dimensional analysis system was in place that takes, for example, **`length/time`**, and implicitly creates a new strong type whose name becomes **`Meters/Seconds`**.

Value Tracking

It wasn't initially my goal to create a system that could track values through a program. I was, perhaps, intimidated by Turing's halting problem, but the issue always intrigued me as I would give talks about PC-lint in the '80s and speculated on our ultimate ability to detect, for example, the flaw in the following code:

```
int a[10];
for( i = 0; i <= 10; i++ ) a[i] = 0; // One too many
```

The humble beginnings of value tracking came in the form of mere initialization tracking. Most static analyzers, like ours, detect that a variable hasn't been initialized within the body of a function prior to use. One day, our German reseller, Peter Kessler, suggested that we find instances of a variable being used where the variable might have been previously initialized but there nonetheless existed a path from its initial declaration to its use that didn't include that initialization.

We introduced this feature in version 5 (1991). I soon realized that this basic facility could be the underpinning for a full-fledged value-tracking scheme. We didn't have to know all the values a variable might have as long as we knew that the possibility existed of having a bad value. Thus, if we have

```
if( x == 5 ) y = 0;
return x / y;
```

we have enough to issue a warning without knowing the value of *x* or *y*.

Some of our value-tracking system was based on inference-driven analysis. There was much to be obtained by carefully examining the predicates of the **if, while, for, do,** and **switch** statements. Thus,

```
if( y ) x = 14;
else x = 20 / y;
```

is a simple example of a divide by 0 that is easily detected by inference.

In version 7 (1995), we were able to incorporate interstatement value tracking (between statements, totally within a single function). The implementation was supported by an in-house theoretical analysis of the effects of C and C++ statements and operators on the possible value states of variables. We were no longer in the simple world of an interpreter where operations take place on simple values but rather a calculus on sets of possible values. A significant requirement for our scheme was reporting (as well as possible) the location in the code where we obtained our information. In a lengthy function, it was vitally necessary not only to indicate that a subscript could be out of bounds but also to show the places within the code that led to that conclusion. This is in sharp contrast to the typical compiler or lint diagnostic, where it's sufficient to merely point to the code that triggered the complaint. But when the complaint has to do with values that originate in code locations many lines from where the problem is discovered, it's vitally important to reveal as much information as possible. Attention to this detail in the interstatement case served us well when we broadened our horizons to interfunction tracking.

With the interfunction value tracking in version 8 (2001), functions are normally parsed into a tree and the tree is walked (or traversed) with no specific set of starting values. This is called the general walk. During this walk, we compute values being passed to functions. So why not utilize these values as starting values for the called function? We came to refer to that approach as a specific walk. For various reasons, we didn't want to initiate a specific walk before finishing the general walk, which led to the need for multiple passes. As an example, consider

```
int f( int n ) { return 10/n; }
int g( int n )
{ if( n < -5 ) return 0;
return f(n) + g(n-1); }
```

If we were to call **g(3),** a sequence of further calls, **g(2),g(1),** and **g(0)** would ultimately result in a divide by 0. This would be detected in the fourth pass through the code.

Our most recent offering (version 9, 2008) added the ability to track static variables (that is, external variables plus nominal static variables). This requires our program to completely scan the entire source code and build a call graph adorned by variable access information before we can comment on which variables have what value. Happily, the multipass facility that we developed earlier allows us to do exactly that, with the user having control over how much of this static information to retain through the static depth option. Each of these innovations, at the time of their release, set a new standard for static analysis. (In these days of nondisclosure, we can only infer this claim because it isn't easy to confirm directly which capabilities lie in other systems.)

Value tracking has also let us supply the programmers with options that they can use to describe arguments and return values of their functions. As a result, we can report on where and how such functions might be abused.

Universality

We received several requests for a version of PC-lint that would work on other operating systems. To resolve the problem of supporting a multitude of different executables, we decided to make the source code available in obfuscated source form using one of our products called The C Shroud. This was almost guaranteed to work well because our customer was almost certain to have access to a C compiler. The new product was called FlexeLint. By using a "plain vanilla" implementation, the programmers could compile using "**cc *.c**" or whatever the equivalent operation was on their system. This meant we wouldn't be plagued with makefile peculiarities.

But compiler peculiarities were a different story. Compilers offered a wide and strange variety of builtin macros, special keywords, non-ASCII headers, and different character encodings, as well as unusual preprocessor directives, include search strategies, interpretations of C++ syntax, language extensions, project configurations (for the Microsoft compiler alone we have `.dsp`, `.vcp`, `.vcproj`, `.vcxproj`, and `.sln`), methods of incorporating assembly language, and so on.

Most of this is handled by way of options that are placed in a command-line extension file custom built-in for each compiler and, if necessary, different versions of the same compiler. We also supply tools such as our scavenger that will seek and find built-in macros and their translations. We have one that will determine the location of compiler-supplied headers. We hope that compilers in the future will provide much of this information directly to third-party vendors in some uniform fashion.

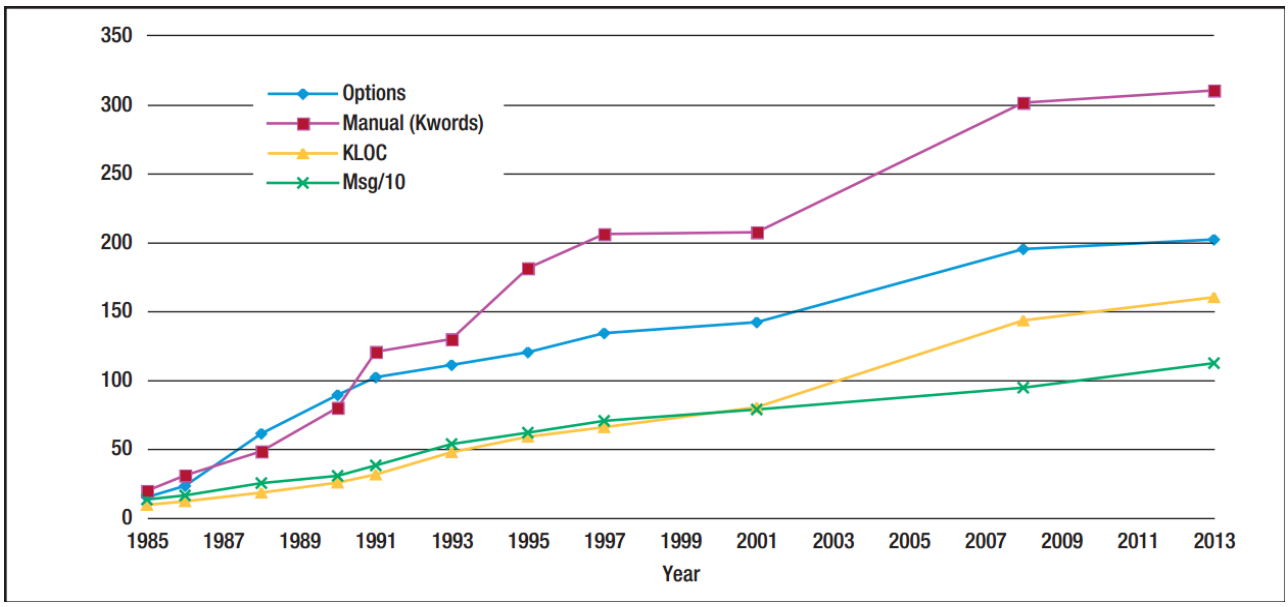

Figure 1. *Various growth properties of PC-lint during several release cycles. The project growth shows how LOC increases with manual size, number of options, and number of messages. Growth rates shown here approximate 11 percent each year and are a little less than the industry norm*.

As a result of this, licenses now number more than 53,000 (83,000 licensed users) for the various versions on many different architectures. Figure 1 shows the system's growth as it spanned several release cycles of both C and C++. The compound annual growth rate of around 1.11 is a little lower than the industry average and reflects the difficulty of the application area and the size of staff. In general terms of software volume, this is similar to other products from previous Impact columns.

Not everything implemented was the result of our own ideas. We've been able to benefit from a variety of authors such as Andrei Alexandrescu, Tom Cargill, James Coplien, Stephen Dewhurst, Les Hatton, Allen Holub, Andrew Koenig, Scott Myers, Robert Seacord, and Herb Sutter. More recently there have been a number of safe/secure programming standards such as MISRA, CERT, JSF, and JPL. In addition to these sources, we have our own vibrant user community, who, in addition to supplying us with suggested improvements (both large and small), have taken the brunt of any mistakes we've made and, in the process, fashioned the product to their own liking.

References

1. M. van Genuchten and L. Hatton, "Compound Annual Growth Rate for Software," IEEE Software, vol. 29, no. 4, 2012, pp. 19–21.
2. M.J. Adams, A.H.M. ter Hofstede, and M. La Rosa "Open Source Software for Workflow Management: The Case of YAWL," IEEE Software, vol. 28, no. 3, 2011, pp. 16–19.
3. D. Avery, "The Evolution of Flight Management Systems," IEEE Software, vol. 28, no. 1, 2011, pp. 11–13.
4. L. Hofland and J. van der Linden. "Software in MRI Scanners," IEEE Software, vol. 27, no. 4, 2010, pp. 87–89.

Dr. James F. Gimpel is the founder and president of Gimpel Software.