Integrated Design and Process Technology, **IDPT-2002**
Printed in the United States of America, June, 2002

STATIC SOURCE CODE CHECKING FOR USER-DEFINED PROPERTIES

Gerard J. Holzmann
Bell Laboratories, Lucent Technologies
600 Mountain Avenue
Murray Hill, NJ 07974

ABSTRACT

Only a small fraction of the output generated by typical static analysis tools tends to reveal serious software defects. There are two main causes for this phenomenon. The first is that the typical static analyzer casts its nets too broadly, reporting everything reportable, rather than what is likely to be a true bug. The second cause is that most static analyzers can check the code for only a fixed set of flaws. We describe a simple source code analyzer, UNO, that tries to remedy these problems. The default properties searched for by UNO are restricted to the most common types of error in C programs: use of **u**ninitialized variables, **n**il-pointer dereferencing, and **o**ut-of-bound array indexing. The checking capabilities of UNO can be extended by the user with the definition of application-dependent properties, which are written as ANSI-C functions.

INTRODUCTION

It would be attractive if we could develop a tool that could intercept *all* defects in a given piece of software with certainty, and with great efficiency to boot. Alas, it has long been known that such a tool cannot exist (Turing 1936). This does not mean that all attempts to build software checking tools are doomed to fail; it does mean that no such tool can promise to be all-encompassing. Not all real errors can always be caught, and not all errors caught can always be real. Increasing the number of real errors caught by a static analysis tool often increases also the number of false reports, and although the former increases the usefulness of the tool, the latter can seriously undermine it.

In the design of UNO we try to accomplish two goals.

- By focusing the tool on the types of software defects that occur most commonly in practice, we can increase the signal to noise ratio of the tool.
- By allowing the user to define precisely targeted, application-specific properties, we can extend the power of the tool in the area of primary interest to the user. This extension of the checking power of a static analyzer is similar to the one found in logic model checking tools, and can be based on similar algorithms (Clarke et al. 1999, Holzmann 1997, 2000).

RELATED WORK

The notion of analysis based on the symbolic execution of code is surprisingly old, e.g. Boyer et al. 1975, Clarke 1976, Osterweil 1976, but never appears to have taken hold in mainstream tools. An well-known example of a broadly distributed, yet still thinly used code analysis tool is *lint*, which dates from 1977 (Johnson 1978). More recent significant extensions of this tool include *lclint* (Evans et al. 1994), and the commercial tools *PClint* and *FlexeLint*, which attempt to cast their nets considerably more broadly than the original. Other well-known tools include *PREfix* (Bush et al. 2000), and its recent adaption *PREfast* (Pincus 2000).

Elaborate static checking capabilities have also been built into commercial tools such as *KLOCwork accelerator* and *PolySpace*, leading to some remarkable promises of defect coverage by the tool vendors. Some commercial code checkers gain power by targeting a restricted class of potential defects, e.g. *Visual Threads* (Savage et al. 1997), *Purify*, and*Insure++*.

The checking power of a tool can also be increased significantly if more information about the purpose of a piece of code is provided through source code annotations, e.g. Detlefs et al. 1998, and the Microsoft Vault tool.

Almost none of the existing tools can be extended with user-defined properties; the few exceptions do not allow for free access to dataflow information in defining additional checks, e.g. ParaSoft's tool *CodeWizard*, Lord 1997, and Cobleigh et al. 2001. An exception is Dawson Engler's *MC*, or Meta-level Compiler (Engler 2000). Engler's *MC* supports the definition of properties for static checking in a more powerful language called *Metal*. There are two main differences with UNO, the tool we

describe in this paper. First, unlike *MC*, UNO properties do not require a special language, but are written as functions in ANSI-C, supported by a small library of primitives that give direct access to dataflow information. Secondly, unlike *MC*, UNO properties need not be compiled and linked with the tool before they can be used: they are interpreted on-the-fly by UNO. As we shall see, the style of property specification in UNO differs substantially from *Metal*, though the length of comparable specifications is very similar in and *Metal*.

CATCHING SOFTWARE DEFECTS

Consider the following trivial, yet flawed, C program.

```
 1 int *ptr;
 2
 3 void
 4 main(void)
 5 {
 6         if (ptr)
 7                 *ptr = 0;
 8         if (!ptr)
 9                 *ptr = 1;
10 }
```

Figure 1 summarizes the output from *lint*, *LCLint*, and for this little program. First, the default run of *lint* produces a fairly cosmetic complaint. Many other things could be complained about with equal cause. The *main* procedure, for instance, should return a result of type *int*, not *void*, and the procedure should expect two parameters for possible command line arguments.

If we use the option -p to *lint* for more feedback, the tool generates 67 lines of output, including copious warnings about small disagreements within default C header files. The one real error in this program is not flagged: the nil-pointer dereference on line 9. (The statement on line 7 is unreachable.)

Lclint, *lint*'s modern incarnation, catches the error, but hides it in a range of messages and explanatory text, as illustrated in Figure 1. If we give UNO the program, the default output is to the point, revealing just the bug.

In a scan of a ten-line program, some verbosity is not much of an issue. For more realistic programs, though, generating hundreds or thousands of lines of righteous warnings can easily obscure the few serious defects that may be hidden in its midst. UNO attempts to generate less output by restricting to a careful check of serious types of software defects only.

UNO is built as an extension of the public-domain compiler front-end tool *ctree* (Flisakowski 1997). We used the same version of *ctree* earlier to build a model extractor for ANSI-C code, enabling the logic verification of multi-threaded systems code directly from its source, using the SPIN model checker as a background verification engine (Holzmann 1997 and 2000, Holzmann and Smith 1999).

The main extensions we built to turn *ctree* into UNO are:

- A dataflow analysis module, collecting basic def-use information for every node in the parse tree.
- A conversion routine that converts the parse tree for each C procedure into a control flow graph.
- Basic analysis routines that run the predefined checks for uninitialized variables, nil-pointer dereferencing, and out-of-bound array indexing on the source code, using the control-flow graphs for the local checks and the function-call graph for the global checks.
- A generic model checking routine that accepts a user-specified property and checks it against the control-flow graphs and/or the function-call graph for the source code being analyzed.
- The addition of a program to perform a global analysis of a program in a separate second pass, based on information gathered in the first pass.

The dataflow analysis module constructs a linked list of all data objects referenced at or below each node in the parsetree. It marks each data object with tags that record whether the object is declared, invoked as a function, evaluated as a variable, dereferenced, assigned a new value, or if it's address is taken. The most commonly used tags are: FCALL (to tag procedure names), DECL (to tag a variable declaration), DEREF (to tag a dereferencing operation), ALIAS (to tag symbols whose address is taken), DEF (to tag symbols whose value is changed), and USE (to tag symbols whose value is used). The dataflow module also collects information about known array bounds and about variables used as array indices. This information is recorded separately, for use in the array bounds analysis. The special tag ANY matches any of the data-flow tags, and the special tag NONE matches none.

The second extension converts the parse tree that is generated for each C procedure by *ctree* into a control-flow graph, interpreting *goto* and *return* statements, and branch and iteration structures. All subsequent analyses are done either via depth-first searches in these control-flow graphs or in the global function call graph that is also constructed here.

The program sources for applications are typically divided over a large number of different source files that can be compiled separately and then linked to form the

final executable. UNO starts by analyzing each source file separately, performing a detailed local analysis on the functions defined in that file, and saving other information for a later global analysis in intermediate files. If the intermediate files are preserved, clearly the analysis of a program source file need not be repeated unless the source file, or any of the files it depends on, was changed since the matching intermediate file was written.

In the first phase of the analysis UNO can check the usage of local variables, and of statically declared global variables. In the second phase, the global analysis is performed based on only the information collected in the collection of intermediate files. The use of all non-static global variables is analyzed, e.g. to find possible dereferences of uninitialized global pointers.

LOCAL ANALYSIS

One of the objectives of the analysis is to find paths in the the control-flow graphs of functions from the point of declaration of a variable to the point of its first assignment (i.e., def) or evaluation (i.e., use). This can easily be interpreted as a classic model checking problem, where the property to be checked defines the required temporal relation between *def* and *use* operations, and the system is given by the control flow graph of a C-function. Both property and control flow graph can be defined formally as labeled transition systems.

Let $\{N, n_0, F, L, T\}$ be a labeled transition system (LTS), where N is a finite set of nodes, $n_0 \in N$ is the start node, $F \subset N$ is the set of final, or accepting, nodes, L is a set of labels on transitions, discussed in more detail below, and $T \subset L \times N \times N$ is the transition relation, assigning a label from set L to each valid transition between nodes from set N in the graph.

The control flow graph for a given C-function is readily formalized as a labeled transition system: the nodes in the graph form set N, set F includes the nodes that are reached immediately after a statement is executed that returns control to a caller of the procedure directly or indirectly (e.g., a return or exit statement), and the label set L contains the set of dataflow tag markings for each statement.

Figure 2 gives the structure of the labeled transition system to capture the *def before use* property that forbids the uninitialized use of a variable. Execution starts at the initial node n_0. When a declaration for the variable is seen without immediate initialization (assuming it is not an array declaration), the property LTS moves to n_1, where it waits to see either a transition with a DEF or with a USE tag, ignoring everything else. In the first case, the property moves to the non-final (and non-accepting) state n_2 from where no further moves are possible. In the second case, the property moves into its final (and accepting) state n_3, corresponding to the detection of a def before use error.

We can check if any execution path in the control flow graph of a given procedure violates the def before use property by computing the product of the two labeled transition systems in a standard way (Holzmann 2000). First note that we can express all paths through the control flow graph of a procedure as a set of strings on L. Call this set S. In a similar way we can also express all *accepting* paths through the reachability graph for the labeled transition system of the property (cf. Figure 2) as a set of strings on the same label set L. Call this set V. If the intersection of S and V is non-empty, the procedure contains at least one execution path that matches an accepting path of the def before use LTS, corresponding to a possible violation of the def before use requirement.

The check for compliance with the def before use requirement then comes down to the computation of the intersection of two languages, standard in model checking. It would be inefficient to perform this check separately for every variable that is declared. We can, however, easily combine the work for all variables in a single depth-first search of the control-flow graph for each procedure. For a given property, such a search has a complexity that increases only linearly with the size of the input (the complexity of the depth-first search as such).

Array-Bound Violations: A check for array bound violations can be done in much the same way as the check for the *def before use* property. The differences are only in the definition of the label set L, and the precise circumstances under which we can declare an accepting run in the LTS for the property.

As a simple example, consider the following little program.

```
1 void
2 main(void)
3 {
4         int a[10], b[5], c[6*8];
5         int i;
6
7         for (i = 1; i < 11; i++)
8                 a[i] = 0;
9 }
```

which triggers the following verdict:

```
$ uno array.c
uno: in fct main, array index can \
     exceed upper-bound (10>9), var 'i'
     statement  : array.c:8: a[i]=0
```

```
declaration: array.c:5: int i;
```

The range of possible values for a scalar variable can be deduced from assignment operations (e.g., $i=0;$) and comparisons in conditional branch instructions. For instance, from the conditional $if(i<5)$ we can conclude that along the true branch scalar variable i can only have values less than 5, and along the false branch it can only have values larger than or equal to 5. The information is combined when multiple conditionals are passed along a path, and fixed at each assignment. A precise indication of a value range is easily lost though. For instance, when the current value range for i is $(i\geq 0 \wedge i<5)$ and the next operation seen is $i++$ then the known range for i reduces to $(i\geq 0)$.

Value ranges are computed conservatively in UNO. The range could be extended by relying on specialized tools for resolving constraint systems, e.g. (Kelly et al. 1996). Limits clearly will remain also in that case, so it is uncertain if such an extension could indeed significantly improve UNO's performance in practice.

Eliminating Infeasible Paths: A path through the control flow graph consists of a simple sequence of transitions. Only two basic types of transitions are of interest to us in determining if a specific path is feasible or not: some transitions correspond to steps through a conditional branch point in the program source (e.g., $(i>5) \equiv \mathsf{true}$ or $(i>5) \equiv \mathsf{false}$); the remaining transitions correspond to unconditional steps (e.g., $i++$). If a path is infeasible it must contain at least one conditional that cannot hold in the given context. It may, for instance, contradict earlier conditionals that appear in the same path. The sequence $(i>5) \equiv \mathsf{true};\ (i<5) \equiv \mathsf{true};$ clearly is not consistent. The conditional can also conflict with earlier assignments. For instance, the sequence $i=6;\ (i>5) \equiv \mathsf{false};$ is not consistent. In many cases UNO can determine if a path is feasible or not. It can use this information to suppress error reports on execution paths that turn out to be infeasible, and it can also, more profitably, use this information to shortcut the search procedure that is used to compute the language intersection of property and system. A resolver tool, such as the *Omega calculator* from (Kelley et al. 1996), can be used to improve accuracy, but as before, not all infeasible paths can always be detected, and therefore a reasonable engineering compromise must be sought.

GLOBAL ANALYSIS

To check if a global variable can be evaluated or dereferenced before it is defined is harder than checking if the same is true for a local variable. Globals in C are by default initialized to zero, so in principle it is impossible to violate the rule that a value must be assigned before the variable is evaluated. An initial value of zero is still a problem, though, when the variable is a pointer. Dereferencing a nil-pointer is always a fatal error. The second phase of the analysis is therefore concentrated on the analysis of dereferencing operations on global pointer variables, all initialized to a nil value by default.

For global variables we need to be able to take into account the possible call chain through the function call graph, not just the information derived from possible execution paths within local control-flow graphs of functions. This can quickly become overly complex, especially for large code bases. UNO therefore uses an approximation method based on the information gathered from the intermediate files from the first pass of the analysis. The function call graph plays a central role in this analysis.

The analysis starts at the *main()* routine and recursively descends into all functions that can be called from that routine, via a depth-first search. Because the search touches all functions reachable from main, as a by product of the analysis, it can also readily identify all functions that are not called, which can capture a remarkable amount of discarded code in evolving programs.

To be able to do the global analysis in a meaningful way, the analyzer must have access to some minimal information from the first pass. It needs to know, for instance, the list of functions that can be called from a given function, and it needs to know on which execution paths pointer variables may be evaluated, dereferenced, or set. The first pass of UNO captures this information by generating, among other information, a highly condensed version of the control-flow graph for each function, that contains only this information. If no globals are set or used, the abstract graph contains only the points where other functions are called. To make the analyses more precise, also information about points in the abstract graph where global variables have known (zero or nonzero) value, are recorded. The check now reduces to the same problem as encountered in the local analyses: a standard model checking problem.

UNO uses a predefined property to capture global nil-pointer dereferencing problems, but also accepts user-defined global properties, again defined as ANSI-C functions in a style we discuss in more detail shortly.

UNO's abstract function graphs do not attempt to compute possible return values. Since the tool is focused on def-use analysis, only assignments are important, not the values being assigned. The tool *PREfix* (Bush et al. 2000) attempts to capture more information by generating functional *models* of each function, based on a restricted symbolic execution of the function source. The user of the tool can control the maximum number of execution

paths that will be generated for each function, and the maximum number of times loops are unrolled. Inevitably, the path conditions can quickly become overly complex, making analysis either very time consuming or undecidable. Although not explicitly stated in Bush et al. 2000, the potential for added accuracy of this type of analysis does not necessarily outweigh the overhead involved. There is great benefit in a fast tool that can do a reasonable, though still approximate, analysis. Much the same observations have lead to a successor tool to *PREfix*, called *PREfast*. The new tool is said to be less precise, but faster and therefore of more immediate use to programmers (Pincus 2000).

DEFINING PROPERTIES

One of the more interesting features of UNO is its ability to accept user-defined properties of application specific requirements. The properties are defined in ANSI-C, but they do not have to be compiled before they can be used. The user can specify the file that contains the definition of a UNO property in the format below on the command line, for instance as:

```
$ uno -prop sample.prop *.c
```

where the check defined in the file *sample.prop* is applied to all functions in all C source files covered by the expansion of **.c*. The extension *.prop* used here is a convention to more easily recognize and categorize UNO property files; it is not required by the tool. The name of the procedure that is defined in the file, though, must be equal to *uno_check()*.

The code in *sample.prop* is parsed, like any other C procedure. The control-flow graph that is prepared is now used to guide the search for errors. UNO interprets the code, calling upon a small library of primitives to access dataflow information where required.

There are two types of predefined primitives in the UNO property definitions language: actions and queries. We mention the most important below; a complete list can be found in Holzmann 2002. UNO *actions* include: error(msg) to print an erro message with the path through the control flow graph, or function call graph, that leads to the point where the action was invoked, mark(N) to mark selected symbols with the integer value N, and unmark() to remove marks from the selected symbols. UNO *queries* include:

```
select(char *name, tag require, tag forbid)
unselect(char *name, tag require, tag forbid)
refine(tag require, tag forbid)
match(int mark,tag require,tag forbid)
marked(int mark,tag require,tag forbid)
path_ends()
```

The parameters used here have the following meanings:

name	specifies a name for the symbol to be matched; the empty string matches any symbol-name.
require	defines one or more required def-use tags for the symbol. The match tag DEF|DECL, for instance, matches if the symbol has a DEF and/or a DECL tag.
forbid	defines one or more xor-ed def-use tags that may *not* be attached to the symbol.
mark	specifies a requirement on a previous marking of the symbol.

The query select() selects symbols from the current statement, based on the criteria specified, erasing any previous selection. The query unselect() excludes the matching symbols from an existing selection. The query match() reduces the existing selection to the matching symbols, which now include a requirement on a pre-existing integer mark. All three above queries return *true* if the resulting selection is non-empty, and *false* otherwise. The query path_ends() returns *true* when the current node in the control-flow graph corresponds to the end of an execution path, just before the function returns to its caller. And finally, the query marked() returns *true* if matching symbols exist that have the mark specified, but it does not change the selection.

Side-Effects in Assertions

A simple first example of a UNO property is a check for the occurrence of side-effects or function calls in the arguments to an *assert* statement. As noted in Engler et al. 2000, such side-effects can introduce bugs in the code when the assertions are disabled at a later stage of code development. The UNO property can be defined as:

```
void
uno_check(void) // side-effects in asserts
{
   if (select("assert", FCALL, NONE))
   if (select("", DEF|FCALL, NONE))
   if (unselect("assert", ANY, NONE))
      error("side effect or fct in assert");
}
```

The property is defined directly in ANSI-C, using three queries and one error-action. UNO runs the check over all paths in the control flow graphs of all functions in the program source, evaluating the queries at each node in a given control-flow graph.

The first call to *select* returns *true* only when a node

in the control-flow graph is reached that contains a call to a function named *assert*. The second call to *select*, executed only when the first call returned *true*, makes a new selection, this time of all symbols that have a DEF or an FCALL tag from the dataflow analysis. This set, because of the match for function calls through the FCALL tag, will of course also match the symbol for the *assert* function call. The call to *unselect*, executed only if the first two calls returned *true*, will remove that symbol from the selection. If any symbol now remains in the selection, either a side-effect or a function call in the argument list of the *assert* call was detected, and an error can be reported.

Def after Def Scenarios

A slightly more interesting check is to look for *def after def* scenarios in the source code. In this case two value assignments can follow each other immediately, with no intervening use of the value. The first assignment can in that case often be avoided. Here is how the property can be written for UNO:

```
void
uno_check(void) // def-after-def errors
{
   if (select("", USE, NONE))
   if (match(1, ANY, NONE))
      unmark();

   if (select("", DEF, NONE))
   {  if (match(1, ANY, NONE))
         error("def after def");
      else
         mark(1);
   }
}
```

The call to *select* catches the symbols for all variables that have a USE tag, meaning that they are evaluated in the current statement. The call to *match* finds the names of symbols that were assigned a mark of one before, i.e., in a DEF context (see below). If there are any such matches, the markings from those symbols are removed, since this means that a USE properly followed the earlier DEF. The next part of the property arranges for new markings to be assigned when a DEF is encountered. The call to *select* finds the symbols with a DEF tag. The call of *match* tries to identify symbols with these names that were already marked. If there are any matches here, an error can be reported. If not, a new marking is assigned.

If we apply this check to the following little program

```
 1 int
 2 main(void)
 3 {   int x;
 4
 5     x = 1;
 6     if (x == 1)
 7     {       x = 2;
 8             x = 20;
 9     } else
10             x = 3;
11     x = 4;
12 }
```

UNO faithfully reports the three possible error paths. The first is:

```
$ uno -prop defdef.prop example.c
uno: 1: main() 'def after def error'
        1:      example.c:2: <main()>;
        2:      example.c:3: <int x; >;
        3:      example.c:5: <x=1>;
C       4:      example.c:6: <(x==1)>
        5:      example.c:7: <x=2>;
        6:      example.c:8: <x=20>;
```

The line marked with a C in the left margin is a conditional in the path. UNO will try to weed out infeasible paths, but it will not always be able to do so completely. The information provided in the error traces is always sufficient for the user to quickly make a final determination of the feasibility of each error though.

Locking Discipline

Locking violations are not difficult to catch in a compiler, if it were not for the unfortunate fact that most applications use their own specific library of lock and unlock calls. There can also be multiple independet locks, so in the analysis we should be able to track them all independently. Let us assume the a lock library is used where the lock routine returns a value that must be used in a subsequent call to the unlock routine. A UNO property for this type of check can be defined as follows.

```
void
uno_check(void) // handle multiple locks
{
   if (select("lock", FCALL, NONE))
   if (select("", DEF, NONE))
   {  if (match(1, DEF, NONE))
         error("lock after lock");
      else
         mark(1);
   }
```

```
    if (select("unlock", FCALL, NONE))
    if (select("", USE, NONE))
    {  if (match(1, DEF, NONE))
          unmark();
       else
          error("unlock without prior lock");
    }

    if (path_ends())
    if (marked(1, ANY, NONE))
       error("lock without unlock");
}
```

Note that the same structure of the property can be used to check other types of parameterized operations that appear in pairs, such as combinations of *fopen* and *fclose*, and of *malloc* and *free*, provided that the requirement is that the pairs must always appear together along every path within each function.

AN APPLICATION

In Holzmann 2002, data for the application of UNO to a small number of public domain software packages, such as *Sendmail*, *Unravel*, and *Linux* is presented. As one example, we summarize the results of the application of UNO's builtin tests to the *Unravel* sources here.

Unravel is a public domain program slicing tool. We applied UNO to the most recent source distribution, dated July 26, 1996. The package contains about 21 KLOC in source files. There are 36 separate C source files in the distribution. The local UNO check on these sources takes 12.1 seconds of system and user time (on the 250 MHz SGI MIPS machine). The global check takes 4.0 seconds.

UNO reports 15 errors in the local analysis. All 15 reports are warnings about the use of a ‘‘possibly uninitialized variable,’’ in various contexts. 12 reports reveal true errors. The remaining 3 reports involve execution paths that UNO could not determine to be infeasible. As one example, this code in *MultiSlice.c*:

```
572        int line,ix,at,h;
573
574        ix = w->slicetext.slicesrc. \
          line[line_from].n_highlight;
575        if (ix+5 > MAXHL) return;
576        ix = w->slicetext.slicesrc. \
          line[line_to].n_highlight;
577        if (ix+5 > MAXHL) return;
578        if(DEBUG) printf ("Setting \
          line %d0,line);
```

triggers the valid report:

```
uno: in fct SliceSet, possibly \
      uninitialized variable 'line'
      statement  : MultiSlice.c:578: \
         printf()
      declaration: MultiSlice.c:572: \
         int line,ix,at,h;
```

Another, perhaps more interesting, example is the following code from the file *slice_driver.c*:

```
113  int stmt_proc;
114
115  clear_active();
116  for (i = 1; i <= n_procs; i++){
117    if (procs[i].file_id == file)
118       if ((stmt >= procs[i].entry) &&
119           (stmt <= procs[i].exit)){
120              stmt_proc = i;
121              break;
122       }
123   }
124   if ((stmt_proc < 1) || \
124      (stmt_proc > n_procs)){
```

which triggers the valid report:

```
uno: in fct do_slice, possibly \
       uninitialized variable 'stmt_proc'
     statement  : auto-slice.c:170: \
       ((stmt_proc<1)||(stmt_proc>n_procs))
     declaration: auto-slice.c:154: \
       int stmt_proc;
```

CONCLUSION

We have described the basic design of a relatively simple static analysis tool called UNO. The tool is meant to intercept the most common types of errors in ANSI-C programs: uninitialized variables, nil-pointer dereferencing, and out-of-bound array indexing. The most interesting part of the tool, however, is it's extendibility with user-defined properties. The properties are written directly in ANSI-C.

The support for user-defined properties can significantly extend the power of a code analyzer. In UNO we have chosen to shift the emphasis in the tool towards the definition of such properties. There is also a drawback in this approach, though. The ideal checking method places no demands on the user: the code need not be annotated and no other guidance is needed to gain benefit from the check. Writing application-specific properties, just like adding annotations into the code, is a way to provide the extra guidance and it takes time and some skill to do it well. UNO properties can be written in an implementation independent way and can be re-used frequently. Small libraries may be developed of typical properties that can act as templates, requiring only minor adaption for a given application.

REFERENCES

W. Bush, J.D. Pincus, and D.J.Sielaff, A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, Vol. 30, No. 7, pp. 775-802.

E.M. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, Boston, 1999.

R.S. Boyer, B. Elspas, and K.N. Levitt, Select— a formal system for testing and debugging programs by symbolic execution. *Proc. Int. Conf. Reliable Software*, April 1975, pp. 234-244.

L. Clarke, *Test data generation and symbolic execution of programs as an aid to program validation.* PhD Thesis, Iniv. of Colorado, 1976.

J.C. Cobleigh, L.A. Clarke, L.J. Osterweil, *Flavers: a finite state verification technique for software systems*. Technical Report UM-CS-2001-017, CS Dept., Univ. of Mass., Amherst, MA 01003, April 2001.

D.L. Detlefs, K.R.M. Leino, G. Nelson, and J.B. Saxe, *Extended static checking.* Technical Report SRC-159, COMPAQ SRC, Dec. 1998.

D. Engler, B. Chelf, A. Chou, and S. Hallem, Checking system rules using system-specific, programmer-written compiler extensions. *Proc. 4th Symp. on Operating Systems Design and Implementation (OSDI), Usenix Organization, San Diego, CA., Oct. 22-25, 2000.*

D. Evans, J. Guttag, J. Horning, and Y.M. Tan. Lclint: A tool for using specifications to check code. *Proc. ACM SIGSOFT Symposium on Foundations of Software Engineering*, December 1994.

S. Flisakowski. CTree distribution, July 1997. `http://www.kagi.com/flisakow/`.

G.J. Holzmann, The model checker Spin, *IEEE Trans. on Softw. Eng.*, Vol 23, No. 5, May 1997, pp. 279-295.

G.J. Holzmann, and M.H. Smith, A practical method for the verification of event-driven software, *Proc. Intern. Conf. on Software Eng. (ICSE99)*, May 1999, Los Angeles, CA, pp. 597-607, (to appear in: *IEEE Trans. on Softw. Eng.*, 2002.)

G.J. Holzmann, Software Model Checking, *Lecture Notes, NATO Summer School*, Marktoberdorf, Germany, August 2000, IOS Press, Computer and System Sciences, Vol. 180, pp. 309-355.

G.J. Holzmann, UNO: Static Source Code Checking for User-Defined Properties, Bell Labs Technical Report, 27 pgs, January 2002. (Full version of the paper. Available from the author.)

S.C. Johnson, Lint, a C program checker, *Unix Programmer's Manual*, 7th Edition, Vol. 2A, Jan. 1979.

W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott, *The Omega calculator and library*, Version 1.1.0. Technical Report November 18, 1996, University of Maryland.

T. Lord, *Application specific static code checking for C programs: Ctool.* Online description, 1997. `ftp://krusty.e-technik.uni-dortmund.de/pub/people/mvo/twaddle.tar.gz`.

L.J. Osterweil, L.D. Fosdick, DAVE-A Validation Error Detection and Documentation System for Fortran Programs, *Software - Practice and Experience* Vol. 6, No. 4, pp. 473-486, 1976.

J. Pincus, Analysis is necessary, but far from sufficient. Invited presentation, *International Symposium on Software Testing and Analysis* (ISSTA), ACM SigSoft, August 2000, Portland, Oregon.

S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T.E. Anderson. Eraser: A dynamic data race detector for multithreaded programming. *ACM Trans. on Computer Systems*, Vol. 15, No. 4, pp. 391-411, 1997.

A.M. Turing, On computable numbers, with an application to the Entscheidungs problem. *Proc. London Mathematical Soc.*, Ser. 2-42, pp. 230-265 (see p. 247), 1936.

D. Wagner, J.S. Foster, E.A. Brewer, A. Aiken, A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities, *Proc. Network and Distributed Systems Security (NDSS 2000)*, San Diego, CA., USA, Feb. 2000.

URL's for Tools Mentioned:

`http://KLOCwork.com`
`http://www.parasoft.com`
`http://www.polyspace.com`
`http://research.microsoft.com/vault`
`http://www.cs.umd.edu/projects/omega`
`http://www.gimpel.com`
`http://hissa.ncsl.nist.gov/unravel.html`

```
$ lint expr.c
declared global, could be static
    ptr                expr.c(1)

$ lint -p expr.c | wc
     67     571    5390

$ lclint expr.c
LCLint 2.5q --- 26 July 2000

expr.c:4:1: Function main declared to return void, should return int
  The function main does not match the expected type.
  (-maintype will suppress message)
expr.c: (in function main)
expr.c:9:4: Dereference of null pointer ptr: *ptr
  A possibly null pointer is dereferenced.  Value is either the result of a
  function which may return null (in which case, code should check it is not
  null), or a global, parameter or structure field declared with the null
  qualifier. (-nullderef will suppress message)
expr.c:1:6: Variable exported but not used outside expr: ptr
  A declaration is exported, but not used outside this module. Declaration
  can use static qualifier. (-exportlocal will suppress message)

Finished LCLint checking --- 3 code errors found

$ uno expr.c
uno: in fct main, possible global nil-ptr dereference 'ptr'
     statement  : expr.c:9: (*ptr)=1
     declaration: expr.c:1: int *ptr;
```

Fig. 1 Output from LCLint and UNO

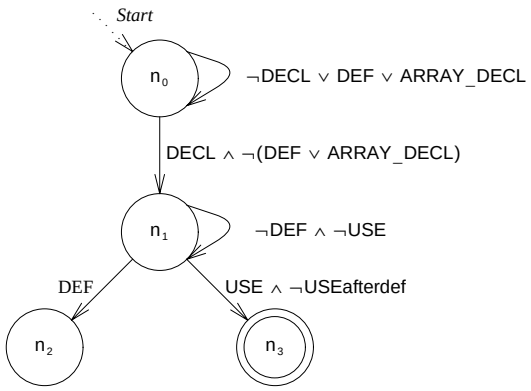

Fig. 2 Property Automaton for Def Before Use