UNO: Static Source Code Checking for User-Defined Properties[1]

Gerard J. Holzmann

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Only a small fraction of the output generated by typical static analysis tools tends to reveal serious software defects. Finding the important defects in a long list of warnings can be a frustrating experience. The output often reveals more about the limitations of the analyzer than about the code being analyzed. 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 predetermined, fixed, set of bugs: the user cannot make the analyzer more precise by defining and experimenting with a broader range of application-dependent properties.

We describe a source code analyzer called UNO that tries to remedy these problems. The default properties searched for by UNO are restricted to the three most common causes of serious error in C programs: use of uninitialized variables, nil-pointer dereferencing, and out-of-bound array indexing. The checking capabilities of UNO can be extended by the user with the definition of application-dependent properties, which can be written as simple ANSI-C functions.

Keywords: Model checking, static analysis, common software defects.

1. 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 [T36]. 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.

The design of UNO is based on the following two principles.

- We avoid trying to build a universal trap for all types of errors. By focusing the tool more sharply on the types of software defects that occur most commonly in practice, we can increase the signal to noise ratio of the tool in an area of primary interest. UNO derives its name from the three main types of software defects that it targets: use of **U**ninitialized variables, dereferencing **N**il-pointers, and **O**ut-of-bound array indexing.
- Rather than extending the set of predefined software defects searched for, we allow the user to define precisely targeted, application-specific properties. This extension of the checking power of a static analyzer is similar to the support for user-defined properties that is traditionally found in logic model checking tools, and can be based on similar algorithms [CGP99,H97,H00].

1. A short version of this paper appears in Proc. 6th World Conference on Integrated Design and Process Technology (IDPT2002), 26-30 June 2002, Pasadena, CA.

2. Related Work

A static analyzer for ANSI-C code needs access to the parse tree of a program, symbol table information, and dataflow information. This means that static analysis routines can profitably be merged into existing front-ends of compilers that for different purposes already collect most of this information. Indeed, many compilers have such options builtin. Understandably, the checks added in the mainstream compilers tend to be conservative, focusing on type types of coding errors that can be identified most efficiently. Many compilers, for instance, can report redundant variable declarations and simple cases of uninitialized variable use.

More detailed checks are usually built into stand-alone tools that focus entirely on code analysis instead of compilation. The notion of analysis based on static analysis or symbolic execution has a respectable history, e.g. [BEL75,C76,O76], 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 [J78]. More recent significant extensions of this tool include *lclint* [E94], and the commercial tools *PClint* and *FlexeLint* [GS], which attempt to cast their nets considerably more broadly than the original. Other well-known tools of this general type include Microsoft's *PREfix*, first introduced in 1996 [BPS00], and its more recent adaption *PREfast* [P00]. Some successful commercial code checkers gain power by targeting only a very restricted class of potential software defects. Examples include Compaq's *Visual Threads* tool, which uses the Eraser algorithm [S97] for identifying potential violations of a locking discipline. Similarly, Rational's *Purify* and ParaSoft's *Insure++*, tools focus primarily on memory usage patterns.

Elaborate static checking capabilities have also been built into commercial tools such as *KLOCwork accelerator* [KLOC] and *PolySpace* [POLY], leading to some remarkable promises of defect coverage by the tool vendors.

Table 1 — Summary of General Code Checking Tools

Tool Name	Requires Annotations	User-Defined Properties	Target Language	Comments
lint	-	-	C	*original version, limited checking*
lclint	-	-	C	*significantly extended version*
PREfix	-	-	C	*targets large code bases*
ESC	+	-	Modula3, Java	*theorem proving engine*
Vault	+	-	C	*targets device driver code*
Flavers	+	+	C, Ada, Java	*property-automata on annotations*
CodeWizard	-	+	C	*general code patterns only*
CTool	-	+	C	*functional specifications in Scheme*
MC	-	+	C	*properties require compilation*
UNO	-	+	C	*properties are interpreted*

Most tools mentioned so far work directly from program source text, requiring no special annotations from the programmer to assist in the checking process. Annotations are used in, for instance, *lclint*, but only to suppress warnings, not to enable more precise checks. The checking power of a tool can be increased significantly if more information about the purpose of a piece of code is provided by the programmer. Examples of tools in this class are Microsoft's *Vault* system for annotated C code [MV], and Compaq's Extended Static Checking tool, *ESC*, for annotated specifications written in Java and Modula3 [D98]. Adding the required annotations does, of course, take time and some sophistication on the part of the programmer. This means that this approach works best for smaller projects, not exceeding a few thousand lines of source code.

None of the tools mentioned so far are truly extendible: they can check only for predefined sets of builtin properties. The concept of user-defined properties in this domain is long known though. Howden [H87], for instance, described the use of finite state machine models for this purpose, and Olender and Osterweil [OO90] described the use of a constraint language based on regular expressions. Not many tools have embraced these ideas. There are some interesting exceptions, though, that we will discuss next. ParaSoft's tool *CodeWizard* [PC] allows the user to specify extra patterns (rules) that the tool can then locate in the

code. The patterns can enforce relatively simple rules of coding style. *Ctool* [L97] supports a broader range of extensions by the inclusion of a Scheme interpreter. The user can define new properties in Scheme, in a functional specification style. The amount of information available in the property definitions is very limited, though. It does not, for instance, allow access to dataflow information. The *Flavers* tool [CCO01] accepts properties specified as finite state automata, or equivalently as regular expressions, over a user-defined alphabet of *events*. Before the checks can be performed, however, the source code must still be annotated with event markers that are matched to the property automata.

Dawson Engler's *MC*, or Meta-level Compiler [E00], supports the definition of properties for static checking in a more powerful language called *Metal*. The main differences with UNO, the tool we will describe in this paper, are that unlike *MC*, UNO properties no not require a special language, but are written directly as functions in ANSI-C. Unlike *MC* also, UNO properties need not be compiled and linked with the tool before they can be used: they are interpreted directly by UNO. Unlike *Metal* also, UNO provides direct access to all relevant def-use information from a dataflow analysis engine.

3. Catching Software Defects

When Steve Johnson first described the design philosophy behind *lint*, he remarked prophetically [J78]: *''Messages of the form "xxx might be a bug" are easy to generate, but are acceptable only in proportion to the fraction of real bugs they uncover.''*

As an illustration of how true this observation is, consider the following trivial, but seriously 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 }
```

Without options, *lint* reports helpfully:

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

True, one of the cosmetic improvements we could make would be to declare the global *ptr* as a static. Since this is the only file we use in our little program, this comment is not of particular core value though. Stylistically, many other things could equally justly be complained about in this little program. The *main* procedure, for instance, should return a result of type *int*, not *void*. The procedure should also expect two parameters for possible command line arguments.

If we use the option -p to *lint* to include a check for portability (one of two options that date back to the original version of this tool), the tool generates 67 lines of output:

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

The output contains copious warnings about minor disagreements within default C header files that were parsed. But, the one real error in this program is not flagged: the nil-pointer dereference on line 9. (The statement on line 7 is unreachable.)

If we give UNO the program, the default output is terse and to the point:

```
$ 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;
```

Lclint, *lint*'s modern incarnation, catches the same error, but hides it in a range of messages and explanatory text:

```
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
```

In this scan of a ten-line program, some verbosity is of course not much of an issue. For more realistic programs, though, generating hundreds or thousands of lines of righteous warnings can easily obscure the relatively small number of serious defects that are hidden in its midst. As clearly indicated in *lclint*'s output, each warning generated may be suppressed, if the user chooses to do so.

UNO attempts to generate significantly less output by restricting to a more careful check of more serious types of software defects only.

UNO — Like other static analysis tools, UNO needs access to the parse tree and symbol table information for the source code that it tries to analyze. Rather than building a parser from scratch, we can use an existing public domain compiler front-end tool. For the first implementation of UNO we used the compiler front-end tool *ctree*, which was developed by Shaun Flisakowski [F97]. The primary appeal of *ctree* is that it is a relatively small and stable parser for ANSI-C. Only minor extensions were needed to adapt the tool to the construction of the static analysis tool. Newer, and substantially larger, versions of *ctree*, supporting C++ as an input language, have since replaced the version of *ctree* (version 0.14) integrated into UNO. 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 verification engine [H97,HS99,H00].

The main extensions we added 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. An overview of the tags that can be assigned is shown in Table 2. 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.

Table 2 — Dataflow Tags

Tag	Meaning	Example
ALIAS	address taken	&symbol
ARRAY_DECL	appears as basename in an array declaration	struct X symbol[8]
DECL	symbol appears as scalar in a declaration	int symbol
DEF	symbol assigned a new value	symbol = x
DEREF	dereferenced	*symbol
FCALL	used as name of function	symbol(x)
PARAM	formal parameter of a function	function(int symbol) { ... }
REF0	dereferenced to access structure	symbol->x
REF1	used as structure name	symbol.x
REF2	used as structure element	x->symbol, x.symbol
USE	evaluated to derive value	x = symbol
USEafterdef	both DEF and USE, but USE occurs after DEF	if ((symbol = x) > 0)
USEbeforedef	both DEF and USE, but USE occurs before DEF	symbol++

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.

3.1. 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, shown in Table 2.

Figure 1 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.

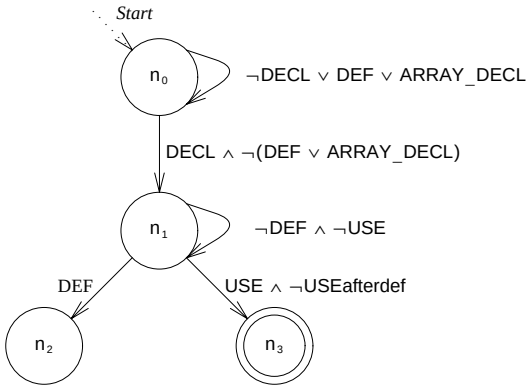

Figure 1 — Property Automaton for Def Before Use Property

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 [H00]. 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 1) 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).

The initial implementation of UNO does not yet include an algorithm for pointer alias analysis. In the built-in checks, therefore, when an ALIAS tag is seen it is treated as if it were a DEF, which means that we stop the check for the variable of which the address is taken. (Note that such variables can be assigned to after the point where the address was taken via pointers, which would go undetected.)

3.2. Array Indexing Errors

A check for possible array bound violations can be done in much the same way as the check for compliance with 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. To perform the check, we want to extend the state information that is directly available in the LTS for the control flow graph a little with information about known variable values, or at least known ranges of possible values for each variable that may appear as an index of an array. The general problem is undecidable, so we cannot hope to solve this problem in its full generality. UNO does not try to check if the value of a *computed* index is within bounds, e.g. if the index is given as a general expression on variables or involves function calls. Even the bound for array indices is not always easy to derive from the program text. But, when the bound

can be determined, and the index is simple (e.g., a single constant or scalar) a reasonable check can be done.

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. [K96]. 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.

3.3. 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 dedicated resolver tool, such as the *Omega calculator* from [K96], could 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.

4. 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. The possibility to dereference a nil-pointer is one of the most commonly occurring defects in C programs. 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.

```
 1 void
 2 uno_global(void)
 3 {
 4       uno_dfs(find_fct("main"));
 5 }
 6
 7 static void
 8 uno_dfs(Fct *f)
 9 {     CallStack *cs;
10       Sym *r;
11
12       if (checked_before(f)) return;
13
14       cs = (CallStack *) emalloc(sizeof(CallStack));
15       cs->f = f;
16       cs->nxt = callstack;
17       callstack = cs;
18
19       check_fct(f);     /* check def-before-use property */
20
21       for (r = f->calls; r; r = r->nxt)
22       {        callstack->r = r;
23                uno_dfs(find_fct(r->s));
24       }
25
26       callstack = callstack->nxt;
27 }
```

Figure 2 — Check for Global Nil-Pointer Dereferences

The main flow of the global analysis procedure is illustrated in Figure 2. The analysis of the use of global pointers starts at the *main()* routine and then 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* [BPS00] 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 [BPS00], 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 [P00].

5. Defining Properties

One of the more interesting features of UNO is its ability to accept user-defined properties of application specific requirements. UNO can check the source code for compliance with these 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 is one predefined integer variable, called *uno_state*, that can be set and tested in UNO properties, e.g. to maintain basic state information. It is only rarely needed, though, since the properties can often be expressed more precisely by maintaining a state tag (or *mark*) in individual symbols, e.g., for variables of interest, as we shall illustrate below.

The most important primitives supported in UNO properties are defined in Table 3. A more complete list is given in the Appendix. The query primitives, select, unselect, match, and marked have three parameters. The query refine has two parameters:

```
select(char *name, tag match, tag donotmatchk)
unselect(char *name, tag match, tag donotmatch)
refine(tag match, tag donotmatch)
match(int mark, tag match, tag donotmatch)
marked(int mark, tag match, tag donotmatch)
```

The parameters have the following meanings:

name	in a select or unselect call specifies a name for the symbol to be matched. If the empty string is given, any symbol-name will match.
match	defines one or more xor-ed def-use tags from Table 2 that must be attached to the symbol in the def-use list for the current statement. For instance, the match tag DEF|DECL matches if the symbol has a DEF or a DECL tag, or both. The special tag ANY matches any tag.
donotmatch	defines one or more xor-ed def-use tags that may *not* be attached to the symbol. The special tag NONE matches no tags, and can be used to bypass this restriction.
mark	specifies a requirement on a previous non-zero marking that was assigned to the symbol, through the use of the *mark* primitive discussed below.

Note that because of the uncertainty of symbolic evaluation, it is quite possible that calls to known_zero()and known_nonzero() both return *false*. In that case the zero- or non-zeroness of the symbol cannot be determined.

To illustrate the use of these primitives, we discuss some examples of user-defined UNO properties next.

Table 3 — UNO Primitives for Use in Property Definitions

Primitive	Meaning
error(msg)	Triggers an error report from UNO when executed, including an execution trace that leads from the point of entry into the function being analyzed to the point where the error is declared.
no_error(msg)	Prints an optional message, given as an argument, but otherwise has no effect. Used for informative purposes, or to satisfy syntax requirements.
list()	Triggers the printing of a list of all symbols that appear on the depth-first search stack at the point of call, and all symbols that appear in the def-use list for the current statement. It also shows all symbols that were selected through use of the primitives select, refine, and unselect.
path_ends()	A boolean function that 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.
select(...)	Selects symbols from the current statement, based on their name or on def-use tags from the dataflow analysis. The call erases earlier results of selections. This is a boolean function that returns true if the resulting selection is non-empty, and otherwise returns false.
unselect(...)	Like refine, but this time the symbols that match this call are excluded from the current selection.
refine(...)	Like select, but this time the selection refines the result of earlier selections, picking a subset of the symbols that match the additional criteria specified here.
match(...)	Reduces the current selection to those symbols that match the additional criteria and that have a specified (non-zero) mark, assigned through an earlier use of the mark primitive.
marked(...)	Returns true if symbols exist that match the criteria specified, and that have the specified (non-zero) mark.
mark(N)	Assigns an integer marking N to all currently selected symbols. Since the default marking is zero, the marking should be non-zero to be detectable later. The marking in effect assigns an individual state to a symbol of interest, so that the evolution of its state can be tracked and checked in the property.
unmark()	Removes the markings (defaulting them back to zero) for all currently selected symbols.
known_nonzero()	A boolean function that returns *true* if it can be determined by static analysis that the currently selected symbol(s) have a non-zero value.
known_zero()	A boolean function that returns *true* if it can be determined by static analysis that the currently selected symbol(s) have a zero value.

5.1. Side-Effects in Assertions

Here is how a UNO property can be used to check for the occurrence of any side-effects or function calls inside the arguments to an *assert* function call. As noted in [E00] such side-effects can introduce bugs in the code when the assertions are disabled at a later stage of code development.

```
void
uno_check(void)                        // find side-effects in assertions
{
   if (select("assert", FCALL, NONE))  // find statements of interest
   if (select("", DEF|FCALL, NONE))    // with symbols tagged DEF or FCALL
   if (unselect("assert", ANY, NONE))  // exclude "assert" itself
      error("side effect or fct call in assert");
}
```

The property is defined here directly in ANSI-C, in a few lines, using four of the predefined primitives we

discussed. UNO runs the check over all paths in the control flow graphs of all functions in the program source, evaluating this error-checking function at every 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. We could make the error report more precise, by individually matching for DEF matches and FCALL matches.

Note that each subsequent call to select erases the current selection, if any, and defines a new selection from scratch.

5.2. Def after Def Errors

A slightly more sophisticated check is to look for *def after def* errors 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))      // find uses of variables
   if (match(1, ANY, NONE))        // if any of these symbols are marked
      unmark();                    // remove the marks

   if (select("", DEF, NONE))      // find defs of variables
   {  if (match(1, ANY, NONE))     // check if any of these are marked
         error("def after def");   // if so, we found an error
      else
         mark(1);                  // otherwise, mark the symbol
   }
}
```

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 three possible error paths:

```
$ uno -prop defdef.prop -allerr 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)> == <_true_>;
        5:      example.c:7: <x=2>;
        6:      example.c:8: <x=20>;

uno: 2: 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)> == <_true_>;
        5:      example.c:7: <x=2>;
        6:      example.c:8: <x=20>;
        7:      example.c:11: <x=4>;

uno: 3: main() 'def after def error'
        1:      example.c:2: <>;
        2:      example.c:3: <int x; >;
        3:      example.c:5: <x=1>;
C       4:      example.c:6: <(x==1)> == <_false_>;
        5:      example.c:10: <x=3>;
        6:      example.c:11: <x=4>;
```

Lines marked with a C in the left margin are the conditionals 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. We gave the option -allerr to get all three paths. By default, UNO will try to limit the number of error reports to one report per statement. So without the extra option the third error path would be suppressed (since it terminates at the same node as the second error path.)

5.3. Locking Disciplines and Interrupt Masking

Interrupt masking and locking violations are particularly easy to catch. Note, however, that the specific name of the required functions can vary from application to applications, and this type of property is therefore hard to encode in a predefined check. A rule we can check is that when a function calls a lock (interrupt disabling) function, it must also call the unlock (interrupt enabling) function within the same function before returning control to the caller.

In the UNO property below we use fct_call(name), which UNO accepts as a shorthand for

```
select(name, FCALL, NONE)
```

We use the predefined integer state variable uno_state here to track the locking state.

```
void
uno_check(void)                    // locking errors, single lock
{
   if (uno_state == 0)
   {  if (fct_call("qunlock"))
         error("unlock while unlocked");
      if (fct_call("qlock"))
         uno_state = 1;
   } else
   {  if (fct_call("qlock"))
         error("lock while locked");
      if (fct_call("qunlock"))
         uno_state = 0;
      else if (path_ends())
         error("lock without unlock");
   }
}
```

Clearly, the same basic type of check may be performed to check interrupt masking and reenabling rules, and any other set of operations that is required to appear in strict pairs. An example of a property for checking the use of interrupt masking in *Linux* device drivers, for instance, can be specified as follows. In *Linux* interrupts are masked with the call cli() and restored with sti(). The normal discipline is to first save the state of the current interrupt mask in a variable, with a call save_flags(flags), then to mask interrupts with cli(), and finally to restore the previous state with restore_flags(flags) (which in turn calls sti()). Since there are four possible calls, and three possible states for the system to be in, there is ample opportunity for mistake. The following UNO property performs a series of checks.

```
void
uno_check(void)                    // check interrupt masking
{
   switch (uno_state) {
   case 0:
          if (fct_call("save_flags"))
             uno_state = 1;
          else if (fct_call("cli"))
             uno_state = 3;
          else if (fct_call("sti"))
             error("sti without cli");
          else if (fct_call("restore_flags"))
             error("restore_flags without save_flags");
          break;

   case 1:
          if (fct_call("save_flags"))
             error("repeated save_flags");
          else if (fct_call("cli"))
             uno_state = 2;
          else if (fct_call("sti"))
             error("sti without cli");
          else if (fct_call("restore_flags"))
             error("restore_flags without cli");
          else if (path_ends())
             error("saved flags not used");
          break;
```

```
   case 2:
          if (fct_call("save_flags"))
             error("repeated save_flags");
          else if (fct_call("cli"))
             error("repeated cli");
          else if (fct_call("sti"))
             uno_state = 0;
          else if (fct_call("restore_flags"))
             uno_state = 0;
          else if (path_ends())
             error("return without reenabling interrupts");
          break;

   case 3:
          if (fct_call("save_flags"))
             error("save_flags atfer cli");
          else if (fct_call("cli"))
             error("repeated cli");
          else if (fct_call("sti"))
             uno_state = 0;
          else if (fct_call("restore_flags"))
             error("restore_flags without save_flags");
          else if (path_ends())
             error("return without reenabling interrupts");
   }
}
```

If we apply this check to the 57 device driver files defined in *Linux* version 2.3.9, a series of violations is quickly found, as first reported by Engler's in an application of the *MC* checker [E00]. Each check takes a few seconds of CPU time. In the UNO check, 10 the 57 files trigger a total of 31 error paths, hitting the following 6 of the 10 possible types of error from the property specified:

```
repeated cli                        3x
restore_flags without cli           5x
restore_flags without save_flags    4x
return without reenabling interrupts 6x
saved flags not used                4x
sti without cli                     9x
```

Arguably, some of the paths reported are benign, for instance a repeated cli(); cli(); is odd, but not likely to cause grief, and similarly the sequence save_flags(flags); restore_flags(flags);, without an intervening cli() to mask interrupts, would be redundant, but not wrong. Some of the warnings about the use of sti() without a preceding cli() also points to initialization code that initializes the interrupts for the first time, so not in error. At least half the error paths point to true bugs in the code though. This calling sequence, for instance, from the file *amiflop.c*, triggers a number of complaints. Most likely, the use of sti() on line 339 is a typo that should have read cli().

```
323 static void fd_deselect (int drive)
324 {
—
338      save_flags (flags);
339      sti();
—
347      restore_flags (flags);
—
350 }
```

One assumption in the examples above is that there is only one type of interrupt mask or lock. If this holds, the property is simple to specify, because we do not have to track the state of individual lock variables. In general, there will be multiple locks to track though, and the situation becomes more interesting. This time

the return value from the lock function, and the parameter given to the unlock function become relevant and must be tracked and matched. A UNO property for this more realistic type of check can be defined as follows.

```
void
uno_check(void)                         // handle multiple locks
{
   if (select("lock", FCALL, NONE))     // find all calls to lock
   if (select("", DEF, NONE))           // select symbols defined in call
   {  if (match(1, DEF, NONE))          // check if any were previously marked
         error("lock after lock");      // if so, complain
      else
         mark(1);                       // else mark the symbol
   }

   if (select("unlock", FCALL, NONE))   // find all calls to unlock
   if (select("", USE, NONE))           // select symbols used in call
   {  if (match(1, DEF, NONE))          // check if previously marked
         unmark();                      // if so, remove the mark
      else                              // else complain
         error("unlock without preceding lock");
   }

   if (path_ends())                     // at any return or exit statement
   if (marked(1, ANY, NONE))            // check if any locks in effect
      error("lock without unlock");     // if so, complain
}
```

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.

5.4. Nil-Pointer Dereferencing

As a last example, we show a more detailed definition of a UNO property that performs a similar type of check as predefined in the tool, targeting the interception of nil-pointer dereferencing operations, but this time restricted to locally declared pointer variables.

```
void
uno_check(void)                       // find nil-pointer dereferencing errors
{
   if (select("", DECL, ARRAY_DECL))  // find non-array declarations
   if (refine(DEREF, DEF))            // of uninitialized local pointers
      mark(1);                        // mark them

   if (select("", USE|DEF, NONE))     // look for all uses and defs
   if (refine(DEREF, ALIAS))          // that involve dereference but no alias
   if (match(1, ANY,  NONE))          // see if any of these symbols are marked
   {  if (known_zero())               // if these variables are known to be zero
         error("dereferencing uninitialized ptr"); // complain
      else
      {  if (known_nonzero())         // if they are known to be non-zero
            no_error();               // keep quiet
         else                         // otherwise give a warning
            error("dereferencing possibly uninitialized ptr");
      }  }
```

```
   if (select("", DEF, DEREF))         // find all defs without dereferencing
   if (match(1, ANY, NONE))            // see if any of these symbols are marked
      mark(2);                         // if so change the mark of the now init ptr

   if (select("free", FCALL, NONE))    // find all calls to free
   if (select("", USE, NONE))          // find the arguments of those calls
   if (match(1, ANY, NONE))            // if any of these variables were marked 1
      error("freeing an uninitialized ptr"); // complain

   if (select("free", FCALL, NONE))    // renew selection of all calls to free
   if (select("", USE, NONE))          // find the arguments of those calls
   if (match(2, ANY, NONE))            // if any were marked 2 (i.e., initialized)
      mark(1);                         // revert this ptr to its uninitialized state
}
```

6. Some Applications

We have applied UNO to a small set of public-domain software packages. We will discuss the application to *Sendmail*, and to *Unravel*. The execution time of the local analyses on these applications is comparable to the time required for straight compilation of the sources. The global analyses are in all cases substantially faster. Table 4 gives run times measured on a 250 MHz SGI MIPS R10000 computer. (For comparison, the times would shrink by a factor of about eight on a 2 GHz Pentium-4 PC.)

Table 4 — Basic Check, Error Counts and Run Times

Application	KLOC	Number of C Files	Valid/Invalid Reports	Run Time (sec.) Local	Global
Sendmail	75	38	3/2	21.9	24.6
Unravel	21	36	12/3	16.6	5.3

As shown in the application of the locking properties test to *Linux* device driver code discussed above (confirming observations made in [E00]), more targeted tests with user-defined properties can be more effective than a scan for predefined properties alone. The results of such tests are harder to classify clearly though. The scan of the 57 *Linux* device driver files reported in Section 5.3, for instance, triggered 31 error reports, with at least half corresponding to true violations of the locking discipline. Below we will look at a separate test for the compliance with locking properties in the core kernel source files from *Linux*.

6.1. Sendmail

We have applied UNO to *sendmail* version 8.11.6. This public domain software consists of close to 75 KLOC (thousand lines of code, measured by a straight wordcount of the source and header files included in the distribution.) Earlier versions (e.g. version 8.9.3, see [W00]) of this software have been the target of static analysis before, so presumably most of the original indigenous bugs were fixed before we ran the UNO tests. The source comes in 38 C files. The local UNO check on these sources takes 21.9 seconds of system and user time. The global check takes 24.6 seconds.

UNO reports 2 errors in the local analysis and 3 execution traces in the global analysis. The two errors from the local analysis are reported as:

```
uno: in fct collect, possibly uninitialized variable 'df'
     statement  : collect.c:316: putc(c, df);
     declaration: collect.c:73: register FILE *volatile df;

uno: in fct logdelivery, uninitialized variable 'now'
     statement  : deliver.c:3673: snprintf()
     declaration: deliver.c:3487: time_t now;
```

Both reports are valid. The first one in context looks like this (showing only the relevant structure and code fragments, as indicated by the gaps in the line numbers below):

```
 66 void
 67 collect(fp, smtpmode, hdrp, e)
—
 72 {
 73              register FILE *volatile df;
—
101      if (!headeronly)
102      {
—
120                      df = bfopen(dfname, FileMode, DataFileBufferSize, sff);
—
122                      if (df == NULL)
123                         {
—
132                         }
133                      dfd = fileno(df);

—
144      }
—
188      for (;;)
189      {
—
311                         switch (mstate)
312                         {
313                           case MS_BODY:
314                                  /* just put the character out */
315                                  if (!bitset(EF_TOOBIG, e->e_flags))
316                                          (void) putc(c, df);
317                                  /* FALLTHROUGH */
318
319                           case MS_DISCARD:
320                                  continue;
321                         }
—
468      }
—
720 }
```

The second bug is caused by a large separation between the point of declaration and the first use of the variable, separated by large blocks of conditionally compiled code containing some of the initializations.

The scenarios intercepted by the global analysis are reported as follows:

```
Trace #1: possible global use or deref before def: [R EventQueue clock.c 336]
        called from setsignal clock.c 436
        called from pend_signal main.c 2320
        called from setsignal main.c 1092
        called from main main.c 113

Trace #2: possible global use or deref before def: [R FileMailer savemail.c 407]
        called from savemail envelope.c 304
        called from dropenvelope main.c 2185
        called from finis main.c 361
        called from main main.c 113
```

```
Trace #3: possible global use or deref before def: [R InChannel err.c 555]
        called from putoutmsg err.c 587
        called from puterrmsg err.c 104
        called from syserr main.c 161
        called from main main.c 113
```

Only the second report is valid. There is just one place in the code where the global pointer *FileMailer* is assigned a value. This happens on line 1399 in main.c, but it is done in conditional code:

```
main.c:1394       st = stab("*file*", ST_MAILER, ST_FIND);
main.c:1395       if (st == NULL)
main.c:1396               syserr("No *file* mailer defined");
main.c:1397       else
main.c:1398       {
main.c:1399               FileMailer = st->s_mailer;
main.c:1400               clrbitn(M_MUSER, FileMailer->m_flags);
main.c:1401       }
```

The procedure *syserr()* looks like an error exit procedure, but it can in fact return to the caller, allowing execution to proceed. In most cases in the remainder of the code where *FileMailer* is dereferenced, a check is first done to secure the non-zeroness of the variables. This does not happen in savemail.c on line 407: the location that is pointed to by UNO:

```
savemail.c:407:                 if (bitnset(M_7BITS, FileMailer->m_flags))
```

The first and last reports, however, are invalid. The statement pointed for the last scenario, for instance, is to is the conditional:

```
err.c:555:      if (InChannel == NULL || feof(InChannel) || ...
```

which after preprocessing turns into:

```
err.c:555:      if (InChannel == ((void *)0) || (((int)(InChannel)->_flag) & 0020) || ...
```

exposing the dereference operation on *InChannel*. The non-zeroness of the pointer is secured here in a non-standard, though completely valid, manner. UNO does not yet interpret the context correctly in this case. The context for the first scenario is similar.

6.2. Unravel

Unravel is a public domain program slicing tool [UNR]. We applied UNO to the most recent source distribution at the time of writing this paper, which was 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 16.6 seconds of system and user time (on the 250 MHz SGI MIPS machine). The global check takes 5.3 seconds.

Optionally UNO generates about 53 valid warnings for functions and variables that are declared but not used anywhere in the source tree, or that are assigned to but never evaluated (i.e., *def without use*). These do not point to coding errors per se, though. More than half of the *def without use* reports are due to the use of static character pointer variables such as *sccs_id*, that are purposely set to a unique identifying string in each source file, but never used.

UNO reports 15 other errors in the local analysis, but none, other than less interesting optional warnings about redundant function declarations, and redundant global variable declarations and structure fields, in the global check. All 15 error reports are warnings the use of a ‘‘possibly uninitialized variable,’’ in various contexts. 3 of these reports are invalid, involving execution paths that the analyzer could not accurately determine to be infeasible. The other 12 reports are true errors in the source code.

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;
```

As is often the case, errors can hide in code that is only conditionally enabled, here for debugging purposes. 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) || (stmt_proc > n_procs)){
```

which triggers the error 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;
```

Without further knowledge, it indeed is uncertain if the variable *stmnt_proc* will always have been initialized at line 124.

6.3. Linux Kernel Sources

As a last application, we checked some standard locking properties for kernel source code from Linux version 2.3.9. The Linux source code has been the subject of several earlier analyses with static analysis tools, e.g. [E00]. For this test we looked at the 20 source files with roughly 10 KLOC in the kernel directory proper (i.e., we did not further consider locking or interrupt masking issues in the device driver code directory). We looked especially at the locking disciplines used in the kernel code.

There are several instances of creative use of ANSI-C, or *gnu* extensions thereof, that are not accepted by the UNO parser. These instances were handled with a small preprocessing filter that was applied before the checks were applied. Eight different pairs of locking primitives are used:

Locking Function	Unlocking Function
read_lock()	read_unlock()
spin_lock()	spin_unlock()
spin_lock_irq()	spin_unlock_irq()
spin_lock_irqsave()	spin_unlock_irqrestore()
lock_kernel()	unlock_kernel()
wq_write_lock()	wq_write_unlock()
wq_write_lock_irqsave()	wq_write_unlock_irqrestore()
write_lock_irq()	write_unlock_irq()

To avoid expansion of these primitives into assembler code, the preprocessing filter also inserted an

underscore in front of the names of each of these primitives, so that their use could be tracked in a UNO property definition. The property definition used is a small variant of the version shown earlier for handling multiple locks. In the case of Linux, the lock primitives do not return a value, so we match on the symbols used in the parameter lists of the calls, instead of on the symbols defined via return values. The locking primitives are defined as macros, which are filled in with the desired specific name for each of the eight checks we ran. (Each check takes about 8 seconds on an 800 MHz Intel machine running Linux.)

```
#define X_LOCK     "_wq_write_lock"
#define X_UNLOCK   "_wq_write_unlock"

void
uno_check(void)                          // handle multiple locks
{
   if (select(X_LOCK, FCALL, NONE))      // find all calls to lock
   if (select("", USE, NONE))            // select symbols defined in call
   {  if (match(1, USE, NONE))           // check if any were previously marked
         error("lock after lock");       // if so, complain
      else
         mark(1);                        // else mark the symbol
   }

   if (select(X_UNLOCK, FCALL, NONE))    // find all calls to unlock
   if (select("", USE, NONE))            // select symbols used in call
   {  if (match(1, USE, NONE))           //check if previously marked
         unmark();                       // if so, remove the mark
      else                               // else complain
         error("unlock without preceding lock");
   }

   if (path_ends())                      // at any return or exit statement
   if (marked(1, ANY, NONE))             // check if any locks in effect
      error("lock without unlock");      // if so, complain
}
```

UNO reports errors in the file *kernel/sched.c*, generating the following type of error traces (*uno_sched.c* is the name of the preprocessed *sched.c* source file that was used in the check):

```
uno: 1: sleep_on_timeout() 'unlock without preceding lock'
         1:       uno_sched.c:1090: <>;
         2:       uno_sched.c:1092: <unsigned long flags; >;
         3:       uno_sched.c:1092: <wait_queue_t wait; >;
         4:       uno_sched.c:1092: <init_waitqueue_entry(&(wait),get_current())>;
         5:       uno_sched.c:1094: <get_current()->state=2>;
         6:       uno_sched.c:1096: <_wq_write_lock_irqsave(&(q->lock),flags)>;
         7:       uno_sched.c:1096: <__add_wait_queue(q,&(wait))>;
         8:       uno_sched.c:1096: <_wq_write_unlock(&(q->lock))>;
```

The source fragment pointed to here reads as follows:

```
1090 long sleep_on_timeout(wait_queue_head_t *q, long timeout)
1091 {
1092     SLEEP_ON_VAR
1093
1094     current->state = TASK_UNINTERRUPTIBLE;
1095
1096     SLEEP_ON_HEAD
1097     timeout = schedule_timeout(timeout);
1098     SLEEP_ON_TAIL
1099
1100     return timeout;
1101 }
```

The locking primitives are hidden in macros, defined earlier in the file, as follows:

```
#define  SLEEP_ON_HEAD                                          \
         wq_write_lock_irqsave(&q->lock,flags);                 \
         __add_wait_queue(q, &wait);                            \
         wq_write_unlock(&q->lock);

#define  SLEEP_ON_TAIL                                          \
         wq_write_lock_irq(&q->lock);                           \
         __remove_wait_queue(q, &wait);                         \
         wq_write_unlock_irqrestore(&q->lock,flags);
```

The sample error report from UNO shown above is triggered by the fact that in the macro SLEEP_ON_HEAD the unlocking primitive *wq_write_unlock()* is called without an earlier call on the matching primitive wq_write_lock(). From the code we can assume that this is most likely intentional.

The error report can be suppressed by treating *wq_write_lock_irqsave()* and *wq_write_lock()* as well as *wq_write_unlock_irqrestore()* and *wq_write_unlock()*, as pairwise equivalent. UNO reports no other violations of the locking properties in this part of the code.

7. Conclusion

We have described the basic design of a relatively simple static analysis tool called UNO. The tool is meant to make it simple to intercept the most three 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 the source language of the checker (ANSI-C) and do not require compilation or linkage with the checker itself.

The use of a special purpose parser for ANSI-C, which in the case of UNO is based on the public domain *ctree* package, has advantages, but also some drawbacks. Among the advantages are the fact that the parser can be fine-tuned for the purpose at hand, without concern for disturbing more traditional uses, e.g. for type checking or code generation. Our parser is forgiving on parse errors, trying hard to produce a parse-tree that can be analyzed for user-defined properties. This feature is useful when analyzing code from foreign platforms (e.g. VxWorks, or Plan9) for which not all system header files may be readily available. If the code can be eased past the parser, it can be analyzed by UNO, also without many of the type definitions that would ordinarily be required for a correct compilation of the code.

There are also drawbacks. Few programmers restrict their code to comply with pure ANSI/POSIX standards. It is tempting to exploit non-standard extensions or even esoteric or accidental features that may exist in a local compiler. In the applications that we have looked at, the code for virtually every different platform turned out to rely on some non-standard features in the local compilers. This problem, of course, will not go away, no matter which compiler front-end one chooses, but it can be addressed to some extent by generalizing the UNO parser.

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 however a way to provide the extra guidance and it takes time and some skill to do it well. The properties, though, are written in a largely implementation independent way and can be re-used frequently. Small libraries may be developed of typical properties, such as locking, that can act as templates, requiring only minor adaption for any given application.

There are some corners of C that make it all too easy to defeat even the best code analyzer. In the *Linux* kernel code we looked at, for instance, large fragments of code are defined in assembler. Our front-end skips over these fragments, which means that possible initializations and variable uses can be missed, causing false error reports. In other places we had to deviate from our rule not to modify the source text that is the subject of the analysis in any way, by adding a few preprocessor constructs to the *Linux* header files, to replace unsupported constructs with simpler ones, e.g.:

```
#ifndef UNO
  #define SPIN_LOCK_UNLOCKED (spinlock_t) { 0 }
#else
  #define SPIN_LOCK_UNLOCKED (spinlock_t) 0
#endif
```

To enable the UNO checks we had to make ten such modifications total, nine in five header files and one in a kernel source file. More troublesome than this is that certain standard constructs from C itself are currently outside the scope of our checks. The functions setjmp() and longjmp() are in this class. A call on longjmp() can transfer control across procedures, back to an earlier point in the execution where setjmp() was last invoked.

The basic checks performed by UNO are most useful if applied as a standard sanity check of all new code written, as it is written. The runtime requirements for these checks is no impediment to frequent use. Checks with user-defined properties, though, can be more revealing than predefined checks. The user-defined properties in UNO apply both to local checks within the control-graph of each function and to global checks that are applied to the function call graph of the program as a whole. Experience will show if the properties are simple enough to define to prove the tool useful for routine checking of code.

8. References

[BPS00] 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.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, Boston, 1999.

[BEL75] 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.

[C76] L. Clarke, *Test data generation and symbolic execution of programs as an aid to program validation.* PhD Thesis, Iniv. of Colorado, 1976.

[CCO01] 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.

[D98] D.L. Detlefs, K.R.M. Leino, G. Nelson, and J.B. Saxe, *Extended static checking.* Technical Report SRC-159, COMPAQ SRC, Dec. 1998.

[E00] 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.*

[E94] 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.

[F97] S. Flisakowski. CTree distribution, July 1997. `http://www.kagi.com/flisakow/`.

[H97] G.J. Holzmann, The model checker Spin, *IEEE Trans. on Software Engineering*, Vol 23, No. 5, May 1997, pp. 279-295.

[HS99] 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 Software Engineering, 2002.)

[H00] 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.

[H87] W.E. Howden, *Functional Program Testing and Analysis*, McGraw Hill, 1987.

[J78] S.C. Johnson, Lint, a C program checker, *Unix Programmer's Manual*, Seventh Edition, Volume 2A, January 1979.

[K96] 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. `http://www.cs.umd.edu/projects/omega/release-1.2.html`.

[L97] 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`.

[O76] 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.

[OO90] K.M. Olender, L.J. Osterweil, Cecil: A sequencing constraint language for automatic static analysis generation, *IEEE Trans. on Softw. Eng.*, Vol. 16, No. 3, March 1990, pp. 268-280.

[P00] 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.

[S97] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T.E. Anderson. Eraser: A dynamic data race detector for multithreaded programming. *ACM Transactions on Computer Systems*, Vol. 15, No. 4, pp. 391-411, 1997.

[T36] 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.

[W00] 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. `http://www.isoc.org/ndss2000/proceedings/`.

URL's Referenced:

[KLOC] `http://KLOCwork.com`

[PC] `http://www.parasoft.com/`.

[POLY] `http://www.polyspace.com`.

[MV] `http://research.microsoft.com/vault/`.

[GS] `http://www.gimpel.com`.

[UNR] `http://hissa.ncsl.nist.gov/unravel.html`.

Appendix: Definition of Property Primitives

The currently implemented set of primitives include five types of basic statements, called *actions* below, and eight types of boolean functions, called *queries*. Table 5 summarizes these primitives, with illustrative names for the arguments. The working of each action and query, and the types of arguments expected, is discussed in more detail in the remainder of this Appendix.

Table 4 — Actions and Queries for Specifying Properties

Actions	Queries
error("typeoferror")	select("name",require,forbid)
no_error("informative")	unselect("name",require,forbid)
no_error()	fct_call("name")
list()	marked(value,require,forbid)
mark(value)	match(value,require,forbid)
unmark()	refine(require,forbid)
	known_zero()
	known_nonzero()

Properties can be specified either *locally*, to be applied individually to each procedure in the program being analyzed, or *globally*, to be applied to the abstracted representation of the global function call graph that can be generated by UNO. Local properties can be used to check properties on local variable use, and/or local usages of procedure names. Global properties can be used to check some properties of global pointer variables without explicit global initialization. For local properties, all data flow tags from Table 2 can be used. For global properties only the tags DECL, DEF, USE, DEREF, can be queried.

There is one predefined global integer variable that can be used to track state within properties. That variable is called *uno_state*. It has an initial value of zero. Actions on this predefined variable are restricted to assignments and simple post-increment and post-decrement operations. Queries are restricted to simple boolean operations, as for instance in:

```
if (uno_state > 0)
        uno_state--;
else
        uno_state = 5;
```

There are generally no limitations on the type of control structure that may be used to specify properties. There are some limitations, though, on the structure of expressions that are used as conditionals. With some minor exceptions, expressions must be simple and consist of only a single query primitive or state comparison. That is, instead of writing:

```
if (select("", DEF, NONE) && uno_state == 3)
        list();
```

or

```
if (select("", DEF, NONE) != 0 && !(uno_state != 3))
        list();
```

one writes:

```
if (select("", DEF, NONE))
if (uno_state == 3)
        list();
```

The reason for this seemingly ad-hoc limitation is that the properties are preprocessed and parsed by the UNO parser, to form the control structure, but the expression from conditionals are interpreted on-the-fly by UNO during the analyses. The interpreter engine is still quite simple (and is even more demanding for global properties than for local properties).

Below we list the syntax, semantics, and possible usage of each of the predefined actions and queries from

the UNO property specificatgion language.

error(STRING)

Execution of this action causes the string that is given as an argument to be printed, followed by a stack-trace of the path that lead to this point in the execution. The stack-trace contains only the names of functions called, unless UNO is invoked with the optional argument –t.

Usage: The error action can be used in the definition of both local and global properties.

no_error(STRING) and no_error()

If a string argument is given, the string is printed when the action is executed, otherwise the action has no effect. Example:

```
if (uno_state == 3)
        no_error();
else
        error("not good");
```

Usage: The no_error action can be used for the definition of both local and global properties.

list()

This action is intended to make it easier to debug property definitions. When the action is executed it causes the list of currently selected and/or marked variables to be printed.

Usage: The list action can be used for the definition of both local and global properties.

mark(CONSTANT) and unmark()

Execution of the mark() action causes all currently selected symbols to be marked with the value of the argument, which should be non-zero. Execution of unmark removes the marks from all selected symbols. A call to unmark() is equivalent to a call to mark(0).

Usage: The mark() and unmark() actions be used for the definition of both local and global properties.

select(STRING,CONST_EXPR,CONST_EXPR)

The execution a select query always erases any existing selection of symbols. (Marks, however, are preserved and can only be erased by a call to unmark.) Symbols (e.g., variable or function names) from the current statement in the program can be selected, and thereby made part of the new selection, if and only if three conditions are met.

1 Either the first argument to select must equal the empty string or it must equal the name of the symbol.

2 At least one of the dataflow tags on the symbol must match at least one of the dataflow tags that are used to form the value of the second argument to select.

3 None of the dataflow tags on the symbol matches any of the dataflow tags that are used to form the value of the third argument to select.

The query returns true if at least one symbol was selected; otherwise it returns false.

The constant expressions used in the last two arguments can be constructed from the tags in Table 2, combined with the bitwise or operator | as in: DEF|USE|DEREF. Alternatively, the special tag ANY can be used to match any of the tags from Table 2, and the special tag NONE can be used to match none of the tags. (NONE is a synonym for zero.)

Usage: The select query be used for the definition of both local and global properties. In the current implementation of UNO, though, the first (string) argument to select must be the empty string .

unselect(STRING,CONST_EXPR,CONST_EXPR)

Like select, but in this case the existing selection of any matching symbols is erased. For a match, the same three conditions apply. The query returns true if at least one selected symbol remains; otherwise it returns false.

Usage: In the current implementation of UNO, the unselect query can only be used for the definition

of local properties.

refine(CONST_EXPR,CONST_EXPR)

Restrict the number of symbols in the current selection to those that satisfy two additional conditions:

1 At least one of the dataflow tags on the symbol must match at least one of the dataflow tags that are used to form the value of the first argument to select.

2 None of the dataflow tags on the symbol matches any of the dataflow tags that are used to form the value of the second argument to select.

Note that with a combination of select and refine queries, a boolean and combinations of conditions can be enforced For example, in the sequence:

```
if (select("", DEF|USE, NONE))
if (refine("", DEREF|ALIAS, NONE))
        list()
```

the call to list would be executed if a symbol is encountered with data flow tags that match ((DEF lor USE) land (DEREF lor ALIAS)).

Usage: In the current implementation of UNO, the refine query can only be used for the definition of local properties.

match(CONST,CONST_EXPR,CONST_EXPR)

The match query checks every symbol that is currently selected (as a result of previous calls to select and refine) and checks if any of these symbols satisfy three additional conditions:

1 The symbol is marked with the (non-zero) value given in the first argument to the match query.

2 At least one of the dataflow tags on the symbol must match at least one of the dataflow tags that are used to form the value of the second argument to select.

3 None of the dataflow tags on the symbol matches any of the dataflow tags that are used to form the value of the third argument to select.

The query returns true if at least one symbol was found that satisfied all three conditions. It returns false otherwise. If the return value is false, the selection that existed before the query was made is left undisturbed. If it returns true, only the symbols that matched remain in the selection.

Usage: The match query can be used for the definition of both local and global properties.

marked(CONST,CONST_EXPR,CONST_EXPR)

This query returns true if symbols exist that satisfy the following same three conditions as a call to match with the same arguments, except that the symbols need not be part of the current selection.

Usage: In the current implementation of UNO, the marked query can only be used for the definition of local properties.

fct_call(STRING)

The query returns true if and only if the current statement in the program contains a symbol with the name given in the argument and dataflow tag FCALL. Execution of the query does not alter the existing selection.

Usage: The fct_call query can be used for the definition of both local and global properties.

known_nonzero()

The query retuns true if all currently selected symbols are known to have a non-zero value. It returns false otherwise. Execution of the query does not change the current selection.

Usage: The known_zero query can be used for the definition of both local and global properties.

known_zero()

The query retuns true if all currently selected symbols are known to have a zero value. It returns false otherwise. Note that it is possible for both known_zero() and known_nonzero() to return false, if the value of the symbol cannot be determined. Execution of the query does not change the

current selection.

Usage: In the current implementation of UNO, the known_zero query can only be used for the definition of local properties.

path_ends()

The query returns true if the current statement in the program has no successor, e.g., because it corresponds to an return statement, or the call of a function known not to return control to the caller, such as exit() or panic(). Execution of the query does not change the current selection.

Usage: In the current implementation of UNO, the path_ends query can only be used for the definition of local properties.