SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **30**:775–802

A static analyzer for finding dynamic programming errors

William R. Bush*, Jonathan D. Pincus and David J. Sielaff

Intrinsa Corporation, Mountain View, CA, U.S.A.

SUMMARY

There are important classes of programming errors that are hard to diagnose, both manually and automatically, because they involve a program's dynamic behavior. This article describes a compile-time analyzer that detects these dynamic errors in large, real-world programs. The analyzer traces execution paths through the source code, modeling memory and reporting inconsistencies. In addition to avoiding false paths through the program, this approach provides valuable contextual information to the programmer who needs to understand and repair the defects. Automatically-created models, abstracting the behavior of individual functions, allow inter-procedural defects to be detected efficiently. A product built on these techniques has been used effectively on several large commercial programs. Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: program analysis; program error checking

INTRODUCTION

There are important classes of programming errors that are hard to diagnose, both manually and automatically, because they involve a program's dynamic behavior. They include invalid pointer references, faulty storage allocation, the use of uninitialized memory, and improper operations on resources such as files (trying to close a file that is already closed, for example).

Finding and fixing such errors is difficult and expensive. They are usually found late in the development process. Extensive testing is often needed to find them, because they are commonly caused by complex interactions between components. Our measurements indicate that in commercial C and C++ code, on the order of 90% of these errors are caused by the interaction of multiple functions. In addition, problems may be revealed only in error conditions or other unusual situations, which are difficult to provoke by standard testing methods.

Traditional checking provided by the error-checking portion of compilers identifies errors relating to the static expression of a program, such as syntax errors, type violations, and mismatches between

*Correspondence to: William R. Bush, 1739 Lexington Avenue, San Mateo, CA 94402-4024, U.S.A.

Received 9 November 1998
Revised 27 December 1999
Accepted 28 December 1999

a function's formal and actual parameters. Compilers also perform more sophisticated analyses, which can include pointer analysis (see [1] for an example), but for the purpose of code generation, not error checking. Some of these techniques can also be applied to certain kinds of error checking (for example, optimizing compilers sometimes identify uninitialized variables), but there are three significant restrictions (see [2] for a first attempt at using pointer analysis techniques for checking). First of all, most analysis is done intra-procedurally, and so problems caused by the interactions between functions are not detected. Secondly, these techniques are not applicable to many categories of defects (such as memory leaks). Thirdly, by virtue of their focus on code generation they are forced to make very conservative assumptions, and hence, for example, only find situations in which a variable is always uninitialized, rather than situations where it is uninitialized only on certain paths.

Specialized checkers such as Lint [3] find broader classes of errors, including stylistic mistakes and potential portability problems, but are still limited and are prone to reporting false errors. Within a single procedure, these checkers take the opposite approach to compilers, and report a much broader set of potential errors, the majority of which are not in fact errors. In particular, these tools tend to report potential problems on paths which are not actually achievable. Because of this, significant deployment of these tools in commercial software development is relatively infrequent, and when it does occur it usually requires substantial investment in developing tools to filter and process the results. As with compilers, the analysis concentrates on intra-procedural problems; with certain limited exceptions, cross-function errors are not reported.

Annotation checkers such as LCLint [4,5], Aspect [6], the Extended Static Checker [7–9] and the monadic second-order logic checker [10] provide better checking by applying techniques from program verification without requiring complete verification. However, they require substantial additional work by the user, who must provide and tune annotations in a manner similar to verification. Additionally, only LCLint handles unsafe (C style) pointers and has been applied to moderately large systems.

Abstract interpretation [11,12] is a form of program analysis that maps programs into more abstract domains. This makes analysis more tractable and potentially useful for checking. The technique requires safety and completeness, however; analysis must be correct for all possible inputs. It also has difficulty in practice with large programs. In contrast, error checking, to be of practical value, must be able to handle large programs. Furthermore, error messages need not always be correct as long as the number and type of spurious ones are below usability thresholds.

In contrast to the above techniques, the debugging tool Purify [13] and similar runtime memory debuggers detect a broad range of errors and require no extra programmer effort to use. They are, however, debuggers, operating on heavily instrumented executables (see, for example, [14]) and requiring test cases, which impose serious limitations.

Thus, the goal of the research reported here was to develop a source code analyzer that could find Purify-like errors with Purify's ease of use, but without needing test cases. This goal led to a few specific requirements.

- *Real world programs written in C and C++ should be checked effectively*. Analysis must therefore handle such difficulties as pointers, arrays, aliasing, structs and unions, bit field operations, global and static variables, loops, gotos, third party libraries, recursive and mutually-recursive functions, pointer arithmetic, arbitrary casting (including between pointer and integer types), and overloaded operators and templates (for C++).

- *Information should be derived from the program text rather than acquired through user annotations*. This is possible because the semantics of a language imply certain consistency rules, and violations of these rules can be identified as defects. For example, the semantics of local variables allow for the detection of defects such as using uninitialized memory.
- *Analysis should be limited to achievable paths; that is, sequences of program execution which can actually occur in practice.* This requires detailed tracking of actual values, not just performing data- and control-flow analysis.
- *The information produced from the analysis should be enough to allow a user to characterize the underlying defects easily*. This is especially important, and hard to achieve, with large programs.

In response to these goals, a new method of analysis was developed, based on simulating the execution of individual functions. The method can be summarized in a few key concepts.

- Simulation specifically consists of sequentially tracing distinct execution paths through the function being analyzed, and simulating the action of each operator and function call on the path on an underlying virtual machine. By tracking the state of memory during path execution, and applying the consistency rules of the language to each operation, inconsistencies can be detected and reported. In addition, by examining the current state of memory whenever a conditional is encountered, the analysis can be restricted to achievable paths. Because of the detailed tracking of paths and values, precise information is available to help the user understand the situation in which the defect manifests itself.
- The behavior of a function is described as a set of conditionals, consistency rules and expression evaluations. This summary of the behavior is called a model of the function. Whenever a function call is encountered during path execution, the model for that function is used to determine which operations to apply.
- The information produced while simulating a function is sufficient to generate a model for that function automatically.
- To apply these techniques to an entire program, or subset of a program, analysis begins with the leaf functions of the call graph and proceeds bottom-up to the root. As each function in turn is simulated, defects are identified and reported, and the model for that function is available for subsequent simulation of its callers.
- This bottom-up approach uses a function's implementation to generate constraints on the callers of that function. This is particularly valuable in situations where the text of the complete program is not available, either because the program is only partially implemented, or because the code under analysis is designed as a component that may fit into many different programs.

An error detection tool for C and C++, called PREfix, was built based on these techniques. It has been used on several large commercial programs. The remainder of this paper discusses in detail the operation of PREfix and presents some experience with it.

AN EXAMPLE

The following example presents a simple imperfect function and the warnings that PREfix generates when analyzing it.


```
1   #include <stdlib.h>
2   #include <stdio.h>
3
4   char *f(int size)
5   {
6        char *result;
7
8        if (size > 0)
9            result = (char *)malloc(size);
10       if (size == 1)
11           return NULL;
12       result[0] = 0;
13       return result;
14  }
```

Figure 1. An example function.

```
example1.c(11) : warning 14 : leaking memory
    problem occurs in function ‘f’
    The call stack when memory is allocated is:
        example1.c(9) : f
    Problem occurs when the following conditions are true:
        example1.c(8) : when ‘size > 0’ here
        example1.c(10) : when ‘size == 1’ here
    Path includes 4 statements on the following lines: 8 9 10 11
    example1.c(9) : used system model ‘malloc’ for function call:
        ‘malloc(size)’
    function returns a new memory block
        memory allocated
```

Figure 2. First error message.

The function in Figure 1 allocates memory and initializes it. More specifically, if the size of memory to be allocated is greater than zero then it allocates memory; if the size of the memory is 1 it then returns NULL; otherwise it sets the first word of the allocated memory to zero. PREfix generates three warnings when analyzing this function.

The warnings have the general form of extended compiler warning messages. They identify the file, function and line in which the defect was detected; the type of defect; the variable involved and where


```
example1.c(12) : warning 10 : dereferencing uninitialized pointer 'result'
    problem occurs in function 'f'
    example1.c(6) : variable declared here
    Problem occurs when the following conditions are true:
        example1.c(8) : when 'size <= 0' here
    Path includes 3 statements on the following lines: 8 10 12
```

Figure 3. Second error message.

```
 example1.c(12) : warning 11 : dereferencing NULL pointer 'result'
     problem occurs in function 'f'
     example1.c(9) : value comes from return value of 'malloc' here
     Problem occurs when the following conditions are true:
         example1.c(8) : when 'size > 0' here
         example1.c(10) : when 'size != 1' here
     problem occurs when 'memory exhausted'
     Path includes 4 statements on the following lines: 8 9 10 12
     example1.c(9) : used system model 'malloc' for function call:
         'malloc(size)'
     function returns 0
         memory exhausted
```

Figure 4. Third error message.

it was declared; which models of called functions were used; the path, as a sequence of line numbers, on which the defect was detected; and the conditions on that path that caused the defect.

The error in Figure 2 occurs on the return of NULL when the size is 1; the allocated memory is neither freed nor returned.

The error in Figure 3 occurs when the function tries to dereference an uninitialized pointer (when size is less than or equal to zero the `malloc` in line 9 will not be executed).

The error in Figure 4 occurs when `malloc` cannot allocate memory and returns a NULL pointer. There is no check for this case, and thus the use of the pointer for initialization is incorrect.

ANALYZER STRUCTURE AND IMPLEMENTATION

PREfix begins by parsing the source code (using a standard C/C++ compiler front end) into abstract syntax trees, determining the order in which to simulate functions (by performing a topological sort based on caller–callee relationships), and loading existing models for any functions which are called

```
while (there are more paths to simulate)
    {
       initialize memory state
       simulate the path, identifying inconsistencies and updating the memory state
       perform end-of-path analysis using the final memory state,
          identifying inconsistencies and creating per-path summary
    }
    combine per-path summaries into a model for the function
```

Figure 5. Pseudo-code for function simulation.

but not implemented in the subset of code being simulated (if they are available). Individual functions are then simulated and defects reported. Recursion is handled by iterating a user-selectable number of times (the default is twice); mutual recursion is done by choosing one function arbitrarily as the initial caller.

Simulating a single function consists of simulating achievable paths through the function. The set of achievable paths through a function consists of the set of all control flow paths through the function that could happen in practice (including control flow through the functions it calls). The size of this set is often less than the apparent number of control flow paths, because the same conditions often control multiple conditional blocks of code (this set is also a superset of possible paths). Since the number of achievable paths through a function is still often quite large (especially for functions high in the call graph), potentially exponential in the number of branches, PREfix selects a representative sample of achievable paths to simulate. The maximum number of paths in that sample is a user configurable setting. If this number is set at least as large as the number of achievable paths through the function, then all achievable paths will be simulated.

Simulating a path involves traversing the function's abstract syntax tree and evaluating the relevant statements and expressions in the tree. The memory state of the function being simulated is updated as a result of evaluating each statement. Many defects (for example, uninitialized memory and NULL or invalid pointer dereferences) can be identified by applying consistency rules to the memory state during path simulation. Others (such as memory leaks and returning a pointer to freed memory) are identified in a separate step at the end of the path.

After a path is simulated, the final memory state of the function is summarized. After all the paths have been simulated, the per-path summaries are combined into a model for the function. (See Figure 5.)

Per-path simulation

Each path through a function begins at the function entry point and continues through some number of statements, where the sequence of statements may be determined by conditional control flow constructs, until the function exits (either by an explicit return, a non-local goto, or falling off the end of the function).

Paths

Because of the basic path tracing approach taken by simulation, a warning can precisely identify the path through the function on which it will occur. Consider the second warning message in the example above (Figure 3). The path on which it occurs – statements 8, 10 and 12 – indicates that this problem occurs when both if clauses are false.

A path includes function calls and their results as well as statements. For example, the third message above (Figure 4) describes a defect that manifests itself only when memory allocation fails. The last two lines of the message describe the code level manifestation of this condition (the call to the function `malloc` returns 0) and the higher level characterization (no memory is available).

Note that the existence of a warning does not mean that it only occurs on the given path, but simply that the given path is one on which the warning occurs. A possible future extension is to remember all the paths on which the warning occurs, and then produce a minimal set of preconditions necessary to cause the error condition.

The notion of simulating execution paths for checking first appeared in an early error checker called SELECT [15]. SELECT traced all paths through a function, constructing the enabling predicates for each path. These predicates were used for generating test cases and for performing formal verification on each path.

Memory: exact values and predicates

The memory used by a function being simulated is tracked as accurately as possible. It is created on an as-used basis. The first reference to any constant, parameter or variable (or any indirect pointer accesses to memory) causes the simulator to lay out the memory, associate with it the source code construct that caused its creation, and give it an initial state. Memory is tracked down to the level of the individual bit, although for efficiency it is handled in larger chunks (bytes and words) whenever possible.

Each piece of memory has a value. There are three basic types of value: a known exact value, an initialized value whose exact value is not known, or an uninitialized value. When the exact value is known it is expressed in terms of the precise bits for that value. Constants, for example, always have their exact values known. Alternatively, the exact value may not be known, but it may be known (or assumed) that the memory is initialized. This happens, for instance, with the initial values of the parameters to a function. Finally, some memory may initially be uninitialized, such as local variables without initializers.

In addition, a set of predicates may be associated with the value. Predicates include a set of unary constructs representing language semantics, such as *initialized*; a set of binary constructs representing the usual relational operators; and ternary constructs expressing more general relations of the form `a=b<op>c`. Binary predicates can represent relations to constants (for example, `x>3`) or relations between variables (`x>y`). Ternary predicates represent relations that include three operands (such as `a=b*2`). Binary and ternary predicates may be combined to express relations of arbitrary complexity between computed values (`x+17!=y*3`, for example). The basic architecture supports arbitrary predicate operations; in order to optimize performance, the current implementation stops short of full-fledged symbolic computation.

There is, in addition, a special operator applicable to pointer values: dereference. It must be possible to get to the pointed-to memory by dereferencing a pointer value.

Operations on memory: evaluate, test and set

The operators applied to memory during simulation are just the unary and binary operators of C and C++. Operators are applied in three ways: setting, testing and assuming. Setting assigns a value to memory, and occurs in common situations such as expression evaluation. Testing evaluates expressions in contexts where a boolean result is required, such as a conditional. Assuming is performed in certain circumstances after testing is done, and the result is not a simple boolean; these circumstances are discussed in the next subsection.

Setting a memory location via an operator (and one or two operands) changes the state of that memory location to reflect the new information. In situations where exact values are known, the result can be computed exactly. Where only predicate knowledge is known, the simulator combines the predicates as best it can. For example, if `a>3` and `b>4`, `a+b>7`, but if `a>3` and `b<4`, no inference can be made about `a+b`.

Expressions are evaluated by recursive descent. Variables and constants are leaf nodes, and an expression referring to one of them simply evaluates to the memory associated with that entity. Evaluating a non-assignment operator returns a new memory location containing the resulting value. Assignment operators copy the source reference to the destination without creating a new memory location.

Because simulation deals with incomplete knowledge, testing involves three-valued logic, where a result may be TRUE, FALSE or DON'T KNOW.

Testing occurs in three contexts: when an expression is being evaluated in a context where a boolean result is required (usually a conditional with control-flow implications); when language semantics are being enforced (in which case a FALSE result may trigger a warning message); and, as discussed later, during the selection of potential outcomes in a model.

Tests to enforce language semantics involve additional, interpolated conditions that are not standard C and C++ operators. An example is the condition that memory must be initialized before being used in a computation. This test is performed before the computation. Another example is pointer validity. Before a dereference operation, the pointer value must be tested, and if it is NULL or invalid, a warning message is generated. In practice, the semantics are slightly more complicated because of the various ways in which a pointer can be invalid; it is useful to distinguish between reporting a NULL pointer dereference, a dereference of a pointer to freed memory, and a dereference of a constant such as 1 (which is used by some C standard library functions to signal an error condition).

An added complication occurs when semantic-enforcing tests are applied to the externals (such as parameters) of a function. In this case, the result is likely to be DON'T KNOW, since the truth or falsity of the test may well depend on the specific value of the parameter being passed, which is unknown due to the bottom-up nature of analysis. Reporting a message in this situation might lead to a flurry of false positives from the user's perspective; on the other hand, losing the information completely would mean missing the valuable knowledge of defects caused between interactions between functions. This problem is addressed by deferring the tests, and instead putting constraints in the model for the function being simulated, which will be tested when the model is used.

Conditions, assumptions and choice points

In addition to a path, the context of a warning can be described in terms of the conditions that will cause it. In the second warning in the example above (Figure 3), for instance, if, on line 8, the size

is zero or less, then the variable `result` will be uninitialized. Conditions correspond to *assumptions* made by the simulator during path simulation.

Assumptions are made when variables are used in relational expressions but their values are unknown, and they are constructed from the expressions involving those variables. In the above example (Figure 1), the value of the size parameter is unknown (because it is an input parameter and simulation is performed bottom-up, callee before caller). The parameter is used in a greater-than comparison with zero, so an assumption is made that it is less than or equal to zero, and the corresponding path is then taken. This assumption is remembered and used in subsequent references to the parameter. The other possible assumption, that the parameter is greater than zero, could as plausibly and arbitrarily be made, and would result in a different path being taken. After making an assumption on line 8, the condition on line 10 must be evaluated. If the condition on line 8 was assumed to be false, then the condition on line 10 will also be known to be false. However, if the condition on line 8 was assumed to be true, the condition on line 10 will have a truth value of DON'T KNOW; another assumption must be made at this point to reflect which path to simulate, either `size==1` or `size!=1`.

When an assumption is made, the state of the relevant memory locations is updated to reflect this information. This updating has a cascading effect when it allows other predicates to be evaluated exactly. For example, if a predicate exists that states `a=b+5`, and a conditional expression `a==2` is assumed to be true, the state of `a`'s memory is updated to reflect the exact value 2, and `b`'s memory state is also updated to reflect that its value is `-3`. The predicate `a=b+5` is then discarded, since having exact values is much more efficient than deducing the same information from predicates.

This situation, where control flow is not completely determined by known values, is called a *choice point*. When a choice point is encountered, the simulator picks one choice for the current path, and investigates the alternate choices on subsequent paths.

Choice points occur whenever non-semantic-enforcing test operations lead to a DON'T KNOW result. Specific examples of this include relational expressions, switch statements, if conditions (which implicitly test against 0), and selection of possible outcomes in a model (see below).

Recording information about the choices made is vital to avoid false paths. For example, if the result of a test of a parameter against 0 is DON'T KNOW, and a result of TRUE is chosen, the parameter's value is set to 0, and that value is used in the simulation of the rest of the path. The situation where the parameter's value is non-zero is investigated on a later path.

End-of-path analysis

After all statements in a path have been simulated, some additional analysis is performed. For example, it is an error if the return value (or, indeed, any external) has been assigned a pointer to stack-based memory such as a local variable (because the values on the stack can be overwritten by a subsequent function call). This error is detected by making a pass over all the memory reachable from any external at the end of the path and finding any local variables. As discussed below, leak analysis is also performed at this time.

Multiple paths

Per-path simulation is repeated on subsequent paths until either all paths have been covered, or the user-configurable maximum number of paths has been simulated. Most well-structured functions have

a relatively small number of paths, but functions which are long and complex may have very large numbers of paths, and functions involving certain kinds of loops have an effectively unbounded number of paths. In practice, as discussed below, relatively small settings for the maximum number of paths suffice to find the large majority of defects.

The order in which the different paths are investigated affects the overall coverage when complete simulation is not possible. Paths are chosen randomly at if and switch statements. For called functions, a heuristic involving equivalence classes of return values is used to maximize the number of statements covered. Paths (more precisely, outcomes, described below) with return values different from ones already encountered are selected first.

Models

A model embodies the behavior of a function. When the simulator encounters a function call, it emulates the called function using the function's model. Models for the basic operating system libraries are provided with PREfix; models for user code are created automatically as a result of simulation.

Model structure

A model for a function represents a set of *outcomes*, possible transformations of the state of memory that can be computed by the function. For example, a simple model for the C standard library function `fopen` has two outcomes: success, in which case an opened file is returned; or failure, in which case a NULL pointer is returned. (Note that other, more detailed, models are possible, in which the single failure outcome is separated into multiple independent outcomes based on the reason for failure.)

A model also includes a list of *externals*, representing the function's interactions with the outside world: parameters, return value, global and static variables accessed by the function, and any memory used or modified by the function that is reachable from an external. For `fopen`, for example, the externals include the function's parameters and the memory pointed to by them (since both parameters are strings), the return value, and the global variable `errno`.

Each outcome in turn is defined as a set of *guards*, *constraints* and *results*, collectively referred to as *operations*. An operation is specified as an operator and one or more operands, which may be references to externals, constants or temporaries created during model emulation and then discarded.

Constraints are preconditions, predicates that must be true when a function is invoked. Violating a constraint results in an error message. In both outcomes of `fopen`, for example, its two parameters must be valid pointers, and the memory they point to must be initialized. LCLint also uses constraints as preconditions.

Results are the function's postconditions. The result of the success case of `fopen`, for example, is the condition that the return value is a valid pointer, pointing to a file that must be closed later. In the failure case, there are two results: the return value is NULL, and the global variable `errno` is set to some positive value. As this example indicates, results can be specified in terms of arbitrary relational operators. It is also possible to represent that a result sets a memory location to an unknown value, or that the value is equal to the value of another external.

Guards are tests used to deal with the fact that a particular outcome may be dependent on the values of the function's inputs, and may only make sense in the presence of specific inputs. Consider, for


```
1  int knot(int j)
2  {
3      if (j==0)
4          return 1;
5      return 0;
6  }
```

Figure 6. A function requiring a guard in its model.

```
1  int deref(int *p)
2  {
3      if (p==NULL)
4          return NULL;
5      return *p;
6  }
```

Figure 7. A dereferencing function.

example, the function in Figure 6. In this function, the return value will be 1 only when the input parameter `j` is 0. The model outcome that has a return value of 1 thus also has a guard to ensure that `j` is equal to 0. Likewise, the outcome where the return value is 0 has a guard to ensure that `j` is not equal to zero.

In contrast to constraints, no message is produced if a model's guards are not true during model emulation. Such an outcome is simply not eligible for selection. The example function in Figure 7 illustrates the importance of this construct.

Additional information can be associated with the model, the outcomes, or indeed particular operations. For example, the line number of a return statement (or, indeed, the entire simulation path corresponding to the outcome) can be associated with an outcome. This information can be very useful in making information about the defect meaningful to the user.

An example function

Consider the simple dereferencing function in Figure 7. The model for this function has three externals.

- The parameter `p`.
- The memory pointed to by parameter `p`.
- The return value.

The model has two outcomes.

```
(deref
        (param p)
    (alternate return_0
        (guard peq p NULL)
        (constraint memory_initialized p)
        (result peq return NULL)
        )
    (alternate return_X
        (guard pne p NULL)
        (constraint memory_initialized p)
        (constraint memory_valid_pointer p)
        (constraint memory_initialized *p)
        (result peq return *p)
        )
    )
```

Figure 8. The model of the dereferencing function.

- Outcome one:
- guard: p equals NULL;
- constraint: p must be initialized;
- result: the return value is NULL.
- Outcome two:
- guard: p does not equal NULL;
- constraint: p must be initialized;
- constraint: p must be a valid pointer;
- constraint: the memory pointed to by p must be initialized;
- result: the return value is equal to the memory pointed to by p.

Note that the called function explicitly tests for NULL, so that when the caller passes in a NULL a valid pointer is not required. Thus, outcome one has a guard that guarantees a NULL parameter value and has no valid pointer constraint. In general, an outcome is only possible in the situation where none of its guards is false (this is a looser requirement than all of its guards being true, because a guard may have an unknown value). Without the concept of guards, it would be impossible to put a pointer validity constraint on p, because the NULL pointer case could not be separated out, and the constraint would then sometimes be applied to NULL; with guards, the outcome where p does not equal NULL can indeed have such a constraint.

The actual text of the model, written in the analyzer's modeling language, appears as in Figure 8. The syntax of the modeling language is LISP-like, because LISP syntax is easy to generate automatically.

The verbal model is an exact transcription of the actual model, with the exception of the externals. In the actual model only function parameters need to be declared. Note also that the result information

(return zero, return non-zero) is duplicated in each outcome header (alternate) as an optimization for outcome selection during emulation, so that outcomes do not have to be scanned for result clauses.

The complete syntax of the modeling language is presented in the Appendix.

Model emulation

The steps taken to evaluate, or *emulate*, a model follow from the nature of its components.

(i) The appropriate model is retrieved. This may require dereferencing a function-valued variable.
(ii) All guards are evaluated to determine eligible outcomes. Possible values for a guard are FALSE (the outcome is not eligible for selection), TRUE and DON'T KNOW (some value or values referenced in the guard are unknown).
(iii) An eligible outcome is selected. This selection creates a choice point.
(iv) All guards in the selected outcome having unknown values are assumed. This is done in the same manner as boolean expressions, discussed above.
(v) All constraints are tested and any violations are reported as warnings.
(vi) All results are evaluated.

Dynamic memory and resources

Modeling allocation and deallocation of memory requires special model operators. The model for `malloc`, in particular, uses an operator that creates memory tracked by the simulator and returns a pointer to that memory. The model for `free` uses a constraint operator that verifies that its operand is a pointer to freeable memory (and not, say, a pointer to a local stack variable). The model for `free` also uses a result operator that marks as freed the tracked memory pointed to by its operand.

As the simulator runs it interpolates constraints on all memory references, checking for invalid pointers and unintialized memory, and making sure that pointed to memory has not been freed. These constraints are passed on to the function's model, as can be seen in the `deref` example function above (Figure 8).

This approach has been generalized to arbitrary resources, such as files, transactions and windows. Model operators support the creation, deletion and state change of resources. Discovering that the program has attempted to write to a file after it has been closed, for example, is directly analogous to discovering that the program is using memory that has been freed.

During end-of-path analysis, the simulator performs a simple mark-sweep operation to discover which memory is reachable from an external. Any unreachable memory or resource that has been allocated, but not freed, is reported.

Model generation

Models are crucial to the accurate operation of PREfix. Their automatic generation is crucial to the usability of PREfix as a real-world tool. Manual creation of models for complex code of any size would be tedious, error prone, and most likely unacceptable to users. (The Aspect specification checker can automatically generate specifications for called functions, for similar ease of use reasons.)

The basic technique of model generation, or *automodeling*, is to remember the relevant state of memory at the end of each path, and then to merge the disparate states into one model at the end of function simulation.

The per-path state is constructed from the tests and assumptions made during the path, along with the result, and is referred to as the path's outcome. The tests made on a path become constraints in the model – they are conditions that must be true when the function is called (and executes that path). A constraint that is not true when the function is called results in a warning message. Assumptions made along a path become guards in the model – they are conditions that enable the execution of the path. An assumption that is not true when the function is called indicates that the path would not be executed; a false guard therefore inhibits evaluation of its associated outcome.

Not all the per-path memory state must be remembered at the end of a path. Only the state that is externally visible must be saved, and that state is just the memory reachable from the function's externals (parameters, return value and globals).

Merging states at the end of function simulation is not necessary semantically. It is, however, necessary for performance; otherwise there would be one outcome per path, and there can be many paths through a function. It is also space efficient – many paths have very similar states.

Exactly equivalent outcomes are merged directly. Outcomes which match exactly, except for a pair of opposite assumptions (for example, one outcome assumes `x>0` and the other assumes `x<=0`), are also merged, eliminating the opposing assumptions. Between 80% and 85% of outcome merging is exactly equivalent merging; the remainder is opposing assumption merging.

Other approaches to merging were tried. States with similar tests and guards can be merged into boolean combinations of inequalities, and overlapping ranges can be combined. These techniques were useful on individual programs, but in large-scale testing never merged more than 5% of outcomes. This benefit was deemed not worth the performance cost, and so the techniques are not used.

Note that all of the merging techniques described above exactly preserve the semantics of the merged outcomes. Experiments were performed that employed aggressive outcome meging at the cost of accuracy, typically by losing some amount of precision. For example, if one outcome had a result `*p=5` and another had `*p=8`, the outcomes were merged with a fuzzy result of '`*p` is initialized'. This aggressive merging resulted in more compact models, but the correctness implications (and resulting quality of defect reports) were surprisingly severe: fewer real errors were detected, and a substantially higher number of false positive messages were reported. As a result, any outcome merging approach which lost information was abandoned.

Incomplete knowledge and conservative assumptions

At many points during analysis, the information in the simulator may be incomplete. For example, function-valued variables are often difficult to disambiguate, and so the simulator does not know which model to apply. Alternatively, sometimes models for called functions may not be available, due, perhaps, to the use of third party components for which source code or models are not available. It is important that PREfix behave benignly when this happens; otherwise the number of spurious warnings can increase significantly.

The general philosophy is to be conservative: avoid generating incorrect messages, even at the potential expense of failing to identify real defects. Note that this meaning of conservative, emphasizing

ease of use over correctness, is different from usual program analysis terminology. The utility of this emphasis is described in the next section.

For unmodeled functions, this conservatism specifically means assuming the following.

- The function can modify any memory it has access to (all memory pointed to by non-constant parameters is set to an unknown initialized value).
- All pointers are used (all pointers are aliased).
- The return value can be anything (so that later use of the return value will be correct in any context).

This leads to correctness at the cost of defect detection. Experiments indicate that PREfix with absolutely no models – performing no cross-function analysis whatsoever – finds only 5% as many defects as PREfix with a full set of models. In practice, however, the models for the base libraries (for the appropriate operating system platform) and the automatically-generated models (for functions which have source code available) are present, and detection rates are correspondingly high.

Another example of a conservative assumption occurs during leak detection. In some situations, PREfix may be unable to discover whether a pointer has been aliased. In this case the pointer is simply marked as 'lost' and not reported as leaked. An interesting future possibility is to report this lost memory, which may assist the user in tracking down issues related to overall program memory consumption.

DEVELOPMENT AND USE OF THE ANALYZER

From the outset the use model for PREfix was Purify, which is striking in its ability to find obscure errors quickly in a large mass of code. This use model was not only employed in the deployment of PREfix, but for the development and improvement of it. Rather than inspect a small body of code for errors and then run PREfix on it, from the outset the analyzer was run on as much code as was available (starting with public domain source and moving as soon as possible to large commercial programs), examining the code after analysis to verify reported errors.

Over 100 million lines of code in a wide range of styles have been analyzed. Typically, during development the analyzer would be taken to a commercial site, installed in the build environment, and run on a source base of several hundred thousand lines. At the end of a week the errors discovered would be reported to senior engineers, and the problems encountered with the analyzer would be used to improve it.

The result of this development strategy was to improve the tool iteratively in a pragmatic fashion, with a focus on reducing spurious and noisy error messages and increasing performance. It was never a goal to provide a guarantee that all errors of a given type would be found.

The key to making the analyzer useful has been managing the exponential nature of analysis (inherent in path simulation) by maximizing the trade-off between quality of information and the cost of acquiring it. The most important mechanisms in that process have been: path selection (including the heuristic described above), model abstractions capturing just enough detail, model simplification through outcome merging (also described above), and parameterizing analysis so that it can be tuned as required by circumstances.

Parameters are used to control the amount of work done in a run. For example, for any set of functions the number of paths explored per function can be limited. The user need not know about these parameters; reasonable defaults have been defined for them based on experience (the default maximum number of paths is 50). Other example parameters include the ability to limit the time spent analyzing a function, to turn off leak checking (since some users don't care about leaks), and to guarantee that memory allocation will never fail (because some users don't care about this error). In general, as with other features of PREfix, these parameters were chosen based on experience.

In practice, modeling and emulation have proven tractable.

It has been sufficient to limit modeling primitives to relational constraints, constructors of sets of bits, and the points to, dereference and offset operators (structs are modeled with these). The information being lost with these primitives includes complex arithmetic relations and string length characteristics; the analyzer may be extended at some point to retain this information.

The size of models has not been a problem. The model file for a source file typically is roughly the same size as its debug object file, even with simple (exact) outcome merging, described above. It is significant that models are not exponentially larger higher up in the call tree. The number of outcomes remains tractable. Models are able to capture efficiently the behavior of all called functions, including those called indirectly. This in turn has made emulation tractable, effectively capping the number of paths (and, in another sense, path length; if direct models had not been adequate, it might have been necessary to emulate all levels of called functions).

PREfix contrasts sharply with annotation-based ESC, LCLint and ASPECT. Given the goal of providing Purify-like results quickly for large code bases, annotation was not feasible. ESC reports [8] a 13.6% annotation overhead in a 20 000 line program. During PREfix development, the commercial code bases checked were at least an order of magnitude larger.

A major reason the Purify use model was chosen was the belief that many software organizations are resistant to methodological change, required for an annotation-based checker. This belief was borne out in practice. A surprising number of organizations did not care about leaks or `malloc` failures. Some organizations did not fix found errors because of the regression testing required to verify the fixes.

Note that the complete PREfix tool includes substantial infrastructure around the simulation engine in addition to parameterizing analysis: mechanisms to store the warnings in a standard SQL database and display them along with annotated source code in a web browser; the ability to filter, order and summarize warning messages; support for user configuration; and mechanisms for integrating with makefiles and other build processes.

Also note that to be completely correct the analyzer must correctly model the operation of the target hardware (with respect to floating point, for example) and the action of any optimizing compiler used in the build process.

RESULTS

Detailed results are presented here for two popular public domain programs, Mozilla and Apache, and one demo program provided by Nu-Mega that demonstrates their BoundsChecker run-time error detection tool. These results are similar to results observed with commercial code. Mozilla is the public domain version of Netscape's web browser, available at www.mozilla.org. The version analyzed here is 1.0, dated 6 April 1998. Apache is a popular Web server, available at www.apache.org. The version

Table I. Performance on sample public domain software.

Program	Language	Number of files	Number of lines	PREfix parse time	PREfix simulation time
Mozilla	C++	603	540 613	2 h 28 min	8 h 27 min
Apache	C	69	48 393	6 min	9 min
GDI Demo	C	9	2655	1 s	15 s

Table II. Warnings reported in sample public domain software.

Warning	Mozilla	Apache	GDI
Using uninitialized memory	26.14%	45%	69%
Dereferencing uninitialized pointer	1.73%	0	0
Dereferencing NULL pointer	58.93%	50%	15%
Dereferencing invalid pointer	0	5%	0
Dereferencing pointer to freed memory	1.98%	0	0
Leaking memory	9.75%	0	0
Leaking a resource (such as a file)	0.09%	0	8%
Returning pointer to local stack variable	0.52%	0	0
Returning pointer to freed memory	0.09%	0	0
Resource in invalid state	0	0	8%
Illegal value passed to function	0.43%	0	0
Divide by zero	0.35%	0	0
Total number of warnings	1159	20	13

tested here is 1.3 beta 5, dated February 1998. The GDI demo program is based on a sample program distributed with Microsoft Visual C++, but modified by Nu-Mega to seed several defects. Interestingly, PREfix identified several memory and resource defects in addition to the ones that were seeded (due to copyright restrictions these additional defects are not presented here).

Table I shows the size of the programs and the performance of PREfix analyzing them (on a 266 MHz Pentium II with 96 MB of memory). For comparison, the build time (without PREfix) for each program is roughly equivalent to the PREfix parse time. The line count is from the standard Unix wc tool.

In the current PREfix implementation, the time to parse and simulate is typically between 2 and 5 times the build time; ratios of 2–4 are typical for C code, 3–5 for C++ code. Certain high-level C++ constructs such as templates increase simulation complexity and hence simulation time.

Table II summarizes the warnings reported. On these examples, the number of warnings identified by PREfix ranges from 1 per 200 LOC (lines of code) to 1 per 2000 LOC (or 0.5 per KLOC (1000 lines of code) to 5 per KLOC). In commercial code, PREfix warning rates per KLOC have ranged from 0.2 to 10.

Table III. Relationships between available Models, coverage, execution time and defects reported.

Model set	Execution time (minutes)	Statement coverage	Branch coverage	Predicate coverage	Total warning count	Using uninit memory	NULL pointer deref	Memory leak
None	12	90.1%	87.8%	83.9%	15	2	11	0
System	13	88.9%	86.3%	82.1%	25	6	12	7
System & auto	23	73.1%	73.1%	68.6%	248	110	24	124

PREfix occasionally incorrectly reports a warning. In these examples, the percentage of such messages ranges from under 10% (GDI demo, Apache) to roughly 25% (Mozilla). All of these messages were due to bugs or implementation restrictions in PREfix, such as incorrect models for system functions and approximations in PREfix analysis. As with warning rates, these percentages are typical of commercial code.

Further experiments were performed with Apache, investigating the relationships between coverage, execution time, modeling strategy and defects reported.

To determine the effects of model availability on warnings, coverage and execution time, Apache was simulated with (i) no models at all, (ii) only system models (libc, etc.), and (iii) system and auto-generated models. (PREfix tracks statement, branch and predicate coverage as it simulates.) See Table III for the results.

The decrease in coverage observable here is caused by two factors. First, models, with their multiple outcomes, cause additional paths to be traced through calling functions. Second, in some cases dead code is revealed when models are used, such as code that checks for (and recovers) return values that never occur in practice.

Table III illustrates that more than 90% of the problems found in practice (with a complete model set) are due to cross-function interactions (that is, they involve models). In the case of leaks, without any models (specifically no models of `malloc` and `free`), no leaks can be detected because there are no memory allocations. Using only system models (without any modeling of user code), the only leaks that are detected are situations involving direct calls to `malloc` and `free`. For uninitialized memory warnings, the much lower number of defects detected without modeling user code is due to the conservative assumptions that must be made for missing models, with respect to both output parameters and fields in structures.

The limit on the maximum number of paths traced through each function was varied to determine the incremental benefit of more simulation. The results are presented in Figure 9. Execution times are in minutes, and the warning counts are totals reported for all of Apache.

Simulation effectiveness is clearly asymptotic. Relative to the maximum path limit, 1% of the paths gets roughly 1/3 of the warnings; 10% gets 3/4, and 25% gets 9/10. This also indicates that the heuristic used for choosing paths at choice points is effective.

The occasional non-monotonicity in the numbers is due to subtleties in the current path selection algorithm when applied to multiple functions. The current approach attempts to optimize branch and statement coverage within each individual function. On occasion, this can decrease overall cross-

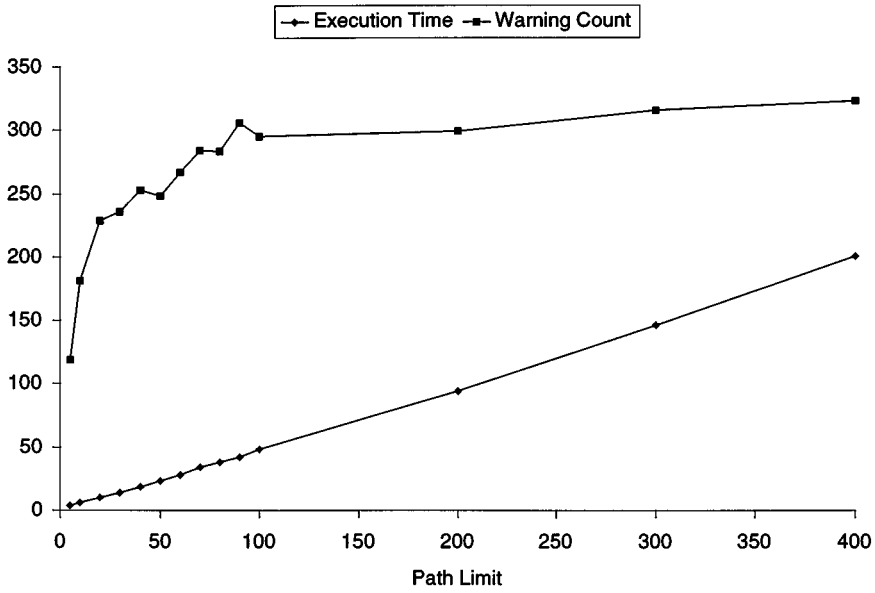

Figure 9. Relationships between path limits, execution time and defects reported.

function coverage or the total number of warnings. Better algorithms for improving coverage are a possible area of future research, but the current implementation has worked well in practice.

The relationships between path limits and types of coverage are investigated in Figure 10.

As one would expect, coverage is also asymptotic. Again, relative to the maximum path limit, 1% of the paths gets roughly 1/2 of the functions with complete coverage; 10% gets 5/6, and 25% gets 9/10. Also as expected, statement coverage is the greatest and complete path coverage is the least.

CONCLUSIONS AND OBSERVATIONS

In general, static analysis answers 'for all' questions, and must be correspondingly complete, while dynamic analysis answers 'there exists' questions, usually based on specific environmental conditions. Although not complete, dynamic analysis must be reasonably thorough: it must be thorough enough in practice to be usable, without being too slow.

In this taxonomy the simulation technique used in PREfix is a dynamic technique, finding specific error-causing conditions. Complexity is managed by using adjustable thresholds on path coverage, merging equivalent outcomes in models of called functions, employing heuristics to choose paths well, and using lazy techniques to trim paths only when necessary. In practice this has allowed PREfix to find obscure defects in large programs efficiently.

Some qualitative observations can be made.

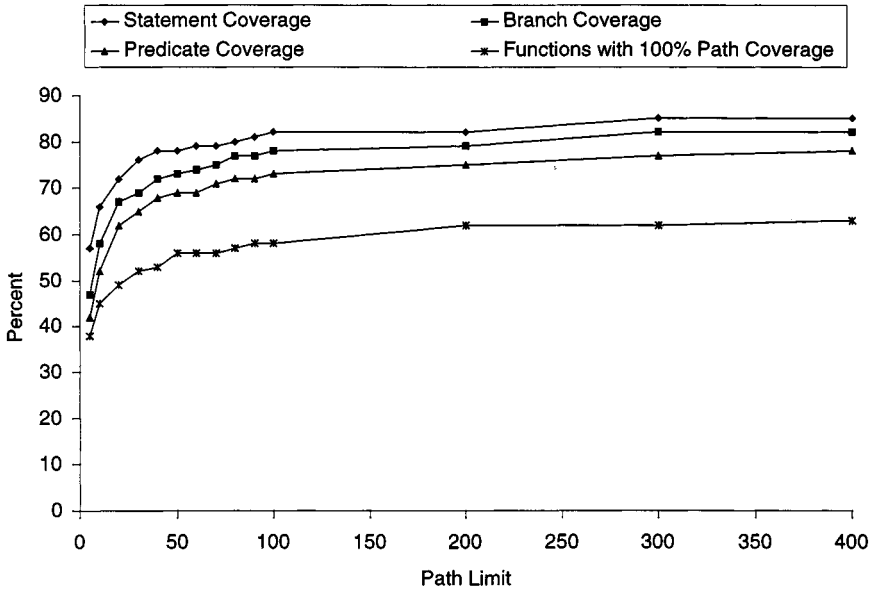

Figure 10. Relationships between path limits and types of coverage.

- The vast majority of defects identified by PREfix are due to the interaction between multiple functions; in many cases, the cause appears to be mismatched assumptions between multiple programmers working on the same project.
- PREfix tends to find errors off main code paths. The main paths are usually reasonably well designed and tested. Error handling and recovery code, on the other hand, is not.
- As predicted by the software engineering literature, there is often an inverse correlation between age of code and error density at any given site; older code has received more testing, and so usually tends to be more reliable.
- Different coding styles and disciplines result in greatly different error rates, particularly with respect to NULL pointers (especially from `malloc`) and leaks. For example, the typical Windows C coding convention of using gotos and cleanup blocks to do error recovery tends to reduce the number of leaks at the expense of increasing the number of uninitialized or NULL memory references.

One interesting result from this work is that PREfix typically finds more defects than the software engineering literature would lead one to expect for a given size and type of application. However, PREfix defects are not necessarily directly comparable to defect counts discussed in the literature, because PREfix sometimes reports several defects for a single underlying cause.

When PREfix is used on a large code base it often reports a large number of defects, many of which arise under complex conditions. Experience with large programs made it clear that tools were needed

to help the user with PREfix output. Tools were developed that present to the user the execution path that leads to a defect, and that help the user sort defects by such categories as importance (priority) and underlying cause.

In practice, 'noise' warnings are a far more significant problem than spurious messages. Noise warnings are technically correct, but for various reasons, the user does not care about them (for example, many developers do not care about leaking memory during a catastrophic shutdown of an application). These warnings are dealt with via sorting and filtering tools.

In summary, the PREfix simulator is useful and mature. Companies are using PREfix on large bodies of code on a daily basis. Continuing work will focus on:

- improvement of the presentation, sorting and filtering tools;
- additional classes of defects, such as race and deadlock conditions in multi-threaded code; and
- applying the technology to additional languages.

APPENDIX: THE MODELING LANGUAGE

This Appendix presents details on models and the PREfix modeling language, amplifying the modeling concepts described above.

Operations and return types

The modeling language supports a variety of operations that can be used in defining constraints, guards and results. The complete list of operations is presented here.

The operations on memory that can be specified in constraints and guards are:

- `memory_initialized`: requires that a variable must be initialized;
- `memory_valid_pointer`: requires that a variable be a valid pointer;
- `memory_valid_offset`: requires that a variable be a valid offset into a structure or array;
- `memory_freeable`: requires that a variable have memory that can be freed;
- `pointer_comparable`: requires that two pointers point into the same chunk of allocated memory; and
- `int_non_zero`: requires that a variable have a non-zero value.

The operations on memory that can be specified in results are:

- `memory_new`: allocates new memory;
- `memory_new_stack`: allocates new memory on the stack;
- `memory_freed`: frees allocated memory;
- `memory_initialized`: asserts that a variable is initialized;
- `memory_valid_pointer`: asserts that a variable is a valid pointer;
- `memory_valid_offset`: asserts that a variable is a valid offset;
- `pointer_comparable`: asserts that two pointers point into the same chunk of allocated memory; and
- `int_non_zero`: asserts that a variable has a non-zero value.


```
[1]     (malloc
[2]         (param size)
[3]         (normal return_X "memory allocated"
[4]             (constraint ine size 0)
[5]             (result memory_new return size) )
[6]         (error return_0 "memory exhausted"
[7]             (constraint ine size 0)
[8]             (result peq return NULL) ) )
```

Figure 11. The model for `malloc`.

The comparison operations, which can appear in constraints, guards and results, are:

- signed integer: `ieq`, `ine`, `ilt`, `ile`, `igt`, `ige`;
- unsigned integer: `ueq`, `une`, `ult`, `ule`, `ugt`, `uge`;
- floating point: `feq`, `fne`, `flt`, `fle`, `fgt`, `fge`; and
- pointer: `peq`, `peq_addr`, `pne`, `plt`, `ple`, `pgt`, `pge`.

As noted above, result information is duplicated in the outcome headers as an optimization. The possible return types are:

- the function returns zero;
- the function returns one;
- the function returns minus one;
- the function returns a value not restricted to one of the above three;
- the return value is unknown;
- the function does not return a value (for void functions);
- the function longjumps; and
- the function exits.

These types were chosen because they represent reasonable trade-offs between speed (few types) and completeness of information (many types).

A manual model of a system function

Consider the model of the `malloc` function (`void *malloc(size_t size)`) in Figure 11. This model was written by hand, based on the `malloc` function presented in [16]. The function has two outcomes: a `normal` outcome, with memory allocated, on lines 3–6, and an `error` outcome, returning NULL, on lines 6–8. The `error` outcome returns NULL (return type `return_0`), with the human-readable description `"memory exhausted"`. The `normal` outcome returns a pointer to a chunk of allocated memory of the requested size (return type `return_X`, for unrestricted value), with the description `"memory allocated"`. The `memory_new` operation takes two arguments,


```
void *wrapper (int size)
{
   return malloc(size);
}
```

Figure 12. A `malloc` wrapper function.

```
[1]     (wrapper "wrapper.c" 1 "03/20/98 11:49:42"
[2]         (param size)
[3]         (alternate return_0
[4]             (constraint ine size@<4:0> 0)
[5]             (result ieq return<4:0> 0) )
[6]         (alternate return_X
[7]             (constraint ine size@<4:0> 0)
[8]             (result memory_new return<4:0> size@<4:0>)
[9]             (result memory_valid_pointer return<4:0>) ) )
```

Figure 13. The model of the `malloc` wrapper function.

the variable to which to assign the resulting memory pointer (`return` in this case), and the size of the chunk of memory to allocate (given by the variable `size` here).

Manual models versus automatically generated models

Contrast the manual model above with the code in Figure 12, a wrapper function around `malloc`, and its automatically generated model in Figure 13.

In general, this model simply propagates the constraints and results of the handwritten model, which was used during the analysis of the wrapper function. There are, however, noteworthy differences.

- Lines 3 and 6 use the keyword `alternate` to identify the outcome, rather than `normal` and `error`. The `normal` and `error` keywords help human model writers identify outcomes, but they do not affect analysis. When generating a model the analyzer does not know the significance of a particular outcome, and simply all labels `alternate`. Similarly, the reasons for the outcomes (`"memory exhausted"` and `"memory allocated"`) are purely human documentation, meaningless to the analyzer.
- The variables in lines 4–5 and 7–9 have parameters. These parameters indicate the byte size of the variable (4 in all cases here), and the offset of the variable in a containing structure or array (0 here because there are no containing structures or arrays). This detail is unnecessary for this

model (again, it wouldn't be if there were structures or arrays), but the analyzer puts all such information in all models.

- The `size` variable in the operations on lines 4, 7 and 8 is followed by an `@` sign. The `@` sign indicates that the value of `size` to be used is its input value, that is, its value on entry to `malloc`. In contrast, the `return` parameter in both results does not have an `@` sign, meaning that the output value should be used. Here the distinction is unnecessary: `size` is not changed by `malloc`, and `return` can only have an output value. The distinction is important in some cases, however, and is always indicated in automatically generated models.
- The model is automatically documented on line 1 in terms of the source file it was derived from, the source line the function definition began on, and the date and time the model was generated.

Resources and resource operations

A resource is an object that has a state, such as a file (which can be in the open or closed state). Operations are used to define constraints and results on resource objects. Functions change the state of such objects (`fclose`, for example, changes the state of a file from open to closed), and resource operations enable the modeling of such behavior. For example, the model of the `fopen` function creates a FILE resource in the open state using the `resource_new` operation:

```
(result resource_new *return (FILE open))
```

The model for fclose requires that its file input parameter be `open`:

```
(constraint resource_state *stream (FILE open))
```

and changes the state of the file to `closed`:

```
(result resource_state *stream (FILE closed))
```

At the lowest level of modeling, resource operations must be added by hand. The analyzer cannot tell, for instance, by analyzing the code for `fopen`, that a file resource needs to be created (files are fundamentally a higher level, abstract concept). But the analyzer will propagate resource operations up to callers of functions that use them; it will, for example, propagate the `resource_new` operation in `fopen` up into models of functions that call `fopen`.

The constraint and guard operations on resources are:

- `resource_state`: requires that the given resource be in the given state; and
- `resource_not_state`: requires that the given resource not be in the given state.

The result operations on resources are:

- `resource_new`: creates the given resource and sets it to the given state; and
- `resource_state`: sets the given resource to the given state.


```
/* void *memcpy(void *s1, const void *s2, size_t n); */
(memcpy
        (param s1 s2 n)
    (normal
        (constraint memory_valid_pointer s1)
        (constraint memory_valid_pointer s2)
        (constraint memory_initialized *s2)
        (constraint memory_initialized n)
        (result memory_initialized *s1)
        (result peq return s1)
        )
     )
```

Figure 14. The model for `memcpy`.

```
/* char *strcpy(char *s1, const char *s2); */
(strcpy
        (param s1 s2)
    (normal
        (constraint memory_valid_pointer s1)
        (constraint memory_valid_pointer s2)
        (constraint memory_initialized *s2)
        (result memory_initialized *s1)
        (result peq return s1)
        )
     )
```

Figure 15. The model for `strcpy`.

Limitations

As mentioned above, modeling has been driven by pragmatic concerns. The language only supports relational constraints, the construction of sets of bits, the points to operator, and successions of dereferences and offsets (which is how naming works and how structures are modeled).

There are obvious limitations of expressiveness in the language as it now stands. Consider for example the models for the system functions `memcpy` (Figure 14) and `strcpy` (Figure 15) [16]. Both functions copy strings of bytes, but `memcpy` uses a count parameter to determine the length of the string to be copied, whereas `strcpy` uses a NULL terminated string.

The only information that can be expressed about the strings is that the length of the source string in `memcpy` should be defined. A number of characteristics cannot be expressed.

- The source string in `strcpy` should be NULL terminated.
- The destination string in `strcpy` will be NULL terminated.

- The lengths of the source and destination strings will be the same length.
- The memory for the source and destination strings should not overlap.
- The length of the source and destination strings combined should not be greater than the size of available memory.

These characteristics are typical of those missing from the modeling language. It would be possible to augment the language to support them, but so far it has not been cost effective to do so. Since there is no theory that, for completeness, requires a fixed set of features, the language has been expanded in an ad hoc fashion to cover significant conditions encountered in real code. Further development of the modeling language is anticipated as needed.

The syntax of the modeling language

The complete model language syntax is given below, in modified BNF. Note that:

- items in square brackets [] are optional;
- items in braces {} can appear one or more times;
- a <NAME> is a symbol; a <NUMBER> is an unsigned integer; a <STRING> is a C string constant (such as "PREfix");
- items with a name, number or string in their names (such as <VARIABLE-NAME>, <LINE-NUMBER>, and <DESCRIPTIVE-STRING>) are names, numbers and strings respectively; and
- comments (in the C form of `/*...*/`) are supported.

The syntax is:

```
<MODEL-SPECIFICATION> ::= ( <ROUTINE-HEADER> { <CASE> } )
<ROUTINE-HEADER> ::=
  <ROUTINE-NAME> <DOCUMENTATION> ] [ <PARAMS> ] [ <GLOBALS> ] [ <STATICS> ] [ <TEMPORARIES> ]
<DOCUMENTATION> ::= [ <FILE-STRING> [ <LINE-NUMBER> [ <DESCRIPTIVE-STRING> ] ] ]
<PARAMS> ::= ( param { <VARIABLE-DEFINITION> } )
<GLOBALS> ::= ( global { <VARIABLE-DEFINITION> } )
<STATICS> ::= ( static { <VARIABLE-DEFINITION> } )
<TEMPORARIES> ::= ( temp { <VARIABLE-DEFINITION> } )
<VARIABLE-DEFINITION> ::= <EXTERN-NAME> [ <BYTE-FIELD> ]
<CASE> ::= ( <CASE-HEADER> [ { <OPERATION> } ] )
<CASE-HEADER> ::= <CASE-TYPE> [ <RETURN-TYPE> ] [ <REASON-STRING> ]
<CASE-TYPE> ::= normal | error | alternate
<RETURN-TYPE> ::=
   return_0 | return_1 | return_minus1 | return_X | return_void | return_type_unknown
  | longjmp | exit
<OPERATION> ::= ( <OPERATION-TYPE> <OPERATOR> <OPERAND> [ <OPERAND> ] )
<OPERATION-TYPE> ::= guard | constraint | result
<OPERATOR> ::=
    memory_new | memory_new_stack | memory_freed | memory_freeable
  | memory_initialized | memory_valid_pointer | memory_invalid_pointer | memory_valid_offset
```


| resource_new | resource_state | resource_not_state
| int_non_zero | ieq | ine | ilt | ile | igt | ige | ueq | une | ult | ule | ugt | uge
| float_non_zero | feq | fne | flt | fle | fgt | fge
| pointer_comparable | pointer_incomparable | peq | pne | plt | ple | pgt | pge
<OPERAND> ::= <VARIABLE> | <CONSTANT> | <RESOURCE-SPECIFICATION>
<VARIABLE> ::= [{ <DEREFERENCE> }] <VARIABLE-NAME> [<BYTE-FIELD>]
<VARIABLE-NAME> ::= <NAME> | <NAME> @
<DEREFERENCE> ::= * [<BYTE-FIELD>]
<BYTE-FIELD> ::= < <SIZE> > | < : <OFFSET> > | < <SIZE> : <OFFSET> >
<SIZE> ::= <NUMBER-OF-BYTES> | all
<OFFSET> ::= <NUMBER-OF-BYTES>
<CONSTANT> ::= <NUMBER> | - <NUMBER>
<RESOURCE-SPECIFICATION> ::= (<RESOURCE-TYPE-NAME> <RESOURCE-STATE-NAME>)

Semantic notes are as follows.

- All parameters must be listed in the parameter list in the order in which they appear in the modeled function.
- A variable name with a trailing @ indicates an input value.
- All guards in a case are implicitly anded (&&) together.
- All constraints in a case are implicitly anded (&&) together. Thus

```
(constraint ige n 0)
(constraint ile n 255)
```

  requires that `0<=n<=255`.
- All results in a case are implicitly anded (&&) together. Thus

```
(result ige n 0)
(result ile n 255)
```

  states that `0<=n<=255`.
- Note that

```
(constraint memory_valid_pointer V)
```

  implies, in order:

```
(constraint memory_initialized V)
(constraint ine V 0)
```

  (where V is a variable, a sized variable, a dereference, or '`...`').
- Note that

```
(constraint <relational-operator> V <argument>)
```

  implies:

```
(constraint memory_initialized V)
```

- Note that `resource_new` creates a resource, and that resource is associated with the given variable. The resource is separate from the variable's normal value. For example, a file descriptor `fd` is an `int` and is laid out as such. It also has an associated resource with a state. This resource is tested and set with the resource primitives, which operate on the associated object.
- Note that resource types and states have no special meaning to the analyzer. They can be arbitrary names. The analyzer has no inherent knowledge of files, for example, or the states files can be in.
- There are a few variable names that have special meaning. Variables cannot have these names:
- return (for the return value);
- ... (for the remaining parameters in a function);
- NULL;
- EOF;
- default.

ACKNOWLEDGEMENTS

We would like to thank Mario Wolczko and the anonymous reviewers for their careful reading and helpful suggestions.

REFERENCES

1. Wilson RP, Lam MS. Efficient context-sensitive pointer analysis for C programs. *SIGPLAN Conference on Programming Language Design and Implementation (PLDI 95)*, La Jolla, CA, 18–21 June; 1–12.
2. Dor N, Rodeh M, Sagiv M. Detecting memory errors via static pointer analysis. *SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 98)*, June 1998; 27–34.
3. Johnson SC. Lint, A C program checker. *Unix Programmer's Manual, 4.2 Berkeley Software Distribution Supplementary Documents*; U.C. Berkeley Computer Science Division, Berkeley, CA, 1984.
4. Evans D. Static detection of dynamic memory errors. *SIGPLAN Conference on Programming Language Design and Implementation (PLDI 96)*, Philadelphia, PA, 21–24 May 1996; 44–53.
5. http://www.sds.lcs.mit.edu/lclint/.
6. Jackson D. Aspect: detecting bugs with abstract dependencies. *ACM Transactions on Software Engineering and Methodology* 1995; **4**(2):109–145.
7. Detlefs DL. An overview of the extended static checking system. *SIGSOFT Proceedings of the First Workshop on Formal Methods in Software Practice*, San Diego, CA, 1 January 1996; 1–9.
8. Detlefs DL, Leino KRM, Nelson G, Saxe JB. Extended static checking. *COMPAQ Systems Research Center Research Report 159*, 1998.
9. http://www.research.digital.com/SRC/esc/Esc.html.
10. Jensen JL, Jørgensen ME, Klarlund N, Schwartzbach MI. Automatic verification of pointer programs using monadic second-order logic. *SIGPLAN Conference on Programming Language Design and Implementation (PLDI 97)*, Las Vegas, NV, 16–18 June 1997; 226–234.
11. Abramsky S, Hankin C. An introduction to abstract interpretation. *Abstract Interpretation of Decarative Languages*, Abramsky S, Hankin C (eds.); Ellis Horwood, 1987.
12. Jones N, Nielson F. Abstract interpretation: a semantics-based tool for program analysis. *Handbook of Logic in Computer Science*; Oxford University Press, 1994; 527–562.
http://www.diku.dk/research-groups/topps/bibliography/1994.html#D-58.
13. *Purify User's Guide*; Pure Software Inc.: Sunnyvale CA, 1994.
14. Austin TM, Breach SE, Sohi GS. Efficient detection of all pointer and array access errors. *SIGPLAN Conference on Programming Language Design and Implementation (PLDI 94)*, Orlando, FL, 20–24 June 1994; 290–301.
15. Boyer RS, Elspas B, Levitt KN. SELECT—a formal system for testing and debugging programs by symbolic execution. *Proceedings of the International Conference on Reliable Software*, Los Angeles, CA, 21–23 April 1975; 234–245.
16. Plaugher PJ. *The Standard C Library*; Prentice Hall, 1992.

