Righting Software

James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fähndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy,
Microsoft Research

What tools do you use to develop and debug software? Most of us rely on a full-screen editor to write code, a compiler to translate it, a source-level debugger to correct it, and a source-code control system to archive and share it. These tools originated in the 1970s, when the change from batch to interactive programming stimulated the development of innovative languages, tools, environments, and other utilities we take for granted.

Three decades later, these are still the primary tools developers use to write software. Although they've been refined, the tools have neither progressed to meet the challenge of complex software nor evolved to exploit faster computers. Today, tools from an era of computational scarcity run on machines four orders of magnitude faster. Developers are struggling to write, understand, and manipulate large, complex software, while vast computational resources sit idle beneath their desks.

Microsoft Research has developed two generations of tools, some of which Microsoft developers already use to find and correct bugs. These *correctness tools* help close the gap that separates a programmer's intent—which can often be concisely stated—from the vast amount of code required to realize that goal. A developer's job is to bridge this chasm; the job of correctness tools is to ensure that the resulting span is straight, level, and connects the right points.

We don't believe that better programming tools—or faster computers—will turn software development into a routine job. Programming is a difficult intellectual task that requires talented people to apply sustained and focused effort. However, just as mechanical devices can unleash creative potential by amplifying physical effort, programming tools can improve software development by helping developers manage details, find inconsistencies, and ensure uniform quality.

Correctness tools can improve software development by systematically detecting programming errors. Microsoft Research has developed two generations of these tools that help programmers find and fix errors early in the development process.

Correctness tools

Program errors detectable by tools fall into two broad categories:

 0740-7459/04/$20.00 © 2004 IEEE

- Errors in using a programming language feature, such as referencing an uninitialized variable. Tools such as compilers and the Lint program[1] find these errors (see the "Related Work" sidebar).
- Errors in API usage, such as closing a file descriptor twice. These errors are more challenging than language errors because they're API specific. A vast number of APIs exist, some public and widely used and others private and used only by one program. As a practical matter, tools for these errors must be extensible, so that programmers can specify new usage rules, without aid from a tool's developers.

Errors in both categories are typically found through *static program analysis*, which explores possible program executions without actually executing the program. The alternative, testing, detects errors when a program executes. The two approaches are complementary. Static analysis can explore all possible execution paths, so it can find an error regardless of input data. However, it is less precise than testing and sometimes identifies spurious errors. Moreover, efficient analyses exist for only a few relatively simple program properties. For now and the foreseeable future, testing is the primary technique to ensure that a program functions correctly and produces the correct answer. Static analysis, however, is useful in that it can find errors that do not manifest during testing or errors that exist on paths that testing doesn't cover.

Underlying program analysis is Turing's halting problem, which shows the general impossibility of deciding whether a program will execute an erroneous action. Consequently, tools approximate a program's behavior, which leads to a fundamental tradeoff between soundness and completeness. A *sound* program model ensures that every program error appears—sometimes including spurious errors that are artifacts of analysis. A *complete* program model ensures that each error in the model is an error in the program, and so none are spurious errors. Given the halting problem, a sound and complete tool that answers nontrivial program behavior questions cannot exist, so tool designers must choose between one or the other. Most choose sound analysis and tolerate the false error reports, although many useful tools have been constructed from heuristics that are neither sound nor complete.

Related Work

Many static analysis tools exist for finding bugs in programs. These range from heuristic tools, such as the Unix utility Lint[1] and its more modern successors (such as LCLint[2] and Engler's Metal[3]), to tools based on sound program analysis, such as theorem proving,[4] model checking,[5] type theory,[6] and abstract interpretation.[7]

Microsoft's original error-detecting tool, PREfix, differs from other heuristic tools in that it can perform interprocedural analysis of ten-million-line programs, it can handle the full complexity of C++, and it has extensive error-filtering features. Microsoft's subsequent correctness tools have pioneered new techniques, such as software model checking, to soundly analyze programs of commercial size and complexity.

References

1. S.C. Johnson, "Lint, a C Program Checker," *Unix Programmer's Manual*, computer science tech. report 65, AT&T Bell Laboratories, 1978.
2. D. Evans et al., "LCLint: A Tool for Using Specifications to Check Code," *Proc. ACM SIGSOFT 2nd Symp. Foundations of Software Eng.*, ACM Press, 1994, pp. 87–96.
3. D. Engler et al., "Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions," *Proc. 4th Symp. OS Design and Int'l* (OSDI 2000), ACM Press, 2000, pp. 1–16.
4. C. Flanagan et al., "Extended Static Checking for Java," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, ACM Press, 2002, pp. 234–245.
5. J. Hatcliff and M. Dwyer, "Using the Bandera Tool Set to Model-Check Properties of Concurrent Java Software," *Proc. 12th Int'l Conf. Concurrency Theory* (CONCUR 2001), LNCS 2154, Springer-Verlag, 2001, pp. 29–58.
6. T. Jim et al., "Cyclone: A Safe Dialect of C," *Proc. Usenix Ann. Conf.,* Usenix Assoc., 2002, pp. 275–288.
7. B. Blanchet et al., "A Static Analyzer for Large Safety-Critical Software," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 03), ACM Press, 2003, pp. 196–207.

To make this point more concrete, consider the following code:

```
FILE* f;
if (complex_calc1())
     f = fopen(...);
...
if (complex_calc2())
     fclose(f);
```

The code executes correctly only if the conditional statements' predicates, `complex_calc1` and `complex_calc2`, produce identical results. A sound model would presume each conditional could execute either way, so each file operation could execute without its mate. Although sometimes wrong, this approach catches the error when the two functions are uncorrelated. By contrast, a complete analysis would report an error only when it could determine the relationship between two predicates—say, one was `x < 0` and the other was

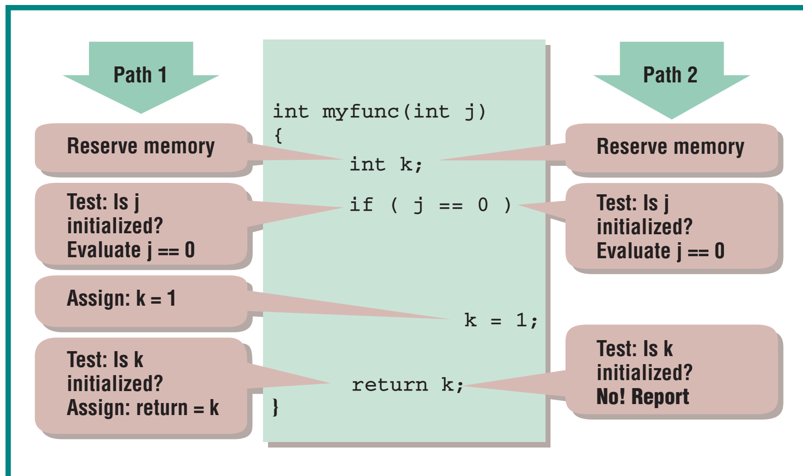

Figure 1. The PREfix tool. PREfix traces two paths through this function, produces a model, and analyzes it for errors.

`x > 0`. When the relationship was unknown, the analysis would report nothing.

In practice, if a tool's goal is to find errors, it need not be sound or complete. Given a choice, soundness is more attractive because it catches all errors, and developers can handle false reports using filtering and sorting techniques. Also, only sound tools can conclusively report that an error cannot occur.

At Microsoft, we divide our tools into two generations. The first-generation tools, PREfix and PREfast, use heuristic and unsound analysis but are robust, production-quality tools. Most of our development groups routinely apply these tools to their code. Our second-generation tools are a product of research into sound program analysis and are only now entering active use. Although these tools focus on API usage errors, they explore a variety of precise and scalable program analysis techniques.

Microsoft's first-generation tools

PREfix and PREfast take complementary approaches to finding errors. PREfix performs a detailed, path-by-path analysis of a complete program to track information across function boundaries. In contrast, PREfast is a lightweight tool that looks for errors locally. It parses individual functions and provides an infrastructure through which custom plug-ins can traverse parse trees to identify and report problematic idioms.

Both tools have proven very effective at finding bugs and are routinely applied to most Microsoft products. Combined, the tools found 12.5 percent of the bugs fixed in Windows Server 2003, which is a significant number given the limited range of errors they detect.

PREfix

PREfix finds a fixed set of bugs in C and C++ programs.[2] Such bugs include null pointer references, improper memory allocation and deallocation, use of uninitialized values, simple resource state errors, and improper library use.

PREfix traverses a program's call graph, analyzing all functions in a bottom-up (leaves to root) fashion. When it analyzes a function, PREfix has already seen routines invoked by the function and it uses models constructed earlier to approximate the called code's effects. Consider the leaf function in Figure 1, for example. PREfix knows nothing about parameter `j` because the program could invoke function `myfunc` at many places. PREfix thus traces the two paths through the body and reports an undefined variable reference along one path. Then, considering both paths, it produces a model recording that the parameter has been referenced and the return value could be uninitialized. PREfix uses this model at calls to `myfunc` to ensure that when the actual argument is 0, it warns about the possibility of referencing an undefined result.

PREfix's heuristic analysis is neither sound nor complete. It makes simplifying approximations to make analysis tractable for large code bases with incomplete information.[2] For example, PREfix examines a fixed number of paths (typically 100) through a function. When the paths are carefully chosen, PREfix can find many errors that occur along multiple paths, though it can't guarantee the absence of a particular error. PREfix makes possibly unsound assumptions about missing functions to reduce spurious errors, at the cost of missing real errors.

PREfast

PREfast performs local analyses on C and C++ programs to find idioms associated with programming mistakes. Developers can easily extend its functionality with plug-ins to augment its library of suspicious patterns. PREfast uses the Microsoft Visual C++ compiler's parser to produce abstract syntax trees for each C++ language construct. PREfast parses a function and then invokes the plug-ins, each of which traverses the function's parse tree looking for idioms that might indicate a misunderstanding or error.

For example, in the code

```
extern void my_wcsncpy(wchar_t *,
  wchar_t *, size_t);
wchar_t pPtr1[5];
my_wcsncpy(pPtr1, input,
   sizeof(pPtr1));
```

PREfast identifies the bold-faced expressions as a likely error, because the function `my_wcsncpy` takes a Unicode string `(wchar_t)` as a parameter, but the last argument is the string's length in bytes, rather than the number of characters:

```
my_wcsncpy(pPtr1, input,
  sizeof(pPtr1)/sizeof(pPtr1[0]));
```

PREfast relies only on local analysis. It thus executes quickly, which facilitates adoption and encourages its regular use. Because PREfast uses the Microsoft Visual C++ compiler's parser, it can parse all source code and also provide the infrastructure that a growing community of researchers and developers is using to build more sophisticated correctness tools.

Microsoft's second-generation tools

Although PREfix and PREfast are successful, they find a limited set of errors and are constrained by unsound analysis. Microsoft Research therefore developed technology to improve our correctness tools in two important ways. First, the new tools start with declarative descriptions of correct and incorrect program behavior and thus can be extended to new errors by writing new rules for APIs. Second, the tools use sound program analysis that does not miss actual errors. As a result, this generation of tools can certify that a particular error does not occur in a program.

Slam

Incorrect event ordering frequently causes errors. For example, code that acquires a lock often fails to release it along all execution paths. Sequencing rules are naturally described by *finite state machines* that accept legal program-event sequences and reject illegal ones. Through their interfaces, FSMs offer a precise, concise sequencing specification that a tool can compare against a program's actual behavior.

How it works. Slam starts with a C program and a usage rule, and either finds plausible program paths that violate that rule or determines that all paths respect the rule.[3] Rules are written in Simple Logic (SLIC), which expresses state machines in a C-like notation. For example, Figure 2 shows the rules governing a lock.

Slam simplifies the analyzed program by eliminating details irrelevant to the target rule. Even with complex rules, it can ignore most program code and data or abstract it to a simpler form. Slam's simplified representation is a Boolean program that contains C's control-flow constructs and only Boolean variables. Each such variable tracks a predicate's value over the C program's state. This simplification offers two benefits:

- *Termination*. Because Boolean programs have a finite number of Boolean variables in scope at any point, they are amenable to automated, terminating program analyses such as dataflow analysis and model checking.
- *Soundness*. Every API usage error in a C program is also present in its Boolean program, making Slam's analysis sound.

```
state {
   enum { Unlocked, Locked} s = Unlocked;  // FSM states
}
AcquireSpinLock.entry {        // Transition on lock acquire
   if (s == Locked) error;
   else s = Locked;
}
ReleaseSpinLock.entry {        // Transition on lock release
   if (s == Unlocked) error;
   else s = Unlocked;
}
```

Figure 2. The rules governing a lock.

```
void example() {
do {
A: AcquireSpinLock();
    nPacketsOld = nPackets;
    req = devExt->WLHV;
    if (req && req->status) {
         devExt = req->Next;
B:       ReleaseSpinLock();
         irp = req->irp;
         if (req->status > 0)
              irp->IoS.Status = S;
         else
              irp->IoS.Status = F;
         nPackets++;
         }
    } while(nPackets!=nPacketsOld);
C: ReleaseSpinLock();
}
```

(a)

```
void example() {
do {
A:  AcquireSpinLock();
    skip;
    skip;
    if (*) {
        skip;
B:      ReleaseSpinLock();
        skip;
        if (*)
          skip;
        else
          skip;
        skip;
        }
     } while (*);
C: ReleaseSpinLock();
}
```

(b)

```
void example() {
do {
A:  AcquireSpinLock();
    b := true;
    skip;
    if (*) {
        skip;
B:      ReleaseSpinLock();
        skip;
        if (*)
          skip;
        else
          skip;
        b := b ? false : *;
        }
    } while (!b);
C: ReleaseSpinLock();
}
```

(c)

Figure 3. An example Slam operation. (a) Program P, which Slam checks against the locking rule code. (b) In the first refinement, Slam generates a Boolean program. (c) Slam then refines the model to eliminate infeasible paths that produce spurious results. (A "*" indicates a nondeterministic choice.)

Another key Slam feature is that it can refine Boolean program abstractions. Because a Boolean program generally has more possible behaviors than its corresponding C program, the Boolean program might contain spurious error paths that are infeasible in the C code. Slam uses a counterexample-driven refinement technique to eliminate spurious paths in the Boolean program.

Example. To illustrate Slam's operation, we can check the C code in Figure 3a against the locking rule in Figure 2, which states that it's an error to acquire (or release) a lock twice in succession. Slam first generates the Boolean program in Figure 3b. At each C program statement, Slam determines the statement's effect on the rule predicates and encodes the effect in the Boolean program. All assignment statements are abstracted to skip because they don't affect any of the predicates.

Slam's next step determines whether there's a feasible path through the Boolean program that violates the SLIC rule. In this crude model, several paths exist, but Slam starts with the shortest one [A, A], which is a double lock acquisition. Next, Slam uses symbolic execution to determine if this path is feasible in the original C program. If feasible, Slam has found a real error. If infeasible, Slam identifies a small predicate set that "explains" the path's infeasibility. In this example, Slam determines that the path is infeasible because it requires that the predicate `(nPackets=npacketsOld)` be both true and false, and it returns the predicate `(nPackets=nPacketsOld)` to explain the infeasibility.

Slam uses this predicate to refine its model (see Figure 3c). The Boolean variable (`b`) represents the predicate `(nPackets=nPacketsOld)`. The original program's assignment statement, `nPacketsOld = nPackets;` makes this predicate true, so the corresponding Boolean program statement is `b := true;`. Slam determines that when the predicate is true before the statement `nPackets++`, it is false afterward, and when the predicate is false before, its outcome is unknown afterward. This relation is captured by the statement `b := b ? false : *`.

The refined Boolean program now contains enough information to show that all paths respect the locking rule because the spin lock is held at the loop's end if and only if `(nPackets=nPacketsOld)` is true. When this predicate is true, the lock is held, the loop exits, and the program releases the lock. When the predicate is false, the lock is not held at the loop's end, and the loop iterates.

Discussion. We used Slam to analyze more than one hundred Windows device drivers against a collection of more than 30 rules, and it found many errors. Slam is successful in this domain because it can separate the driver's control path from its data path. Most driver safety rules concern proper kernel resource use, and Slam can abstract away the driver

```
State Closed
State Opened
State Error

Creation Event Open
     { _object_ FUNCTIONCALL { SYMBOL "fopen" } { _anyargs_ } }

Event Close
     { FUNCTIONCALL { SYMBOL "fclose" } { _object_ } }

Transition _ -> Opened on Open
Transition Opened -> Closed on Close
Transition Closed -> Error on Close "File already closed"
```

Figure 4. OPAL rules for opening and closing files.

portion that shuttles data to hardware, so it can focus on checking the control path. As a result of Slam's success with device drivers, the Windows development organization has deployed the tool within Microsoft.

ESP

Error detection via scalable program analysis (ESP)[4] is similar to Slam, except that it focuses on very large C and C++ code bases. Its global analysis technique also differs from Slam's and trades precision for scalability.

ESP offers a pragmatic solution to the large-program analysis problem by combining precise analysis within a component with less precise but more scalable analysis across components. ESP ensures that the C or C++ program obeys a rules set written separately in the simple Object Property Automata Language (OPAL), which combines an FSM with syntactic code patterns. ESP is based on sound analysis, so it finds all instances of a particular problem.

OPAL rules. Consider the following code, which conditionally opens and closes two files:

```
void main ()
{
  if (dump1) /* B1 */
   fil1 = fopen(dumpFile1,"w");

  if (dump2) /* B2 */
   fil2 = fopen(dumpFile2,"w");

  if (flag)
      x = 0;
   else
      x = 1;

  if (dump1) /* B3 */
      fclose(fil1);

  if (dump2) /* B4 */
      fclose(fil2);
}
```

Code that manipulates a file must obey well-known rules, such as that calling `fclose` can close a file only if it has not been previously closed. OPAL formally states these rules as shown in Figure 4.

The OPAL specification consists of two parts. The first is an FSM consisting of three states, two events, and state transitions labeled by events. The other is a set of syntactic patterns that identify events. Stateful file handles are created by the `Open` event, which is triggered by calls to `fopen`, and closed by the `Close` event. File handles that the `Open` event creates move to the `Opened` state. OPAL is an intentionally simple, restricted-rule language; it can only express properties that we can check efficiently.

ESP analysis engine. We can explain the main idea behind ESP's analysis using the specification just presented and the OPAL example in Figure 4. One way to analyze code is to symbolically evaluate each path through a program. Along each path, ESP maintains a symbolic state that records everything inferred about program execution. In addition, the engine can update an FSM at each event along a path. In the OPAL code just presented, for example, this approach might recognize the correlation between branches B1 and B3. Unfortunately, the technique is impractically expensive, as the symbolic states can double at each branch.

Figure 5. The bold-faced annotations identify the socket value to be tracked and specify a state machine that describes the acceptable sequences of operations on values of this type.

```
interface SOCKET {
 type sock;
 variant domain [ `UNIX | `INET ];
 variant comm_style [ `STREAM | `DGRAM ];
 tracked(@raw) sock socket(domain, comm_style, int);
 struct sockaddr { ... };
 void bind(tracked(S) sock, sockaddr)          [S@raw->named];
 void listen(tracked(S) sock, int)             [S@named->listening];
 tracked(N) sock accept(tracked(S) sock, sockaddr)
                                          [S@listening, new N@ready];
 void receive(tracked(S) sock, byte[])         [S@ready];
 void close(tracked(S) sock)                   [-S];
}
```

As an alternative, we could avoid tracking paths by associating FSM states with locations in a program. This offers an efficient analysis but reports too many false errors. In our example, a tool of this sort could only report that the file might not be open at the two `fclose` statements, because it couldn't track correlations between predicates and file state.

We based ESP's algorithm on the insight that most branches are irrelevant to a particular rule. In our example, the intermediate conditional statement has no bearing on file behavior. ESP identifies and tracks relevant branches with the heuristic that a conditional statement is relevant if an FSM associated with an object moves to different states along the branch's arms. If this happens, code later in the program might again rely on the correlation between the branch and the FSM state. ESP tracks this correlation and doesn't merge information from the two arms. If, on the other hand, the FSM state is identical along both paths, ESP merges information and loses track of the branch correlation. This heuristic leads to a polynomial-time algorithm efficient enough for large programs.

Another important analysis aspect is determining whether an event triggers a change in an FSM for the target object. For instance, does `fclose(fil2)` change the state of `FILE fil1`? ESP answers these questions by performing a sophisticated alias analysis and tracking value flow through the program.

Discussion. We've used ESP to find buffer overruns in large-scale system code and to validate an OS kernel's security properties. The latter application demonstrates ESP's strength: the tool checked all execution paths in a million-line code base against more than 500 properties, requiring only a few minutes per property. ESP reported only 25 false errors.

Vault

The Vault project takes a different approach than Slam and ESP, incorporating error detection into a programming language rather than a distinct tool.[5] The Vault language is a safe version of C. Its type system lets developers record usage rules in an interface's type signatures, and its type checker ensures that client code using the interface obeys the rules. Like Slam and ESP, Vault's rules describe execution-ordering constraints.

A major difference between Vault and other tools is Vault's modular checking, which lets developers independently check a compilation unit (function). As with conventional type checking, Vault's type checking is efficient and scalable to any program size. Its drawback is that a programmer must supply more annotations. As with datatype declarations, these annotations document a developer's intent and let the checker verify properties quickly and locally. Moreover, annotations let Vault find errors early, when developers first compile their code.

Sockets example. Consider, for example, the well-known socket API, which breaks the connection-establishment process into several steps. Omitting one or more steps is a common mistake. The bold-faced code in the Vault interface describes the rules (see Figure 5).

Vault's interface associates abstract states (`raw, named, listening, ready`) with program objects to enforce the sequence of steps required to create a connection. The function `socket` creates a new socket. Its annotation, `tracked(@raw)`, tells Vault to *track* (at compile time) the returned value's state, which starts at `raw`. In the subsequent tracked annotations, the name in parentheses is a *key*, which is the compile-time name that refers to an object at a particular point. A function's effect clause expresses pre- and postconditions on an object's state. For example, the `receive` function requires that its socket be in state `ready`, and the function `bind` changes the socket from the required state `raw` to state `named`.

This interface is simplified, as it ignores the possibility of failure. In reality, function `bind` returns an error code indicating whether the operation succeeded. If `bind` is successful, the socket's new state is `name`; if it fails, it remains `raw`. We can describe this situation precisely using a slightly more elaborate signature:

```
variant status<key K>
  [ `Ok {K@named}
  | `Error(error_code) {K@raw} ];

tracked status<S>
  bind(tracked(S) sock, sockaddr)
    [-S@raw];
```

The variant type status has two constructors: `` `Ok`` for the successful case and `` `Error`` for the failure case (which includes an error code explaining the error). Both constructors accept key `K,` which identifies the socket object. In the `` `Ok`` constructor, the socket has the state `named`, whereas in the `` `Error`` case, it has state `raw`. The effect clause of the `bind` function states that the socket must be in the state `raw` on entry, but it's unavailable on exit (indicated by the "`-`" sign). The result status effectively guards access to the socket after the call. The result's variant type forces a developer to check the status returned from `bind` to regain access to the socket, which ensures that they're aware when `bind` fails.

Expressiveness. Vault goes beyond describing an object's finite-state protocols. An object can be in one of (finitely) many states. Function signatures express preconditions on object states and describe how an object's state changes as the result of a call. Vault can also express resource allocation and deallocation protocols through annotations that explicitly state when objects are created and when they become unavailable. The Vault checker ensures that unavailable objects are not referenced.

Using these mechanisms, Vault can enforce memory safety, even for programs using explicit deallocation (such as `free`) rather than garbage collection. In addition to its value as a general-purpose language, this unique aspect makes Vault valuable for writing safe, low-level system code. Even in garbage-collected languages, enforcing correct resource management offers benefits. In such languages, many resources—such as file handles, database connections, and so on—must be explicitly released by code. Vault can track and check these resources.

Checking. Vault separately checks each program function. The function's effect is translated into a precondition and a postcondition on each object parameter. The checker performs a control-flow and dataflow analysis along all function paths. At each statement, it checks any statement-specific conditions against objects' compile-time state as follows:

- After each statement, the checker verifies that no objects become inaccessible and thus leak.
- At a function call, the checker verifies that the current state satisfies the function's precondition. If so, the current state is updated using the function's postcondition.
- At control-flow join points, the checker verifies that states on incoming edges are compatible.
- Finally, at function exits, the checker verifies that the state agrees with the function's postcondition.

Checking is fast, simple, and feasible because the checker enforces strong heap invariants. If, for example, we have two tracked objects at a program point that are denoted by keys with distinct names, then the two objects are distinct as well. This nonaliasing property between tracked objects lets the checker track object states without making overly conservative assumptions about pointer aliasing.

Vault separately checks each program function. The function's effect is translated into a precondition and a postcondition on each object parameter.

About the Authors

James R. Larus is an assistant director and senior researcher at Microsoft Research and is an affiliate associate professor at the University of Washington. His research interests include programming languages, compilers, and parallel computation. He received his PhD in computer science from the University of California, Berkeley. Contact him at Microsoft Research, One Microsoft Way, Redmond, WA 98052; larus@microsoft.com.

Thomas Ball is a senior researcher and head of the Testing, Verification, and Measurement Research group at Microsoft Research. His research interests include applying program analysis and programming language technology to hard problems in software correctness and reliability. He received his PhD in computer science from the University of Wisconsin-Madison. Contact him at Microsoft Research, One Microsoft Way, Redmond, WA 98052; tball@microsoft.com.

Manuvir Das leads the Scalable Program Analysis research group at Microsoft Research and is an affiliate assistant professor at the University of Washington. His primary interests are programming languages and compiler technology, and their application to software reliability. He received his PhD in computer science from the University of Wisconsin-Madison. Contact him at Microsoft Research, One Microsoft Way, Redmond, WA 98052; manuvir@microsoft.com.

Robert DeLine works in software engineering at Microsoft Research, most recently on static tools for software verification. He received his PhD in computer science from Carnegie Mellon University. Contact him at Microsoft Research, One Microsoft Way, Redmond, WA 98052; rdeline@microsoft.com.

Manuel Fähndrich is a researcher at Microsoft Research. His research interests include program analysis and programming languages. He received his PhD in computer science from the University of California, Berkeley. Contact him at Microsoft Research, One Microsoft Way, Redmond, WA 98052; maf@microsoft.com.

Jon Pincus is a senior researcher at Microsoft Research, currently focusing on security and privacy. He has previously developed and deployed program analysis-based tools such as PREfix and PREfast and was founding CTO of Intrinsa. He received his MS in computer science from the University of California, Berkeley. Contact him at Microsoft Research, One Microsoft Way, Redmond, WA 98052; jpincus@microsoft.com.

Sriram K. Rajamani leads the Software Productivity Tools group at Microsoft Research. His research interests are in tools and methodologies for building reliable systems. He received his PhD in computer science from the University of California, Berkeley. Contact him at Microsoft Research, One Microsoft Way, Redmond, WA 98052; sriram@microsoft.com.

Ramanathan Venkatapathy is a research development manager at Microsoft Research. His interests include program analysis and parser and security tools development. He received his MS in computer science from Indiana University, Bloomington. Contact him at Microsoft Research, One Microsoft Way, Redmond, WA 98052.

Microsoft Research's correctness tools are helping our developers find and fix bugs, and we're working on new tools that systematically detect other types of bugs and are more easily extensible. Microsoft is also actively working to turn some of our correctness tools into products for the benefit of developers outside the company. Further in the future, we can envision shipping software with rules, to provide precise descriptions that can help developers understand and correctly use APIs.

At the same time, our goals for these tools are limited. Although they check important and necessary properties, they're far from sufficient to ensure program correctness. A collection of finite-state properties—no matter how rich and varied—will never fully describe the behavior of complex software.

A huge amount of work remains if we're to exploit computers' potential to improve software development. Machines are fast enough, storage is deep enough, and program analysis is well developed enough to produce far more powerful tools. A deep understanding of program semantics and behavior may one day enable tools that handle routine tasks, thereby freeing developers and testers to focus on software development's more creative aspects.

References

1. S.C. Johnson, "Lint, a C Program Checker," *Unix Programmer's Manual*, computer science tech. report 65, AT&T Bell Laboratories, 1978.
2. W.R. Bush, J.D. Pincus, and D.J. Sielaff, "A Static Analyzer for Finding Dynamic Programming Errors," *Software—Practice and Experience*, vol. 30, no. 7, 2000, pp. 775–802.
3. T. Ball and S.K. Rajamani, "The Slam Project: Debugging System Software via Static Analysis," *Proc. 29th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages* (POPL 2002), ACM Press, 2002, pp. 1–3.
4. M. Das, S. Lerner, and M. Seigle, "ESP: Path-Sensitive Program Verification in Polynomial Time," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 02), ACM Press, 2002, pp. 57–69.
5. R. DeLine and M. Fähndrich, "Enforcing High-Level Protocols in Low-Level Software," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 01), ACM Press, 2001, pp. 59–69.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.