Formal Specifications on Industrial Strength Code: From Myth to Reality

Manuvir Das

Principal Researcher
Center for Software Excellence
Microsoft Corporation

Talking the talk …

- SAL source code annotations
  - Deployed on Windows Vista and Office 12
  - Incremental approach is the key to success
- OPAL defect specifications
  - Lower cost, lower coverage option
  - Range of applicability is the key to success
- The right approach for the right problem
  - SAL: focus on a small set of critical properties
  - OPAL: apply to a wide range of quality priorities

… walking the walk

- CSE impact on Windows Vista
  - Found 100,000+ *fixed* bugs
  - Added 500,000+ specifications
  - Answered thousands of emails
- We are program analysis researchers
  - But we measure our success in *adoption*
  - And we feel the pain of the customer

Buffer overruns

- Defect: a buffer access index is out of bounds
- Detection: check that index is within bounds

- Problem: where are the buffer bounds stored?
  - Tools must track buffer size from allocation to access
  - Exhaustive global analysis is infeasible

- Solution: turn global analysis into local analysis
  - Specify buffer sizes at function interfaces
  - Perform modular (one function at a time) analysis

BO example

- Prototype of function **SetupGetStringFieldW**

```
BOOL WINAPI SetupGetStringFieldW(
                IN PINFCONTEXT Context,
                IN DWORD FieldIndex,
                OUT PWSTR ReturnBuffer,
                IN DWORD ReturnBufferSize,
                … );
```

- Body of function **CheckInfInstead**

```
…
WCHAR szPersonalFlag[20];
…
SetupGetStringFieldW(&Context,1,szPersonalFlag,50,…);
…
```

BO example

```
BOOL WINAPI SetupGetStringFieldW(
  …
  __out_ecount(ReturnBufferSize)
  OUT PWSTR ReturnBuffer,
  IN DWORD ReturnBufferSize,
  …);

WCHAR szPersonalFlag[20];
…
SetupGetStringFieldW(&Context,1,szPersonalFlag,50,NULL);
```

NT# 587620 PREfast: \nt\inetsrv\iis\setup\osrc\dllmain.cpp dllmain.cpp(112) : warning 202: Buffer overrun for stack buffer 'szPersonalFlag' in call to 'SetupGetStringFieldW': length 100 exceeds buffer size 40.

SAL example 1

- **`wcsncpy`** [precondition] destination buffer must have enough allocated space

```
wchar_t wcsncpy (
 wchar_t *dest, wchar_t *src, size_t num );

wchar_t wcsncpy (
 __pre __notnull __pre __writableTo(elementCount(num))
 wchar_t *dest,
 wchar_t *src, size_t num );

wchar_t wcsncpy (
 __out_ecount(num) wchar_t *dest,
 wchar_t *src, size_t num);
```

SAL example 2

- **memcpy**

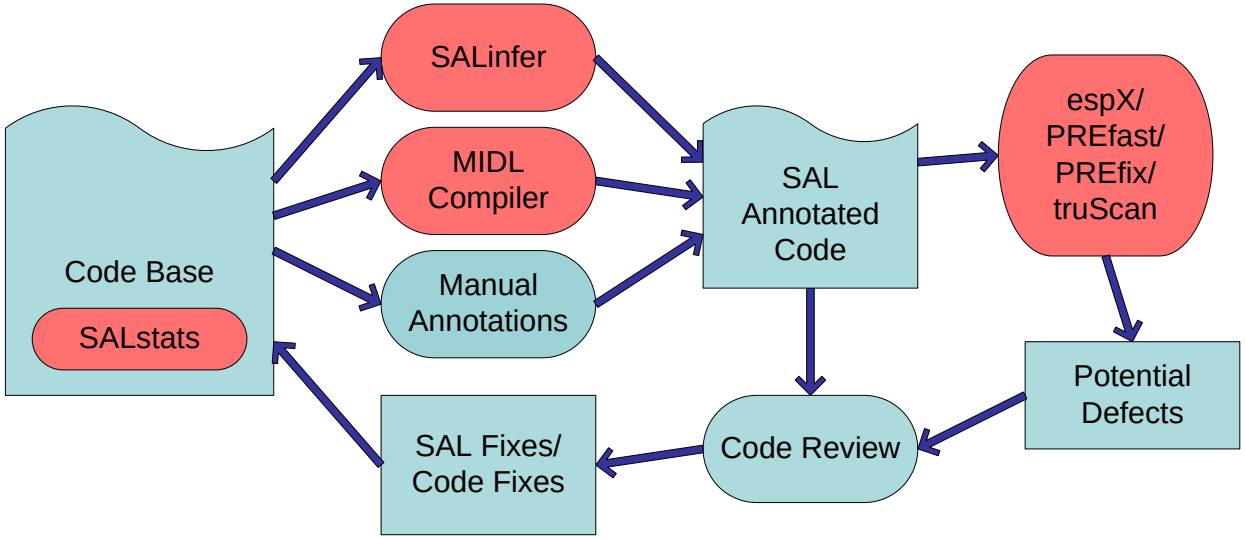
```
void * memcpy ( void * dest, void * src, size_t num );

void * memcpy (
  __pre __notnull __pre __writableTo(byteCount(num))
  __post __readableTo(byteCount(num)) void * dest,
  __pre __notnull __pre __deref __readonly
  __pre __readableTo(byteCount(num)) void * src,
  size_t num );

void * memcpy (
  __out_bcount_full(num) void * dest,
  __in_bcount(num) void * src, size_t num );
```

Standard Annotation Language

- Usage example:

  a_0 *RT func(*a_1 … a_n *T par)* a_i : SAL annotation
- Interface contracts
  - pre, post, object invariants
- Basic properties
  - null, readonly, valid, range, …
- Buffer extents
  - writableTo(size), readableTo(size)
- Buffer size formats
  - (byte|element)Count, endPointer, sentinel, …

SAL ecosystem



- espX/PREfast/… : Use annotations to find defects
- SALstats : Identify parameters that should be annotated
- MIDL Compiler : Translate MIDL directives to annotations
- SALinfer : Infer annotations using global static analysis

SALinfer example

```
void work() {
    int tmp[200];
    wrap(tmp, 200);
}

void wrap(int *buf, int len) {
    int *buf2 = buf;
    int len2 = len;
    zero(buf2, len2);
}

void zero(int *buf, int len) {
    int i;
    for(i = 0; i <= len; i++)
        buf[i] = 0;
}
```

size(tmp,200)

size(buf,len) write(buf)

size(buf2,len)

size(buf2,len2) write(buf2)

size(buf,len) write(buf)

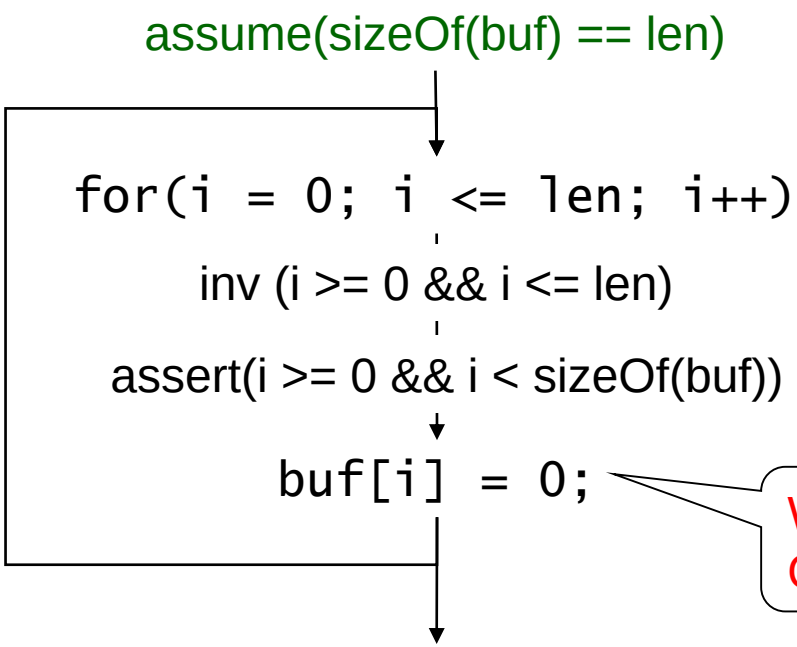
write(buf)

SALinfer example

```
void work() {
    int tmp[200];
    wrap(tmp, 200);
}

void wrap(__out_ecount(len) int *buf, int len) {
    int *buf2 = buf;
    int len2 = len;
    zero(buf2, len2);
}

void zero(__out_ecount(len) int *buf, int len) {
    int i;
    for(i = 0; i <= len; i++)
        buf[i] = 0;
}
```

espX example

```
void zero(__out_ecount(len) int *buf, int len) {
    int i;
    for(i = 0; i <= len; i++)
        buf[i] = 0;
}
```



Constraints:

(C1) i >= 0

(C2) i <= len

(C3) sizeOf(buf) == len

Goal: i >= 0 && i < sizeOf(buf)

Subgoal 1: i >=0 by (C1)

Subgoal 2: i < len FAIL

Warning: Cannot validate buffer access.
Overflow occurs when i == len

SAL impact

- Windows Vista
  - Mandate: Annotate 100,000 mutable buffers
  - Developers annotated 500,000+ parameters
  - Developers fixed 20,000+ bugs
- Office 12
  - Developers fixed 6,500+ bugs
- Visual Studio, SQL, Exchange, …
- External customers
  - CRT + Windows headers SAL annotated
  - SAL aware compiler shipped with VS 2005

SAL evaluation

Vista – mutable string buffer parameters

- Annotation cost:
  - [–] 100,000 parameters required annotations
  - [+] 4 out of 10 automatic
- Defect detection value:
  - [+] 1 buffer overrun exposed per 20 annotations
- Locked in progress:
  - [+] 9.4 out of 10 buffer accesses validated

SAL priorities

- Crashes
  - Annotate possibly-NULL pointers (SALinfer)
  - Enforce NULL pointer checking (PREfast)
- Error handling
  - Annotate failure conditions (SALinfer, typedefs)
  - Enforce error handling in callers (PREfast)
- AppCompat
  - Annotate public APIs (MaX, WINAPI macros)
  - Prohibit signature changes (SD)
- Resource usage, drivers, …

Annotations summary

- Ensure correct behavior by extending the type system with SAL annotations
  - [+] Checkers validate correct behavior
  - [–] Requires investment in annotation effort
  - [–] Requires investment in developer education
- SAL is a high cost, high return approach
  - Applicable to a small class of critical defects

OPAL – defect by example

- Problem
  - A defect is discovered through internal testing, or in the field (MSRC, Watson)
- Diagnosis
  - Identify the code pattern that caused the bug
- Detection
  - Specify the code pattern formally in OPAL
  - Use checkers to find instances of the pattern

RegKey leak defect

```
status = RegOpenKeyExW( HKEY_LOCAL_MACHINE,
   L"SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Perflib",
   0L, KEY_READ, & hLocalKey);

if (status == ERROR_SUCCESS) bLocalKey = TRUE;

… block of code that uses hLocalKey …

if (bLocalKey)
  CloseHandle(hLocalKey);
```

- Bug: registry key is closed by calling the generic CloseHandle API
  - May fail to clean up some data that is specific to registry key data structures

RegKey leak code pattern

- Search for code paths along which a registry key is opened, and then closed using the generic CloseHandle API
- Specification:
  - define a sequence of relevant actions
  - e.g. A(k)…B(h)
  - define the actions (e.g. A, B, k and h)

RegKey leak specification

```
defect RegKeyCloseHandle
{
  // A(x)…B(x)
  sequence OpenKey(key);CloseHandle(handle)
  message “Registry key closed using generic CloseHandle API!”

  // A(x)
  pattern OpenKey(key)
     /RegOpenKeyEx[AW](@\d+)?$/ (_,_,_,_,&key)
         where (return == 0)

  // B(x)
  pattern CloseHandle(handle)
     /CloseHandle(@\d+)?$/ (handle)
}
```

This is the entire specification effort for the codebase

OPAL – under the hood

- Requirements for checkers
  - Customizable analysis engine
  - Path-specific static or dynamic analysis
- Checking support for OPAL
  - Vista: ESP (global static analysis)
  - Vista: PREfast (local static analysis)
  - truScan (execution trace analysis)

OPAL impact

Windows Vista – Finished		
Issue	*Fixed*	*Noise*
Security – RELOJ	386	4%
Security – Impersonation Token	135	10%
Security – OpenView	54	2%
Leaks – RegCloseHandle	63	0%
Windows – In Progress		
Issue	*Found*	
Localization – Constant strings	1214	
Security – ClientID	282	
…		

OPAL priorities

- Concurrency
  - Specify incorrect lock usage
- Localization
  - Specify usage of culture-sensitive strings
- Accessibility
  - Specify usage of hard-coded fonts and colors
- DLL loading
  - Specify cyclic dependencies from DLLMain
- Security, drivers, serviceability, …

Specifications summary

- Rule out specific patterns of incorrect behavior by writing OPAL specifications of observed failures
  - [+] Specifications are written once per codebase
  - [+] Education is limited to a few experts
  - [–] No validation (“how far are we from done?”)
- OPAL is a low cost, lower return approach
  - Applicable to a broad range of quality priorities

Lessons

Forcing functions for change

- Gen 1: Manual Review
  - Too many code paths to think about
- Gen 2: Massive Testing
  - Inefficient detection of simple errors
- Gen 3: Global Program Analysis
  - Delayed results
- Gen 4: Local Program Analysis
  - Lack of calling context limits accuracy
- Gen 5: Specifications

Developers like specifications

- If you make them incremental
  - No specifications, no bugs
- If you make them useful
  - More specifications, more real bugs
- If you make them informative
  - Make implicit information explicit
  - Avoid repeating what the code says

Defect detection myths

- Soundness matters
  - sound == find only real bugs
  - The real measure is Fix Rate
- Completeness matters
  - complete == find all the bugs
  - There will never be a complete analysis
- Developers only fix real bugs
  - Developers fix bugs that are easy to fix, and
  - Unlikely to introduce a regression

Theory is important

- Fundamental ideas have been crucial
  - Hoare logic
  - Dataflow analysis
  - Abstract interpretation
  - Graph algorithms
  - Context-sensitive analysis
  - Alias analysis

Summary

- Goal: Use formal specifications to move enforcement of code quality upstream
  - Testing → Specifications → Compiler
- Two complementary solutions:
  - Source code annotations (SAL), targeted to a small set of critical properties
  - Defect specifications (OPAL), applied to a wide range of quality priorities
- Testing → OPAL → SAL → Compiler

http://www.microsoft.com/cse
http://research.microsoft.com/manuvir