LLNL-TR-587361

NEAMS Static Analysis

B. Oliver, T. Dahlgren, M. C. Miller

October 3, 2012

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

NEAMS Static Analysis

Bill Oliver, Klocwork Administrator, oliver4@llnl.gov
Tamara Dahlgren, SQA Engineer, dahlgren1@llnl.gov
Mark C. Miller, NEAMS ECT Technical Area Lead, miller86@llnl.gov

Table of Contents

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Introduction to static analysis

Why use Static Analysis Tools

Advanced Static Analysis tools can find defects in source code without actually running the code. These tools are capable of finding a large set of defects depending on which checkers are enabled. The tool used on the NEAMS project is developed by a company named “Klocwork.” For C and C++ codes it has more than 150 checkers. These checkers can find issues in source code inter-procedurally including such defects as memory/resource leaks, buffer overflows, use of uninitialized variables, null pointer dereferences and many others. Some of these defects can also be found with dynamic tools such as Valgrind. However the Klocwork static analysis tool has several advantages over dynamic analysis tools like Valgrind. First, Klocwork can find more types of defects and it looks in more places to find defects. Valgrind, for example, requires that the code be executed to find defects. That means valgrind can find defects only in the code paths that are actually executed. In addition, running a code with Valgrind can cause the code to run significantly slower. On the other hand, the time it takes to analyze a code with Klocwork is only on the same order as the time it takes to compile the code. Changes to fix defects are confirmed by a short re-compile instead of a potentially very long re-execution. In addition static analysis provides 100 percent code coverage whereas dynamic analysis provides only as much coverage as the portion of code actually executed. We are not suggesting static analysis should replace dynamic analysis. Neither approach can find all defects. We recommend the use of a static analysis tool in conjunction with a dynamic tool. This facilitates the discovery of the largest possible set of defects.

Anecdotes from Industry

Klocwork has over 1000 customers which include Motorola, Intel, Qualcomm, and John Hopkins to name just a few. In a case study by John Hopkins titled “Enhancing software reliability and developer productivity while building the next generation of human prosthetics”, a DARPA project, Klocwork was shown to increase the reliability of the code by finding critical defects that went undetected by conventional testing techniques. In summarizing the benefits of using Klocwork that study states the following: “Since the software is an embedded real-time system, defects such as array bounds violations, use of uninitialized data, null pointer references, and thread synchronization errors are all examples of potentially critical issues that were found by the tool.”

In another case study, Motorola asserts “large scale deployment of static analysis leads to measurable productivity and quality improvements.” In that study, Motorola found that in the first year of using Klocwork they experienced a two-fold reduction in the number of defects found during system test.

The University of Colorado’s Laboratory for Atmosphere and Space Physics (LASP) conducts research in Atmospheric science, space physics, solar influences, and planetary science. Due to reliability requirements, LASP began routine analysis of the GOES-R software with the Klocwork tool set. Don Woodraska, Professional Research Assistant in the Data Systems group, has indicated that Klocwork found Array Bounds Overflow type defects that were critical to the software for that project. Their case study indicated that by using Klocwork they were able to improve the reliability of the code and the productivity and development skills of their developers.

Introduction and description of the 4 NEAMS codes

Four codes were chosen to run through the Klocwork static analyzer; HDF5, MOAB, AMP and VisIt. Three codes, HDF5, MOAB, and VisIt are open source. AMP is divided into an export controlled piece and a piece that is not export controlled. The non-export controlled portion was analyzed with Klocwork.

According to the HDF Group web site: "The HDF5 technology suite is designed to organize, store, discover, access, analyze, share, and preserve diverse, complex data in continuously evolving heterogeneous computing and storage environments. HDF5 supports all types of data stored digitally, regardless of origin or size. Petabytes of remote sensing data collected by satellites, terabytes of computational results from nuclear testing models, and megabytes of high-resolution MRI brain scans are stored in HDF5 files, together with metadata necessary for efficient data sharing, processing, visualization, and archiving." HDF5 is a C I/O library used by most of the other codes on the NEAMS project.

The Mesh-Oriented datABase (MOAB) is a software component developed at Argonne National Laboratory for representing and evaluating mesh data. It includes support for parallel mesh reading, writing, and communication. Also included are other important mesh-based capabilities: mesh-to-mesh solution transfer, fast ray tracing, and interfaces to key services such as parallel partitioning, mesh visualization, and geometric modeling. A maximal configuration of MOAB makes use of several third party libraries including HDF5, NetCDF, CGM, VTK, Zoltan, ParMETIS and Chaco. Our configuration tested HDF5 and NetCDF. The functional interface to MOAB is simple yet powerful, allowing the representation of many types of metadata commonly found on the mesh. MOAB also is optimized for efficiency in space and time. MOAB is written in C++ though it also supports interface bindings for Fortran and C.

Advanced Multi-Physics (AMP) is a general purpose computational environment developed at Oak Ridge National Laboratory. It includes implementations of coupled diffusion, mechanics, and fluid dynamics that allow the user to build a multi-physics application code from existing diffusion and mechanics operators. It also allows for the extension with user-defined material models and new physics operators. AMP makes use of the following third party libraries: BLAS, BOOST, HDF5, HYPRE, LAPACK, LibMesh, MATPRO, PETSC, Silo, SUNDIALS, Trilinos, and VisIt for visualization. AMP is used in the simulation of Nuclear Reactors. AMP is also written in C++.

VisIt is an open source, interactive, parallel visualization and graphical analysis tool for viewing scientific data on Unix and PC platforms. Users can quickly generate visualizations from their data, animate them through time, manipulate them, and save the resulting images for presentations. VisIt contains a rich set of visualization features so that a user can view their data in a variety of ways. It can be used to visualize scalar and vector fields defined on two- and three-dimensional (2D and 3D) structured and unstructured meshes. VisIt was designed to handle very large data set sizes in the terascale range and yet can also handle small data sets in the kilobyte range. VisIt makes use of a large number of third party libraries including MPI, VTK, Qt, Python, HDF5, Silo, as well as a large number of optional I/O libraries. Because VisIt is an order of magnitude larger code base than the other codes we tested, the configuration of VisIt we tested was simplified to expedite progress in performing static analysis.

Static Analysis Approach

Since all but one of the codes is developed by other organizations outside of Lawrence Livermore National Laboratory, we set up a Klocwork server on our outwardly facing (Green) network. The Klocwork administrator coordinated with the code teams to obtain the requirements for doing a native build using standard compilers such as gcc and g++. In most cases this required downloading, and compiling many third party libraries. Once the native build was working the Klocwork administrator ran the analysis and reviewed the results. Accounts were given to the code team members and a brief tutorial was conducted over the phone.

For every defect Klocwork finds, it maintains an associated “State” and “Status”. The State is not modifiable by a user but is instead set by the tool and includes the following values: New, Existing, and Fixed. The first time a defect is found during an analysis, its state is set to “New”. On subsequent analyses, the state is either “Existing” if a fix has not been put in the code or “Fixed” if it has. In this way we are able to keep track of which defects have been fixed as well as filter out existing and newly introduced defects.

Bill Oliver, the Klocwork administrator, and Tammy Dalghren, both of Lawrence Livermore National Laboratory ASQ Group performed a review of the results and modified the status of the defects to help identify as many false positives as reasonably possible and to indicate which defects required a closer look. This process is known as *defect triage*. Triage is accomplished via a web based interface. Developers on the code teams that were given accounts can log in remotely and use the same web interface to see and triage results themselves similarly. With the exception of Mark Baird of the AMP team, none of the developers from the other code teams took an opportunity to use the web interface to review and/or triage defects. Due to resource constraints, LLNL personnel triaged only the highest priority defects.

Another application in the Klocwork tool set allows a developer to create a local project and synchronize analysis results to the Klocwork server. Developers can then fix defects in their own local checkout, quickly run a static analysis and determine if the defect was fixed (and that new defects were not introduced in the process). However, this application is available only to developers who have accounts on LLNL’s yellow network. Since code team members outside the lab do not have accounts on the machine they cannot take advantage of this local tool.

For AMP, one of the AMP developers, Mark Baird, was given the responsibility to start putting fixes in the code, test the code and prepare a code release to be given to the Klocwork administrator. The Klocwork administrator then reran the analysis and verified that the defects were fixed.

For MOAB, HDF5 and VisIt Mark Miller, the NEAMS ECT TAL, already had permission to check out each of the codes and to also check fixed code back in to the repository. For MOAB, HDF5, and VisIt, Mark used the local, desktop client to fix the defects on a local, private checkout as described above. He assisted each of the code teams by putting in fixes for selected defects, running each code’s test suite to confirm he had not introduced any test failures and checked his changes back into a branch. Each code team then reviewed Mark’s changes on this branch and then merged the changes back into their mainline (trunk) development. Once Mark's changes were merged to the trunk, the Klocwork Administrator checked out a fresh copy and reran the analysis to confirm the defects were reported as fixed.

The MOAB code required the installation of multiple libraries. Fortunately MOAB includes a shell script to help build and install MOAB and its required libraries. This script required minor modifications to get it to build on the machine where Klocwork is installed. Once the initial build and analysis was complete a script was created by the Klocwork administrator and scheduled to run once a week, automatically via a cron job.

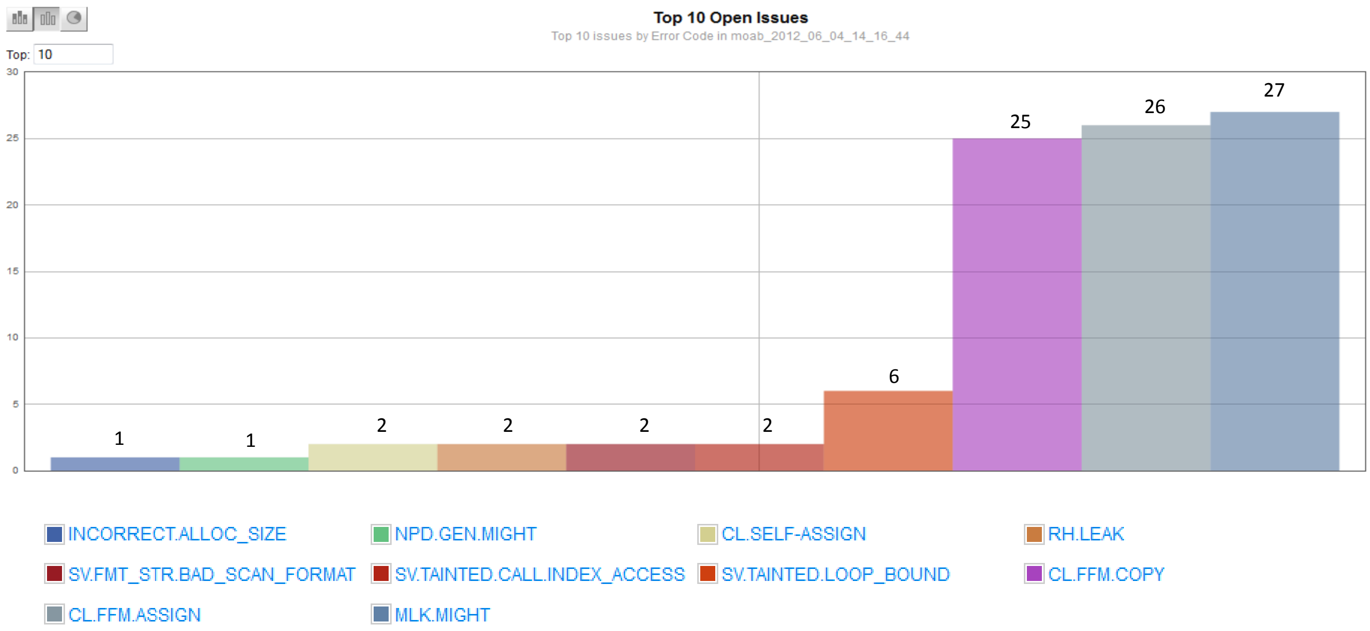
For the AMP code there were a number of issues that had to be overcome when building the large number of libraries. Fortunately the AMP team assigned their main build engineer, Mark Baird, to assist in working out the issues. Even so this turned out to be quite time consuming and took several weeks of iterations with the AMP build engineer to complete. In addition, Klocwork was crashing when analyzing AMP. This turned out to be a bug with Klocwork. What made resolving this Klocwork bug particularly difficult was that we could not simply give the AMP code to Klocwork developers due to export control constraints. We addressed this by winning approval from Oak Ridge to allow one of Klocwork's sales engineers who is an American Citizen to view the code and receive debug files. He was able to recreate the problem and Klocwork developers were able to provide a patch that resolved the bug and allowed us to complete static analysis of AMP.

VisIt is developed at Lawrence Livermore National Laboratory with other developers outside the lab contributing. The VisIt analysis was moved to the swordfish server in order to allow developers from outside the lab to view the results. The checkout of VisIt and build with Klocwork was very straight forward. The Klocwork build was automated and also runs once per week. Any new defects discovered are automatically emailed to the team for immediate resolution. In this way, the VisIt team prevents new defects from entering the code. Nonetheless, there is also a backlog of defects VisIt developers plan to address as time and resources permit.

Analysis Results

MOAB

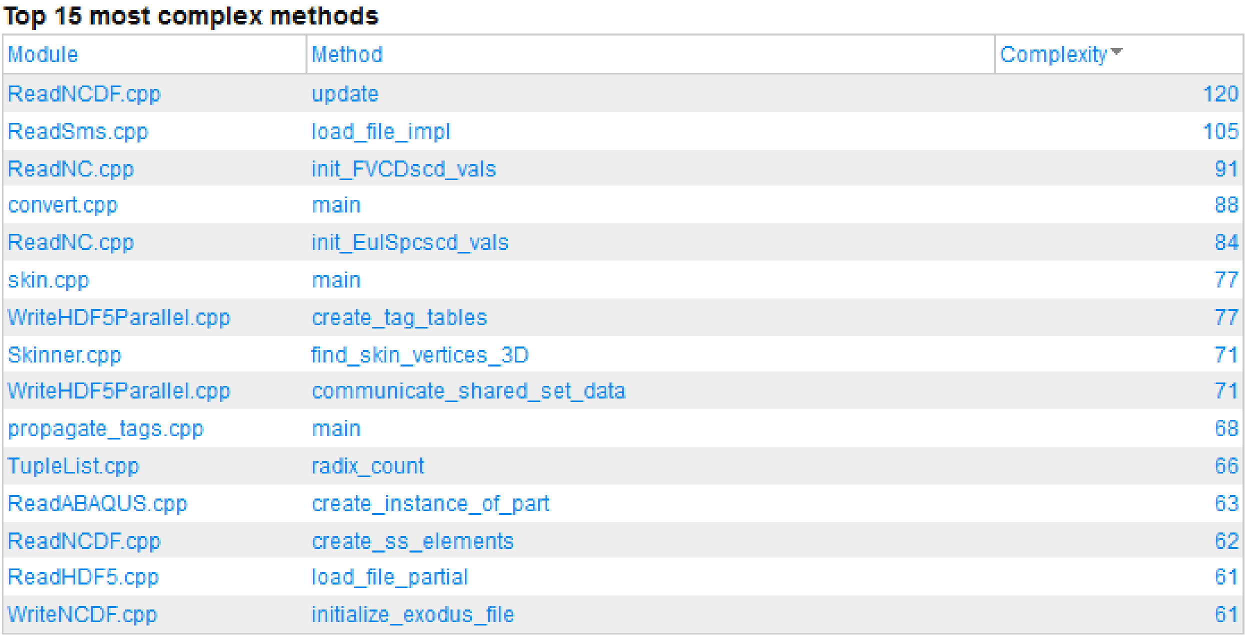
Of the 190 open defects in 104,566 lines of code analyzed 19 defects were thought to be False Positives or about 10%. The defect density for this code is 1.82 defects per thousand lines of code. To date 40 of those defects have been fixed and verified by another Klocwork analysis. Of the 94 issues triaged and deemed needed to be fixed, the breakdown of the issue types is shown below:



The defect nomenclatures, a brief description of the type of defect and the number found are listed in the following table.

Nomenclature	Description	Number Found
MLK.MIGHT	Memory Leak Possible	27
CL.FFM.ASSIGN	Freeing of freed memory due to missing assign operator=	26
CL.FFM.COPY	Freeing freed memory due to missing copy constructor	25
SV.TAINTED.LOOP_BOUND	Un-validated input used as a loop boundary	6
SV.FMT_STR.BAD_SCAN_FORMAT	Missing width field for format	2
SV.TAINTED.CALL.INDEX_ACCESS	Un-validated input used in array indexing by function call	2
RH.LEAK	Resource Leak	2
CL.SELF-ASSIGN	Memory leak in assign operator=	2
NPD-GEN.MIGHT	Possible assigned null-pointer constant value may be dereferenced	1
INCORRECT.ALLOC_SIZE	Incorrect allocation size	1

In addition the Klocwork tool calculates cyclomatic complexity. To obtain measures of cyclomatic complexity, each function is modeled as a complete graph and then the number of linearly independent paths through the function (graph) is calculated. Studies have shown that there is a direct correlation between cyclomatic complexity and the testability and maintainability of the code. A standard target complexity measure is 20 or less. Within MOAB, 2,849 functions had a cyclomatic complexity of 20 or less and 212 functions had complexities greater than 20. Below is a breakdown of the files that contained the highest complexities along with the maximum complexity in that file.

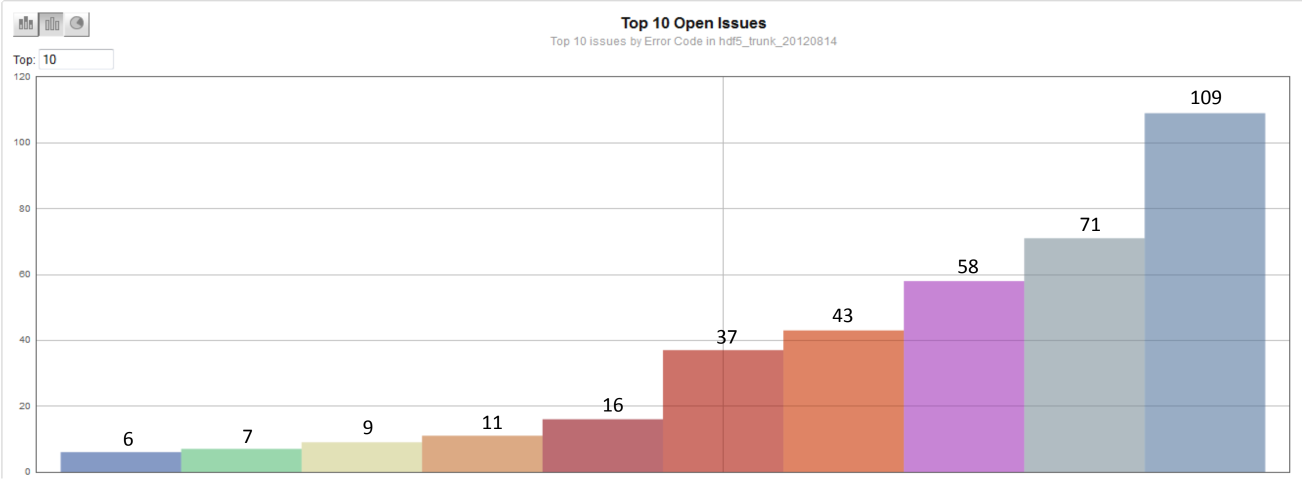


Top 15 most complex methods

Module	Method	Complexity▾
ReadNCDF.cpp	update	120
ReadSms.cpp	load_file_impl	105
ReadNC.cpp	init_FVCDscd_vals	91
convert.cpp	main	88
ReadNC.cpp	init_EulSpcscd_vals	84
skin.cpp	main	77
WriteHDF5Parallel.cpp	create_tag_tables	77
Skinner.cpp	find_skin_vertices_3D	71
WriteHDF5Parallel.cpp	communicate_shared_set_data	71
propagate_tags.cpp	main	68
TupleList.cpp	radix_count	66
ReadABAQUS.cpp	create_instance_of_part	63
ReadNCDF.cpp	create_ss_elements	62
ReadHDF5.cpp	load_file_partial	61
WriteNCDF.cpp	initialize_exodus_file	61

HDF5

Of the 512 open defects in 340,993 lines of code 68 defects were thought to be False Positives or about 13 percent. The defect density for this code is 1.50 defects per thousand lines of code. To date 70 defects have been fixed and verified by another Klocwork analysis. Of the 367 defects issues triaged and deemed needed to be fixed, the breakdown of the number and type of defect reported is shown below.



Top 10 Open Issues

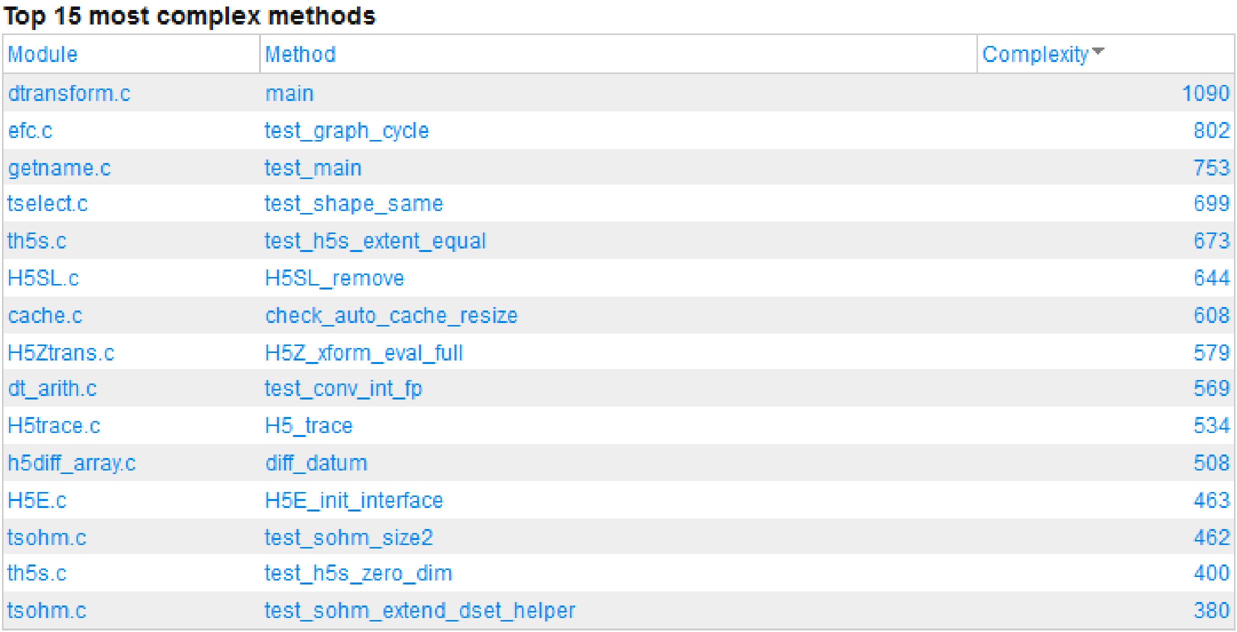
Top 10 issues by Error Code in hdf5_trunk_20120814

🟦 RH.LEAK 🟩 SV.TAINTED.ALLOC_SIZE 🟨 ABV.STACK 🟧 SV.FMT_STR.BAD_SCAN_FORMAT
🟥 RNPD.DEREF 🟥 ABR 🟧 NPD.FUNC.MUST 🟪 UNINIT.STACK.MIGHT
⬜ MLK.MIGHT 🟦 SV.TAINTED.FMTSTR

The defect nomenclatures, a brief description of the type of defect and the number found are listed in the following table.

Nomenclature	Description	Number Found
SV.TAINTED.FMTSTR	Un-validated input in format string	109
MLK.MIGHT	Memory Leak Possible	71
UNINIT.STACK.MIGHT		58
NPD.FUNC.MUST	Possible null pointer is dereferenced	43
ABR	Buffer Overflow	37
RNPD.DEREF	Suspicious dereference of pointer before null check	16
SV.FMT_STR.BAD_SCAN_FORMAT	Missing width field for format	11
ABV.STACK	Buffer overflow—local array index out of bounds	9
SV.TAINTED.ALLOC_SIZE	Un-validated input used in memory allocation	7
RH.LEAK	Resource Leak	6

Within HDF5 of the 6,116 functions analyzed 4,396 had a cyclomatic complexity of 20 or less leaving 1,720 methods with a complexity of greater than 20. Information on the most complex methods is shown below.



Top 15 most complex methods

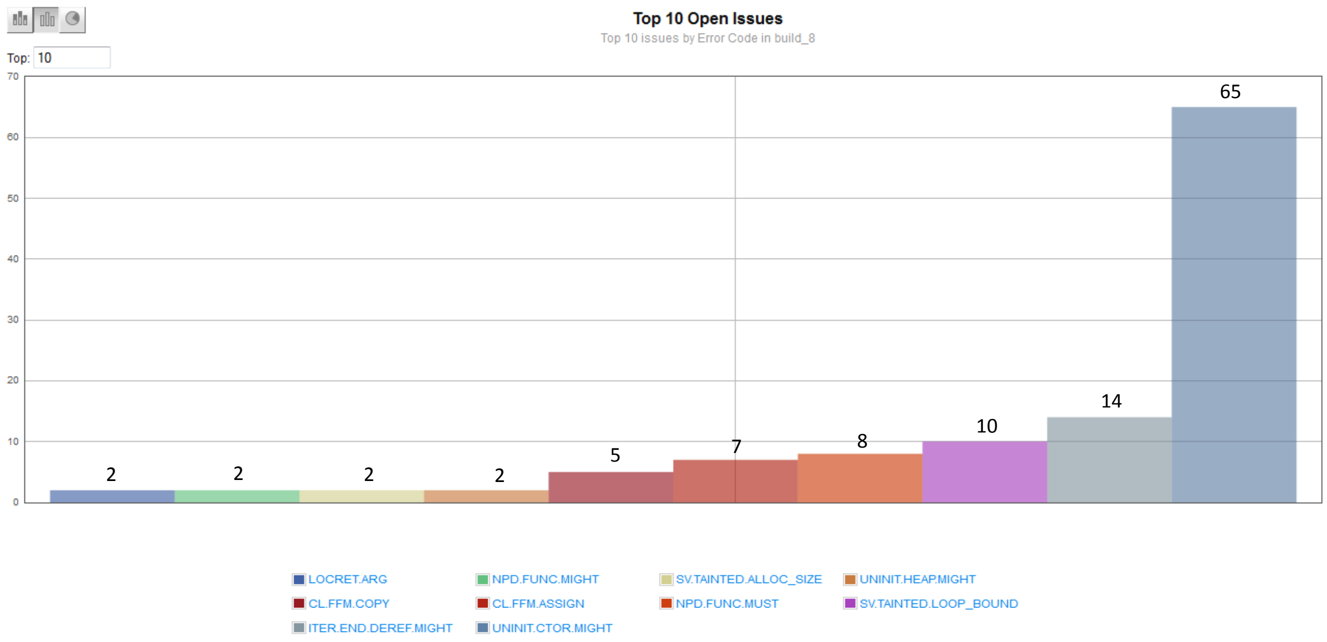
Module	Method	Complexity
dtransform.c	main	1090
efc.c	test_graph_cycle	802
getname.c	test_main	753
tselect.c	test_shape_same	699
th5s.c	test_h5s_extent_equal	673
H5SL.c	H5SL_remove	644
cache.c	check_auto_cache_resize	608
H5Ztrans.c	H5Z_xform_eval_full	579
dt_arith.c	test_conv_int_fp	569
H5trace.c	H5_trace	534
h5diff_array.c	diff_datum	508
H5E.c	H5E_init_interface	463
tsohm.c	test_sohm_size2	462
th5s.c	test_h5s_zero_dim	400
tsohm.c	test_sohm_extend_dset_helper	380

A reason for surprisingly high cyclomatic complexity measures in many of theses cases is that a majority of these routines are actually part of HDF5's test suite and not part of the HDF5 library itself. The test suite involves more complex code because of the wide variety of ways in which it attempt to drive HDF5 and the various means of detecting if a test has failed.

When reviewing fixes to HDF5, lead software engineer for HDF5, Quincy Koziol remarked “Very cool. I think it's a nice win-win scenario, so I'm really glad to participate. I'm a big proponent of static analysis tools. They aren't a panacea, but they do help quite a bit.”

AMP

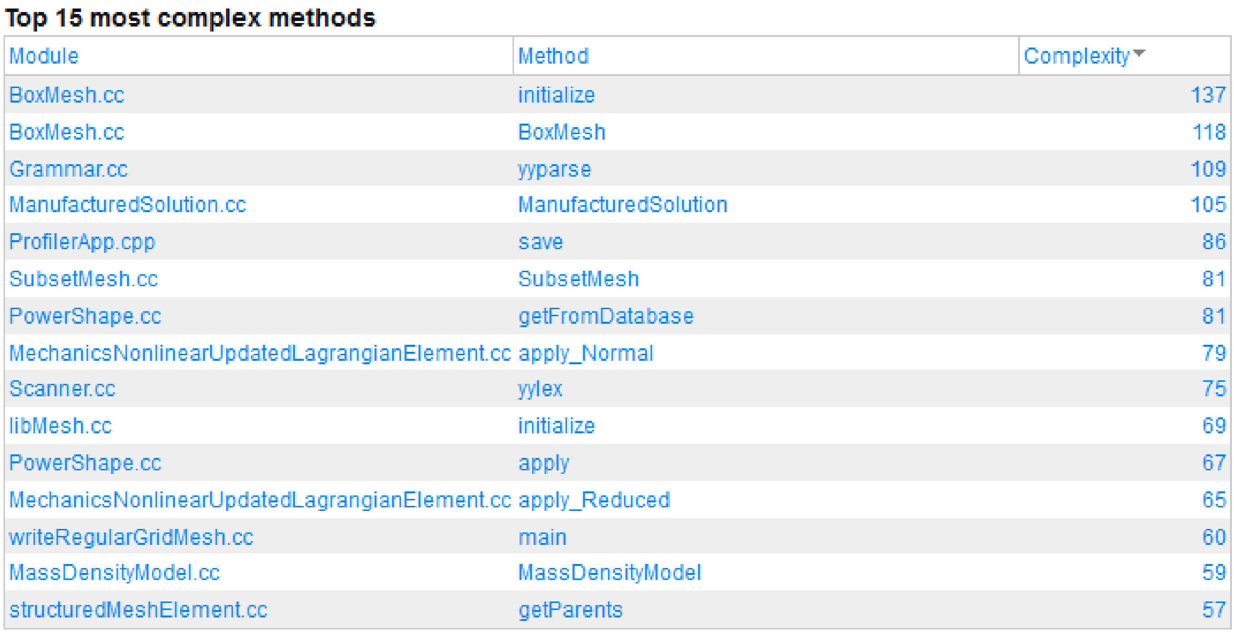
A total of 296 defects have been reported in 103,727 lines of code. Of those defects 90 have yet to be triaged leaving 206 that have been triaged. The defect density of this code is 1.99 defects per thousand lines of code. Of the 206 triaged defects, 17 were considered to be False Positives or 8 percent. A total of 124 have been identified as needing to be fixed and a total of 62 defects have been fixed. The distribution of the types of defects that have been identified as needing to be fixed is shown below.



The defect nomenclatures, a brief description of the type of defect and the number found are listed in the following table.

Nomenclature	Description	Number Found
UNINIT.CTOR.MIGHT	Uninitialized variable in constructor possible	65
ITER.END.DEREF.MIGHT	Dereference of 'end' iterator	14
SV.TAINTED.LOOP_BOUND	Un-validated input used as a loop boundary	10
NPD.FUNC.MUST	Possible null pointer is dereferenced	8
CL.FFM.ASSIGN	Freeing freed memory due to missing assign operator=	7
CL.FFM.COPY	Freeing freed memory due to missing copy constructor	5
UNINIT.HEAP.MIGHT	Uninitialized heap use possible	2
SV.TAINTED.ALLOC_SIZE	Un-validated input used in memory allocation	2
NPD.FUNC.MIGHT	Possible null pointer may be dereferenced	2
LOCRET.ARG	Function returns address of local variable	2

Within AMP of the 3,429 functions analyzed 3,309 had a cyclomatic complexity of 20 or less leaving 120 functions with a complexity greater than 20. A listing of the top 15 functions and their complexity values are shown below.



Top 15 most complex methods

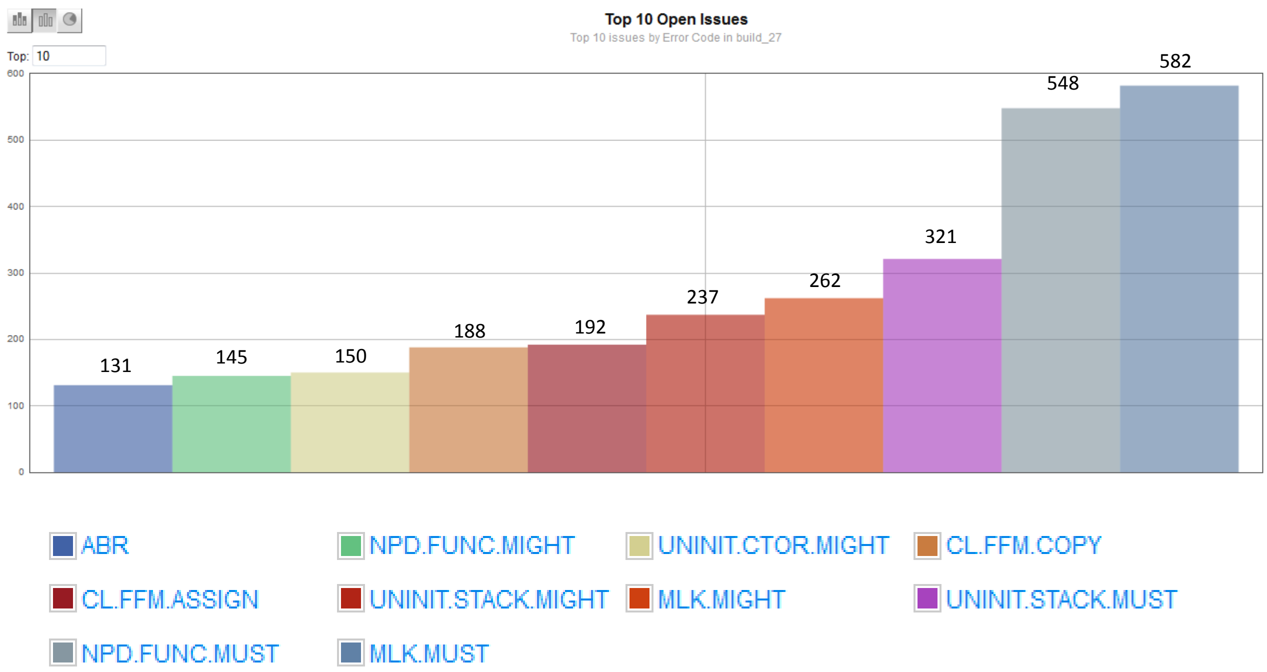
Module	Method	Complexity▾
BoxMesh.cc	initialize	137
BoxMesh.cc	BoxMesh	118
Grammar.cc	yyparse	109
ManufacturedSolution.cc	ManufacturedSolution	105
ProfilerApp.cpp	save	86
SubsetMesh.cc	SubsetMesh	81
PowerShape.cc	getFromDatabase	81
MechanicsNonlinearUpdatedLagrangianElement.cc	apply_Normal	79
Scanner.cc	yylex	75
libMesh.cc	initialize	69
PowerShape.cc	apply	67
MechanicsNonlinearUpdatedLagrangianElement.cc	apply_Reduced	65
writeRegularGridMesh.cc	main	60
MassDensityModel.cc	MassDensityModel	59
structuredMeshElement.cc	getParents	57

Mark Baird provided the following feedback regarding the inter-lab collaboration and the usefulness of performing static analysis.

“The information the AMP (Advanced Multi-Physics) development team received from the Klocwork static analysis tool through the NEAMS project was a valuable asset from a development standpoint. The analysis provided to the developers from the Klocwork graphical user interface allowed the developers to not only view possible coding and memory issues but the surrounding code for easy determination of purpose and structure of the analyzed code. This allowed the developers to put the source in proper context. The web based user interface was also easy to learn and presented the information in an organized manor. The interface also provided a means to track and trend the development of the source over time, which could allow developers to see what form errors were being typically generated and develop processed to limit the future introduction of similar errors. Overall we were happy with the information we gained from Klocwork. In the future I think it would be a great benefit for everyone involved if a method could be developed early in the process that would allow for an easier transfer of source code from the developers to the LLNL team. I do want to compliment the people involved with the Klocwork analysis, they were very professional and very knowledgeable of not only the interface, but programming in general.”

VisIt

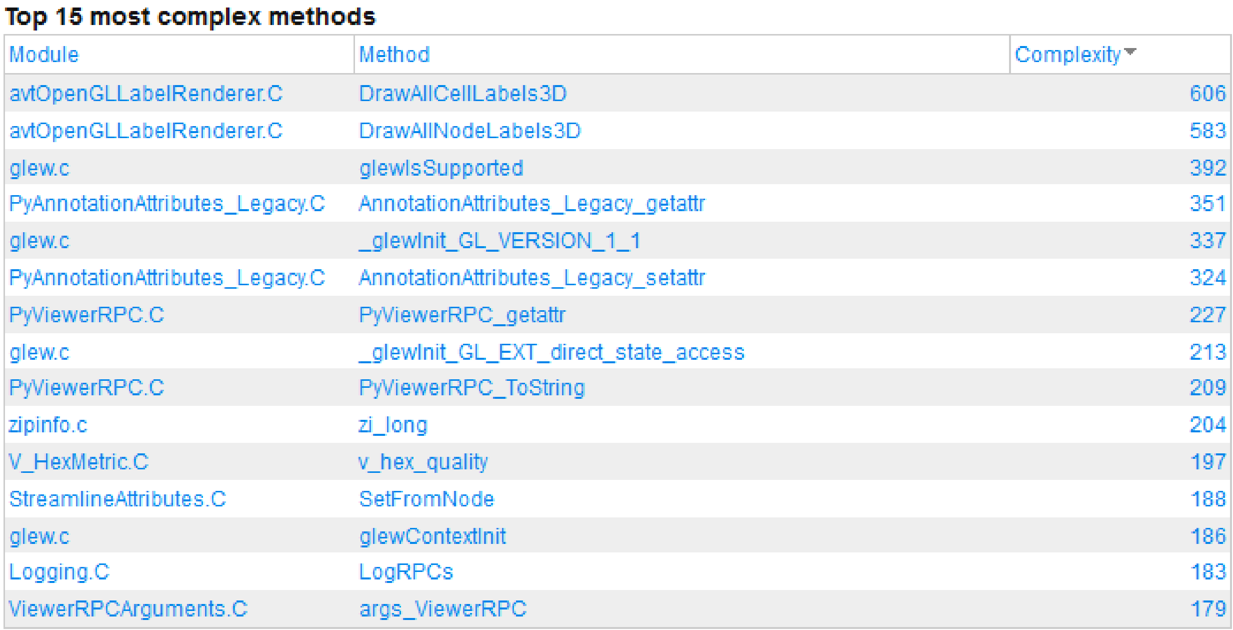
A total of 2,756 defects were reported in 1,195,082 lines of code. However, insufficient triage work has been conducted to estimate an accurate False Positive rate. Preliminary indications show that the False Positive rate may be as high as 20% and may be primarily due to Klocwork’s analysis of VTK object usage. Using 20% as the False Positive rate, there are approximately 2213 real defects. This produces a defect density of 1.85 defects per thousand lines of code and is consistent with the other codes we tested. Rather than attempt to triage the rather large number of defects, the team chose to focus on only certain classes of defects, memory leaks. As of the latest Klocwork analysis the tool reports that 84 defects have been fixed. The breakdown of the distribution of defect types is shown below.



The defect nomenclatures, a brief description of the type of defect and the number found are listed in the following table.

Nomenclature	Description	Number Found
MLK.MUST	Memory leak	582
NPD.FUNC.MUST	Possible null pointer is dereferenced	548
UNINIT.STACK.MUST	Uninitialized variable	321
MLK.MIGHT	Memory leak possible	262
UNINIT.STACK.MIGHT	Uninitialized variable possible	237
CL.FFM.ASSIGN	Freeing freed memory due to missing assign operator=	192
CL.FFM.COPY	Freeing freed memory due to missing copy constructor	188
UNINIT.CTOR.MIGHT	Uninitialized variable in constructor possible	150
NPD.FUNC.MIGHT	Possible null pointer may be dereferenced	145
ABR	Buffer overflow—array index out of bounds	131

Of the 29,242 functions that were analyzed 1,259 had cyclomatic complexities greater than 20. A listing of the top 15 functions and their complexity values are shown below. Several of the functions listed in the table below are actually part of third party libraries (Glew and Verdict libraries) which are automatically built-in to VisIt and exist inside of its source code tree. In addition, some of the functions are the result of code generation techniques which are notorious for generating code with high cyclomatic complexity.



Top 15 most complex methods

Module	Method	Complexity▼
avtOpenGLLabelRenderer.C	DrawAllCellLabels3D	606
avtOpenGLLabelRenderer.C	DrawAllNodeLabels3D	583
glew.c	glewIsSupported	392
PyAnnotationAttributes_Legacy.C	AnnotationAttributes_Legacy_getattr	351
glew.c	_glewInit_GL_VERSION_1_1	337
PyAnnotationAttributes_Legacy.C	AnnotationAttributes_Legacy_setattr	324
PyViewerRPC.C	PyViewerRPC_getattr	227
glew.c	_glewInit_GL_EXT_direct_state_access	213
PyViewerRPC.C	PyViewerRPC_ToString	209
zipinfo.c	zi_long	204
V_HexMetric.C	v_hex_quality	197
StreamlineAttributes.C	SetFromNode	188
glew.c	glewContextInit	186
Logging.C	LogRPCs	183
ViewerRPCArguments.C	args_ViewerRPC	179

In his experience using Klocwork for VisIt, Mark Miller writes…

”In most cases, the leaks we found with Klocwork were relatively small and unlikely to have caused any serious problems in VisIt. However, in a few cases, the memory leaks involved allocations related to input database mesh size (e.g. were problem sized) which could have easily driven VisIt into an out of memory (and therefore crash) condition when iterating over a time series. So, fixing these leaks definitely improves VisIt’s robustness. The speed with which fixes to code could be tested using the Klocwork desktop client (as well as the reporting of potentially new issues introduced by such fixes) was impressive and significantly helped productivity in finding and fixing defects. The Klocwork administrator, Bill Oliver, was instrumental in providing technical support for these activities.”

Summary of Results

In the table below, we summarize the results of static analysis activities on these 4 codes.

Code	#Lines	#Defects	Rate	False +	#Triaged	#Fixed	%Complex
MOAB	104566	190	0.182%	19 (10%)	94	40	6.93
HDF5	340993	512	0.150%	68 (13%)	367	70	28.1%
AMP	103727	296	0.199%	17 (8%)	206	62	3.5%
VisIt	1195082	2756	0.185%	20%	N/A	84	4.31%

Conclusions & Recommendations

The defects found by the Klocwork tool set were defects that had previously gone undetected by whatever code review and or testing that are currently being done. Defects related to use of uninitialized variables or uninitialized heap memory could easily cause the code to produce erroneous or results that vary depending on configuration. Fixing these defects can have the effect of making the code more robust.

All of the codes had a defect density of between 1 and 2 defects per thousand lines of code. This has been typical of other RL3 and RL4 HPC codes we have analyzed for other programs.

In MOAB, HDF5 and VisIt a majority of memory leaks were in error-handling portions of the code which are often difficult to test dynamically. A common source of leaks of this nature have to do with early returns from functions where all cleanup from temporary allocations is handled at the bottom of the function only.

For all of the codes analyzed the vast majority of the functions had cyclomatic complexities of 20 or less and for MOAB and AMP the top complexity values were less than 140. In many cases, higher complexities are associated with test code. This is to be expected as test code often involves extensive logic to test a large variety of related but different behaviors all in the same test function harness.

To reduce barriers to entry in the introduction of static analysis on a larger scale in a program like NEAMS, a probable best practice is to first stand up procedures to eliminate the introduction of *new* defects. For example, the first time static analysis was applied to VisIt, there were already more than one million lines of code in VisIt. Static analysis revealed a large number of defects, too numerous for existing resources to address immediately. The VisIt team therefore took the approach of addressing this backlog of pre-existing defects on an as-time-permits basis. However, the Klocwork administrator, Bill Oliver, set up weekly static analysis to identify any *new* issues introduced on VisIt's mainline development trunk. VisIt developers fix new issues found. In this way, we prevent new defects from being introduced.

All of the codes analyzed had development teams that were significantly under staffed. Going back and fixing legacy defects was a matter of prioritization. In the case of MOAB and HDF5 we applied our own manpower resources, Mark Miller, to assist in that effort. Mark was also involved in fixing VisIt defects since he is also one of the developers on that code team. The AMP team assigned that task to Mark Baird who did an excellent job. The advantage of having the Livermore people fix defects stems from the access setup. Livermore employees actually have login accounts on the Klocwork server whereas non-Livermore employees only have access to the web interface. Livermore employees can take advantage of the use of the Klocwork desktop client which allows local checking of analysis results prior to the next server analysis. This allows for verifying that defects are fixed and more importantly that no new defects have been introduced in the process. So when that code is committed to the repository and a new server analysis is completed we have 100% certainty that defects are fixed and no new defects introduced.

Based on this advantage it is recommended that in the future Livermore employees actually fix defects (at least the low hanging fruit) and provide the code team with patches (or wholesale commits to a special branch set aside for static analysis work) so that the team can just run it through their test suite and if all is well commit the code to the repository.