Tool Operational Requirements (TOR) Framework for CodeHawk C Analyzer (Tool B)

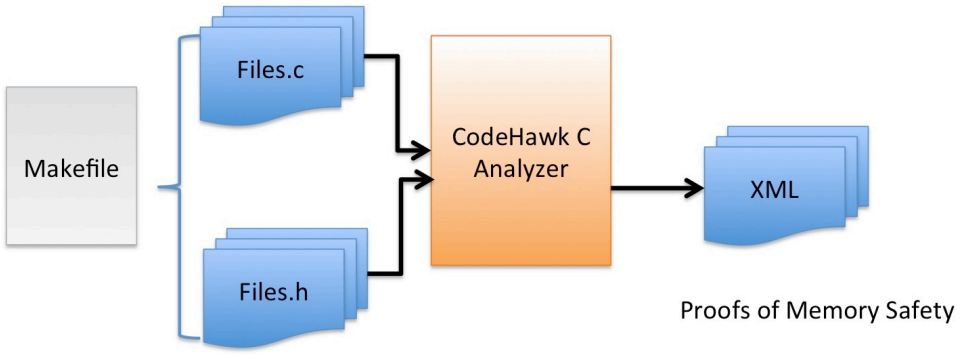
1 Introduction

This document is part of a research Case Study simulating a Formal Methods Tool qualification exercise under DO-330, for the CodeHawk C Analyzer static analysis tool. An actual qualification exercise would produce, among other documents, a Tool Operational Requirements (TOR) document for this analyzer as specified by DO-330 10.3.1. Because this is a research study, in which there is no actual qualifying organization and accompanying context, and because the tool under consideration is a research implementation without specific versions, release control, user documentation, or standard configurations, some of the sections of an actual TOR will not be relevant. This document is thus a Framework for an actual TOR, organized according to the DO-330 enumeration of required contents in section 10.3.1. Accordingly, some subsections will represent content that is concrete enough to be part of an actual TOR; some will discuss how a more concrete implementation of this tool might fulfill the required contents; and some will not be applicable for the purposes of this research simulation.

2 TOR Framework

[From DO330-10.3.1] **The Tool Operational Requirements define the tool's functionality and interface from a software life cycle process perspective (that is, the process which uses the tool). The Tool Operational Requirements shall include, as applicable:**

A. Description of the context of the tool use, including interfaces with other tools and integration of the tool output files into the resultant software.



The CodeHawk C Analyzer is a standalone static analysis tool for proving (or disproving) memory safety properties of C programs via abstract interpretation. Its inputs are the set of source files that would normally constitute a C application for a compile and link process described by a Makefile. Its output is a series of intermediate and final XML files that record the analysis results. It requires the original C application to be first pre-processed by the CIL intermediate C language pre-processor [1] (bundled with the tool). The CIL pre-processor will, in turn, call the gcc compiler for parsing and macro expansion. The CodeHawk C Analyzer requires no other external tool interfaces for its normal operation.

B. Description of the tool operational environment(s) (where the tool will be installed).

Discussion

The CodeHawk C Analyzer is a research project developed on Mac and Linux. It is written in Ocaml and thus can be compiled to run on any platform supported by the Ocaml compiler, including Windows. It has only been built and tested on Mac/Linux to date. If this were a TOR for an actual tool qualification, there would be a specific version of the tool that was being qualified, and thus it would have a specific installation/execution platform described here.

C. Description of input files, including format, language definition, etc.

The CodeHawk C Analyzer is implemented as seven separate programs, compiled from Ocaml sources, that collective analyze a complete C application for memory safety. The seven modules are executed separately from the command line by seven shell scripts, and communicate with each other via persistent XML files. Some of the intermediate XML files may be regarded as both input and output (output from one analysis phase and input to another). The formats and language definitions of the XML files are specific to each analysis phase. Only one of these is created by the end user (see below). The others are created exclusively by the tool phases.

The ultimate input is a collection of C source programs (.c and .h files) that constitute a complete application for analysis, from the point of view of a build script for the application such as a Makefile. CodeHawk uses the open source CIL C preprocessor as a front-end, which in turn reads the input project's Makefile, to collect all of the .c and .h sources and preprocess them (by invoking **gcc**) into a canonical semantic representation.

CodeHawk itself (the Ocaml software distinct from the CIL pre-processor front-end) takes the following proximal input files to begin its analysis:

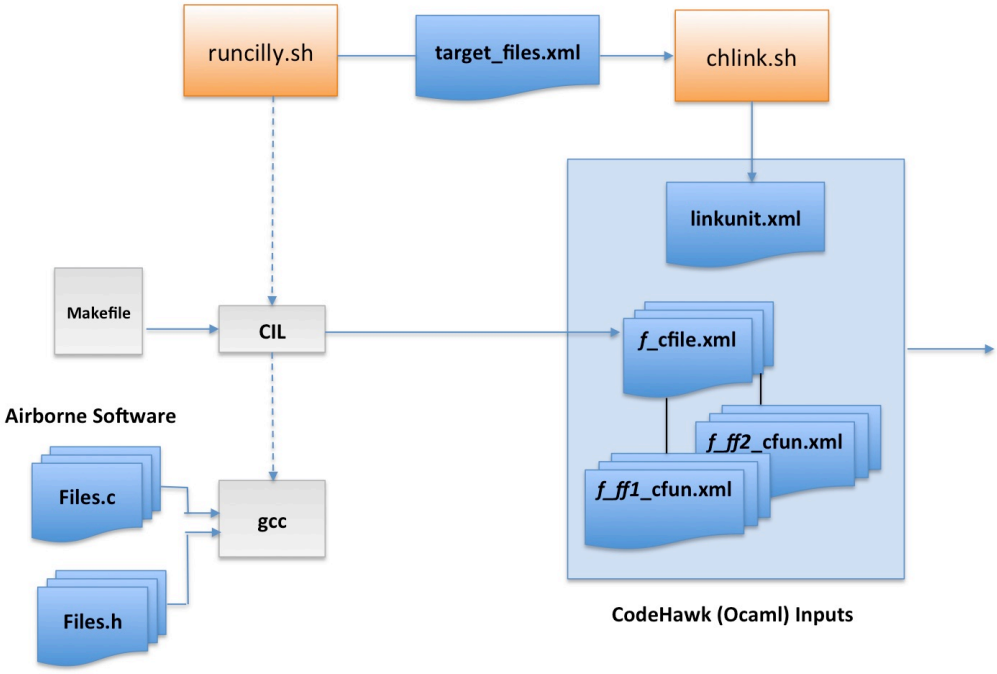
f*_cfile.xml**: file information for each file ***f compiled by CIL

f*/*f_ff*_cfun.xml**: the function semantics for each function ***ff in ***f***

target_files.xml: a listing of all source files included in the linking unit to be analyzed

linkunit.xml: lists all source files included in this linking unit and relates local variable and type id's to global id's

To create the proximal input files, the user first runs the shell script **runcilly.sh,** which in turn invokes CIL on the C source files and produces the ***f*_cfile.xml** , ***f*/*f_ff*_cfun.xml** and **target_files.xml** files. The user then runs the shell script **chlink.sh** on this input file to create the **linkunit.xml** file.



D. Description of output files, including format and contents.

The output files from CodeHawk are a persistent collection of XML files, some of which serve as intermediate input to various phases of the analysis, but still may be inspected independently by the user.

For each file ***f*** listed in the **linkunit.xml** file, CodeHawk produces:

- ***f*_cfile.xml** : definition and declaration information for compilation unit ***f***

For each function ***ff*** in file ***f***, CodeHawk produces:

- ***f/f_ff*_cfun.xml**: function semantics
- ***f/f_ff*_api.xml** : function API
- ***f/f_ff*_ppo.xml** : primary proof obligations
- ***f/f_ff*_spo.xml** : secondary proof obligations
- ***f/f_ff*_pev.xml** : primary proof obligation evidence
- ***f/f_ff*_sev.xml** : secondary proof obligation evidence
- ***f/f_ff*_invs.xml**: function invariants

For an example of how to relate discharged (proven) and open (unproven) proof obligations to their source code language constructs, see Appendix A of the *Theoretical Soundness Issues Report* produced as deliverable 2 for this project.

E. Requirements for all the tool functions and technical features used to satisfy the identified software life cycle process(es).

The certification objectives to be pursued by defining these Tool Operational Requirements are those described in DO178C-6.3.4(c) & (f):

> *c. Verifiability: The objective is to ensure the Source Code does not contain statements and structures that cannot be verified and that the code does not have to be altered to test it.*
>
> *f. Accuracy and consistency: The objective is to determine the correctness and consistency of the Source Code, including stack usage, memory usage, fixed point arithmetic overflow and resolution, floating-point arithmetic, resource contention and limitations, worst-case execution timing, exception handling, use of uninitialized variables, cache management, unused variables, and data corruption due to task or interrupt conflicts. The compiler (including its options), the linker (including its options), and some hardware features may have an impact on the worst-case execution timing and this impact should be assessed.*

The CodeHawk C Analyzer can be used to automatically check those aspects of verifiability, accuracy and consistency of C Source Code that relate to memory safety.

With respect to the verifiability objective (c), a program contains statements that cannot be verified only relative to some method of verification. All executable statements accepted by the C compiler have a proscribed semantics that change the program state in

some way. Any given set of test inputs may not execute all of these statements on all paths, and some of the intermediate values computed may not reach visible output, so if the method of verification is testing, some program statements may not be verifiable. But CodeHawk is an *abstract interpreter* that considers all possible program behaviors over all paths, *at some level of abstraction*, independent of any test inputs. It attempts to prove formal conjectures about the possible behaviors of program statements by computing constraints (invariants) on the range of values that they may compute. These constraints are then used to exclude or include certain classes of runtime behavior. The CodeHawk C Analyzer attempts to prove that every aspect of memory reference proscribed by the C Language Standard for every executable statement will be within the bounds of a well-defined memory state. *Every* statement will be subject to this proof, so relative to the verification method (abstract interpretation) there can be no program statements whose memory safety properties cannot be verified. Relative to the precision of a given abstract domain (intervals, values sets, linear equalities and inequalities), some statements may not be able to be proven memory-safe that actually are safe, but all statements are amenable to the verification method. Thus CodeHawk's approach to objective (c) is not to check for unverifiable statements, but to actually perform the verification of a particular behavior (memory safety) for all statements. It is the verification method itself (abstract interpretation) that guarantees complete coverage.

With respect to the accuracy and consistency objectives (f), this TOR specifies requirements to prove the subset of behaviors listed above that relate to memory safety. This includes aspects such as stack usage, memory usage, fixed point arithmetic overflow and resolution, use of uninitialized variables, cache management, and unused variables. In addition, CodeHawk will prove other memory-related properties not listed above.

E.1 Operational Requirements for Memory Safety Proof

Memory safety is usually associated with bounds checking. To establish memory safety, however, requires a substantial number of additional properties to be checked, including integer under/overflow, whether variables have been properly initialized, whether memory has not been freed, and, often overlooked, the validity of casting of operations.

MITRE Corporation has created a standard, numbered taxonomy of program vulnerabilities called the Common Weakness Enumeration (CWE). A subset of these deal with undefined memory operations, and the numbering scheme is often used as a way to classify memory safety properties. Below we enumerate the Tool Operational Requirements for CodeHawk according to our own, more coarse-grained classification of these properties that forms the basis for proof obligations, including references to the CWE's covered. It should be noted, however, that the mapping from proof obligation categories to CWE's is only approximate because of the informal definition of most CWE's. Furthermore, the analysis of individual categories of properties cannot be considered in isolation, as discharge of many of the proof obligations depends on other property classes via the inductive hypothesis (see Tool Requirements Framework for CodeHawk C Analyzer, section D).

E.1.1 Bounds Checking Requirement

Requirement E.1.1. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no memory location in the analyzed C application can be referenced (read or written) outside of its declared bounds.

The proof obligations for bounds checking include index array bounds, pointer dereferencing, and pointer arithmetic, and many of the standard library functions that write to pointers as a side effect, such as strcpy. They roughly cover the CWE's shown in Table 2.

CWE	description
118	Improper access of indexable resource (range error)
119	Improper restriction of operations within the bounds of a memory buffer
120	Buffer copy without checking size of input
121	Stack-based buffer overflow
122	Heap-based buffer overflow
123	Write-what-where condition
124	Buffer underwrite (buffer underflow)
125	Out-of-bounds read
126	Buffer over-read
127	Buffer under-read
128	Wrap-around error
129	Improper validation of array index
130	Improper handling of length parameter inconsistency
131	Incorrect calculation of buffer size
188	Reliance on data/memory layout
193	Off-by-one error
242	Use of inherently dangerous function
252	Unchecked return value
466	Return of pointer value outside of expected range
467	Use of sizeof() on a pointer type
468	Incorrect pointer scaling
469	Use of pointer subtraction to determine size
680	Integer overflow to buffer overflow
758	Reliance on undefined, unspecified, or implementation-defined behavior
785	Use of path manipulation function without maximum-sized buffer
786	Access of memory location before start of buffer
787	Out-of-bounds write
788	Access of memory location after end of buffer
805	Buffer access with incorrect length value
806	Buffer access using size of source buffer
822	Untrusted pointer dereference
823	Use of out-of-range pointer offset
824	Access of uninitialized pointer
839	Numeric range comparison without minimum check

Table 2: CWE's covered by bounds checking

E.1.2 Initialized Variable Requirement

Requirement E.1.2. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no memory location in the analyzed C application can be referenced (read) before it has been initialized.

The proof obligations for initialized variable include conditions for uninitialized variables, uninitialized struct fields, uninitialized array elements, and uninitialized regions that are read by library functions such as strcpy. These conditions roughly cover the CWE's listed in Table 3.

CWE	description
391	Unchecked error condition
456	Missing initialization of variable
457	Use of uninitialized variable
665	Improper initialization
824	Access of uninitialized pointer
909	Missing initialization of resource (memory)

Table 3: CWE's covered by initialized variable conditions

E.1.3 Valid Memory Requirement

Requirement E.1.3. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no memory location in the analyzed C application can be referenced (read or written) before it has been properly allocated, or after it has been freed or gone out of scope.

These conditions roughly cover the CWE's listed in Table 4.

CWE	description
415	Double free
416	Use after free
822	Untrusted pointer dereference
825	Expired pointer dereference

Table 4: CWE's covered by valid memory conditions

E.1.4 Null Dereference Requirement

Requirement E.1.4. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no pointer in the analyzed C application can be dereferenced (read or written) while it is NULL.

This condition roughly covers the CWE's listed in Table 5.

CWE	description
476	Null pointer dereference
252	Unchecked return value
690	Unchecked return value to NULL pointer dereference

Table 5: CWE's covered by null-dereference conditions

E.1.5 Casts Requirement

Requirement E.1.5. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that every cast variable in the analyzed C application has the properties expected of the target type.

The proof obligations for cast operations include both the numeric conversions and pointer casts, as well as casts from void. Because types are the basis of discharge for many of the bounds-checking proof obligations and casts can arbitrarily change types, it is important to verify at the place of the cast that the memory that is being cast has all of the properties expected of the target type. Furthermore, when casting from a larger numeric type to a smaller numeric type, or from a signed type to unsigned type or vice versa, it must be checked that no loss or modification of value occurs. The CWE's covered by cast conditions are shown in Table 6.

CWE	description
188	Reliance on data/memory layout
192	Integer coercion error
195	Signed-to-unsigned conversion error
196	Unsigned-to-signed conversion error
197	Numeric truncation error
681	Incorrect conversion between numeric types
843	Access of resource using incompatible type (type confusion)

Table 6: CWE's covered by cast conditions

E.1.6 Arithmetic Properties Requirement

Requirement E.1.6. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that every operation on an arithmetic variable in the analyzed C application results in a value consistent with the arithmetic type.

The proof obligations for arithmetic properties include checking for integer underflow and integer overflow and division by zero. They cover the CWE's listed in Table 7. We include the underflow/overflow conditions to enable the use of the inductive hypothesis that values have not wrapped around when used in other operations. We include the division by zero condition, because it may give rise to crashes. Note that these conditions do not include conditions related to pointer arithmetic; conditions related to pointer arithmetic are included in the bounds-check conditions.

CWE	description
190	Integer overflow or wraparound
191	Integer underflow
369	Divide by zero

Table 7: CWE's covered by arithmetic conditions

E.1.7 Other Undefined Behaviors Requirement

Requirement E.1.7. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no reference to a memory location in the analyzed C application can lead to the kinds of undefined behaviors enumerated below.

This remaining category of proof obligations includes a variety of properties that are not necessarily related, but grouped together because they are relatively few in number. They all relate to conditions that are necessary to avoid undefined behavior, yet for several of them we have not found corresponding CWE's. This may be the case because present-day compilers in such cases tend to exhibit default behavior that avoids undefined behavior, but there is no guarantee that they will continue to do so when more and more optimizations are introduced in compilers, and so we do include them.

• **Improper null termination**: Proof obligations shall be created for all string operations that rely on null termination for correct operation;

• **Uncontrolled format string**: Proof obligations shall be created for all format arguments to be string literals;

• **Shift operation arguments**: Proof obligations shall be created for all shift operations that the amount to be shifted does not exceed the size of the operand to be shifted;

• **Common base**: Proof obligations shall be created for all pointer subtraction and pointer comparison operations that both operands reside in the same memory area, and, for subtraction, in the same array.

• **No overlap**: Proof obligations shall be created for all calls to library functions that require their pointer arguments to not overlap.

• **Allocation base**: Proof obligations shall be created for all calls to free to ensure that the pointer provided points to the start of a heap allocated memory region.

CWE	description
134	Uncontrolled format string
170	Improper null termination
590	Free of memory not on the heap
763	Release of invalid pointer or reference

Table 8: CWE's covered by other conditions

F. Requirements to address the abnormal activation modes or inconsistent inputs that should be detected by the tool. These requirements should consider the impact of those modes on the functionality and outputs of the tool. (This item is not applicable to TQL-5.)

There are no requirements for the CodeHawk C Analyzer to detect inconsistency inputs because there is no opportunity for users to create inconsistent inputs for the analyzer portion of CodeHawk. The input is a complete C application, determined by a Makefile. The CIL front-end will first attempt to parse and pre-process the various source files by calling the gcc compiler. The compiler itself determines that the input is a valid and complete C application and rejects it (with standard compiler error messages) otherwise.

Discussion

The user can exercise some control over which abstract domains will be used, and in which order, during the generation of invariants. These domains are chosen from a fixed list represented by single letters:

l: linear equalities

s: symbolic sets

v: valuesets

i: intervals

A string of these letters, with possible repeats, submitted by the user instructs the abstract interpreter to propagate invariants over the program, in successive passes, using the domains specified. It is thus possible that a user could submit an input string with unrecognized characters, so we could include a requirement in this section for the CodeHawk C Analyzer to reject unrecognized characters in the parameter string (normal command line validation). It should be noted, though, that unrecognized characters (if

skipped over silently) cannot cause CodeHawk to behave incorrectly or abnormally. Any sequence of valid domain characters describes a valid propagation strategy for the abstract interpreter.

In its current (research) implementation involving seven separate shell scripts to be executed by the user in a specific order, CodeHawk has some exposure to abnormal activation if a user executes the scripts in the wrong order. This current method of open invocation was designed for the benefit of researchers, so CodeHawk itself has no specific detection mechanisms or remediations for incorrect shell execution orders. It is unlikely, however, that a product version of CodeHawk would be released with this unregulated invocation structure. The developers are already experimenting with a single Perl script interface that will automate the execution order and validation of the constituent shell scripts.

G. The applicable user information, such as a user manual and installation guide or a reference to it, if not provided as part of the Tool Requirements data.

Discussion

An actual TOR for the qualification of an actual formal methods tool would refer to the user documentation for the released version of the tool for which qualification is being sought. The CodeHawk C Analyzer that is the object of this case study was produced by Kestrel Technology as part of a DHS research project. It is not a publically released product, and has no user documentation at present.

H. Description of the operational use of the tool (including selected options, parameters values, command line, etc.).

The various modules of the CodeHawk C Analyzer are executed independently by the user from the command line on a common linking unit, with the following shell scripts:

> **runcilly.sh** creates the definition/declaration files for each compilation unit and the semantics files for each function (referred to below as ***f*.xml** and ***f/f_ff*.xml**) and the **target_file.xml** file which enumerates the files that will comprise the analyzed link unit.
>
> **chlink.sh** creates a linkunit file that contains all global definitions and declaration for the linking unit
>
> **createprimary.sh** creates primary proof obligations and initializes the secondary proof obligation files and function API files
>
> **createsecondary.sh** creates secondary proof obligations

generateinvariants.sh generates invariants for individual functions

checkpo.sh checks proof obligations and saves the evidence

report.sh reports the current status of the proof obligations

To run an analysis, the user performs the following steps: (the first step must be run in the project directory that contains the main makefile; all other steps are to be run in the project analysis directory):

1. Run **runcilly.sh** (in the directory where the main makefile is located) to create the file declaration and function semantics files.

The script creates:

- ***f*_cfile.xml**, for each file compiled

- ***f*/*f*_*ff*_cfun.xml**, for each function in ***f***

- **target_files.xml** file listing files included in the linking unit

2. Run **chlink.sh** to create the linking file.

The script reads :

- **target_files.xml** file listing files included in the linking unit

- for each file ***f*** listed: ***f*_cfile.xml**

The script creates:

linkunit.xml

The script updates:

for each file ***f***: ***f*_cfile.xml** (adds linking information)

3. Run **createprimary.sh** to create primary proof obligations.

The script reads :

- **linkunit.xml**

- for each file ***f*** in linkunit: ***f*_cfile.xml**

- for each function **ff** in **f: f/f_ff_cfun.xml**

The script creates:

- for each function **ff** in **f**:
  - **f/f_ff_api.xml**
  - **f/f_ff_ppo.xml**
  - **f/f_ff_spo.xml**

4. Run **generateinvariants.sh** to generate invariants of different types (per file **f**)

The script reads :

- **f_cfile.xml**
- for each function **ff** in **f**:
  - **f/f_ff_cfun.xml**
  - **f/f_ff_invs.xml** (if present)
  - **f/f_ff_ppo.xml**
  - **f/f_ff_spo.xml**

The script creates or updates:

- for each function **ff** in **f**:
  - **f/f_ff_invs.xml**

5. Run **checkpo.sh** to check proof obligations against the invariants generated (per file **f**)

The script reads:

- **f_cfile.xml**
- for each function **ff** in **f**:
  - **f/f_ff_cfun.xml**
  - **f/f_ff_invs.xml**

- **f/f_ff_api.xml**

- **f/f_ff_ppo.xml**

- **f/f_ff_spo.xml**

- **f/f_ff_pev.xml** (if present)

- **f/f_ff_sev.xml** (if present)

The script creates or updates:

- **f/f_ff_pev.xml**

- **f/f_ff_sev.xml**

The script updates:

- **f/f_ff_api.xml**

6. Run **createsecondary.sh** to create secondary proof obligations from the assumptions generated by **checkpo**

The script reads:

- **linkunit.xml**

- for each file **f** in **linkunit**:

  - **f_cfile.xml**

- for each function **ff** in **f**:

  - **f/f_ff_cfun.xml**

  - **f/f_ff_spo.xml**

  - **f/f_ff_api.xml**

The script updates:

- **f/f_ff_spo.xml**

Repeat steps 4, 5, 6 until no new secondary proof obligations are generated. At any point one can run **report.sh** to view the current status of proof obligations.

I. Performance requirements specifying the behavior of the tool output.

Discussion

The output of the CodeHawk C Analyzer is a report – a collection of XML files specifying which memory safety conjectures were proved (or not proved) at each program location that references memory. The output thus has no "behavior" that would be subject to a performance measure. Its size will always be linearly proportional to the size of the input C application. The runtime performance of the analyzer itself can vary widely depending on program size and complexity of the memory references. Although abstract interpretation guarantees finite termination in general, by analyzing a conservative abstraction of actual program behaviors, the actual runtime for a particular analysis may take unacceptable long, or require unacceptably large amounts of memory, to complete. The CodeHawk C Analyzer has many built-in heuristics for limiting memory size and running time, but there are no one-size-fits-all performance requirements per se, other than the requirement to terminate eventually. This is fairly typical of formal methods analyzers that attempt to prove properties about infinite sets of behaviors.

If CodeHawk runs out of available memory, it will be abnormally terminated by the operation system. If a user decides that a given analysis run is taking too much time, the user will have to terminate CodeHawk abnormally (via operating system intervention). CodeHawk will always terminate eventually, though there is no way for the user to gauge its progress. Because this is currently a research implementation, there is no guarantee of usefulness for partial XML results after abnormal termination. Researchers often do restart with partial results. An eventual product implementation would likely close-off incomplete XML items and flush write buffers on abnormal termination, if the partial results are useful for restarts. Otherwise, it would likely erase any existing XML files in the working directory before starting up.

References

[1] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In R. Nigel Horspool, editor, CC, volume 2304 of Lecture Notes in Computer Science, pages 213–228. Springer, 2002.

[2] ISO/IEC. ISO/IEC 9899:1999(E) Programming Languages - C. 1999.

[3] Henny B. Sipma. A Gold Standard for Benchmarking C source code static analysis tools: Measures and Metrics. Technical report, Kestrel Technology, LLC, May 2013.

[4] Gogul Balakrishnan and Thomas W. Reps. Analyzing memory accesses in x86 executables. In Evelyn Duesterwald, editor, CC, volume 2985 of Lecture Notes in Computer Science, pages 5–23. Springer, 2004.

[5] Michael Karr. Affine relationships among variables of a program. Acta Inf., 6:133–151, 1976.