Tool Requirements (TR) Framework for CodeHawk C Analyzer (Tool B)

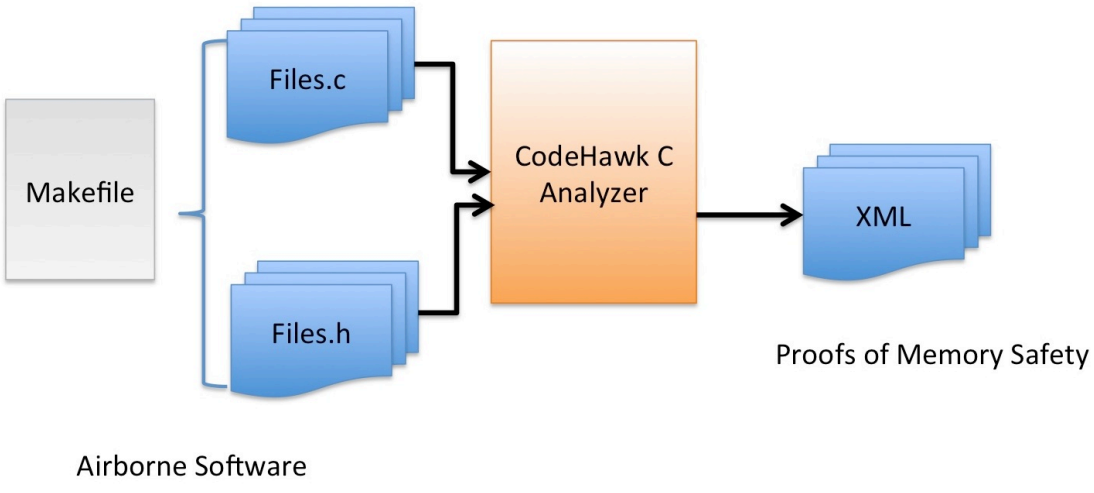
1 Introduction

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 Requirements document (TR) for this analyzer as specified in DO-330 10.2.1. 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 TR will not be relevant. This document is thus a Framework for an actual TR, organized according to the DO-330 enumeration of required contents in section 10.2.1. Accordingly, some subsections will represent content that is concrete enough to be part of an actual TR; 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 TR Framework

[From DO330-10.2.1] **The Tool Requirements describe all the tool functionality. This data should include:**

A. A description of the tool functions and technical features, including modes of operation.



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). It requires no other external tool interfaces for its normal operation.

A more detailed functional overview can be found in *Tool Qualification Plan (TQP) Framework for CodeHawk C Analyzer,* section C.

B. User instructions, installation instructions, list of error messages, and constraints. This is often packaged as a user manual. The user manual may be part of the Tool Requirements or it may be packaged in one or more documents.

Discussion

An actual TR 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. Also, because it is a research implementation, it does not yet have an externally identified accounting of error messages. In an actual qualification of an actual end-user tool, these user documentation artifacts would be reported here.

One characteristic that sets CodeHawk apart from other kinds of static analysis tools is its lack of constraints on the forms of program inputs that it can accept and analyze. Because it is an abstract interpreter defined over the full C language, it can accept and analyze programs of arbitrary complexity. Some C constructs may cause CodeHawk to produce less precise approximations than others, but CodeHawk will always produce an analysis result that is a safe over-approximation of the actual behavior of the program. So, although a real productization of the research implementation under study here would have a user manual, installation instructions, and a list of error messages to be described (or referenced) in this subsection, the CodeHawk C Analyzer would still not have any input constraints to describe here.

C. Requirements to describe the user's ability to customize the tool.

The technological core of the CodeHawk C Analyzer is a generic abstract interpretation engine designed to be customized to specific analysis tasks. A research user, with access to this core implementation, is not only able to customize the analyzer to different source languages and different properties to prove, but indeed must do so to realize a concrete analysis tool. The C Analyzer under study here, however, represents a particular customization of the core technology in which the source language (C), and the properties

to be proved (memory safety) are permanently fixed. The end of user of this tool would thus have no opportunity to further customize the input language or the properties to be proved. The CodeHawk C Analyzer does permit the user to customize the types and order of abstract domains that will be used by the abstract interpreter to compute 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. We could thus include a customization requirement in this subsection that the CodeHawk C Analyzer implement this scheme (which it already does).

D. Functional requirements, with the appropriate level of detail (see 5.2.1.2.k).

D.1 Memory Safety Requirements

The Tool Operational Requirements for CodeHawk (see *Tool Operational Requirements (TOR) Framework for CodeHawk C Analyzer (Tool B)*) are enumerated according to a coarse-grained taxonomy of memory safety properties that groups individual properties into seven classes. An important claim that CodeHawk makes as a formal methods tool, however, is that it can do a *complete* memory safety analysis of *all* memory safety properties in a C program. To establish functional requirements for such a claim, we need an enumeration of requirements that is exhaustive against some standard of "all memory safety properties." To achieve this, the CodeHawk C Analyzer enumerates its checked memory safety properties according to the C Language Standard itself [2].

This more fine-grained taxonomy of memory safety properties that we use to describe *requirements* below is derived from that standard. CodeHawk uses this taxonomy to guide its actual placement of proof obligations in the C code because it enumerates the memory safety properties from the point of view of the language definition. The C Standard contains a long list of *undefined* behaviors that may result from otherwise syntactically correct constructs. Since it is not feasible for a compiler to detect, or automatically prevent such undefined conditions, the standard allows a compiler to produce an executable object without enforcing these semantic conditions, thus producing programs with potentially undefined behaviors. The CodeHawk C Analyzer simulates an inductive proof over the transfer function that defines how each statement changes the program state. The undefined behaviors described below are only possible if the execution of some prior statements can put the accumulated memory state into an undefined condition. Accordingly, CodeHawk places formal conjectures (proof

obligations) at every statement location where such a transition could occur, stating that a defined memory state before the statement execution will be preserved by the statement execution itself. A program, all of whose proof obligations on the transfer function can be discharged, will thus be proven memory-safe if it starts execution from a well-defined memory state.

The requirements described below reference the aggregated sites, from the C Standard, where proof obligations must be discharged to rule out transition to an undefined memory state. The reference numbers following each refer to sections in the C Standard.

D.1.1 Object Use After Free

Requirement D.1.1. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no data object can be referred to outside of its lifetime (6.2.4).

Trace to TOR E.2.3: Valid Memory Requirement.

D.1.2 Pointer Use After Free

Requirement D.1.2. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no value of a pointer to an object whose lifetime has ended can be used (6.2.4).

Trace to TOR E.2.3: Valid Memory Requirement.

D.1.3 Indeterminate Use of Automatic Storage

Requirement D.1.3. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no value of an object with automatic storage duration is used while it is indeterminate (6.2.4, 6.7.8, 6.8).

Trace to TOR E.2.3: Valid Memory Requirement.

D.1.4 Unrepresentable Integer Conversion

Requirement D.1.4. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no conversion to or from an integer type produces a value outside the range that can be represented (6.3.1.4).

Trace to TOR E.2.5: Casts Requirement and E.2.6: Arithmetic Properties Requirement.

D.1.5 LValue Designates Object

Requirement D.1.5. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that every lvalue designates an object when evaluated (6.3.2.1).

Trace to TOR E.2.3: Valid Memory Requirement.

D.1.6 Function Pointer Type Mismatch

Requirement D.1.6. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no pointer is used to call a function whose type is not compatible with the pointed-to type (6.3.2.3).

Trace to TOR E.2.5: Casts Requirement.

D.1.7 LValue Type Mismatch

Requirement D.1.7. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no object has its stored value accessed other than by an lvalue of an allowable type (6.5).

Trace to TOR E.2.5: Casts Requirement.

D.1.8 Invalid Unary * Operand

Requirement D.1.8. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no operand of the unary * operator has an invalid value (6.5.3.2).

Trace to TOR E.2.3: Valid Memory Requirement.

D.1.9 Invalid Pointer Conversion

Requirement D.1.9. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no pointer is converted to other than an integer or pointer type (6.5.4).

Trace to TOR E.2.5: Casts Requirement.

D.1.10 Zero Divide

Requirement D.1.10. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no value of the second operand of the / or % operator is zero (6.5.5).

Trace to TOR E.2.6: Arithmetic Properties Requirement.

D.1.11 Invalid Pointer Arithmetic

Requirement D.1.11. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object (6.5.6).

Trace to TOR E.2.1: Bounds Checking Requirement.

D.1.12 Invalid Pointer Arithmetic Reference

Requirement D.1.12. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary * operator that is evaluated (6.5.6).

Trace to TOR E.2.1: Bounds Checking Requirement.

D.1.13 Invalid Pointer Subtraction

Requirement D.1.13. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no pointers that do not point into, or just beyond, the same array object are subtracted (6.5.6).

Trace to TOR E.2.1: Bounds Checking Requirement.

D.1.14 Array Index Out of Range

Requirement D.1.14. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no array subscript is out of range, even if an object is apparently accessible with the given subscript (as in the lvalue expression a[1][7] given the declaration a[4][5]) (6.5.6).

Trace to TOR E.2.1: Bounds Checking Requirement.

D.1.15 Valid Type for Pointer Subtraction

Requirement D.1.15. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that every result of subtracting two pointers is representable in an object of type ptrdiff t (6.5.6).

Trace to TOR E.2.5: Casts Requirement.

D.1.16 Out of Bounds Shift Operation

Requirement D.1.16. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no expression is shifted by a negative number or by an amount greater than or equal to the width of the promoted expression (6.5.7).

Trace to TOR E.2.1: Bounds Checking Requirement.

D.1.17 Invalid Pointer Comparison

Requirement D.1.17. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that all pointers compared using relational operators point to the same aggregate or union (and not just beyond the same array object) (6.5.8).

Trace to TOR E.2.1: Bounds Checking Requirement.

D.1.18 Invalid Object Assignment

Requirement D.1.18. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no object is assigned to an inexactly overlapping object or to an exactly overlapping object with incompatible type (6.5.16.1).

Trace to TOR E.2.1: Bounds Checking Requirement and E.2.5: Casts Requirement.

D.1.19 Out of Bound Pointer Array Reference

Requirement D.1.19. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no attempt is made to access, or generate a pointer to just past a flexible array member of a structure, when the referenced object provides no elements for that array (6.7.2.1).

Trace to TOR E.2.1: Bounds Checking Requirement.

D.1.20 Non-positive or Non-constant Array Size Expression

Requirement D.1.20. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no size expression in an array declaration is other than a constant expression and evaluates at program execution time to a non-positive value (6.7.5.2).

Trace to TOR E.2.6: Arithmetic Properties Requirement.

D.1.21 Overlapping Object Copy

Requirement D.1.21. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no attempt is made to copy an object to an overlapping object by use of a library function, other than as explicitly allowed (e.g., memmove) (clause 7).

Trace to TOR E.2.1: Bounds Checking Requirement.

D.1.22 Invalid Library Array Parameter Value

Requirement D.1.22. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no pointer passed to a library function array parameter has other than a value such that all address computations and object accesses are valid (7.1.4).

Trace to TOR E.2.3: Valid Memory Requirement.

D.1.23 Unrepresentable Integer Function Results

Requirement D.1.23. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that every value of the result of an integer arithmetic or

conversion function can be represented (7.8.2.1, 7.8.2.2, 7.8.2.3, 7.8.2.4, 7.20.6.1, 7.20.6.2, 7.20.1).

Trace to TOR E.2.6: Arithmetic Properties Requirement.

D.1.24 Valid Format Function Arguments

Requirement D.1.24. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing sufficient arguments for the format in each call to one of the formatted input/output functions, and that each argument has an appropriate type (7.19.6.1, 7.19.6.2, 7.24.2.1, 7.24.2.2).

Trace to TOR E.2.7: Other Undefined Behaviors Requirement.

D.1.25 Formatted Output Function Arguments Properly Terminated

Requirement D.1.25. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no s conversion specifier is encountered by one of the formatted output functions, where the argument is missing the null terminator (unless a precision is specified that does not require null termination) (7.19.6.1, 7.24.2.1).

Trace to TOR E.2.7: Other Undefined Behaviors Requirement.

D.1.26 Invalid Arguments to Formatted Input Functions

Requirement D.1.26. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no A c, s, or [conversion specifier is encountered by one of the formatted input functions, where the array pointed to by the corresponding argument is not large enough to accept the input sequence (and a null terminator if the conversion specifier is s or [) (7.19.6.2, 7.24.2.2).

Trace to TOR E.2.7: Other Undefined Behaviors Requirement.

D.1.27 No Get Buffer Use after Read Error

Requirement D.1.27. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no contents of the array supplied in a call to the fgets, gets, or fgetws function are used after a read error occurred (7.19.7.2, 7.19.7.7, 7.24.3.2).

Trace to TOR E.2.2: Initialized Variable Requirement.

D.1.28 Allocated Memory Buffers Used as Objects

Requirement D.1.28. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no non-null pointer returned by a call to the calloc, malloc, or realloc function with a zero-requested size is used to access an object (7.20.3).

Trace to TOR E.2.3: Valid Memory Requirement.

D.1.29 Use After Freed Pointer Value

Requirement D.1.29. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no value of a pointer that refers to space deallocated by a call to free or realloc function is used (7.20.3).

Trace to TOR E.2.3: Valid Memory Requirement.

D.1.30 Valid Arguments to free and realloc

Requirement D.1.30. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that every pointer argument to the free or realloc function matches a pointer earlier returned by calloc, malloc, or realloc , or the space has been deallocated by a call to free or realloc (7.20.3.2, 7.20.3.4).

Trace to TOR E.2.3: Valid Memory Requirement.

D.1.31 malloc Allocated Object Used as Value

Requirement D.1.31. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no value of the object allocated by the malloc function is used (7.20.3.3).

Trace to TOR E.2.5: Casts Requirement.

D.1.32 Out of Bound realloc Use

Requirement D.1.32. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no value of any bytes in a new object allocated by the realloc function beyond the size of the object are used (7.20.3.4).

Trace to TOR E.2.1: Bounds Checking Requirement.

D.1.33 Out of Bound Array Access by String Utility Functions

Requirement D.1.33. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no string or wide string utility function is instructed to access an array beyond the end of an object (7.21.1, 7.24.4).

Trace to TOR E.2.1: Bounds Checking Requirement.

D.1.34 Invalid Pointer Access by String Utility Functions

Requirement D.1.34. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that no string or wide string utility function is called with an invalid pointer argument, even if the length is zero (7.21.1, 7.24.4).

Trace to TOR E.2.7: Other Undefined Behaviors Requirement.

D.1.35 Invalid strxfrm, strtime, wcsxfrm, wcsftime Use

Requirement D.1.35. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that contents of the destination array are never used after a call to the strxfrm, strftime, wcsxfrm, wcsftime function in which the specified length was too small to hold the entire null-terminated result (7.21.4.5, 7.23.3.5, 7.24.4.4, 7.24.5.1).

Trace to TOR E.2.7: Other Undefined Behaviors Requirement.

D.1.36 Non-null Pointer Argument to strtok, wcstok

Requirement D.1.36. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that the first argument in the very first call to the strtok or wcstok function is a non-null pointer (7.21.5.8, 7.24.4.5.7).

Trace to TOR E.2.4: Null Dereference Requirement.

D.1.37 Valid Pointer Argument to wcstok

Requirement D.1.37. CodeHawk shall generate and attempt to discharge proof obligations guaranteeing that in each call to the wcstok function, the object pointed to by ptr has the value stored by the previous call for the same wide string.

Trace to TOR E.2.7: Other Undefined Behaviors Requirement.

D.2 Additional Requirements

Discussion

The "appropriate level of detail" referenced in the Section D heading above refers to this sentence in DO-330 5.2.1.2(k):

> *The Tool Requirements should be defined to a level of detail appropriate to ensure proper implementation and to assess correctness of the tool (for example, defining underlying models or mathematical theories).*

The memory safety requirements enumerated in D.1 above define the visible functional *results* of CodeHawk. They also serve to define the underlying mathematical theory of completeness with respect to such safety conjectures – namely, that the variety and placement of the safety conjectures is complete enough to support an inductive proof of whole-program memory safety over the semantic transfer function for the C language.

One interpretation of the 5.2.1.2(k) requirement, however, is that this level of detail should include all of the *internal* underlying models and mathematical theories of the tool as well. CodeHawk has several of these internal technologies related to the theory of abstract interpretation. The functional requirements of D.1 do not specify how the memory safety conjectures are internally proved. These could conceivably be proved by technologies other than abstract interpretation. Thus a Tool Requirements document for

CodeHawk that follows the more granular interpretation of "detail" would also need to list specific correctness requirements for its internal implementations of abstract interpretation technology.

The enumeration of such requirements, in this document, will eventually specify the test cases that will by used to verify the requirements. Since, for the purposes of this research study, we do not intend to specify additional test cases for these internal requirements, we will simply outline the kinds of internal technology requirements that might go here, in the next four sections.

D.3 Abstract Domain Requirements

Discussion

The theory of Abstract Interpretation [2] makes it feasible to analyze the behavior of a program over all possible paths by interpreting the program over abstract equivalence classes of its data values rather than the concrete values themselves. The concrete program must first be translated to an abstract version in which all concrete operations over concrete data types become operations over the abstract domains of the concrete values. Thus a program defined over real-valued variables might be abstractly interpreted over the domain of real-valued intervals. Each interval (represented by bounding minimum and maximum values) denotes all of the actual values it covers. Each such abstract domain must define operations over the domain values that mirror those over the concrete values they abstract. Addition over reals, for example, would need a corresponding operation that adds two real-intervals to produce a summed real-interval.

Requirements for the correctness of CodeHawk's underlying use of abstract domains would thus need to be specified for each abstract domain that it uses. In the case of the CodeHawk C Analyzer, these domains would be linear equalities, symbolic sets, value sets, and intervals. For each of these domains, there would be requirements specifying that their set of abstract operations cover the corresponding concrete operations of C programs, and that each operation preserves (modulo abstraction) the function of it's C counterpart.

CodeHawk reuses Ocaml implementations of these domains from open source libraries available to the research community. The requirements for these implementations (and the corresponding test cases) could thus possibly be reused from the same sources.

D.4 Abstract Translation Requirements

Discussion

Unlike the domain implementations themselves, which represent the prior, vetted work-product of others, the translation of a C program into a corresponding abstract program that uses the domain operations is a custom work-product of Kestrel Technology. There could thus be a set of requirements specifying that this translation is a faithful representation of the C semantics. An enumeration of such requirements would likely

mirror the statement types of C and specify the corresponding abstract operation target for each applicable domain.

D.5 Abstract Interpretation Requirements

Discussion

The actual (abstract) interpretation process is performed by an iterative algorithm that computes abstract domain values for every variable of interest, at every location of interest in the program, until the changes in abstract values reaches a fixpoint. The resulting abstract values represent invariant constraints on the actual values of the program's variables – i.e., the constraints are true on every actual execution path. These invariants will be used as lemmas in subsequent passes to prove the memory safety conjectures enumerated in the D.1 requirements above. Requirements for the correct implementation of this underlying technology might specify, for instance, that the iteration will always converge in finite time, and that the invariants at each location are maximally informed by all upstream invariants on which they depend. This requires a correct interpretation of the translated graphs that represent C control flow and data flow.

D.6 Proof Discharge Requirements

Discussion

The final proof discharge operations are independent of abstract interpretation technology. They attempt to prove that the invariants deposited by abstract interpretation logically imply the safety conjecture at the same location. This relies on the correct implementation of first-order proofs. Requirements for this condition would specify simply that all such derived implications are valid. The CodeHawk's prover is a custom work-product, but it is a simple implementation of a rather low-power prover, relying typically on simplification.

E. Specific requirements, if necessary, for compliance with tool operational environment.

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 run on Mac/Linux to date. If this were a TR 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 operational environment. Any requirements specific for CodeHawk's compliance to this environment would be described here.

F. Specific requirements to address the failure modes and response to inconsistent

inputs.

There are no requirements for the CodeHawk C Analyzer to respond to inconsistent 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.

Because of this externally enforced stability of inputs, the CodeHawk C Analyzer has no failure modes specific to user-triggered actions. It deals with every construct available, at some level of abstraction, in the C language specification. It may fail, as all software might, due to bugs or design flaws in its implementation, but since such possible failures can't be anticipated, there can be no specific requirements listed to address them.

Discussion

There currently are no requirements to address potential problems with user inputs and activation modes because this is a research implementation (and researches expect to bear this burden). A product version of CodeHawk would likely involve the design of a more robust user interface, and the requirements for that design would typically be listed here.

G. The expected responses of the tool under abnormal operating conditions.

In the few instances where the CodeHawk C Analyzer takes user arguments in some of its shell scripts, it does a standard validity check for type and number, and halts with error messages when these are incorrectly specified. Most of the shell scripts read from XML files written from previous shell scripts, so there is no user intervention in these.

As with all abstract interpreters, CodeHawk is exposed to combinatorial explosion in memory and computer resources. It will take whatever resources are available to complete its analysis. If it exceeds available memory resources, it will be terminated by the operating system as with all such execution situations. It may take unacceptably long to finish a result, but since it has no fixed limit on computing time, it will simply continue executing until the analysis is completed or it is terminated by the user. In the case of user termination, the incremental writing of intermediate XML files for various of the items computed during analysis provides some means of recording the current state of progress.

Discussion

As with section F above, there currently are no requirements to address responses under abnormal operating conditions because this is a research implementation. A product version of CodeHawk would likely involve the design of a more robust management of abnormal termination, and the requirements for that design would typically be listed here.

H. For a collection of tools, interface requirements between the tools within the collection.

Discussion

As mentioned in other sections and documents, the CodeHawk C Analyzer uses the CIL C pre-processor form Berkeley [1] as it's front-end. The interface between CodeHawk and CIL is via an external file written by CIL and read by CodeHawk. When used in its native research setting, it is not uncommon for CIL and CodeHawk to be run on separate machines, and even under separate operating systems (due to some problems building CIL on MacOS). An end-user productization would most likely solve these build issues and bundle the two tools so that the interface is transparent to the user, as if CIL were an integral part of CodeHawk. In that case, which is the context in which CodeHawk would be undergoing a real qualification exercise, there would be no interface requirements listed under this subsection.

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] Cousot, Patrick and Cousot, Radhia. "*Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints.*" POPL (1977) , 238-252.