Framework for Tool-Specific Information in the Plan for Software Aspects of Certification (PSAC) for CodeHawk C Analyzer (Tool B)

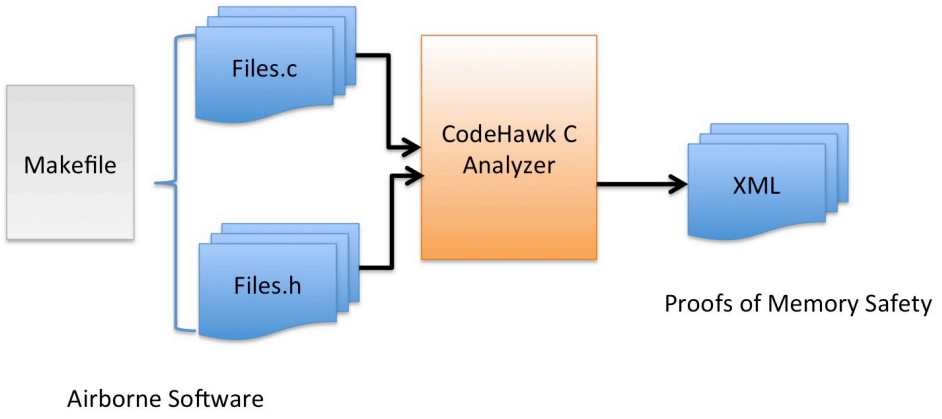
1 Introduction

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 Plan for Software Aspects of Certification (PSAC) which would include tool-specific information for this analyzer as specified by DO-330 10.1.1. 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 PSAC will not be relevant. This document is thus a Framework for an actual PSAC, organized according to the DO-330 enumeration of required contents in section 10.1.1. Accordingly, some sections will represent content that is concrete enough to be part of an actual PSAC; 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 PSAC Tool Information Framework

[From DO330-10.1.1] **The PSAC should include the following information about the need for tool qualification:**

A. Identification of the tool and its intended use, including its impact in the software life cycle process.



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.

CodeHawk's impact in the software life cycle is to assist, automate or eliminate verification and validation steps for the C application under development.

B. Details of the certification credit sought through tool use for eliminating, reducing, or automating the process(es).

B.1 DO178C Objectives

The specific certification objectives to be pursued by this hypothetical qualification exercise 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 *automate* (see DO330-10.1.1(b)) the checking of those aspects of verifiability, accuracy and consistency of C Source Code above that relate to memory safety.

B.2 Full Verifiability Credit Sought for DO178C Objective 6.3.4(c)

With respect to the verifiability objective (c) above, 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. For this reason, this PSAC would seek full credit for DO178C objective 6.3.4(c).

B.3 Partial Accuracy and Consistency Credit Sought for DO178C objective 6.3.4(f)

With respect to the accuracy and consistency objectives (f) above, the CodeHawk C Analyzer can be used 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 can be used to prove other memory-related properties not listed above. Because memory safety properties are a subset of the accuracy and consistency objectives listed in (f), this PSAC would seek partial credit for DO178C objective 6.3.4(f).

Discussion

Whenever a tool claims partial credit for an objective, it is often the case that the qualifying organization will list any additional actions required to meet that objective completely. But in this case, partial credit is sought because the tool *completely* satisfies a subset of a list of features that comprise the objective – the subset that covers memory safety. To verify accuracy and consistency for the non-memory safety features enumerated in 6.3.4(f) one would need a different tool. There are no more actions that can be taken with this tool to qualify for full credit.

C. Substantiation of the maturity and technical background of any technology or theory (for example, mathematical theory) implemented in the tool to show its applicability.

The CodeHawk C Analyzer achieves its proofs of memory safety by implementing a static analysis technique known as *abstract interpretation*. The theory of abstract interpretation was originally developed by Patrick and Radia Cousot in the 1970s [1], and has been widely studied and implemented since. This method attempts to get around the infeasibility of proving properties over all possible (potentially infinite) executions of a program by finding a suitably abstract, over-approximation of all program behaviors (that *is* feasible to compute) over which the property also holds. Since the over-approximation

of behaviors contains all actual behaviors as a subset, the proof extends to all actual behaviors.

The over-approximation is necessarily imprecise, so for some abstract results, a property that holds for all actual behaviors may not hold for the over-approximation. There is typically a trade-off between precision and computing resources. More precision (better chance of containing proofs) requires more resources. The trick is to find an abstraction precise enough for a proof that it is still feasible to compute.

Instead of using actual input values, the method interprets the program over sets of its possible input values. Abstract data value domains are used to constrain the possible values in these sets. So a program that takes integers as inputs, for examples, might be interpreted over the domain of integer *intervals* instead. Each instruction in the program that operates on integers will be abstracted to one that operates on intervals. A subtraction instruction ($A := B - C$), for instance, would yield $A = [-4,4]$ if $B = [1,7]$ and $C = [3,5]$. The interpreter walks the branches in the program's control flow graph breadth-first, propagating all data flows in parallel. Predicates on conditional branches *narrow* the population of the abstract data sets by constraining the possible values that flow through to their true and false branches. Cross edges and back edges that join multiple paths *widen* these abstract sets by forming the union of sets of the incoming branches. Loops must be iterated to a fixpoint, which typically widens the sets due to back edges.

The aim is to use an abstract domain that is precise enough to prove the property of interest. After the interpretation is finished, each <variable, location> pair of interest will have an abstract invariant expressing an over-approximation of the actual values at that point (for all executions). If the goal is to prove that an array reference can never be out of bounds, for instance, we will attempt to prove that the interval invariant on the index variable is contained within the declared upper and lower bounds of the array. It doesn't matter that the invariant may contain values that will never occur, as long as they are all contained within the bounds.

D. The TQL proposed for the tool and supporting justification.

Formal methods tools are considered criteria 2 tools for determining the Tool Qualification Level because they are generally used to prove properties, analyze source code, and generate tests. As such, they do not produce output that can be part of the system software and insert errors. This means the TQL depends on whether the certification approach will use formal analysis results to eliminate or reduce verification processes beyond those automated by the tool or development processes for the system software. For example, a formal method result that division by zero is not possible could justify absence of protection mechanisms in the system. This would be a reduction in a development process, for systems requiring Software Level (AL) A or B, the formal method tool must be qualified with TQL-4.

E. Tool source (for example, in-house, COTS, third party).

Discussion

This subsection refers to the tool source from the point of view of the organization seeking qualification credit. From the point of view of the authors of this document, simulating a qualification exercise for research purposes, the tool source would be "in house," since we are the developers of the tool. If the CodeHawk C analyzer should find it's way into an actual qualification exercise, it is likely that that it would be by license from the developers, and thus the source would be "COTS."

F. The stakeholders involved in the tool qualification and their specific roles and responsibilities, including who is responsible for satisfying specific objectives.

Discussion

Since this is a research simulation of a qualification, there is no qualifying organization to have stakeholders or roles and responsibilities to document here.

G. Description of the Tool Operational Requirements definition process (see 5.1), tool operation integration process (see 5.3), and tool operational V&V process (see 6.2) (or a reference to where these processes will be described).

Discussion

This subsection would only be applicable to an actual organization seeking to qualify a tool. As a side note, we see no reason why formal methods tools would be treated any differently by these requirements than any other tools.

H. Description of the tool operational environment in which the tool will be used.

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.

Discussion

If this were a PSAC 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.

I. If the tool qualification data is reused, identify previous applications of the tool, the strategy and justification for reuse, and any applicable re-qualification activities. In the case of a third party tool or a COTS tool, information about previous application can be provided by the supplier since it may not be available from the users of the tool. See sections 11.2 and 11.3 for reuse of previously qualified tools and COTS tools.

Discussion

This subsection is not applicable to the present research exercise, since it references the prior qualification history of a qualification seeking organization that does not exist.

J. Reference to the TQP, or if no TQP is generated (for TQL-5), reference the data to support tool qualification.

See the separate document: *Tool Qualification Plan (TQP) Framework for CodeHawk C Analyzer (Tool B)*.

References

[1] 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.