Tool Qualification Plan (TQP) Framework for CodeHawk C Analyzer (Tool B)

1 Introduction

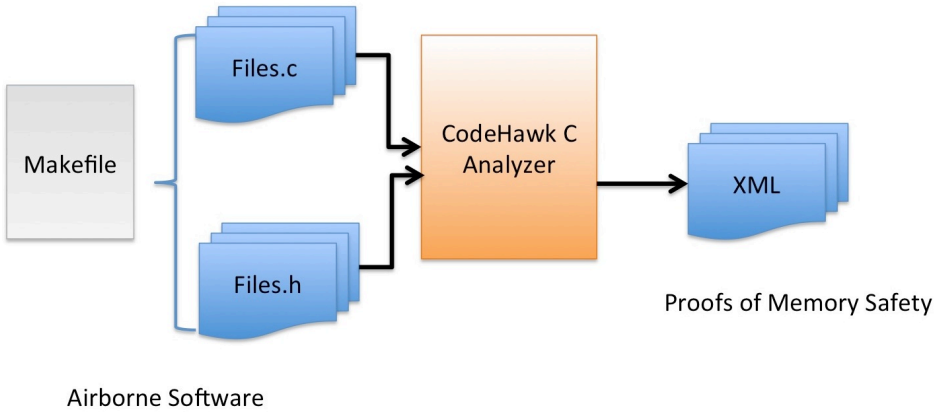
This document is part of a research Case Study simulating a Formal Methods Tool qualification exercise under DO-330, for the CodeHawk C Analyzer static analysis tool. An actual qualification exercise would produce, among other documents, a Tool Qualification Plan (TQP) for this analyzer as specified in DO-330 10.1.2. 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 TQP will not be relevant. This document is thus a Framework for an actual TQP, organized according to the DO-330 enumeration of required contents in section 10.1.2. Accordingly, some sections will represent content that is concrete enough to be part of an actual TQP; 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 TQP Framework

[From DO330-10.1.2] **For a tool to be qualified at TQL 1, 2, 3 or 4, the TQP should describe the tool qualification process. This plan should include:**

A. Identification of the tool, and, if applicable, user configuration.

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.

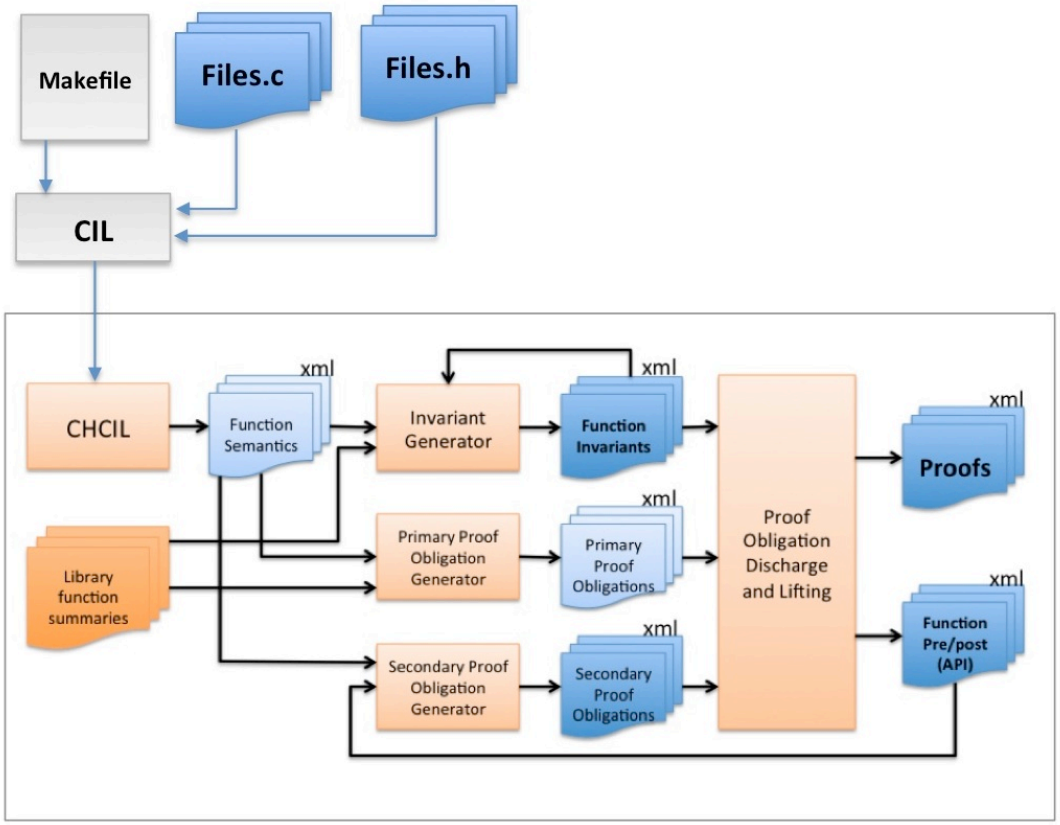


B. Qualification considerations, including the proposed TQL and means of compliance with the objectives of this document.

We will qualify the CodeHawk C Analyzer to TQL-4, as the activities are a superset of those required for any official qualification of a formal method tool. The Case Study Preparation report outlines the approach the case studies will use to meet the requirements of the objectives relevant to TQL-4. Tables T-0 through T-10 are present with explanations for each objective. Other than the objectives/activities that do not apply to TQL-4, some that relate to agreement from the certification authority (T-10 items 1 and 2) or the tool lifecycle (T-9 items 2 and 5, T-10 item 3) are ruled out of scope for these studies.

C. A functional overview of the tool, its interfaces, and its architecture. Additionally, any external components should be identified.

The CodeHawk C Analyzer is implemented in Ocaml as a collection of modules that communicate with each other via XML files, as shown in Figure 1



Below we describe each of the components in some more detail.

C.1 CHCIL: Parsing and Linking

A C application consists of a collection of source files combined with instructions on how these source files are to be compiled, usually in the form of a Makefile. To analyze a C application, the source files must be parsed according to the instructions provided in the Makefile and all global data structures and global variables in the individual source files must be linked to a shared set of data structures and global variables.

CodeHawk uses CIL [1] as a parser front end to parse the individual source files. CIL, in turn, uses the project Makefile to identify all source files that belong to a linking unit and uses the gcc preprocessor to expand all macros and include the header files. For each C file, CIL creates a data structure that is a complete semantic representation of the source file. The CodeHawk C Analyzer converts this data structure into a set of xml files that form the basis of all further analysis.

The declaration of shared global variables and data structures is typically duplicated in many files to enable independent compilation of individual source files. Before analysis, these global variables and data structures must be linked. For global variables, this entails creating a mapping between each external global variable declaration and its corresponding, unique definition, based on the variable name. Linking of data structures is more difficult because it cannot be based on name equivalence, but must be determined by structural equivalence. Determining structural equivalence is complicated by the possibility of circular dependencies introduced by pointers to other data structures [1].

All shared global variables and data structures are collected in a single file and each given a unique identifier. The individual source files maintain their local copies of these variables and data structures and a mapping is created from these local copies to their shared counterparts. This arrangement allows the decoupling of the local analyses, performed per source file, from the global analysis that is performed at the application level, and enables a more modular analysis.

C.2 Generation of Primary Proof Obligations

Primary proof obligations represent the necessary conditions that must be true at given locations to ensure absence of undefined behavior. They are generated for each program construct that requires preconditions for its action to be defined [3]. Primary proof obligations are fully determined by the semantic constructs and generated once for each function in each source file. Each primary proof obligation is given a unique identifier and associated with an AST-based semantic path expression that uniquely identifies the statement or expression to which the proof obligation applies.

C.3 Standard Library Function Summaries

As described in [3], primary proof obligations are not only generated for given language constructs, but may also be required for certain standard library functions that are called by the application. The CodeHawk C Analyzer has a collection of 250 summaries of standard library functions. These summaries include a specification of the preconditions that must be true for the function to be invoked safely, which are used to generate the

corresponding primary proof obligations for the function call, instantiated with the arguments passed to the function. The function summaries also provide post conditions that represent constraints on their return values and side effects that describe how the environment changes as a result of the call, related to pointer variables passed to the function and the global environment. Standard library function summaries are made available in XML, organized by the header files in which they are defined.

C.4 Invariant Generator

Discharging proof obligations requires knowledge about the state of the computation at the location of the proof obligation, that is, invariants. Invariants are statements about the values of variables and relationships between variables at a particular location. The CodeHawk C Analyzer uses the CodeHawk Abstract Interpretation Engine to generate these invariants. It requires the semantic representations of the functions to be translated into the internal language of the Abstract Interpretation Engine (CHIF). The CodeHawk C Analyzer generates invariants in four domains: linear equalities [5], intervals, symbolic sets, and value sets [4]. Invariants are extracted from the engine and associated with proof obligations based on control-flow context information kept with each invariant location.

Invariants are generated for individual functions. Context information is provided by function summaries of functions called in the form of postconditions and side effects. Furthermore, the invariant generation process is incremental. After each round of invariant generation, the invariants are stored in XML files. Each new round loads the current invariants and uses them as a starting point for the new cycle of invariant generation.

C.5 Proof Obligation Discharge and Lifting

Proof obligation discharge is performed by a separate component that reads both the proof obligations and the invariants and connects them via the AST-based path expressions associated with each. Proof obligations can be discharged by determining their validity in three different ways, as described below.

Statement Validity A proof obligation is statement-valid if its validity can be determined solely based on the statement itself, program constants, and type declarations. In particular, this method of proof discharge does not require any invariants. If a proof obligation is found to be statement-valid, an explanatory message is saved in the evidence file and the proof obligation is closed.

Function Validity A proof obligation is function-valid if it is implied by the invariants generated for the location of the proof obligation. If so, the proof obligation is saved in the proofs file, with indication of the invariant used to establish “function validity”.

Context Validity A proof obligation is conditionally valid if, using invariants, the proof obligation can be propagated back to the function API or to a return value from the call to another function. This is possible only if all variables involved in the proof obligation can be shown to have a linear relationship to an external value such as a parameter or a function call return value. If this is the case, the proof obligation is saved in the proofs

file as “conditionally safe”, accompanied by the assumption on the API or return value on which it relies for its validity. It also saves the assumption as a precondition (in case of an assumption on a function parameter) or a postcondition (in case of an assumption on a function return value) in a separate assumption file, to be used to generate secondary proof obligations on the corresponding callers and callees.

C.6 Secondary Proof Obligation Generator

This component takes as input the function semantics of a function and determines for all its callees that are application functions, whether these functions imposed any assumptions (preconditions) on its arguments. If so, these preconditions are localized to the calling function and applied to the arguments with which the function is called resulting in a secondary proof obligation that must hold at the call site. This secondary proof obligation is then saved in a separate file, to be included in the next proof obligation discharge step.

C.7 External Components

Discussion

As detailed in C.1, the CodeHawk C Analyzer employs an external, open source application, CIL, which itself calls the gcc compiler. In an actual qualification exercise, the qualifying organization would have to decide whether these two external components are to be qualified as part of the total CodeHawk package, or whether they depend on, and thus require, their own independent qualifications. In the latter case, CodeHawk would have to be qualified against the specific version of CIL and gcc that have been independently qualified. Since CodeHawk is not yet a product, this decision is currently hypothetical. When it is productized, it is likely that a specific version of CIL will be bundled into the final set of executables, and thus that requirements and test cases specific to CIL would be part of the qualification exercise for CodeHawk. It is also likely that gcc will *not* be bundled in with CodeHawk. CodeHawk will default instead to the version available in the user’s environment, as is the industry custom. In that case, the qualifying organization would need to separately qualify their installed version of gcc along with the qualification of the installed version of CodeHawk.

D. Description of the tool operational environment(s), and if different, the tool verification environment(s).

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 TQP 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.

E. Description of the means the applicant will use to provide the certification authority with visibility of the activities of the tool life cycle processes so tool reviews can be planned.

Discussion

This subsection describes characteristics of the organization that is applying for qualification. It is not applicable for this research exercise because there is no such organization.

F. Tool life cycle description and the qualification activities to be performed.

We will qualify the CodeHawk C Analyzer to TQL-4, as the activities are a superset of those required for any official qualification of a formal method tool. The Case Study Preparation report outlines the approach the case studies will use to meet the requirements of the objectives relevant to TQL-4. Tables T-0 through T-10 are present with explanations for each objective. Other than the objectives/activities that do not apply to TQL-4, some that relate to agreement from the certification authority (T-10 items 1 and 2) or the tool lifecycle (T-9 items 2 and 5, T-10 item 3) are ruled out of scope for these studies. The requirement in this subsection raises no special considerations for CodeHawk in particular, or formal methods tools in general.

G. Tool qualification data to be produced.

Discussion

This subsection is specific to an actual tool qualification. It is not applicable to this research exercise.

H. Any additional considerations that may affect the qualification process, for example, deactivated code, COTS tools, reuse, tool qualification (of other tools used to develop or verify the tool), alternate means of qualification, tool service history, and means to ensure the determinism of the tool per the last paragraph of section 2.0 of this document.

Discussion

These considerations would typically arise from the context of a released tool in an actual qualification setting, and are thus underdetermined for this hypothetical case study.

It is worth noting, however, that the CodeHawk C Analyzer would not need any means to ensure determinism per the referenced paragraphs in section 2.0 above. The operable sentences from that section are these:

For a tool whose output may vary within expectations, it should be shown that the variation does not adversely affect the intended use of the output and that the correctness of the output can be established. However, for a tool whose output is part of the software and thus could insert an error, it should be demonstrated that qualified tool functions produce the same output for the same input data when operating in the same environment.

CodeHawk's output is an analysis of memory safety, and thus does not constitute part of the avionics application software. But the CodeHawk C Analyzer is strictly deterministic in any case. The input is a C application and a requested order and variety of abstract domains to be used in analysis. The properties to be proved are built in, and the abstract interpretation and proof algorithms are themselves deterministic. So the same C application with the same domain order specification will always yield the same analysis. Different domain order specifications may yield different analyses for the same C program, but all of these analyses will be sound, over-approximations of the C behavior.

I. Organizational responsibilities within the tool life cycle processes.

Discussion

This subsection is specific to an actual tool qualification. It is not applicable to this research exercise.

J. If suppliers are used, a means of ensuring that supplier processes and outputs will comply with approved tool plans and standards

Discussion

This subsection refers to suppliers of tool verification/qualification services contracted for by the organization applying for qualification. Because this research case study is

hypothetical in nature, there is no actual qualifying organization (and thus no service suppliers).

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.