RTT-STO:
Source-To-Object Analyzer

Effective source-to-object code (STO) analysis for safety-critical avionics software

RTT-STO is a software analysis tool-suite that automatically performs static program analyses of C code and assembly required to receive certification credit for source-to-object code validation in the context of safety-critical avionics software.

For more information visit

www.verified.de/products/rtt-sto

Verified Systems International GmbH
Am Fallturm 1
28359 Bremen
Germany

Source-Code-To-Object-Code Traceability Analysis for Avionics Software

For safety-critical systems software verification, many activities are performed on source code level. As a consequence, the validity of these verification results depends on the consistency between source code and object code. In some application domains, this issue is addressed by utilizing validated compilers. This approach, however, is not accepted in the avionics domain. The current standard for software development for airborne systems, RTCA DO-178C, states clearly that any automation tool applied in the development or verification process can only be qualified for the specific target system under consideration [1, Chap. B-1]. For compilers, the standard requires an approach where the object code produced is verified by means of tests and analyses, so that a qualification of compilers is not necessary; [2, Chap. 4.4.2] states that:

“Upon successful completion of verification of the software product, the compiler is considered acceptable for that product.”

To support this approach, the standard requires to perform various verification activities which show that the executable object code complies with the high-level and low-level requirements, that it is robust with respect to these requirements, and that it is compatible with the target computer [2, Chap. 6.4]. In order to show that the requirements-driven tests performed during the verification activities suffice, a structural coverage analysis has to be performed [2, Chap. 6.4.4.2]. Structural coverage analysis detects whether some code structures or interfaces have not been exercised during testing. This code has to be removed if it does not contribute to the realization of the requirements, or it may lead to refined tests or analyses, if the requirements-driven tests performed so far had been too coarse-grained to exercise all case distinctions reflected by the uncovered code.

For software of the highest criticality – this is Design Assurance Level A (DAL-A) – additional analyses have to be performed on object code level; this activity is called Source-Code-to-Object-Code (STO) Traceability Analysis. Its main objective is to verify that any additional object code which has been generated by the compiler but is not directly traceable to the source code does not introduce any errors and has been adequately covered by tests and/or analyses [2, Chap. 6.4.4.2 b.].

STO analysis certainly is a non-trivial task, because in principle, compilers may add, delete, or morph code during compilation, and the need for STO analysis imposes a significant workload on developers of airborne software systems.

STO Traceability Analysis using RTT-STO

The RTT-STO tool-suite automatically performs four different analysis passes in order to show that compilation has not introduced any problems:

- **Branching analysis** compares the control flow implemented in source code and object code and detects deviations between these two program representation. It turns out that compilers frequently add branches on object code level, which implies that additional tests have to be performed in order to achieve 100% assembly branch coverage.
- In some situations, compilers replace seemingly simple operations in source code by calls to built-in functions. For example, a 64-bit integer division on a 32-bit PowerPC platform has to be emulated by a sequence of instructions. Compilers may then call a built-in function, rather than inserting the sequence of instructions. **Hidden library function analysis** automatically detects such situations, which warrant additional verification in order to receive certification credit.
- The **memory allocation analysis** checks whether the object code contains data allocations (on the heap, on the stack, or using registers) where the size of the allocated memory region does not conform to the size expected from the type declarations in the source code.
- A quite subtle observation is that an erroneous compiler may have inserted undesired store operations targeting some memory addresses. Since requirements-based tests typically only examine the effects of desired store operations in the expected results — but not all possible alterations of the memory state — such malicious behavior is likely to be missed during testing. The **store analysis** provided by RTT-STO analyzes all memory accesses implemented in the object code and traces these accesses back to source code, which guarantees the absence of the aforementioned malicious store operations.

Together, these analyses cover all STO analysis requirements defined by the RTCA DO-178C standard, and as interpreted in the DO-178C guide [3]. It is important to stress that the analysis method provided by RTT-STO is associated with a verification workflow that has been approved by certification authorities for the verification of DAL-A software. Of course, RTT-STO can be qualified for such projects.

Verification Project with RTT-STO

All in all, a verification project conducted using RTT-STO consists of five tasks overall. First, the code base is imported into the tool, which is referred to as the “preprocessing phase”. During this preprocessing phase, the tool analyzes the code base and extracts data that is used among all following analysis passes. Then, the four analysis passes, which have been described before, are performed one after another. This section sketches the workflow of a typical STO verification project using RTT-STO.

After program startup, a project wizard guides tool-users through the preprocessing phase. The setup is required so as to configure the tool with respect to the build process. For example, the directories that contain the source code and object code have to be configured, the compiler used has to be chosen from a list of supported compilers, and the build flags used during the project have to be defined. The Illustration 1 shows the preprocessing configuration in RTT-STO. Once finished, the same preprocessing configuration is reused for all following analysis passes.

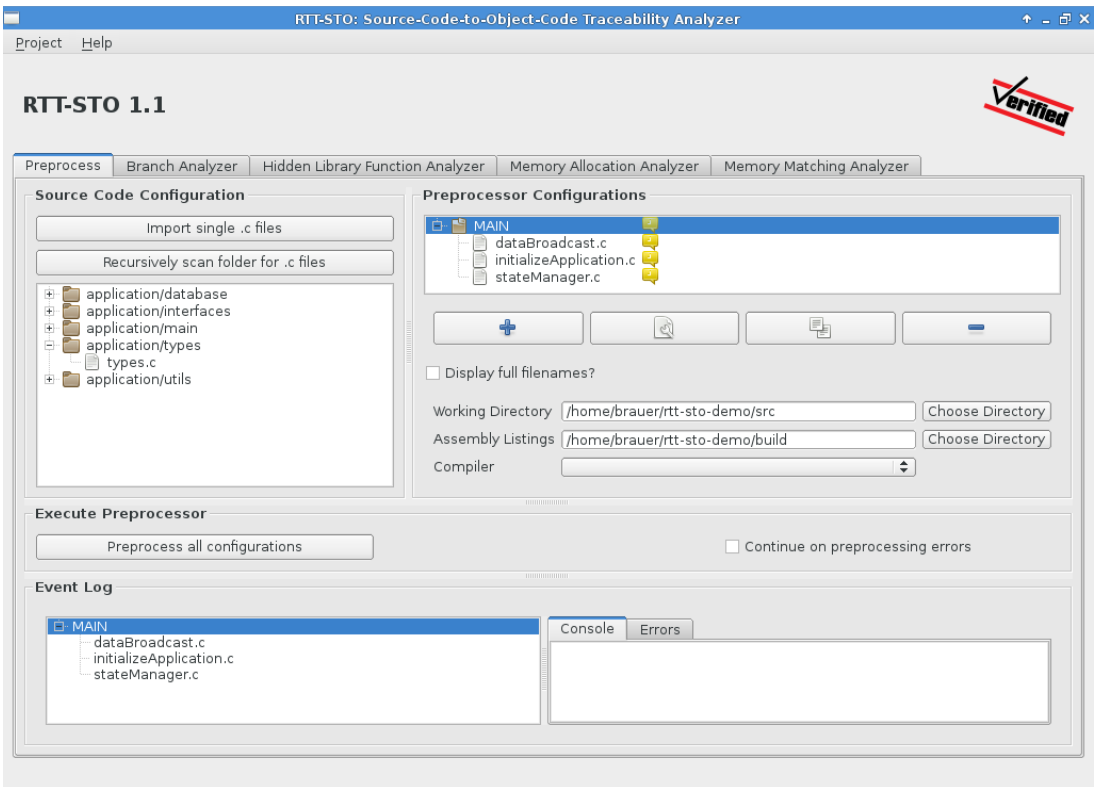

Illustration 1: Configuration dialog of RTT-STO

The configuration of each following analysis is extremely simple, as the configuration just consists of selecting the files that shall be analyzed. The analysis are then performed automatically, without any manual intervention required. Illustration 2 shows the results of the hidden library function analysis. The tool explicitly highlights which statement in C-code has induced which in object code, and both representations can be examined directly in RTT-STO. In the example, RTT-STO has detected that the compiler has mimicked a 64-bit unsigned integer division via a call to a built-in function called __gh_udiv64. The compiler has thereby introduced additional control flow.

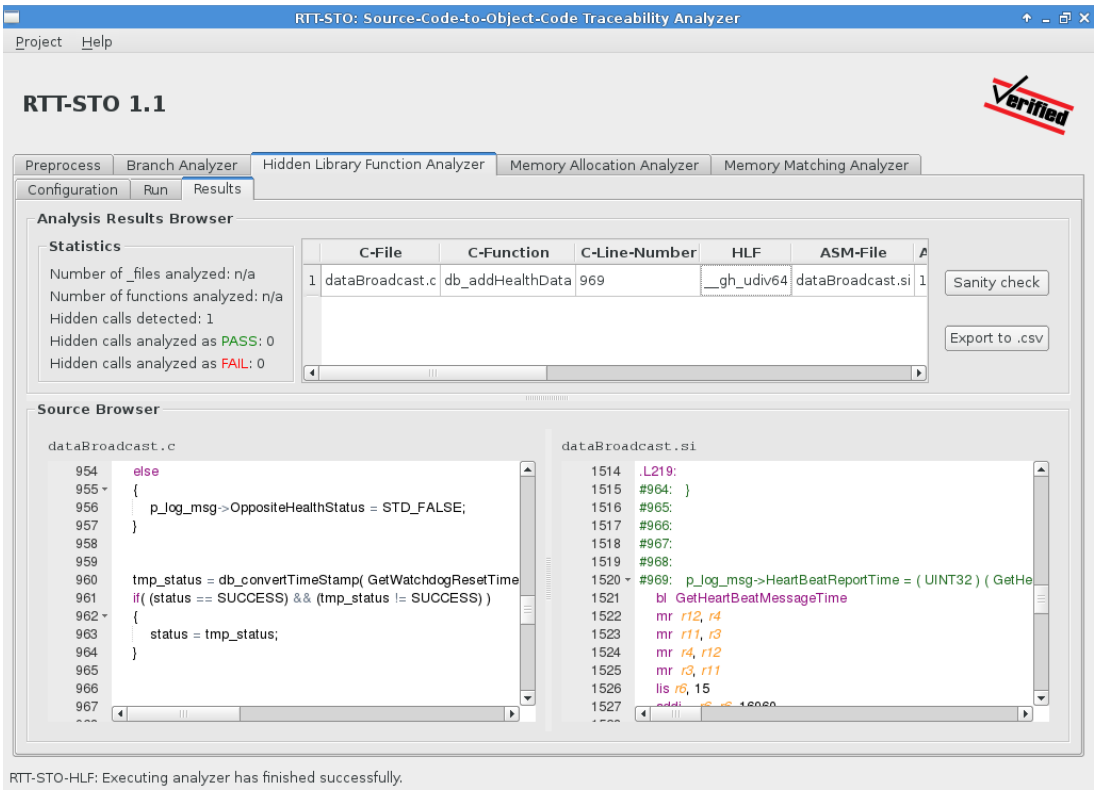

Illustration 2: Verification of hidden library function calls in RTT_STO

Likewise, Illustration 3 shows the tool outputs for store analysis (also referred to as memory matching analysis). For each memory access, RTT-STO establishes traceability between the store operation in the object code (as shown on the left-hand side) and the source code fragment that induces the store operation (as shown on the right-hand side). The results table indicates these code fragments as well as the variable symbol that is accessed. For typical projects, RTT-STO automatically proves correctness for more than 90% of all memory accesses.

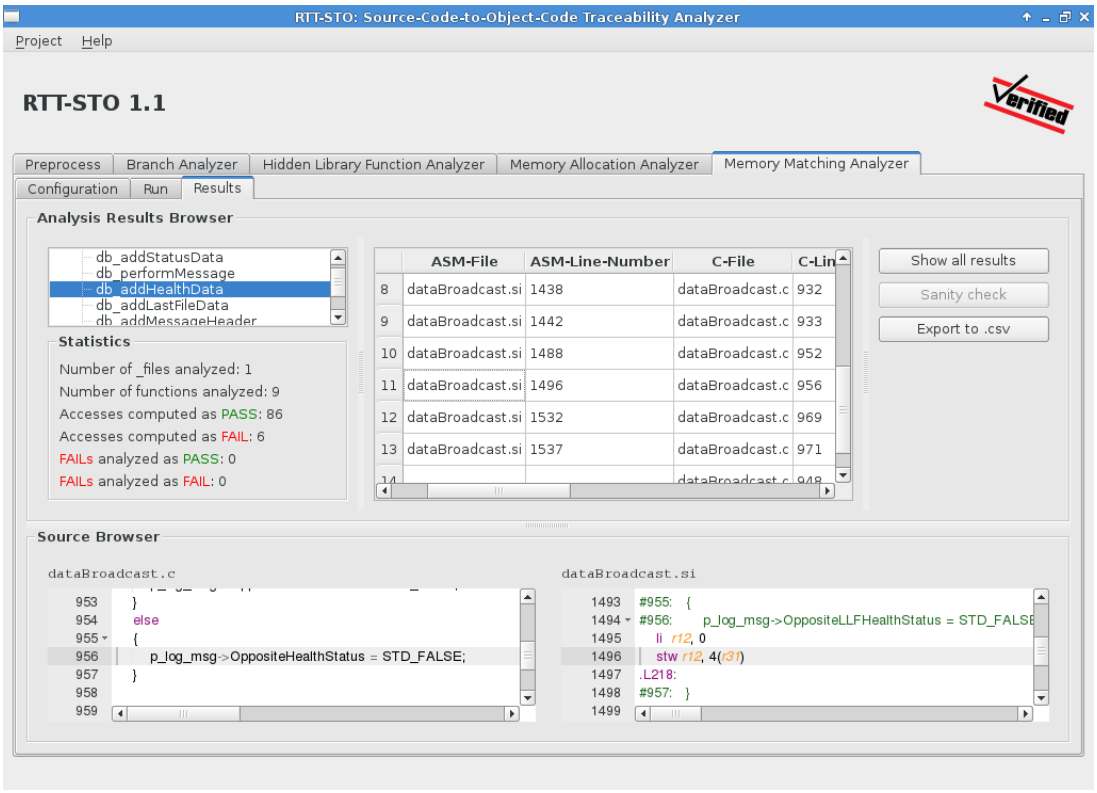

Illustration 3: Verification of stores in RTT-STO

Conclusion

RTT-STO outputs a results spreadsheet for each analysis pass with detailed verification results. The analyses are designed as conservative program analyses, which means that they output a verdict “FAIL” whenever the tool cannot guarantee that the analyzed item is correct. However, RTT-STO outputs detailed information about the parts of the code base that have caused verdict “FAIL”, so that the outputs guide the manual verification efforts. Instead of having to scan for those code fragments that need to be verified, verification engineers are provided with a detailed list of code fragments that need to be examined. This approach significantly reduces the human workload, and thus the cost, for verification.

Traditional approaches to STO traceability analysis manually examine a small fragment of the overall code base, whereas RTT-STO covers the entire source code and object code. The verification data that is presented to the certification authorities is thus much stronger because RTT-STO generates complete verification evidence.

RTT-STO ships with a custom graphical user interface that guides tool-users through the workflows and runs on both Windows 7 and Linux (CentOS 7 64-bit).

Contact

Verified Systems company has more than 15 years of experience with certification-related services for the avionics domain and offers a wide variety of verification and testing services beyond STO traceability analysis. Please contact us via info@verified.de.

References

[1] RTCA SC-205/EUROCAE WG-71: Software Considerations in Airborne Systems and Equipment Certification. No. RTCA DO-178C, RTCA, Inc.
[2] RTCA SC-205/EUROCAE WG-71: Software Tool Qualification Considerations. No. RTCA DO-330, RTCA, Inc.
[3] Rierson, Leanne: Developing Safety-Critical Software. CRC Press (2013)