Proceedings of the SDR 11 Technical Conference and Product Exposition, Copyright © 2011 Wireless Innovation Forum All Rights Reserved

THE BENEFITS OF STATIC COMPLIANCE TESTING FOR SCA NEXT

James Ezick (Reservoir Labs, New York, NY 10012; ezick@reservoir.com)
Jonathan Springer (Reservoir Labs, New York, NY 10012; springer@reservoir.com)

ABSTRACT

The next generation of the Software Communications Architecture (SCA) specification (SCA Next) introduces several features that will affect the compliance certification process. Among the more significant and high-profile changes are the introduction of multiple supported platform models and support for both CORBA® and non-CORBA transport layers. These features, which further abstract the specification away from a uniform hardware/software interface, make constructing an all-purpose dynamic testing platform more difficult. This suggests that a certification plan that relies more heavily on static testing might provide a more versatile and cost-effective approach. In this paper, we describe the benefits and implications of static compliance testing in the context of SCA Next. This includes a discussion of which requirements are and are not amenable to static testing, the complexity of defining and customizing tests, the expected performance and limitations of those tests, and a summary of our experiences from the development of R-Check™ SCA, our platform for static SCA 2.2.2 compliance testing.

1. INTRODUCTION

Static testing provides power and versatility by directly testing the software source code, across potentially multiple file formats, as it is written. This unbiased inspection permits simultaneous testing of all software paths and can be used to find latent issues that do not manifest on particular platforms or in scripted test executions. Further, static testing can be run as code is being developed, allowing non-compliant code to be fixed earlier in the development cycle. This immediate code-test-repair cycle also provides a mechanism to educate the developer on the SCA by making direct links between lines of source code and specification directives. Compiler-driven static analysis mirrors the build process and scales to millions of lines of code. Advances in static analysis and model checking techniques allow precise specification of error conditions in terms of well-defined operations that limit the rate of false positives and, in some cases, can provide correctness guarantees.

R-Check SCA [1] is a compliance testing tool being developed in partnership with the Joint Tactical Radio System (JTRS) Test & Evaluation Lab (JTEL) [2] that uses static source code analysis to check requirements contained in the SCA 2.2.2 specification. R-Check SCA uses a compiler-grade static analysis engine combined with off-the-shelf tools and data formats to test SCA-specific requirements that cut across C/C++ source code, CORBA IDL, and SCA XML descriptor files and generate concise, reproducible incident reports. R-Check SCA has been used to analyze production waveforms and operating environments and is now being expanded in anticipation of supporting SCA Next.

2. STATIC ANALYSIS

Static analysis refers to analysis performed by inspecting a program source or binary code without requiring the code to be executed. For source code static analysis, this also implies that the code does not need to be compiled to machine-dependent object code or linked. Static analysis methods provide several advantages that make them a useful complement to traditional dynamic (runtime) testing.

- Static methods are not influenced by common vs. exceptional case behavior and analyze all program paths without bias.
- Since they do not require the code to be compiled or executed, static methods can be applied to code in an intermediate (potentially incomplete) state.
- Static methods can be integrated into development environments and provide a foundation for automated, reproducible tests.
- Static methods can frequently produce counter-examples from violations and be used to pose "what-if" questions to aid code comprehension and perform hypothesis testing.

For the SCA, these advantages translate to a system of analysis methods that can be used to automate testing of several specification requirements. Through integration with a visual development environment, such as Eclipse [3],

analysis can be fast and transparent enough to execute with each source file save operation. The advantage to the SCA developer is near instantaneous feedback when a potential violation is introduced. Even while the code is not yet ready for compilation and runtime testing, and without the need to construct any unit or regression test cases or specialized test harness, the seeds of SCA violations can be found and reported. Again, through integration with a development environment, errors can be directly linked to SCA reference material that provides a concise and up-to-date description of each reported issue. Taken together, these capabilities allow potential bugs to be found and corrected at the *soonest possible point* in the development cycle through direct action by the responsible developer.

In implementation, the term *static analysis* encompasses everything from search-and-inspect type analyses, to compiler-driven data-flow methods, to more advanced symbolic execution and model checking techniques. These techniques vary greatly in scalability, power, and precision. Search-and-inspect techniques, which have been used by JTEL for SCA 2.2 and 2.2.2 compliance testing, scale to enterprise code, but have limited power and precision. The most modern techniques are extremely powerful and precise, but are frequently limited in applicability or require expertise in formal methods to apply.

With R-Check SCA, we have taken a middle ground that uses a compiler-grade language parser for C/C++ that breaks code down into intermediate data structures and then uses well-established data-flow methods of analysis to perform requirements checking. The intent in choosing this approach was to increase automation and analysis speed while improving on the power and precision of the current forms of analysis being used and still meeting the requirement that code should be analyzable "as is."

Data-flow methods date to the 1970's with the discovery that facts that may or must hold at each program point can be computed as a fixed-point of a system of equations defined over a lattice of fact subsets [4]. Later work demonstrated how these techniques could be abstracted to support reasoning over paths through reused or recursive procedures [5]. As they are commonly applied, these techniques function over a control-flow representation of a program, in which nodes are program statements and edges reflect the statement successor relation. Each statement can be seen as transforming the "state" of some fact of interest (e.g., is a variable live). In this way, a set of states that hold at an initial point in the code are propagated through the graph and "solved" for each program point, yielding the set of states that may or must (depending on how the transformers are defined) hold at each program point. The exact implementation depends on the requirement being checked, but in many cases the analysis can be restricted to single functions. For cases where the analysis must span multiple files, summary information can often be collected and used to seed the analysis for other files.

These methods are the foundation of complier-driven optimization and are commonly used to optimize data and control flow, simplify redundant computations, and, in some cases, detect basic errors of usage such as using a variable before it is defined. The compiler front end used by R-Check SCA naturally creates the intermediate data structures necessary to implement these analyses. To meet the requirement of checking incomplete code, we have extended the C/C++ language parser to ignore or infer missing information rather than halt and report a parsing error as a compiler would. However, even with the partial-code capability, the resulting underlying representation and accessing API remains the same.

3. LESSONS FROM SCA 2.2.2

Depending on how a requirement is written, static testing can vary from straightforward to very difficult. This section describes experience with three illustrative examples from the SCA 2.2.2 specification for applications [6].

1. **AP0603**, from Section 3.2.1: *"Applications shall be limited to using the OS services that are designated as mandatory in the SCA Application Environment Profile (AEP) (Appendix B)."*

The core of this requirement is identifying procedure calls in the code and determining if they refer to IEEE POSIX® routines designated as NRQ in Appendix B. Although this requirement is straightforward, there are several nuances that must be addressed to provide a complete evaluation of compliance.

- Preprocessor directives must be properly interpreted, including macro expansions and the inclusion or exclusion of conditionally compiled code.
- The analysis must recognize procedure calls in context, or, to detect calls through pointers, address-capturing call references.
- The analysis must be able to distinguish between calls that refer to POSIX routines as opposed to locally redefined calls (which are permissible under the SCA).
- Ideally, the analysis should be restricted to those parts of the code (components) that are subject to the Appendix B AEP restrictions.

A compiler-driven data-flow approach is able to address these nuances directly by functioning over an intermediate representation. Preprocessor directives are executed as part of the parsing process. The context of each code construct is

readily accessible through API calls and traversal functions over the intermediate representation. Meta-information attached to procedure calls includes declaration information (typically used by a compiler for error reporting), so distinguishing between POSIX and non-POSIX references can be done in constant time. Type information, accessible through symbol table lookup, can be used to restrict the analysis to particular SCA elements.

As a data-flow analysis, this requirement does not require a complex fact transition system – validity with respect to the requirement can be determined from an independent evaluation of each program statement. Thus, evaluation of this requirement is dominated by the time to parse the code (shared across all requirement analyses) and the time to test library-declared procedure calls against the list of proscribed NRQ function names.

2. **AP0604**, from Section 3.2.1.1: *"Applications shall perform file accesses through CF file interfaces."*

While a straightforward statement, static testing of this requirement for C/C++ code requires an enumeration of the methods of file access that are not allowed.

- For C, this includes several standard system calls, for example, `open()`, `close()`, `fread()`, and `fwrite()`. Note that POSIX calls such as `fprintf()` are covered under the AP0603 requirement.
- For C++, this includes derivatives of the file stream classes: `ifstream` and `ofstream`. While these are classes in C++, they are usually used as interfaces through class inheritance. Any method that overloads the file access methods is potentially violating.

In addition to the API and traversal functions over the program intermediate representation leveraged in the analysis for AP0603, a proper analysis for this requirement depends on domain knowledge of the C and C++ languages. It is worth noting that there is no direct path from the text of the requirement "shall" statement to an automated implementation. However, as with AP0603, once the list of proscribed actions has been defined, the analysis can be performed with a trivial fact-transition system.

3. **AP0075**, from Section 3.1.3.1.2.5.2.3: *"The releaseObject operation shall release all internal memory allocated by the component during the life of the component."*

While a simple statement of an intuitively desirable property, requirements such as this – essentially a restriction against leaking memory – represent the most difficult class of problems (in the strictest sense, undecidable) for static analysis tools. The complexity of analyzing this requirement stems from the fact that correctness depends on a precise understanding of exactly which statements will execute in a program and in which sequence.

As a tractable first approximation of a static analysis for this requirement, allocation operations for which no corresponding deallocation statement can be found are treated as suspect and passed back the developer. The developer must then assert that the code is non-violating. In practice, this approach can have a false-positive rate (based on feedback from developers) of more than 50%.

In addition to generating false-positive results, this process also has the potential, depending on the precision with which the matching step is carried out, of generating false-negative results. A false-negative result occurs when the test fails to detect an error that is within the scope of the requirement being tested. For C/C++, there are multiple examples of how this can happen.

Ex 1: Memory Leaks through Pointer Reassignment

```
Component::method_a() {
     p = malloc(…);
     …
     p = malloc(…);
}

Component::releaseObject() {
     free(p);
}
```

Second `malloc()` leaks memory allocated by first `malloc()`.

In this first example, the memory allocated by the first assignment to "p" would not be `free`'d by the `releaseObject()` method. This is a violation of the requirement. While the two `malloc()` statements are co-located in the example for brevity, they may not be so co-located in actual codes. A simple matching algorithm would not detect this violation.

Ex 2: Memory Leaks through Control Flow

```
Component::method_a() {
     if (A) {
          p = malloc(…);
     }
}

Component::releaseObject() {
     if (B) {
          free(p);
     }
}
```

If "`A`" evaluates to true, but "`B`" does not, then memory allocated by `malloc()` will be leaked.

In this second example, the memory allocated by the assignment to "`p`" is dependent on condition "`A`". The release of the memory is dependent on condition "`B`". The code is correct if and only if "`A`" implies "`B`"; that is, "`B`" is true whenever "`A`" is true. More specifically, "`B`" must be true when the "`B`" conditional *is reached* whenever "`A`" was true when the "`A`" conditional *was reached*. This distinction is important, as it reflects that either "`A`" or "`B`" or both might depend on variable assignments that could change during the life of the object.

These example program statement patterns occur frequently in submitted code. More complicated examples can be constructed that introduce leaks through looping constructs (e.g., `while`) or through combinations of reassignment and control flow. **Note, however, that it is impossible to catch instances of these more complex control-flow issues if the only lines of code that are inspected are the isolated `malloc()` and `free()` lines.** To have any chance to catch these defects, it is necessary, at a minimum, to inspect operations in the broader context of the surrounding statements. This tremendously increases both the complexity and the time required to perform the analysis.

Refinements to the basic analysis for this requirement must balance between eliminating false-negatives, limiting false-positives, and speed. This opens the door to the deepest types of analysis available today. The data-flow based model extends to support refinements that take into account precise statement sequencing and, in the most advanced implementations, can tag data-flow facts with branch predicates and test them for compatibility. Although R-Check SCA does not yet implement this level of analysis, in 2005 a prototype implementation of an analysis of this type, run over the Linux kernel source tree, discovered more than a hundred instances (in this case POSIX `lock()` API mismatches) of previously undetected bugs [7] in a few hours with a low rate of false positives.

The lesson from these examples is that, to provide a meaningful contract, it is important that requirements not be written in isolation from the developers or test tool authors. In many cases, a strong, enforceable specification requires concessions from developers on how code will be written and awareness of how a requirement will be tested. The SCA Next [8] guidelines for C/C++ best practices are a positive step in this direction. Specifications should be targeted for static or dynamic testing (or both). The designation of requirements, done as part of the SCA Next development, into "automatable" vs. other categories is, likewise, a positive step.

In total, with the current design of R-Check SCA, we have identified more than two dozen core SCA 2.2.2 requirements that can be automated. All of these requirements can be tested over the common intermediate representation, meaning that the tests can be performed from a single parse of each program file. Beyond the source code analyses described here, some of these analyses also require consistency across supporting SCA XML domain profile and CORBA® IDL files. These files are parsed separately in R-Check SCA with relevant facts condensed into a summary file that is then reparsed to initialize the C/C++ analysis engine. In this way, the static analysis approach extends to support specifications over heterogeneous and interdependent file types.

4. BENEFITS FOR SCA NEXT

We expect certification of SCA Next to rely even more heavily on methods of static program analysis as new features introduce challenges that could make dynamic (runtime) analysis harder.

- More flexibility in the interface (CORBA vs. No-CORBA).

  For SCA 2.2.2, CORBA provides a uniform interface layer to code being tested. For code that bypasses the CORBA transport layer, it will necessary to construct an equivalent test harness.

- More flexibility in choice of supported capability through units of functionality and multiple supported SCA profiles.

  The increase in supported capability sets complicates the process of runtime testing. Typically, each capability set requires a customized environment and suite of tests.

- Less accessibility to component interfaces (common access through the Domain Manager)

  Less accessibility to individual components makes it more difficult to perform individual unit tests. This pushes runtime testing to later in the development cycle and requires more detailed test scripts to exercise individual components.

Carrying over from the SCA 2.2.2, the SCA Next also retains requirements concerning non-termination, memory cleanup, and exception handling that are difficult or impossible to test dynamically. For termination and memory, adequate tests require deeper analyses that can only be built on robust analysis platforms that understand code structure and context. C++ exceptions are an example of a code feature that can be particularly hard to exercise at runtime. Here, static testing offers the advantage of being able to test exception paths without requiring the construction of exception-inducing behavior.

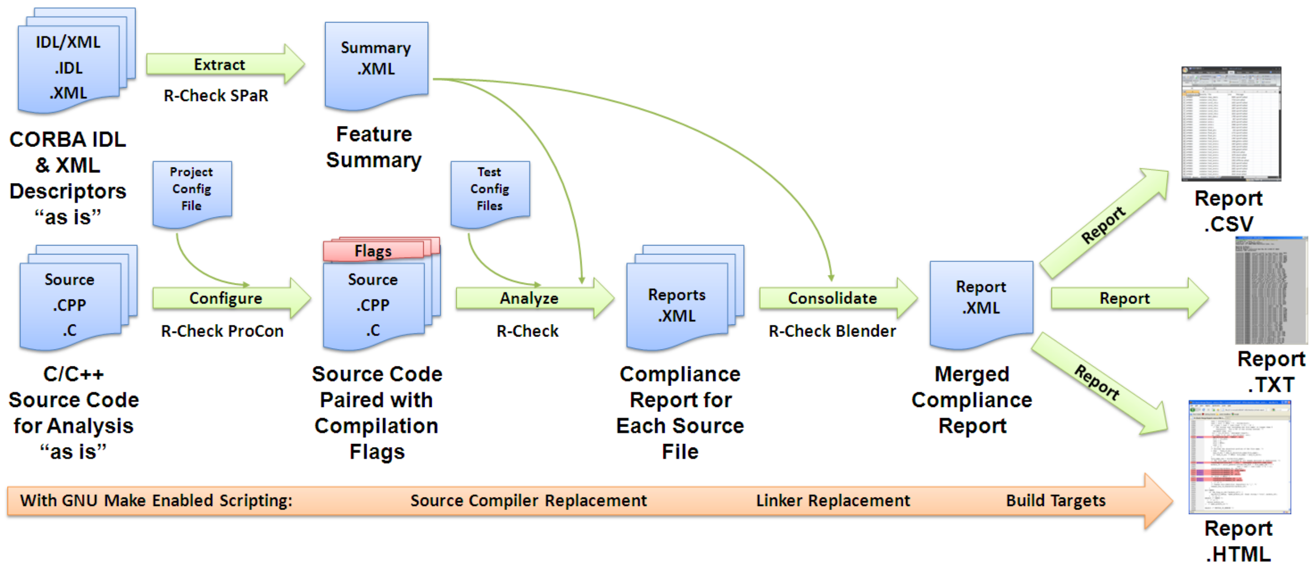

Figure 1. R-Check SCA workflow.

5. STRUCTURE OF R-CHECK SCA

Figure 1 illustrates the R-Check SCA workflow, encompassing both C/C++ source code and SCA XML domain profile and CORBA IDL file static analysis for SCA 2.2.2. The structure of R-Check SCA mirrors that of a traditional compiler and linker. The R-Check static analysis engine processes source files independently, interpreting the usual range of compiler arguments, and generates a summary report, in XML, for each. These files can be interpreted independently within a development environment to provide immediate feedback (see Figure 2) or can be merged by a separate report post-processing utility (R-Check Blender) into a consolidated view. Natively, R-Check can produce reports in plain text, CSV, and HTML formats. The HTML reports include hyperlinks to highlighted syntax and include descriptive information about the nature of the violation with references back to the SCA specification itself (see Figure 3).

Using augmented data-flow techniques, we plan to ultimately automate more than two dozen SCA 2.2.2 requirements with R-Check SCA. Moving toward SCA Next, we are extending the core workflow to support dependencies between source files (needed for true interprocedural analysis) using a two-pass analysis system. In the first pass, summary information is collected for each file and in the second pass the analysis engine draws upon the aggregated summary information to perform deeper analyses. This structure mirrors our approach to consistency checking over C/C++, SCA XML, and CORBA IDL files in which a summary of the supporting files is passed to the source code analysis engine.

6. SUMMARY

This paper describes the benefits of static compliance testing for the current SCA 2.2.2 and for the forthcoming SCA Next specifications. Static analysis provides specific advantages that make it a natural complement to dynamic testing. Static testing can be automated and, through integration into a development environment, can provide the soonest possible indication of a potential violation. Through this sort of reinforcement, static analysis can also serve as an aid in teaching the specification to the developer. This results in more knowledgeable developers and tighter test-debug cycles with fewer issues manifesting in post-development certification and validation testing.

Static analysis methods will be an important part of the certification process for SCA Next. New features in the specification will make uniform dynamic testing more difficult and several other features carried over from SCA 2.2.2 map best to static methods. Efforts to address coding practices and to categorize requirements by their ability to be automated are positive steps toward a more testable specification.

R-Check SCA provides a platform for implementing modern static analysis techniques, capitalizing on nearly forty years of academic and industry experience. The use of compiler methods of paring and analysis provides reasonable power and precision, a platform for implementing more advanced techniques, and scalable performance for enterprise codes.

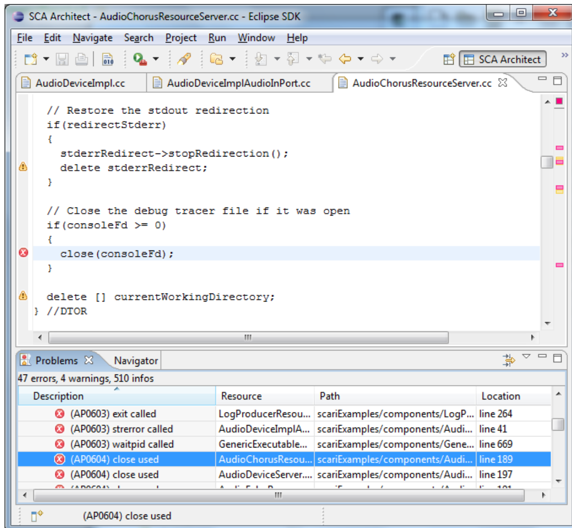

Figure 2. R-Check SCA running inside SCA Architect/Eclipse.

Through a plug-in to SCA Architect [9], a save operation initiates a reanalysis of a source code file. R-Check SCA can evaluate incomplete code and find violations even before the first compilation. In this case, a violation of the SCA 2.2.2 AP0603 requirement is highlighted. Once the plug-in is enabled, the developer sees the report and listed errors only a few seconds after the file is saved.

Figure 3. AP0603 description linked from HTML report.

In R-Check SCA, violations can be linked directly to source reference material. This accelerates the knowledge transfer process, especially useful for new or evolving specifications such as SCA Next, and leads to fewer bugs in submitted code.

7. ACKNOWLEDGEMENTS

Development of R-Check SCA was funded under Navy SBIR contract N00039-09-C-0118. The authors would like to thank our technical contact at SPAWAR, John Thom, who has been extremely supportive of this effort and tireless in helping us improve R-Check SCA. We would also like to thank the team at JTEL for their patience and wealth of valuable feedback. Finally, we would like to thank Jim Kulp of Parera Information Services for his insight and many valuable contributions.

8. REFERENCES

[1] J. Ezick, J. Springer, V. Litvinov, D. Wohlford, "A Path Toward Cost-effective SCA Compliance Testing," *Proc. of the SDR '10 Technical Conference and Product Exposition, December 2010.*

[2] https://jtel.spawar.navy.mil.

[3] http://www.eclipse.org.

[4] G. Kildall, "A Unified Approach to Global Program Optimization," *1st Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages, 1973.*

[5] S. Sharir, A. Pnueli, "Two Approaches to Interprocedural Data Flow Analysis," *in Jones and Muchnik, editors, Program Flow Analysis: Theory and Applications. Prentice-Hall, 1981.*

[6] SCA 2.2.2 Applications Requirements List version 2.2 Release Notes, *JTRS Test & Evaluation Laboratory, July 2010.*

[7] Y. Zie, A. Aiken, "Saturn: A Scalable Framework for Error Detection Using Boolean Satisfiability," *ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 29 No. 3, May 2007.*

[8] http://www.public.navy.mil/jpeojtrs/SCA/Documents/

[9] http://www.crc.gc.ca/en/html/crc/home/research/satcom/rars/sdr/products/sca_architect