

A PATH TOWARD COST-EFFECTIVE SCA COMPLIANCE TESTING

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

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

Vassily Litvinov (Reservoir Labs, New York, NY 10012; vass@reservoir.com)

David Wohlford (Reservoir Labs, New York, NY 10012; wohlford@reservoir.com)

ABSTRACT

We present R-Check™, a versatile architecture used to develop R-Check SCA, an SCA-specific static-analysis-based compliance testing tool for software radio waveforms. R-Check SCA was developed for JTEL and is intended to provide a cost-effective replacement for several of their search-and-inspect-based compliance testing procedures. The R-Check architecture makes use of several off-the-shelf components and open standards and is specifically engineered to integrate into the widest possible range of vendor development environments, an essential feature for addressing a modest but heterogeneous market space such as software radio. R-Check is capable of functioning over both complete and incomplete code bases and is efficient enough to run as part of the everyday edit-recompile-test cycle. In testing at JTEL, R-Check has demonstrated potential as a powerful, but low-cost platform on which to build additional test capability. For requirements related to restrictions to core POSIX® and Minimum CORBA® interfaces, R-Check produces reports in minutes that currently require hours or days of JTEL code inspection. We discuss the R-Check architecture, R-Check SCA, and design choices that allowed us to make the business case for addressing the SCA market.

1. INTRODUCTION

The Software Communications Architecture (SCA) is an open architecture framework that governs the structure and operation of the US Military's Joint Tactical Radio System (JTRS) and is gaining momentum in the private and international government sectors as a guiding path for software radio development. As an evolving specification for waveform development, the SCA references the IEEE POSIX® and OMG CORBA® standards, placing restrictions on the use of POSIX operations and the structure, definition, and operation of CORBA components. For the US Military, the JTRS Test & Evaluation Lab (JTEL) [1] acts as the test authority for compliance testing of the SCA and is charged with developing and maintaining that test capability. Other organizations and individual vendors have expressed

interest in using JTEL tools and procedures as a starting point for their own respective formal evaluation processes.

Certifying compliance with the SCA is a difficult and time-consuming task. The SCA references both structural and runtime behaviors, and places cross-referencing requirements on source code, XML configuration, and CORBA Interface Description Language (IDL) files. As an open architecture framework, SCA guidelines apply not only across platforms, but also across Operating Environments (OEs) and CORBA implementations, often with subtle differences in tool and supporting library structure. While several general-purpose analysis tools exist, few, if any, provide a cost-effective foundation for SCA compliance testing. Because they were not engineered specifically for the SCA, most commercial products are either deficient in dealing with one or more aspects of the SCA's scope, or else offer a range of features that make them prohibitively expensive for the front-line development process. In some cases, general-purpose tools also prove to be a poor fit with the specialized development environments, compilers, and operating systems pervasive in the software radio community. For vendors, the length of the certification process can exceed the interval of their upgrade-release cycle [2]. The lack of test tools targeted to the SCA represents a critical gap in the lifecycle process that is a limiting factor in the deployment of new capabilities and threatens the acceptance of the SCA as a viable commercial development specification.

The scope and complexity of the SCA requires test and evaluation tools that can draw upon and synthesize several types of analysis. The nature of the commercial SCA market, which is highly heterogeneous but still relatively modest, coupled with the existence of general-purpose tool solutions, however imperfect, place severe development cost restrictions on prospective tool vendors. The evolving nature of the SCA, which imposes an unpredictable maintenance obligation, further complicates the business case for profitable commercial test tool development.

We present our approach to the development of a low-cost, targeted test solution for the SCA: R-Check™ SCA, an SCA compliance tool, intended for both government and commercial use that we are developing with support from

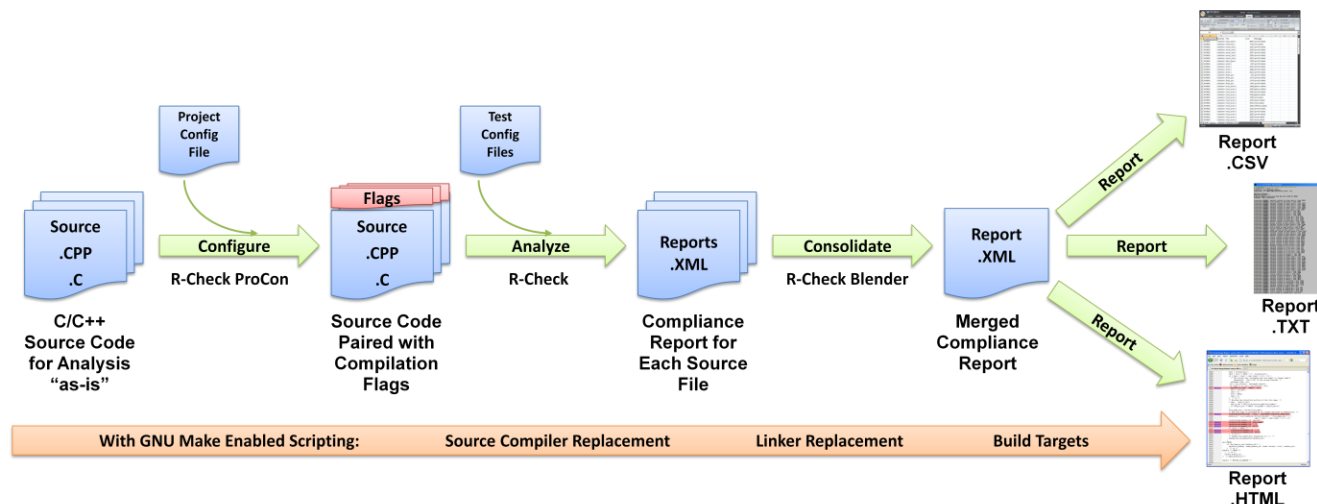


Figure 1. R-Check SCA workflow.

JTEL. Our intent is to offer our experience as insight into how to address the conflicting demands of a complex problem, modest but competitive market, and uncertain risk. Our presentation will include a discussion of the positive and negative external factors affecting the case for focused SCA test tool development. Our hope is to motivate interest in creating a broader and more hospitable commercial tool marketplace.

2. R-CHECK SCA OVERVIEW

R-Check SCA is a static analysis tool for ANSI C and C++ with secondary support for the sort of XML and IDL files associated with SCA projects. R-Check is built on the Edison Design Group (EDG) family of compiler front ends [3], which allows it to parse the full ANSI C and C++ languages with recognition of compiler flags, proper support for all preprocessing directives, and retention of file and line number information for error reporting. From the abstract representation produced by the front end, R-Check is capable of performing analysis with complete awareness of the code structure and context. This capability improves on JTEL's search-based testing procedures by identifying instances of non-compliance in an environment in which macro definitions have been expanded, conditional code has been included or excluded, comments have been filtered, and usage context can be discerned. The decision to use a commercial front end both improves performance and reduces the likelihood of encountering problematic code. The result for JTEL is a substantially faster analysis for several requirements, with fewer human steps, that generates reproducible error reports with fewer false positives – fewer false positives meaning that less time is spent in manual post-inspection of code.

To support the manner in which code is submitted to JTEL for evaluation, we have relaxed the type and syntax checking systems in the front end to accommodate analysis of files where OE or IDL-generated headers are unavailable. The result is a mode of conservative analysis that still allows a deep inspection, but is robust when executed on machines that are not configured or licensed as full development platforms.

The structure of R-Check SCA mirrors that of a traditional compiler and linker (see Figure 1). R-Check processes source files independently, interpreting the usual range of compiler arguments, and generates a summary report, in XML, for each. A separate report generation tool then merges the individual reports into a single composite report. Separate post-processing capabilities support various transform, filter, comparison, and presentation functions. Presently, 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 2). The choice to segment the tool in this way means that the analysis and synthesis features can piggy-back on almost any build process. Smart report updating can be trivially accomplished using any standard dependency-driven build tool (e.g. GNU Make). This same functionality also allows analysis of source files in parallel, with no development effort on our part, using the parallel build functionality included with most dependency-based tools. The use of open standards and tools have substantially reduced our development costs while at the same time providing more, not less, capability to the end user.

Presently, R-Check provides coverage of several requirements related to core POSIX and Minimum CORBA restrictions. Coverage of several additional requirements on CORBA components that span XML configuration, IDL, and source code files is in testing now.

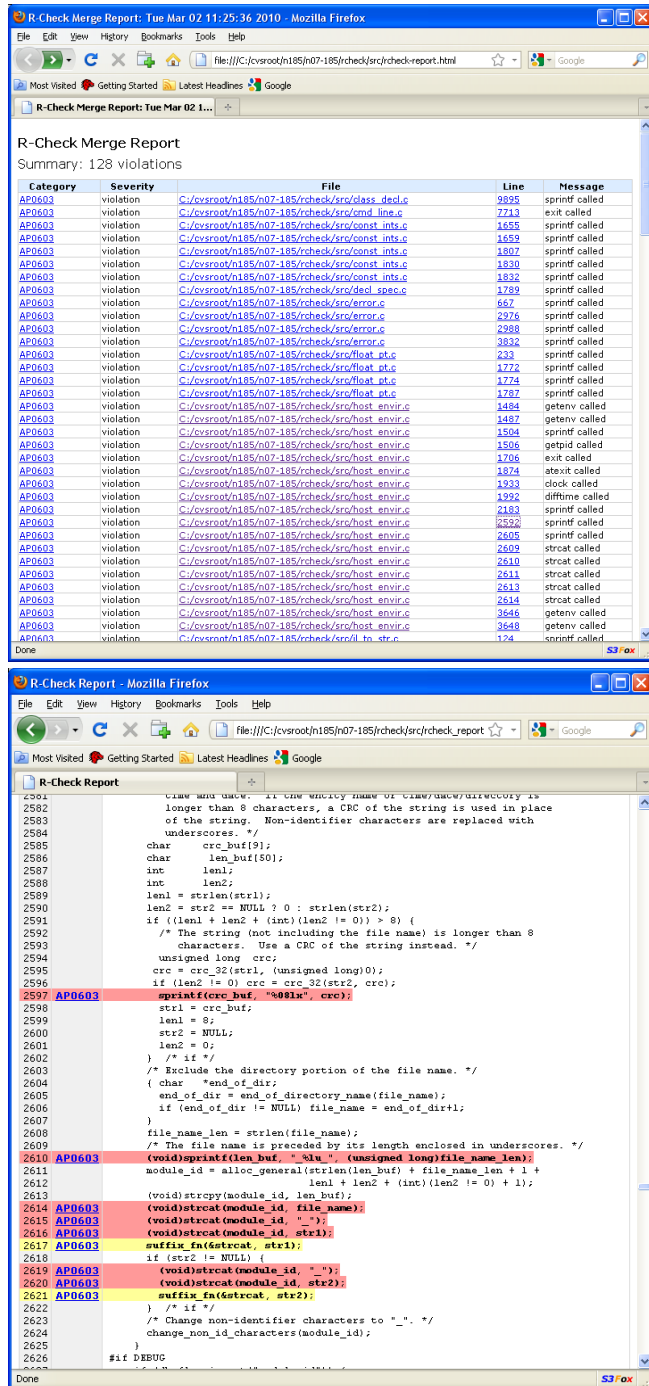


Figure 2. R-Check HTML report for the JTEL AP0603 core POSIX requirement (SCA 2.2.2 Appendix B).

Lines 2617 and 2621 are shaded as sources of a potential violation because the address of an NRQ POSIX call is taken and passed as a function pointer argument.

3. POWERFUL, LOW-COST, ROBUST DESIGN

The business case for developing R-Check depended on being able to provide a value-adding test capability for a multitude of platforms, without having to expend resources developing non-essential functionality. To accomplish this, the R-Check architecture relies heavily on off-the-shelf components and open standards.

For parsing source code, R-Check uses the same EDG suite of front ends used by several other commercial source compiler and analysis products. Although R-Check doesn't actually build code, having a compiler front end, and the associated intermediate representation it generates, gives us a prebuilt set of data structures over which to define analyses. These data structures allow analyses to draw on the syntactic context of program elements and support immediate access to the file and line number information necessary to effectively report violations. While some vendors of general-purpose tools have claimed that the off-the-shelf front end is inadequate for dealing with the variety of code seen from customers [4], we have found it very capable of processing typical SCA source bases. Given the sheer size and complexity of the C++ language in particular, writing a ground-up parser was not an option for us.

For dealing with project-level configuration, we developed a command-line utility, called ProCon. R-Check ProCon uses a simple plain text file format for mapping file names (with associated directories) to compiler flags and analysis options. Mapping uses familiar regular-expression syntax making it easy to associate options with specific files, file types, modules, or the entire project. The R-Check ProCon file format was designed as an accessible target for any sort of tool or script capable of crawling over IDE-specific (e.g., Green Hills Integrity® or Microsoft Visual Studio®) build files or build logs.

R-Check itself uses a text-based configuration file format with references to files containing default configuration options for each supported analysis. The analyses are further individually customizable and support modifying lists of keywords, disallowed functions, etc. In this model, a waiver becomes a command-line option designating a modified test configuration. Combined with R-Check ProCon, this feature makes it easy to record and implement waivers at the file, module, or project level. One option supports explicitly setting the root directory from which the analysis should be run, which allows alignment of relative include paths between the R-Check analysis and the normal build process. For each source file, R-Check generates a summary compliance report in XML.

On the back end, we developed a post-processing utility called Blender that is capable of generating aggregate reports from individual file reports and performing common operations on reports such as merges, slices, and comparisons. R-Check Blender supports source-to-source

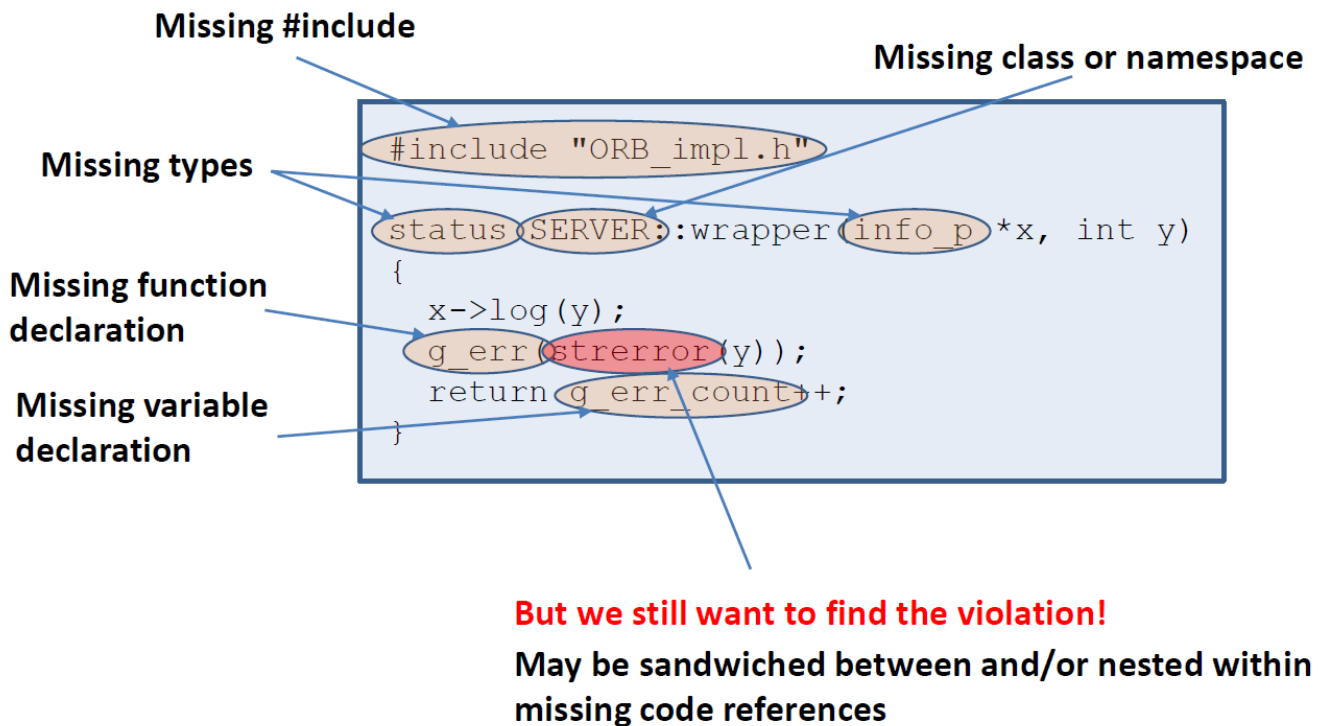


Figure 3. Challenges of checking incomplete code.

processing, taking one or more XML reports to a target XML report, and also supports generating reports in any choice of plain text, CSV, and HTML formats. Aggregate reports include tables of violation-counts-per-module and a record of the tests and options sufficient for reproducibility. The comparison capability can be used for tracking regressions across runs. The HTML report format supports syntax highlighting and hyperlinks for locating violations and providing background information on the nature of the violation linked directly to the text of the SCA specification.

We rely exclusively on third-party tools to view reports. The use of HTML provides a reasonable graphical presentation capability without the need to reply on a separate GUI or integration into any particular IDE (e.g., Eclipse). This choice provides more flexibility to the user while minimizing project costs. We provide the XML schema used for reports, so there is no barrier to end-users writing their own companion post-processing utilities.

Finally, in place of an IDE or IDE-plugin, we use GNU Make to tie the entire pipeline into an end-to-end analysis system. Using R-Check as a compiler replacement and R-Check Blender as a linker replacement, GNU Make provides all of the necessary functionality for a complete analysis. In addition we get, for free, advanced functionality such as smart re-checks (only check files that have changed since the last analysis) and support for parallel file analysis. We provide a complete GNU Make-based system with targets bound to different operations and report types. Additional operations include support for generating code

metrics, forcing reanalysis of designated files, and summarizing analysis progress. In terms of development cost, about 80% of the current GNU Make-based user interface functionality was prototyped by a single developer in less than two days. Also, the Makefiles and wrapping shell scripts are provided in plain text, allowing full customizability. These files also serve as a template for integrating R-Check into any type of development build environment. R-Check accepts standard compiler options, and, in most cases, R-Check and R-Check Blender can be used as one-to-one substitutions for the compiler and linker in a build process. In this way, R-Check SCA can be run in parallel with the daily edit-recompile-test development cycle.

4. CHECKING INCOMPLETE CODE

R-Check has the ability to provide a conservative analysis over incomplete source code packages. By incomplete, we mean packages that are missing some source or included header files. For software radio waveforms that use CORBA, this feature means that R-Check SCA can be run on machines that do not have a full CORBA ORB installation (or license for installation) and thus might not have the header files or the IDL compiler that a development machine would require. For JTEL, this means that their test capability is not dependent on having access to each of the different ORB and OEs used by vendors. For both JTEL and vendors, this feature means that testing can

begin during the development process, on any amount of syntactically correct source code.

The development of this capability was the single most technically challenging aspect of R-Check's creation. R-Check uses a commercial compiler front-end with full parsing capability for the C and C++ languages. For that reason, R-Check natively processes (and thus requires) all of the included header files (both system and IDL-generated) that are required by a compiler. Implementing an incomplete code checking capability, called Partial Code Model (PCM), required relaxing the type and syntax checking systems to both suppress errors arising from missing information and to infer information from usage that would normally be provided by an explicit declaration. Development of this capability, which is orthogonal to the analyses performed, required a significant amount of compiler expertise.

Figure 3 illustrates some of the issues that PCM needs to resolve in order to surmise that a violation has or may have occurred. In this example, a call to the POSIX function `strerror()` would be a violation, as this is a call to a function designated "Not Required" (NRQ) in Appendix B of the SCA 2.2.2 specification. Whether or not this is an actual violation depends upon (1) whether or not the call is to an actual POSIX call (violation) rather than to a locally defined function with the same name (not a violation) and (2) whether or not the argument to `g_err()` is evaluated (violation) rather than passed through a macro to be used as, say, syntax substituted verbatim into a `printf` statement (not a violation). As a benefit of R-Check's ability to analyze code in context, in its native (complete-code) mode R-Check makes these distinctions automatically. Using PCM, R-Check provides the best conservative analysis possible from the available information.

The resolution of macros, in particular macro-defined functions, in cases where the defining header is not included is a particularly tricky issue. From a pure analysis standpoint, this problem is tricky since, before the expansion occurs, the code need not be syntactically correct. Without the definition, a best effort must be made to analyze the existing code while ignoring errors that arise from simply not having the macro definition. Our method for accomplishing this involves traversing the available part of the syntax tree while suppressing errors from the grammar. Of course, missing macros could permute even apparently compliant code into non-compliant code. To handle this, we flag the potentially offending code for post-inspection, but do not raise a specific violation.

With PCM, it is not strictly necessary to provide an R-Check ProCon configuration file – R-Check will skip (and report) include files it cannot find. Combined with our GNU Make-based build system, which includes support for finding all of the source files in a directory that can be analyzed, R-Check SCA can be run, out-of-the-box by simply changing the working directory to the root of a

source tree and running a single command with no arguments: `rcodeck.make`.

5. TEST COVERAGE & PERFORMANCE

The current version of R-Check SCA includes tests distilled from the SCA 2.2 and SCA 2.2.2 specifications into JTEL requirements. The tests are named by each requirement and can be individually included or excluded from the test suite. The current version includes tests related to the use of core POSIX and Minimum CORBA, as these tests were identified by JTEL as among their most time intensive. Additional tests are in development now that include integrating information between the source code, XML configuration files, and unprocessed IDL files. We are developing an additional stand-alone utility to pull information from the non-source-code files and store it for access and comparison within R-Check. This functionality will enable testing of requirements relating to the use of ports, consistency checks between component declarations and implementations, and completeness of component implementations. As with the other parts of the R-Check architecture, this utility will use a text-based open format, will use a context-aware intermediate representation, and will be executable through a command-line interface. The IDL-parsing capability will be reusable and will further extend the scope of the R-Check architecture to support a range of CORBA-related specifications.

In working with JTEL, R-Check has been tested against more than a dozen commercial waveforms contained in the Network Enterprise Domain Information Repository (NED IR). Running in PCM on standard JTEL workstations (that is, without either a CORBA ORB or OE installation), R-Check successfully processes 100% of the tested waveform files, without modification, and generates uniform, reproducible reports that are as or more complete than the reports generated from the existing methodology for the requirements covered. Performance on other waveforms tends to track closely with expected compilation times. On the open Calit2 version of FM3TR [5], analysis takes two to three minutes on a typical desktop machine. For the largest waveforms, R-Check automatically processed more than ten million total lines of code (in PCM with available headers expanded) in about ninety minutes. The equivalent manually intensive search-and-inspect analysis for the same covered requirements typically requires two or more days of dedicated test engineer time. Again, the GNU Make-based system supports smart re-checking of modified files and can take advantage of multi-processor machines with its built-in parallel execution option. The report synthesis and post-processing tools scale to reports generated from thousands of files without difficulty.

6. SUMMARY

R-Check SCA is a commercially available [6] solution for statically checking source code compliance with the SCA version 2.2 and/or 2.2.2 specifications. Built using a range of off-the-shelf components and open data formats, the R-Check architecture demonstrates a cost-effective, but still powerful and robust analysis platform. The context-aware intermediate representation generated by the compiler-grade front end permits almost any aspect of the source code to be reasoned about. The use of command-line utilities to manage configuration and report generation permits R-Check to be customized for integration into almost any type of development environment. Leveraging external viewers and existing build tools like GNU Make not only substantially reduce development and maintenance costs, but also provide more customizable functionality to the end-user than a custom, closed tool would. The use of simple interfaces and open data formats means that R-Check SCA imposes virtually no restrictions on the end-user's development process and does not require additional third-party software purchases. On the contrary, the open interfaces facilitate creation of custom capabilities that fit the customer's needs and practices. With PCM, R-Check SCA can provide an immediate capability without any setup or customization. The performance of R-Check is comparable to a traditional source compiler, meaning that it is efficient enough to integrate into the daily development cycle.

Development of the R-Check v1.0 architecture and R-Check SCA required, in total, approximately ten developer-months, of which roughly half were devoted to implementing PCM. Purchase of an EDG license was the only other significant development cost. Approximately 80% of the development effort required some level of compiler or static analysis expertise. From a business standpoint, we expect to amortize this development cost by using the R-Check architecture as the basis of a service business developing other low-cost custom static analysis capabilities, of which R-Check SCA is the first example.

Looking forward, a significant source of development risk is the volatility of the SCA itself, and in particular, the implications of the forthcoming SCA Next. Currently, analyses must be coded into R-Check and then exposed through a command-line interface. Analyses typically have several customization options, but the basic analysis type is "baked-in" to the product. We are experimenting now with several analysis template and pattern matching ideas that would allow new analyses to be specified through an open format. We expect this feature to mitigate future risk. A second, perhaps more significant, source of risk is the volatility of the C++ language. To reduce costs, many parts

of the R-Check architecture are built directly on top of the front-end code base. A significant change to the language, for example the adoption of C++0x [7] for waveform development, would require a significant reimplementing effort on our part.

Beyond meeting JTEL's testing needs, one of our forward goals for R-Check SCA is to facilitate on-site testing and certificate generation. In this model, compliance testing could be done on-site, directly by the vendor, with the result being a certificate of compliance linked to the source code. Clever use of digital signatures and checksums could even allow this process to be implemented in a non-intrusive manner. Formal certification could then be accomplished through a simple hash check that would confirm that the certificate generated is, in fact, linked to the submitted code. Wherever final testing occurs, the ability to run equivalent tests as part of the daily edit-recompile-test development cycle should significantly reduce the need for expensive "dry-runs" and iterative submission for certification.

From several conversations with waveform vendors, we understand that is a strong desire for test tools that not only match, but exceed JTEL standards. With the R-Check architecture we can offer custom test solutions within a robust framework at minimal cost.

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. We would also like to thank Jim Kulp of Mercury Federal Systems for his insight and many valuable contributions.

8. REFERENCES

- [1] <https://jtel.spawar.navy.mil>.
- [2] M. Turner, "Global Military SDR Solutions – Practical Methods for SCA Radio Compliance and Deployment," *Proc. of the SDR '09 Technical Conference and Product Exposition, December 2009*.
- [3] <http://www.edg.com>.
- [4] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World," *CACM*, Vol. 53, No. 2, pp 66-75, February 2010.
- [5] <http://jtrs.calit2.net/>.
- [6] <https://www.reservoir.com/rcheck>.
- [7] <http://www.open-std.org/jtc1/sc22/wg21/>.