ACCELERATING SCA COMPLIANCE TESTING WITH ADVANCED DEVELOPMENT TOOLS

Jonathan Springer (Reservoir Labs, New York, NY, USA; springer@reservoir.com)
Steve Bernier (NordiaSoft, Gatineau, Québec, Canada; steve.bernier@nordiasoft.com)
James Ezick (Reservoir Labs, New York, NY, USA; ezick@reservoir.com)
Juan Pablo Zamora Zapata (NordiaSoft, Gatineau, Québec, Canada; juan.zamora@nordiasoft.com)

ABSTRACT

In this paper, we explore the potential for combining model-based development environments supporting automatic code generation with novel static testing technology to accelerate the SCA compliance testing process. Model-based development and automated testing yield higher regularity and predictability, reducing testing complexity and sidestepping some issues for software intended for deployment on multiple hardware platforms. Further, integrating test tools into the development environment can provide immediate feedback on compliance issues during the development process. As testing moves upstream, the load on certification entities is reduced, and correction of issues becomes more straightforward. Pushing the testing upstream also opens the door to increased customization. We introduce Pitchfork, a language technology that allows users to define specifications as sequences of patterns that can be identified in source code. With Pitchfork, it becomes possible to encapsulate both SCA and API properties in a precise, automatically checkable way and distribute them across the SDR community for immediate integration. A natural evolution of this concept is self-certification, in which a robust set of test tools provides a capability for a developer to offer strong evidence of compliance without a formal certification process. Conclusions in this paper are supported with experiences from the use of the NordiaSoft™ SCA Architect IDE and Reservoir Labs' R-Check® SCA compliance test tool.

1. INTRODUCTION

The Software Communications Architecture (SCA) has provided proven benefits to the defense communications community, including reduced risk, cost and time-to-market, enhanced communications interoperability and simplified insertion of new communications capabilities. Based on this success, governments worldwide are mandating the adoption of SCA standards in their own defense communications infrastructure and forward-looking stakeholders in electronic warfare [1], radar, and robotics are evaluating the SCA for its potential to accelerate their own projects. This pattern of adoption is leading to a proliferation of developers being introduced to the SCA, growth in the SCA tools marketplace and increased interdependencies and opportunities for sharing among nations and project groups. As the stamp of "SCA Compliant" transitions from desirable to essential, there must be credible tools and processes for efficiently demonstrating SCA compliance. If the SCA is to survive as a respected international standard, it is essential that the SCA community be able to defend against free-riders who would erode the reputation of the SCA by claiming compliance not backed by any outside validation.

Both the prevailing SCA 2.2.2 and emerging SCA 4.1 specifications pose difficult compliance testing challenges. For SCA 2.2.2 testing in the US, more than 100 separate application requirements and more than 500 separate operating environment requirements have been enumerated [2][3]. Thorough testing requires the application of both static and dynamic tools. Although the trend in SCA 4.1 has been toward greater platform abstraction and therefore an increased need for static testing tools, both specifications include requirements that can, at best, be only approximately tested [4]. Nations adopting the SCA for their defense communications infrastructure should expect to make a considerable investment in compliance testing facilities, and contractors outsourcing SCA development should plan to include specific provisions for verification of SCA compliance in their agreements.

In this paper we illustrate, using technologies that are being developed today, ways in which SCA certification can be accelerated with advanced development tools. By integrating testing with model-based development environments, testing can be focused on business logic and compliance issues can be caught earlier in the development process. In this model, it becomes possible to catch errors as they are introduced and offer instructive remedies that prevent similar errors from being introduced throughout the code base. By providing compliance tools directly to the developer, possibilities are created for allowing limited self-testing that would reduce the time and cost of ultimate

compliance verification. Tools that permit new rules to be defined allow for knowledge capture and sharing across projects and teams. Languages for defining these rules provide a guiding path toward writing new, automatically testable, requirements.

The remainder of this paper is organized as follows. Section 2 introduces the concept of a model-based development environment as the foundation of an advanced development platform. Section 3 discusses how model-based environments can accelerate testing for a developer and references a recent illustrative success in which advanced tools played a pivotal role for GateHouse to achieve SCA-compliance for the BGAN SDR waveform. Section 4 describes the role of advanced development tools in the context of a test lab and discusses the potential for accelerating compliance through pre-testing. Section 5 introduces Pitchfork, a language technology that allows users to capture testable requirements as sequences of patterns that can be identified in source code. Section 6 outlines future prospects for evolving pre-testing to self-certification. Finally, Section 7 summarizes conclusions and highlights subjects for future work.

2. MODEL-BASED DEVELOPMENT ENVIRONMENTS

In spite of the projected gains associated with object-oriented programming languages, evidence has shown that Java only yields approximately 20% better productivity than BASIC [5] and barely 10% better than C++ [5][6]. Further, very small productivity gains have been achieved since the latest object-oriented languages like Java and C# have been introduced. Recent gains are mostly associated with the broader use of design patterns and agile methods [7]. Software projects are becoming more complex as their size keeps increasing. A recent study made by the Standish Group [8] shows that in 2012 only 39% of projects were delivered on time and on budget. The Standish study also shows that the larger the project is, the less chance it has of succeeding. Projects costing less than $1 million USD in labor costs had a 76% success rate, while projects over $10 million USD only had a 10% success rate. This reality led to a recognition of the need to raise the level of abstraction above object-oriented programming.

The SCA is an exemplar of Component-Based Development (CBD), a fairly new paradigm that raises the level of abstraction above fourth-generation programming languages and Unified Modeling Language (UML). While the majority of fourth-generation programming languages are aimed at addressing specific domains such as database management, mathematical optimization, GUI/HCI development, or web development, CBD can be used for a wider range of domains. In fact, CBD is considered by many as the paradigm that can lead to the industrialization of software [7][9][10][11][12].

The salient feature of CBD is that it shifts the emphasis from *programming software* to *composing software systems* [12]. Using CBD, the smallest unit of functionality is called a *software component*. CBD applications are assembled by wiring together several software components. This composition process takes place well after the compilation and linking phases. Just as it is not possible to add new pins to a chip post-manufacturing, software components are not meant to be modified by third-party users. Consequently, software components are designed to be as reusable as possible. Components offer a well-defined and concise functionality that is controlled via pre-defined interfaces. Associated metadata describes the characteristics of each component so that they can be used by third-party developers. The metadata also contains information about the operating environment requirements, the runtime capacity requirements, and other information relevant to the use of the component in composition. The metadata for software components is the equivalent of the technical specification sheet for a chip. The goal of adopting CBD as the basis of the SCA was to replicate the success of the electronic components market, which relies on reusability and third-party composition.

From SCA Components to Model-Based Development

While the CBD paradigm defines the manner in which systems are organized and deployed, it does not dictate the manner in which systems are developed. Model-Based Development Environments (MBDEs) extend the level of abstraction defined by the CBD paradigm all the way to the developer. MBDEs allow developers to build complex systems using high-level model concepts that treat components as first-class citizens. MBDEs can represent software components graphically, based on their associated metadata. Application developers, assisted by modeling tools, access and use the metadata to discover the interfaces and properties a component implements. Since it is the modeling tools that maintain the metadata in sync with the implementation of a component, developers need not manually create or modify it. MBDEs significantly improve productivity [6] by allowing developers to craft whole systems using models from which source code is then automatically generated. This frees developers to focus on writing the most valuable part of the SCA component, the business logic, which implements the algorithm for a filter or error correction code or any other functionality of their waveform application. Developers thus work in an environment where they graphically drag-and-drop SCA components onto a canvas, and later establish connections

between them. In essence, SCA development using MBDEs raises the level of abstraction above programming languages.

The SCA is the first CBD standard that is targeted at heterogeneous embedded distributed systems. It mandates the use of standard POSIX APIs for things such as thread creation, thread synchronization, timers, file access, and other core facilities. At the heart of the SCA is a Core Framework (CF) that enables third-party composition and component deployment. At runtime, the SCA CF is tasked with the deployment of the SCA application, which consists of the deployment and interconnection of individual components.

Advanced SCA modeling tools generate source code and metadata for SCA components that a) might run on different operating systems, b) run on different processors, c) are able to interact with third-party components that may or may not run in the same address space and d) are either local or remote. Generated source code handles the component life-cycle, from instantiation to termination, including the handling of runtime composition. SCA modeling tools can also generate source code that adapts to different runtime environments, which is not an easy task for heterogeneous embedded distributed systems. This level of independence is critical to increasing productivity by allowing third-party developers to reuse components. Implementing platform independence requires intricate infrastructure source code, and that type of source code constitutes a large proportion of any heterogeneous distributed embedded system. Using MBDEs, the infrastructure source code can be automatically generated. A recent study conducted with car producers, suppliers and technology consulting companies found that, although the amount of generated code varied substantially between companies and projects, 40% of participants reported generated code above 95% levels [13].

Over the last decade, the SCA has led to demonstrated productivity gains [14][15][16]. SCA MBDEs with graphical modeling of components and automatic code generation represent the future of software development. Companies can now build complex SCA-based systems by assembling software components that originate from different sources. The resulting development process extends well beyond company borders, which increases the importance of quality assurance and certification testing [17].

3. SCA COMPLIANCE TESTING IN MODEL-BASED DEVELOPMENT ENVIRONMENTS

The SCA defines a set of conventions that provides a means for components to be deployed and interconnected in a standard manner. The conventions are defined via a number of standard Application Programming Interfaces (APIs), behavioral requirements, and metadata structures.

Compliance testing encompasses hundreds of requirements that span the proper presentation and implementation of interfaces, the proper execution of components within those interfaces, and the overall consistency of the component implementation with the associated metadata. In practical terms, 100% assurance of SCA compliance is not achievable, owing to the existence of requirements that are worst-case undecidable and thus simply not exhaustively testable. In categorizing a test for a requirement, it is important to be clear about what "guarantee" the test really provides. In some cases, "pass" simply means "no obvious error." In other cases, "pass" is an exhaustive proof. Understanding the quality of the result provided by a test is an important discriminator in selecting tests to cover requirements. For some requirements, completeness requires a test process that spans multiple, independent tests – both static and dynamic.

SCA Modeling tools can generate component source code and metadata from high-level SCA models. Using modeling tools, developers can concentrate on writing domain-specific source code (also referred to as business logic). When implemented correctly, modeling tools thus have the potential to simplify the testing process by reducing the amount of code to be tested. This considerably decreases the development time required to create proper components by driving the focus of development and testing predominantly to the business logic. The remainder of this section describes methods of testing that can be applied and potentially accelerated, or not, in the context of a MBDE.

Components: 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 [4].

- 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 that link errors directly to violating code.

For the SCA, these advantages translate to a system of analysis methods that can be used to automate, and thus

accelerate, testing of several specification requirements. Through integration with a MBDE, analysis can be fast and transparent enough to execute concurrent 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, any introduction of an SCA violation 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.

Components: Automated Unit Testing

When adding business logic, developers can mistakenly introduce non-compliant behavior or structures. Perfectly compliant generated components can become non-compliant, negatively affecting their potential for reuse and therefore reducing productivity. While modeling tools cannot prevent developers from adding source code that violates the specification, they can automatically generate unit tests to catch deviations from the specification. Models are used to generate the specific unit tests a component must pass in order to adhere to the specification. Unit tests can be used to verify a very large portion of the behavioral requirements for a component. The automatically generated unit tests provide a safety net for the developers in terms of conformance. They also provide the opportunity to *test early* and *test often,* a best practice to help contain development costs, as well as to keep the project schedule under control.

Framework Testing

While modeling tools can help test a large portion of the requirements related to a) the structure of components, b) their metadata, and c) the way they use the operating environment, modeling tools are of limited help in testing the component framework itself. The Component Framework is a key element of the overall CBD approach. It serves as the virtual backbone, allowing components to be deployed and to communicate with each other. Using Model Based Testing (MBT) for the validation of the SCA CF only helps generate a small proportion of the tests needed to validate all of the requirements. However, an SCA MBDE can be used to generate the bulk of the source code used to validate the framework requirements.

As described in [18], using MBT, test source code can be generated based on models of the requirements of a specification. For instance, a harness can be generated to test if an SCA FileManager refuses mount-points with an invalid name. This can be done because there is a one-to-one relationship between the invalid parameter and an expected failure code. However, most framework requirements are not as straightforward to validate. A component framework is used to deploy an application made of several components with one simple API call. With such an API, it is not sufficient to simply test a return code to validate all of the requirements. Verifying deployment requirements involves interacting with components that have been deployed on remote processors. This implies creating a number of components and applications that will be deployed by a test harness to create different use cases that will help validate the long list of deployment requirements.

In the context of the SCA, the artifacts (i.e., components, applications, and metadata) required for testing the deployment engine inside the CF represents many times more source code than the test harnesses. One way to significantly accelerate the development of a CF test tool is to use an SCA MBDE to generate all of the test artifacts. The majority of the SCA CF requirements are related to the deployment of components on heterogeneous embedded distributed systems. MBT tools cannot generate the required framework test artifacts. MBT tools have been shown [18], however, be useful for generating test harnesses for strictly component-level requirements.

SCA Architect and R-Check SCA

NordiaSoft's SCA Architect™, an SCA MBDE, provides a deterministic graphical modeling language that supports every concept of the SCA. It performs full behavioral and structural code generation. Beyond the traditional skeletal code generation, SCA components generated by SCA Architect are fully functional and can be built and executed without adding a single line of source code. SCA Architect supports component-level model-based testing. Starting from the models, SCA Architect produces source code for unit tests that can instantiate the component-under-test to dynamically verify several aspects of runtime compliance. For instance, automatically generated unit tests can verify whether a component does indeed support all of the configuration properties it has declared. They can also validate that each provided interface implements the specified APIs.

SCA Architect integrates seamlessly with Reservoir Labs' R-Check® SCA, a sophisticated static source code analysis tool. R-Check SCA is capable of testing whole SCA applications and operating environments, but when integrated with SCA Architect, it provides continuous coverage of business logic. This capability complements the presumed (but still verified) correctness of the code automatically

generated by the SCA Architect MBDE by providing compliance testing for correct AEP and CF interface usage, the presence of exception and error case handling, and the completeness of component memory reclamation at tear down. By focusing on the business logic and linking directly to each save operation in SCA Architect, source code errors can be caught and explained as soon as they are introduced. This helps eliminate misunderstandings about non-compliant constructs, mitigates the likelihood of the same error being propagated throughout the source code tree, and ultimately accelerates testing by making it possible to catch and fix errors at the earliest possible point in the development process. The integration of SCA Architect and R-Check SCA provides a very high level of confidence regarding compliance throughout the development life-cycle.

GateHouse Case Study

As a recent illustrative success, SCA Architect with integrated R-Check SCA support, played a pivotal role for the Danish firm GateHouse to achieve SCA-compliance from the Joint Tactical Radio System Test and Evaluation Laboratory (JTRS JTEL) for its BGAN SDR waveform designed in cooperation with Inmarsat [19]. BGAN SDR allows the global government market to support the full suite of BGAN bearers, such as packet switched data, circuit switched data and Integrated Services for Digital Network (ISDN) and is the first SCA-compliant commercial satellite communications waveform. SCA Architect was used to create software models of the components. Using automatically generated code, automatic unit test support and continuous R-Check SCA static analysis of business logic during the development process, the entire SCA and API implementation verification was reduced to only four days. When the waveform was submitted for certification, the effort benefited from the fact that the certification authority had already assessed and verified compliance of the SCA Architect generated code. By having pre-tested their business logic with R-Check SCA, GateHouse was able to submit their waveform with a high degree of confidence that the final certification process would not yield any major surprises. This success demonstrates not only the capability of MBDEs to support developers in writing compliant code, but also the potential for advanced tools to accelerate the compliance certification process itself.

4. IMPACT OF MODEL-BASED DEVELOPMENT ENVIRONMENTS ON AN SCA TEST LAB

In an ecosystem such as the SCA's, with multiple software producers distinct from the software consumers, a model that has proven effective is to have a designated, accredited test organization whose primary purpose is to ensure that software adheres to the specification. Organizationally separating the process of writing software from that of certifying it for compliance greatly increases confidence in the validity of the compliance assessment. Within the SCA ecosystem, an *SCA Test Laboratory* is an independent facility that provides controlled conditions to achieve reproducible test results [20]. With an independent test lab, both vendors and procurement entities have a neutral authority for ensuring that all software adheres to the specification. Vendors submit software source code to the test lab, which checks it against the specification. If software does not pass inspection, the test lab communicates back to the vendor the specific parts of the specification that were not met.

In considering the impact of MBDEs on an SCA Test Lab, we identify three progressively more advanced forms of compliance testing:

1. **Pre-Testing**: A vendor uses tools known to, and used by, the Test Lab to develop and test their software. Test results are thus, presumably, known to the vendor prior to submission for formal compliance certification.
2. **Pre-Certification**: The vendor uses tools accredited by the Test Lab and submits the output of those tools for review by the Test Lab as proof of compliance.
3. **Self-Certification**: The vendor uses tools accredited by the certification authority and directly publishes the product and test results from those tools as proof of compliance.

As they exist today, MBDEs known to a Test Lab offer the potential to accelerate pre-testing in multiple ways:

- The visual layout provided by a MBDE can be used to generate a test plan and makes plain what needs to be tested
- Automatically generated code should be correct by construction
- MBDEs can support integration of static test tools equivalent to those used by the Test Lab
- Automatically generated unit tests increase confidence in the performance of components
- MBDEs encapsulate knowledge about the existence and structure of components that can be used to automatically select, inform and initiate dynamic tests

The move from pre-testing to the possibility of pre-certification, and ultimately self-certification, requires the certification of tools and the means to establish trust that the certified tools have been configured and executed in accordance with published test procedures. The nature of how this trust might be established is discussed in Section 6.

A key step toward enhancing support in MBDEs for pre-testing and bridging the gap between pre-testing and pre-certification and self-certification is providing a way to express and distribute testable requirements in a format that can be automatically executed within a MBDE. In the next section, we describe a novel extension to R-Check SCA for formally defining automatically checkable SCA and API requirements.

5. PITCHFORK

As specification-checking software is promoted more widely, both across test labs responsible for differing specifications and across vendors, its brittleness becomes more of an issue. Typical checking software builds in checks for specific aspects of the specification, and these checks are not able to be changed without significant development effort. Thus, when a specification is modified, new versions of the checking software must be developed and distributed throughout the ecosystem. Even if specification change is infrequent, the simple existence of multiple varying specifications (such as SCA, ESSOR, and SVFuA) means that there is a development burden on the maintainers of the checking software. Furthermore, vendors may find it convenient to be able to adjust or augment the checks that are done in areas unrelated to the ecosystem-level specification, and to share these adjustments with others.

To address this issue requires adding a level of configurability or programmability to the checking tool in the form of a specification language that the tool understands. Specification languages such as Larch [21] that are designed to express program invariants at a specific program point (e.g., method entry) do not fit this need. We are interested in expressing more flexible properties, which may span a larger region of code. A few specification languages such as Metal [22][23] have been designed to express general-purpose static analyses. However, expressing the desired specification is complex in those languages, requiring advanced programming skills.

To fill this gap, we propose a language, Pitchfork, based on the matching of code patterns, which allows most static properties to be expressed in a more straightforward manner. The Pitchfork rule language is a pattern-matching language over C and C++ source syntax. The concept is to allow the user to write the C/C++ source code that needs to be matched and reported. To make it convenient to match sets of structurally similar programs, Pitchfork augments the base C/C++ syntax with a metalanguage, providing regular expression syntax, pattern-match variables, wildcards, and other helpful features.

A prototype of Pitchfork has been implemented in R-Check SCA. For clarity of presentation, we have elided a few details that are not central to the language concept.

Syntax

A Pitchfork specification consists of a sequence of rules, each of which describes an independent property to be checked. A rule consists of a left-hand side (LHS), describing the property to be checked, and a right-hand side (RHS), describing what to do in the event that the LHS is triggered. The LHS is called the "pattern," while the RHS is the "action."

Some rules are "positive," by which we mean that they activate when the LHS pattern matches a specific code region. Other rules are "negative," meaning they only trigger if the property expressed is not found in the code. Rule polarity is implied by the action: *incident* actions create positive rules, while *API* actions create negative rules.

```
exp goto @label; @stmt
  => incident "Found dead code";
```

Figure 1 Example Pitchfork rule

Figure 1 shows a very simple example rule, which has the form "LHS => RHS." Here, the leading `exp` indicates the LHS is an expression pattern; the rest of the LHS is the pattern itself. In the RHS, the "incident" keyword indicates a positive pattern, and the quoted text is a message that will be reported (along with the location of the matched code). The net effect is a rule that reports when there is code directly following an unconditional branch in the program. We discuss the parts of a rule in more detail below.

Patterns

The pattern on the LHS is the heart of Pitchfork. It describes syntactic constructs that should be recognized and acted upon. Patterns cover statement and expression syntax as well as variable and function declarations.

As mentioned above, pattern syntax is C-like, however it is not specifically C or C++, as these languages (particularly the latter) are quite large and contain many details that are not relevant. Instead, an abstraction of C is used that captures the relevant algorithmic constructs without the clutter of isomorphic syntax (like **for** vs. **while**). The focus is on the essential core of the statement and expression parts of the syntax. Most expression syntax is supported, such as function calls, dereferencing, and primitive numeric operations. Statement syntax is more abstracted: conditionals are elided (our framework is flow-insensitive) and there is a single loop construct for any form of iteration.

Function calls are a special case. Calls to libraries or APIs are modeled, but calls to locally defined functions need not be written into a pattern. The checker, where possible, implicitly traces through function calls for which the callee body is known. The program is thus modeled in principle as

a Control Flow Graph (CFG) of statements. In practice, not all calls within the program text will be seeable, due to indirect calls or efficiency considerations, so the actual result may be a pessimization (in the direction of false negatives for positive rules and false positives for negative rules) of the ideal.

Several kinds of declarations are needed: functions, variables, exceptions, and structs. Each matches if the source code contains a file-level, class, or namespace definition of the corresponding construct. The identifiers naming the declarations must be C++-style qualified names indicating the specific class or namespace it must be defined in (if any). It is expected that declaration patterns will usually be used with API actions on the RHS, but they are valid for incident actions as well.

Metalanguage

On top of the core syntax, Pitchfork layers a regular-expression capability. Since regular expression syntax overlaps with the source language syntax, it is necessary to make it syntactically distinct. Pitchfork does this by prefixing an "@" character to all metalanguage constructs. Such constructs include grouping ("@{" and "@}"), zero-or-more iteration ("@*"), one-or-more iteration ("@+"), optionality ("@?"), disjunction ("@|"), and conjunction ("@@"). All are straightforward, although conjunction merits further explanation. Pattern conjunction means the first pattern and the second pattern are found, in that order in the code, possibly with intervening source code. Patterns that are simply juxtaposed (without the @@ connective) match only without any intervening code. Another way to think of @@ is as a wildcard pattern that matches any amount of code. Note that @@ is a *may-follow* relation; it is only necessary to have some path that connects the two constructs. The @!@ syntax can be used for the *must-follow* relation, in which all paths from the first construct must lead to the second one (again, with possible intervening code).

Pattern variables comprise another component of the metalanguage. A pattern variable is just an identifier distinguished by a leading "@" character. Where used, a pattern variable matches an arbitrary block of code. Furthermore, if the same pattern variable is used elsewhere in the LHS, it is required to match the same source construct as the first instance, or the match fails. This pattern variable can also be used on the RHS of a rule, where it is replaced by the matched program code.

A subtlety that has been so far elided concerns how to handle program variables, meaning normal C/C++ identifiers. Note that these are distinct from pattern variables in syntax (not being preceded by @) and in meaning (not matching code blocks). We treat program variables as unique up to alpha-renaming, as is standard. This avoids unintentional variable capture, where a script variable that is intended to be free (unbound) takes on some meaning due to a coincidental choice in the program being analyzed. However, in our case, such variable names are intended to match program variables, as for example when referencing system call names. Thus, the convention is established that program variables written into the pattern (unless declared otherwise in the script) match bound program variables appropriate to the scope where matching is being done.

Actions

There are two kinds of built-in actions in Pitchfork: incident and API. An incident action simply indicates that if the LHS matches, then that fact must be reported as an incident. An API action indicates that the construct specified in the LHS must be found in the program, or else the property is violated and the fact must be reported.

Declarations

In addition to the rules, a Pitchfork specification can include a set of declarations. Declarations take two forms. First, a declaration *@id*: *type* can be used to restrict a pattern variable to a particular type or type pattern. As with expressions, the syntax for a type pattern uses a tag character to escape from the concrete syntax. Additionally, there is a small set of "wildcard" tokens that allow some common groups of types to be expressed, such as "all pointer types." The second form of a declaration *id = val* assigns an identifier to a value. This value is then simply used in place of the identifier wherever it appears in subsequent declarations and rules.

Example Specifications

To get a better sense for Pitchfork in practice, we give some examples that exercise a number of different features of the language. The first shows an incident rule involving statement sequencing. The second extends the concept of sequencing to include checking for the absence of preceding or following code. The third illustrates checking that software properly implements an API.

Sequencing and Shared Variables

```
exp scanf(@buf, @str)        @@
    strcat(@filename, @buf) @@
    open(@filename)
 => incident "User input in filename";
```

Figure 2 Taint tracking example

A common security problem with user-facing software is the use of unsanitized user input in system calls. The rule in Figure 2 looks for a particular example of this in which a

scanf() call is used to fill a buffer, which is then appended to a path prefix using strcat() before being passed to open(). In this example, the three calls are sequenced using @@, indicating they need not be adjacent. The calls are tied together, however, through their shared use of the metavariables. The pattern only matches if the target of the scanf(), represented by @buf, is the same as the suffix given to strcat(), and the target of the strcat(), represented by @filename, is the same as the argument of open().

Constraining API Usage

```
exp @! pthread_attr_setdetachstate
       (@att, PTHREAD_CREATE_DETACHED)
    @@ pthread_create
       (@thread, @att, @start, @arg)
 => incident "Thread not detatched";
```

Figure 3 API usage example

Sometimes an API needs to be used in a certain way, either to honor the API contract or for reasons unique to the client software. This kind of constraint can be modeled in Pitchfork using the negation metasyntax (@!), which matches if its expression is not found. In the example in Figure 3, the pattern matches, and an incident is generated, if a pthread_create() call is found that is not preceded by a pthread_attr_setdetatchstate() call with the appropriate argument.

```
Exp
 pthread_mutex_init(@mutex, @att)
 @@
 @! @( pthread_mutexattr_setprotocol
       (@att, PTHREAD_PRIO_INHERIT)
       @|
       pthread_mutexattr_setprotocol
       (@att, PTHREAD_PRIO_PROTECT) @)
 @@
 pthread_mutex_lock(@mutex)
 => incident "Bad mutex attributes";
```

Figure 4 Complex API usage example

Figure 4 shows a more complex constraint on API usage. Here, we require that a particular mutex used for locking a thread must have either one of two different attributes. At the outermost level, the pattern is a sequence of three expressions. The "@att" variable is used to link the attribute setting (second sub-pattern) to the mutex (first sub-pattern), while the "mutex" variable links it to the thread lock statement (third sub-pattern). The negation syntax (@!) is used as before, but it modifies a group (@(and @)) in which there is a disjunction operator (@|). The net effect is to match a mutex created and then used for locking without having its protocol set to one of the two approved values.

API Implementation

```
exc Audible::InvalidToneId(
      msg : string)
 => api "Audible API";

str Audible::SimpleToneProfile(
      frequencyInHz        : uint4,
      durationPerBurstInMs : uint4,
      repeatIntervalInMs   : uint4)
 => api "Audible API";

fun Audible::createTone(
      SimpleToneProfile)
    : uint4 raises InvalidToneProfile
 => api "Audible API";

fun Audible::startTone(
      uint4)
    raises InvalidToneId
 => api "Audible API";

fun Audible::stopTone(
      uint4)
    raises InvalidToneId
 => api "Audible API";
```

Figure 5 API implementation example

When writing software that is supposed to conform to a published API, a Pitchfork specification can be written to ensure that this API is implemented properly. Figure 5 shows Pitchfork rules for a small extract of the AudioPort API (AudibleAlertsAndAlarms has been renamed for brevity). Three different kinds of declarations are illustrated: an exception (introduced with the "exc" keyword), a struct ("str"), and three functions ("fun"). In each case, the structure of the declaration is represented, including fields of the exception and the struct, the arguments of the methods, and the exceptions thrown by the methods (via the "raises" keyword). To match, a compatible implementation must use the same entity names and types. Any that are not found will generate a report from the Pitchfork checker.

Pitchfork in Practice

We have implemented a prototype of Pitchfork as a part of R-Check SCA. Pitchfork specifications are written into a ".pf" file and passed to the tool through a command-line argument. R-Check SCA uses this specification as it processes the submitted software suite, generating negative

reports only after the whole program has been analyzed. Pitchfork reports can be included alongside SCA incidents, allowing a unified vendor testing process.

Using Pitchfork specifications, certification authorities can independently extend API-based testing to include APIs that are proprietary (i.e., not-public) to each national program, without having to divulge sensitive information to the testing tools vendors. This added independence should be useful for national programs that need to enhance the public APIs (SCA, JTRS, ESSOR, WInnF, etc.) with their own national or regional requirements.

Future Directions

We have defined Pitchfork patterns in terms of an abstracted procedural syntax, which works well for C-like languages, but this could be augmented for other kinds of structured syntax such as XML. To support this, the LHS would need a tag indicating the kind of syntax intended for the pattern. It may also be useful to establish constraints on when rules could be applied, for example only within particular files.

Constraining the application of rules in the above and other ways could be handled conveniently through the addition of side-condition syntax to the LHS. A side condition is an expression, referring to variables bound in the pattern, that evaluates to true or false. The LHS is considered to match only if the expression evaluates to true.

Pitchfork adopts regular expressions as its central idiom for describing patterns of code to be matched. This has two main advantages: the descriptions are straightforward for the user and they are efficient for R-Check SCA to implement. Some code patterns, however, require a more powerful language (e.g., some form of grammar). A fully-general way to enhance matching is through extension of the action part of the language. An "event" action could be defined that has the effect of setting a named event state. A corresponding named event pattern would be defined for use on the LHS, allowing the match only when the event state is set. The net effect would be to allow meta-rules to be defined, comprised of groups of rules linked by named event states.

6. TOWARD PRE-CERTIFICATION AND SELF-CERTIFICATION

While pre-testing eases the task of certifying vendor software against the specification, the Test Lab nevertheless remains a bottleneck in the SCA compliance testing process because it must test from scratch. A natural idea for addressing this bottleneck is to enlist the aid of the software producers themselves to do pre-certification. Software producers are already motivated to check their own software in order to prevent costly delays due to certification failures. Having the software producer actually perform the checking and deliver the results to the Test Lab would effectively parallelize part of the testing workload. In this scenario, the Test Lab would simply need to perform a check on the results themselves, which could be done much more quickly.

The prime difficulty of vendor-assisted pre-certification is retaining the high confidence in the results that stems from the use of an independent certification authority in the first place. One possible way to increase this confidence is by using code-signing techniques similar to those used for cryptographic applications. The checking software would produce a checksum as part of the report that is unique to the combination of test software, test configuration, test invocation options, SCA XML project files, IDL, source code and test results. The vendor would deliver the test results, with included checksum, which could then be quickly checked for validity using a simple "result checker" utility against the other artifacts. If the test results or any of the artifacts are altered, the checksum will not match. If the checksum matches, the result can be treated as authoritative for those software artifacts and test configuration.

Self-certification of compliance with automatically testable SCA requirements would require that the "result checker" be available to the software consumer. In cases where the vendor is unwilling to make source code available (such as for intellectual property reasons), the test result would have to be linked to the binary image. A separate assurance that the binary is the product of the tested source code would then have to be held in escrow by a trusted entity. We leave further discussion of such a scheme as future work.

The security of the signing approach depends on the integrity of the checksum. The vendor should not be able to manually calculate the checksum for any given code base and set of test results; only the test software approved by the Test Lab should be able to do this. To achieve this, the test software could have embedded a cryptographically secure hash function, such as SHA-1 [24]. Subverting this checksum would, however, still be possible by reverse-engineering the test software. Preventing this kind of subversion would require a client-server architecture, where vendor software is submitted to a server running the checking software under the control of the Test Lab or trusted tool vendor.

7. CONCLUSION

Reaping the benefits of the SCA – reduced risk, cost and time-to-market, enhanced communications interoperability, and simplified insertion of new communications capabilities – depends on developing and deploying truly SCA-compliant applications and core frameworks. The SCA spans hundreds of requirements of varying complexity, and assuring compliance requires investment in tools and processes dedicated to the task. Fortunately, modern

development environments provide opportunities to simplify and accelerate the compliance testing process.

Manually implementing systems based on software components can be very difficult. MBDEs automate the generation of the code required for the deployment and interoperation of components. For business logic, static testing provides unbiased inspection of all software paths and can be used to find latent issues that do not manifest on particular platforms or in scripted test executions. Component models admit automated unit tests that can be used to verify proper execution. Dynamic, model-driven methods use known examples and error conditions to check for correctness. All of these methods can be run as code is being developed, allowing non-compliant code to be fixed earlier in the development cycle.

When the testing tools used by the certification authority are commercially available, vendors can engage in pre-testing by applying to their SCA artifacts the same test software that the certification authority uses. Figures indicate that in the early days of the Joint Tactical Radio System (JTRS) program, when most of the development was being done manually, certifying an SCA system would take several months at a cost of over $200,000 per radio platform product release [25]. As demonstrated with the GateHouse experience, pre-testing enabled by advanced development tools has the potential to substantially accelerate the SCA certification process.

The trend in both software development and compliance testing is towards increased automation. We expect this trend to continue and intensify. Advances in static analysis, reflected through languages such as Pitchfork, will allow more precise and flexible specifications spanning all artifacts of MBDE-generated software (XML and IDL as well as C and C++ source code). In addition, there will be increased cooperation between and among software developers and compliance testers, as they share knowledge expressed as mechanically-checkable rules, share the certification task, and share the benefits of interoperability.

8. REFERENCES

[1] W. Kishaba, Raytheon, "Testimonials – Combined Presentation", SCA 4.1 Standard Preview Workshop, October, 2014, pp 49-52.

[2] JPEO JTRS Test and Evaluation Laboratory (JTEL) SCA 2.2.2 Application Requirements List version 2.2 Release Notes, July 8, 2010.

[3] JPEO JTRS Test and Evaluation Laboratory (JTEL) SCA 2.2.2 OE Requirements List version 2.2 Release Notes, November 4, 2010.

[4] J. Ezick and J. Springer, "The Benefits of Static Compliance Testing for SCA Next," Proceedings of the SDR'11 WInnComm Technical Conference, November 2011.

[5] Caper Jones, "Programming Languages Table", Software Productivity Research, 1996.

[6] S. Kelly, "Improving Developer Productivity with Domain-Specific Modeling Languages", DeveloperDotStar.com, 2005.

[7] J. Greenfield and K. Short, "Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools", from John Wiley and Sons. 2004.

[8] The Standish Group, "CHAOS Manifesto 2013".

[9] B. Councill, G. T. Heineman, "Component-Based Software Engineering: Putting the Pieces Together", Addison-Wesley Professional, 2001.

[10] D. S. Frankel, S. Cook, "Domain-Specific Modeling and Model Driven Architecture", MDA Journal, 2004.

[11] J. Dong, "Research on Heterogeneous Component Assembly Problems Based on Software Product Line", Proceedings of the 2nd International Conference On Systems Engineering and Modeling (ICSEM-13), 2013.

[12] P. C. Clements, "From Subroutines to Subsystems: Component-Based Software Development", November 1995, Software Engineering Institute.

[13] M. Broy, S. Kirstan, H. Krcmar, and B. Schatz, "What is the Benefit of a Model-Based Design of Embedded Software Systems in the Car Industry?", In Emerging Technologies for the Evolution and Maintenance of Software Models. ICI, 2011.

[14] Wireless Innovation Forum, "SCA Standards for Defense Communications: Global Adoption, Proven Performance", 2014.

[15] M. Turner, "Waveform Applications Porting – Experience in Moving Past the Myths and Legends", Proceedings of the SDR'10 Technical Conference, March, 2010.

[16] K. Dingman, A. Dibernado, "Porting…It's More Than Just Software", Waveform Portability Workshop, January, 2014.

[17] C. Rathfelder, H. Groenda, R. Reussner, "Software Industrialization and Architecture Certification", Industrialisierung des Software Management. 2008.

[18] J. Botella, E. Jaffuel, B. Legeard, F. Peureux, "Model-Based Testing for SCA Conformance", In Proceedings of WInnComm Europe 2014.

[19] NordiaSoft, "NordiaSoft Technologies Pave the Way to SCA Certification for GateHouse", http://www.nordiasoft.com/#!news/c1jg6

[20] SDR Forum, Test and Certification Guide for SDRs based on SCA, Part 1: SCA, SDRF-08-P-0007-V1.0.0, April 2009.

[21] S. J. Garland, J. V. Guttag, J. J. Horning, "An Overview of Larch," in Functional Programming, Concurrency, Simulation and Automated Reasoning, Peter E. Lauder (editor), LNCS 693, Springer, July 1993, pp 329-348.

[22] D. Engler and M. Musuvathi, "Static Analysis Versus Software Model Checking for Bug Finding," in *Verification, Model Checking, and Abstract Interpretation*, LNCS vol. 2937, 2004, pp 191-210.

[23] S. Hallem, B. Chelf, Y. Xie, D. Engler, "A system and language for building system-specific, static analyses," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation* (PLDI), 2002, pp 69-82.

[24] The Internet Society, US Secure Hash Algorithm 1 (SHA1), 2001. http://tools.ietf.org/html/rfc3174

[25] M. Turner, "Global Military SDR Solutions – Practical Methods for SCA Radio Compliance and Deployment", Proceedings of the SDR'09 Technical Conference, December, 2009.