Using DITA to Create Security Configuration Checklists

A Case Study

Joshua Lubell
Computer Scientist
National Institute of Standards and Technology
<lubell@nist.gov>

Abstract

Many software tools use security configuration checklists expressed in the Extensible Configuration Checklist Description Format (XCCDF) to monitor computers and other information technology products for compliance with security policies. But XCCDF syntax is checklist author-unfriendly. And complex relationships and dependencies between and among checklist rules, checking instructions, and software platforms make it difficult to reuse or repurpose existing XCCDF content in new checklists. The Darwin Information Typing Architecture (DITA) can tame XCCDF syntax and facilitate content management and reuse. A case study comparing the use of specialization and other DITA features with a currently-deployed ad hoc XCCDF authoring system demonstrates the advantages of the DITA approach.

Table of Contents

1. Introduction
2. XCCDF Overview and Example
3. The SCAP Security Guide Approach
4. Specialized DITA Element Types for XCCDF
   Rule Element Type
   Profile Element Type
   Other Element Types
5. Reuse and Implementation Considerations
   Block and Inline Content Reuse
   Code Reuse
6. Conclusion
Appendix. List of Acronyms

1. Introduction

checklist developers the pain of having to learn multiple proprietary formats and lowers deployment barriers for automated configuration checking. XCCDF-expressed checklists can be used with any SCAP-conforming software product.

XCCDF is a powerful and versatile language, but it is not checklist author-friendly. Although XCCDF enables automation of security configuration scans and generation of human-readable documentation, it is of limited use in helping authors reuse existing content when developing a new checklist. XCCDF is designed in a modular manner to facilitate reuse of sub-elements across different checklists. However, as checklist target types proliferate, efficient management of such content becomes increasingly challenging. Also, XCCDF’s syntax is verbose and cluttered with namespaces, making checklist authoring labor-intensive and error-prone.

These shortcomings were of less concern in the recent past when most computing occurred on commodity hardware running one of a small number of popular operating systems. But today the mobile computing revolution and the “Internet of Things” are spawning a greater variety of devices, operating systems, and applications that need protection from cyber-threats. Creating XCCDF content for this multitude of platforms is becoming untenable. Checklist authors need better tools to cope with this new diversity — and with the ever-increasing need to prevent security incidents.

This paper explores how the Darwin Information Typing Architecture (DITA) OASIS-DITA can improve life for XCCDF checklist authors. Section 2 provides an overview of XCCDF and introduces a sample checklist to be used as a case study. Section 3 describes the ad hoc approach the SCAP Security Guide (SSG) OpenSCAP-SSG project uses to address XCCDF and SCAP authoring and reuse challenges. Section 4 illustrates how DITA “topic” and “map” element type specializations can be used to improve the XCCDF authoring experience. Section 5 discusses how DITA can meet reuse challenges in a more robust and maintainable manner than the SSG approach described. Section 6 summarizes the case study results and offers some concluding remarks.

2. XCCDF Overview and Example

The XCCDF specification Waltermire defines XCCDF’s data model and processing semantics. The current version of XCCDF is 1.2. The specification references a schema XCCDF as XCCDF’s normative XML representation. This schema's `<Benchmark>` element is the root element of a checklist, or “benchmark document” using XCCDF terminology. A `<Benchmark>` element contains a collection of `<Rule>`, `<Value>`, and `<Group>` elements. A `<Rule>` specifies a single item to check, such as a firewall's default setting. A `<Rule>` also specifies how the checking should be done, such as with an implementation-specific scripting language or with the Open Vulnerability Assessment Language (OVAL) OVAL, an SCAP standard for representing system configuration information, assessing machine state, and reporting assessment results. A `<Value>` represents a named parameter that can be used within rules and tailored for a particular configuration scenario. A `<Group>`
collects <Rule>, <Value>, and other <Group> elements into an aggregation that is meaningful to a checklist user, for example, a collection of firewall configuration settings.

A <Benchmark> element also contains one or more <Profile> elements. A <Profile> is a named collection of references to <Group>, <Rule>, and <Value> elements. Profiles allow different combinations of groups and rules to be enabled so that they are included in a series of tests, or disabled so that they are not. Profiles can also tailor the value of the named parameters in <Value> elements. Profiles allow a single benchmark to support different test scenarios by enabling different combinations of tests and parameter values. XCCDF's <Profile> element increases a benchmark document's versatility, but also increases the document's complexity by adding many cross references.

This paper uses as an ongoing example a simple XCCDF checklist consisting of four ungrouped rules for version 7 of Red Hat Enterprise Linux (RHEL7) [2], none of which use any <Value> parameters. The example was extracted from a more complex XCCDF checklist developed by the Center for Internet Security CIS. Two of the rules pertain to SELinux StPierre, a Linux kernel module for Mandatory Access Control (MAC). MAC is the security principle of limiting the ability of a user or running application to access system resources Hu. The other two rules apply to firewall configuration. The checklist defines two profiles:

- “Firewall with MAC” referencing all four rules,
- “Firewall” referencing only the two firewall-related rules.

Figure 1

```
<Benchmark xmlns="http://checklists.nist.gov/xccdf/1.2"
  id="xccdf_gov.nist_benchmark_Red_Hat_Enterprise_Linux_7_Benchmark"
  style="SCAP 1.2">
  <status date="2016-06-02">interim</status>
  <title>Red Hat Enterprise Linux 7 Benchmark</title>
  <description>…</description>
  <version>2.1.0</version>
  <metadata>…</metadata>
  <Profile id="xccdf_gov.nist_profile_Firewall">…</Profile>
  <Profile id="xccdf_gov.nist_profile_Firewall_with_MAC">…</Profile>
  <Rule id="xccdf_gov.nist_rule_Ensure_SELinux_not_disabled_in_bootloader_configuration" selected="false">…</Rule>
  <Rule id="xccdf_gov.nist_rule_Ensure_SELinux_is_installed" selected="false">…</Rule>
  <Rule id="xccdf_gov.nist_rule_Ensure_iptables_is_installed" selected="false">…</Rule>
  <Rule id="xccdf_gov.nist_rule_Ensure_firewall_rules_exist_for_all_open_ports" selected="false">…</Rule>
</Benchmark>
```

XCCDF checklist representing RHEL7 example.
Figure 1 shows the high-level XCCDF representation of this checklist. Ellipsis symbols indicate lower-level details omitted for brevity. In this checklist, all rules have a \texttt{@selected} attribute set to “false”. Thus, no rules in the benchmark document are processed by default. Instead, each profile must explicitly select the rules it wishes to include in its test scenario (as shown later in Figure 3).

Figure 2 shows the checklist loaded as input into SCAP Workbench OpenSCAP-Workbench, a security configuration scanner user interface. The user has selected the “Firewall with MAC” profile. The “Rules” pane displays the title and description of each rule in the profile. If the user were to click on the “Scan” button, the software would scan a remote machine to determine compliance with the four profile rules and report the results back to the user.

Figure 3 shows the XCCDF representation of the “Firewall with MAC” profile. The \texttt{<select>} elements reference the rules comprising the profile. Each \texttt{<select>} element has an \texttt{@idref} attribute identifying a rule and a \texttt{@selected} attribute set to “true” to explicitly include the referenced rule for use in the profile's test scenario.

The XCCDF markup shown in Figure 1 and Figure 3 exhibits the following characteristics:
• Identifiers are long. XCCDF requires that identifiers be both unique and descriptive, and that they contain a reverse-Domain Name System (DNS) style string associated with the content author. The profile identifier and four rule identifiers all meet these requirements.

• Rule identifiers are repetitive. They all start with "xccdf_gov.nist_rule_".

These characteristics promote interoperability and completeness, but at the expense of author-friendliness and maintainability.

Now consider Figure 4, the XCCDF representation of the first of the four rules referenced by the “Firewall with MAC” profile. This rule checks whether SELinux is enabled. The <description> and <rationale> elements document the rule and its justification. The <complex-check> element represents a Boolean expression of <check> elements, which in turn reference the code for determining whether the system being scanned complies with the rule. In this example, checking is done using OVAL. Each <check-content-ref> element's @href attribute references an external file containing OVAL definitions, and the @name attribute identifies the particular OVAL definition used to perform the check. Like XCCDF, OVAL requires identifiers to be descriptive and use reverse-DNS syntax.

![Figure 3: Profile in XCCDF](image)

The XML in Figure 4 exhibits the same XCCDF characteristics as the profile XML shown in Figure 3. Additionally:

• There are multiple Uniform Resource Identifiers (URIs): the XCCDF default namespace http://checklists.nist.gov/xccdf/1.2 and the OVAL URI http://oval.mitre.org/XMLSchema/oval-definitions-5.

• The markup representing the checking of the rule is verbose and complex.

The rule's <complex-check> markup complexity is partly because the check requires a Boolean expression referencing three OVAL definitions, but the <check-content-ref> syntax and
repetitive @system attributes further exacerbate matters. This added verbosity is perhaps necessary to allow for possibilities such as check systems other than OVAL, and embedded code for performing a check. However, given that SCAP-conforming tools are required to support OVAL as a checking language in XCCDF, and most XCCDF benchmarks use OVAL, this extra verbosity is generally not needed in practice.

Figure 4

```xml
<Rule xmlns="http://checklists.nist.gov/xccdf/1.2" id="xccdf_gov.nist_rule_SELinux_not_disabled_in_bootloader_configuration" selected="false">
  <title>Ensure SELinux is not disabled in bootloader configuration</title>
  <description>Configure SELinux to be enabled at boot time and verify that it has not been overwritten by the grub boot parameters.</description>
  <rationale>SELinux must be enabled at boot time in your grub configuration to ensure that the controls it provides are not overridden.</rationale>
  <complex-check operator="OR">
    <complex-check operator="AND">
      <check system="http://oval.mitre.org/XMLSchema/oval-definitions-5">
        <check-content-ref href="CIS_Red_Hat_Enterprise_Linux_7_Benchmark_v2.1.0-oval.xml" name="oval:gov.nist.redhat_redhat_enterprise_linux_7:def:1058"/>
      </check>
      <check system="http://oval.mitre.org/XMLSchema/oval-definitions-5">
        <check-content-ref href="CIS_Red_Hat_Enterprise_Linux_7_Benchmark_v2.1.0-oval.xml" name="oval:gov.nist.redhat_redhat_enterprise_linux_7:def:1059"/>
      </check>
    </complex-check>
    <check system="http://oval.mitre.org/XMLSchema/oval-definitions-5">
      <check-content-ref href="CIS_Red_Hat_Enterprise_Linux_7_Benchmark_v2.1.0-oval.xml" name="oval:gov.nist.redhat_redhat_enterprise_linux_7:def:1060"/>
    </check>
  </complex-check>
</Rule>
```

Rule in XCCDF.

The discussion so far has focused on XCCDF characteristics that impede authoring such as verbosity and redundancy. An additional concern is that XCCDF and other SCAP languages do not facilitate reuse of content applicable to multiple platforms Lubell2017. To see why, consider as an example the third rule shown in Figure 2, “Ensure iptables is installed.” Iptables, an application for configuring the Linux kernel firewall, is available in all Linux distributions. Therefore, this rule is potentially applicable to all Linux systems.

A reasonable way to check for compliance with this rule would be for the rule's OVAL definition to invoke RHEL's package manager, rpm, to verify that iptables is installed. But unlike iptables, not all Linux distributions come with rpm. Fedora, an open source Linux distribution closely related to
RHEL, has rpm. On the other hand, Debian and Ubuntu (a Debian derivative) both use dpkg for package management. Figure 5 shows the relationships between the Debian, Ubuntu, RHEL, and Fedora Linux distributions in terms of their shared kernel and package manager components.

This example illustrates the problem of platform fragmentation. Platform fragmentation occurs when the same operating system, software application, or hardware component is bundled by multiple entities — with each “bundler” providing different customizations Vecchiato. Although XCCDF and other SCAP languages allow for associations between rules or collections of rules and platforms, the SCAP ecosystem currently does not provide the kind of guidance shown in Figure 5 to checklist authors. Therefore, checklist authors need other methods and tools outside the scope of SCAP to help them leverage taxonomic and other relationships to maximize reuse when creating new checklists. The next section discusses how this is done in the SSG project.

3. The SCAP Security Guide Approach

The SSG OpenSCAP-SSG is an open source project whose output is a growing collection of SCAP-expressed content (security guides) for Linux distributions and software applications. This SSG-generated SCAP content is widely used in government and industry. Current and recent users include cloud computing providers, national defense agencies, and the financial services and airline industries [3]. The SSG “source code” consists of:

- Scripts and Extensible Style Language Transformations (XSLT) W3C-XSLT for generating SCAP content,
- XML files that serve as input to the scripts and XSLT.

To deal with platform fragmentation and to facilitate source code management, the source code is modularized. Shareable module files applicable to two or more security guides reside in a single “shared” directory tree. Platform-specific module files applicable only to an individual security guide reside in platform-specific directory trees. The XML input files are further modularized into logically-related checklist components such as profiles, groups, and OVAL definitions. Building SCAP security guides from the source requires running scripts that perform XML transformations, macro substitutions, and merging of the XML input modules into bigger SCAP-conforming files.
SSG contributors create rules, profiles, and other XML input modules using a shorthand XML vocabulary. The shorthand is less verbose and namespace-heavy than the XCCDF markup shown in Section 2. The SSG build process converts the shorthand to XCCDF and OVAL. This build process is complicated and requires SSG contributors to understand not only SCAP, but also the one-off way the SSG source files are organized and structured Preisler.

Figure 6 shows a high-level view of the SSG source code directory structure. The shared directory subtree shown in the left-hand column contains content, XSLT, and scripts applicable to more than one Linux distribution. Platform-specific subtrees contain content, transforms, and scripts applicable only to a single distribution or application. The right-hand column shows the subtree for RHEL7. Each distribution-specific directory tree has a Makefile with targets needed for building SCAP content for that distribution. A master Makefile in the source code root directory has targets for building all the SCAP content.

All rules and profiles in the source code are defined using the SSG shorthand XML. Rules that are common across multiple Linux distributions, such as the rule for determining whether SELinux is enabled, reside in shared/xccdf. Rules specific to a particular Linux distribution are in the distribution's input directory subtree. Since all rules for RHEL7 come from shared/xccdf, the RHEL7 input directory has no xccdf subdirectory. Profiles are distribution-specific and reside in the distribution's input/profiles directory.

XSLT stylesheets in a platform-specific transforms directory generate SCAP-conforming XML, such as the RHEL7 benchmark shown in Figure 1. These stylesheets use <xsl:include> elements to incorporate stylesheets from ../../../shared/transforms. The stylesheets first combine the individual shorthand XML files (both shared and distribution-specific) into a single shorthand benchmark file. They then transform the combined shorthand benchmark into an XCCDF-conforming benchmark document.
The SSG XSLT stylesheets support reuse not only of structural checklist components such as rules, but also of inline fragments. The stylesheets automatically insert frequently-occurring fragments when feasible to do so. For example, the value of the @system attribute in all XCCDF rules using OVAL is the same URI: http://oval.mitre.org/XMLSchema/oval-definitions-5. Therefore, the shorthand XML syntax omits @system and its value, saving authors the trouble of having to repeatedly specify it.

For reusable fragments where the author determines placement, the SSG shorthand employs “-macro” elements. For example, a RHEL7 checklist author might want to use the full name of the product to which the checklist applies — “Red Hat Enterprise Linux 7” — in multiple places within elements where mixed content is allowed. To save authors the trouble of having to specify a full product name for every occurrence, the RHEL-specific XSLT in RHEL/7/transforms contains the following variable declaration:

```
<xsl:variable name="product_long_name">Red Hat Enterprise Linux 7</xsl:variable>
```

The stylesheet in shared(transforms for converting the combined shorthand XML components into an XCCDF benchmark contains the following template rule:

```
<xsl:template match="product-name-macro">
  <xsl:value-of select="$product_long_name"/>
</xsl:template>
```

The template rule and variable declaration enable an SSG author to specify a product name inline as <product-name-macro/>. Doing so not only minimizes the opportunity for undetected typos, but also maximizes the reusability of the <rule> or other structural element containing the product name.

![Diagram](image)

SCAP Security Guide transformation from shorthand to XCCDF.

Figure 7 summarizes the overall SSG transformation process discussed in the preceding paragraphs. The stylesheets first combine individual shorthand XML files and then, using constants defined as variable declarations and macros defined as template rules, produce an XCCDF-conforming benchmark document.
4. Specialized DITA Element Types for XCCDF

The DITA standard OASIS-DITA is an XML-based architecture for creating, managing, reusing, and delivering technical content. A variety of authoring applications and processors — commercial as well as free and open source — conform to DITA OASIS-DITA-xml.org. DITA has two basic building blocks: the “topic” and “map” element types. A “topic” is a reusable chunk of information. A “map” is an aggregation of topics or other maps. DITA supports reuse not only of topics and maps, but also of inline XML elements and fragments.

DITA information types are specializable. Specialization is an information modeling technique that helps avoid inconsistency and facilitate interoperability Krima. A specialized information type is a refinement of an existing base type and therefore must be at least as constrained. Specialization imposes some discipline on designers of new DITA information types, with the upside that implementations can easily leverage other DITA-conforming implementations Kimber. DITA includes built-in specialized element types based on the “topic” base type, such as “concept”, “task”, “reference”, and “glossary group”. DITA also allows for definition of new specialized element types based on “topic”, “map”, or other specialized types.

In this section, I define new DITA specialized element types to support the authoring of XCCDF rules and profiles. I also provide examples of other XCCDF elements for which DITA specializations could be defined. These types have the same advantages as the XCCDF shorthand vocabulary, but with the added benefits of validation and more author-friendliness when using DITA-aware XML editing software. The specialization approach follows guidance from the DITA standard OASIS-DITA and from Eliot Kimber’s DITA Configuration and Specialization Tutorials Kimber.

**Rule Element Type**

To develop a specialized DITA information type for XCCDF rule authoring, let us begin by revisiting the “Ensure SELinux is not disabled in bootloader configuration” rule shown in Figure 4. A DITA “rule” element type should mitigate the three XCCDF authoring barriers highlighted in Section 2:

1. Repetitive and overly long identifiers,
2. Namespace proliferation,
3. Overly verbose check expressions.

I choose to define the new “rule” element type as a specialization of the DITA “concept” topic type. A “concept” has the loosest content model of DITA's built-in topic-based types, making it easy to specialize. `<concept>`, the “concept” topic type's root element, contains a `<title>` followed by `<conbody>`, the main body element. `<conbody>` may contain DITA `<section>` elements, which subdivide content within a topic and are not allowed to be nested. DITA `<sectiondiv>` elements subdivide content within a `<section>` and may be nested inside one
another. `<section>` and `<sectiondiv>` may contain mixed content including `<xref>`, a DITA linking element. `<xref>` has an @href attribute, which may reference another topic, map, or non-DITA resource.

The next step is to create a “rule” document type shell based on the “concept” document type shell. A DITA document type shell defines which elements and attributes are allowed in a DITA document, and is usually specified using Document Type Definition (DTD) syntax. The DITA standard provides a modular architecture for document type shells to facilitate creation of new shells. A recommended practice is to make a copy of an existing shell (the “concept” shell in our case) and modify the modules as needed for the new document type.

Figure 8 shows the “Ensure SELinux is not disabled in bootloader configuration” rule represented in a manner similar in spirit to the SSG shorthand syntax, which mitigates the XCCDF rule authoring barriers. The new “rule” document type shell will specify a grammar for authoring rules such as the rule in Figure 8.

**Figure 8**

```dita
<rule id="SELinux_not_disabled_in_bootloader_configuration">
  <title>Ensure SELinux is not disabled in bootloader configuration</title>
  <rulebody>
    <description>Configure SELINUX to be enabled at boot time and verify that it has not been overwritten by the grub boot parameters.</description>
    <rationale>SELinux must be enabled at boot time in your grub configuration to ensure that the controls it provides are not overridden.</rationale>
    <check>
      <OR>
        <AND>
          <oval href="oval/1058.dita"/>
          <oval href="oval/1059.dita"/>
        </AND>
        <oval href="oval/1060.dita"/>
      </OR>
    </check>
  </rulebody>
</rule>
```

Rule represented using specialization of concept DITA topic [4].

Looking at Figure 8 in the context of the DITA “concept” element type, one observes the following:

- The `<oval>` element can be specialized from DITA's `<xref>` element, without additional constraints.
- `<OR>` and `<AND>` can be specialized from DITA's `<sectiondiv>` element, constrained to allow one or more `<oval>` elements or Boolean expressions as content.
- `<check>` can be specialized from DITA's `<section>` element, constrained to allow either a single `<oval>` element or Boolean expression of `<oval>` elements as content.
• `<description>` and `<rationale>` can be specializations of `<section>` without additional constraints.
• `<rulebody>` can be specialized from `<conbody>`, but constrained to allow only a `<description>`, `<rationale>`, and `<check>` as content.
• `<rule>` can be specialized from `<concept>`, but constrained to allow only a `<title>` and `<rulebody>` as content.
• The DITA `<title>` element can be used as is.

After creating a “rule” document type shell reflecting these observations, I add DITA @class attributes with default values to each specialized element: `<rule>`, `<rulebody>`, `<description>`, `<rationale>`, `<rationale>`, `<check>`, `<AND>`, `<OR>`, and `<oval>`. Default values are used to hide the DITA specialization machinery from authors, who have no need to see it when editing content. The @class attribute specifies the element's specialization hierarchy — a mapping from the element name to its more generalized DITA concept and topic element equivalents. For example, the `<rule>` element's @class attribute has the value "- topic/topic concept/concept rule/rule " , which specialization-aware DITA processors interpret to mean

The `<rule>` element in the “rule” element type specializes `<concept>` from the “concept” element type, which in turn specializes `<topic>` from the “topic” element type.

<table>
<thead>
<tr>
<th>Element</th>
<th>Specialization Hierarchy (@class value)</th>
<th>Document Type Shell Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;rule&gt;</code></td>
<td>&quot;- topic/topic concept/concept rule/rule &quot;</td>
<td>(title, rulebody)</td>
</tr>
<tr>
<td><code>&lt;rulebody&gt;</code></td>
<td>&quot;- topic/body concept/conbody rule/rulebody &quot;</td>
<td>(description, rationale, check)</td>
</tr>
<tr>
<td><code>&lt;description&gt;</code></td>
<td>&quot;- topic/section concept/section rule/description &quot;</td>
<td>none</td>
</tr>
<tr>
<td><code>&lt;rationale&gt;</code></td>
<td>&quot;- topic/section concept/section rule/rationale &quot;</td>
<td>none</td>
</tr>
<tr>
<td><code>&lt;check&gt;</code></td>
<td>&quot;- topic/section concept/section rule/check &quot;</td>
<td>(OR</td>
</tr>
<tr>
<td><code>&lt;OR&gt;</code></td>
<td>&quot;- topic/sectiondiv concept/sectiondiv rule/OR &quot;</td>
<td>(OR</td>
</tr>
<tr>
<td><code>&lt;AND&gt;</code></td>
<td>&quot;- topic/sectiondiv concept/sectiondiv rule/AND &quot;</td>
<td>(OR</td>
</tr>
<tr>
<td><code>&lt;oval&gt;</code></td>
<td>&quot;- topic/xref concept/xref rule/oval &quot;</td>
<td>none</td>
</tr>
</tbody>
</table>

Table I shows the specialization hierarchy and document type shell constraint for each new element in the “rule” element type. The specialization hierarchy effectively specifies the default specialization-aware DITA processing behavior. DITA specialization hierarchies maximize reuse
of existing markup and transformation code Priestley. The document type shell constraints specify requirements for valid XML.

Suppose an XCCDF rule author were to use a DITA specialization-aware XML application such as Oxygen XML Editor. Figure 9 shows the “Ensure SELinux is not disabled in bootloader configuration” rule as presented in Oxygen's user interface. The screen capture occurred when the user was about to insert a new element following the <oval> element whose @href value is oval/1059.dita. A simple Cascading Style Sheet (CSS) W3C-CSS augments Oxygen's CSS styling for the “concept” element type by adding “Rule:”, “Description:”, “Rationale:”, and “Check:” labels, as well as the parentheses and “OR” and “AND” in the check expression. This is all the CSS code does. Other presentation aspects such as the fonts used and the clickable @href links are derived using the specialization hierarchies shown in Table I.

![Figure 9](image)

**Rule: Ensure SELinux is not disabled in bootloader configuration**

- **Description:** Configure SELINUX to be enabled at boot time and verify that it has not been overwritten by the grub boot parameters.
- **Rationale:** SELinux must be enabled at boot time in your grub configuration to ensure that the controls it provides are not overridden.
- **Check:**

  (OR
  (AND <oval/1058.dita> <oval/1059.dita>)
  <oval/1060.dita>)

Rule in DITA specialization-aware authoring environment.

Analogous to the CSS augmentation of the default DITA presentation, the rule document type shell constraints shown in Table I augment the default DITA visual editing and validation capabilities. The drop-down list of elements shown in Figure 9 limits the user's choices to inserting an <AND>, <OR>, or <oval> element, or adding a new <AND> after the current <AND> element. Any DITA “concept” topic document type shell constraints not overridden by the “rule” document type shell remain. For example, the user can add any content inside the <description> or <rationale> elements permitted by the “concept” document type shell's content model for <section>.
**Profile Element Type**

I choose to define the new “profile” element type as a simple specialization of the DITA “map” element type. `<map>`, the “map” type's root element, may contain a `<title>`, followed by a `<topicmeta>` element, followed by a sequence of `<topicref>` elements. `<topicmeta>`, in this context, specifies metadata applicable to all topics in the map, and may contain a brief description of the map in a `<shortdesc>` element. The `<topicref>` element uses `@href` to reference a map resource.

I first create a “profile” document type shell based on the “map” document type shell. Figure 10 shows the “Firewall with MAC” profile represented in a manner similar in spirit to the SSG shorthand syntax, and that mitigates the XCCDF authoring barriers discussed in Section 2.

---

**Figure 10**

```xml
<profile id="Firewall_with_MAC">
  <title>Firewall with MAC</title>
  <profilemeta>
    <shortdesc>This profile extends the "Firewall" profile to check configuration of Mandatory Access Control (MAC).</shortdesc>
  </profilemeta>
  <rule href="rules/SELinux_not_disabled_in_bootloader_configuration"/>
  <rule href="rules/SELinux_is_installed"/>
  <rule href="rules/iptables_is_installed"/>
  <rule href="rules/firewall_rules_exist_for_all_open_ports"/>
</profile>
```

Profile represented using specialization of DITA `<map>`.

---

**Table II**

Specialization hierarchies and document type shell constraints for each “profile” element.

<table>
<thead>
<tr>
<th>Element</th>
<th>Specialization Hierarchy (@class value)</th>
<th>Document Type Shell Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;profile&gt;</code></td>
<td>&quot;- map/map profile/profile &quot;</td>
<td>(title, profilemeta, rule+)</td>
</tr>
<tr>
<td><code>&lt;profilemeta&gt;</code></td>
<td>&quot;- map/topicmeta profile/profilemeta &quot;</td>
<td>none</td>
</tr>
<tr>
<td><code>&lt;rule&gt;</code></td>
<td>&quot;- map/topicref profile/rule &quot;</td>
<td>none</td>
</tr>
</tbody>
</table>

Looking at Figure 10 in the context of the DITA “map” document type, one observes the following:

- `<rule>` can be specialized from DITA's `<topicref>` element, without additional constraints.
- `<profilemeta>` can be specialized from DITA's `<topicmeta>` element, without additional constraints.
- `<profile>` can be specialized from DITA's `<map>` element, but constrained to allow only a `<title>`, `<profilemeta>`, and one or more `<rule>` elements as content.
- The DITA `<title>` and `<shortdesc>` elements can be used as is.
Table II shows the specialization hierarchy and document type shell constraint for each new element in the “profile” type. Figure 11 shows the Oxygen presentation with clickable @href links. Minimal CSS styling adds the “Profile:” label. All other presentation characteristics are inherited from the “map” element type's CSS styling.

![Profile: Firewall with MAC](image)

Profile in DITA specialization-aware authoring environment.

**Other Element Types**

The previous subsections demonstrated how shorthand XML vocabularies for XCCDF rules and profiles can be implemented as specializations of the DITA “concept” and “map” element types respectively. Specialization can also be used to create DITA element types for XCCDF groups and values as well. A “group” type could be specialized from the DITA “map” element type, and “value” can be specialized from the base “topic” type or one of the DITA built-in types specialized from “topic”.

An XCCDF benchmark document as a whole could be represented using a “benchmark” element type specializing “map”. Doing so would explicitly represent the high-level checklist structure, which the SSG build system represents implicitly in Makefiles and in the XSLT implementing the “Transform shorthand to XCCDF” box in the Figure 7 flowchart. DITA maps have additional benefits besides making the transformation process more explicit and self-documenting for XCCDF authors. As the next section will discuss, maps enable reuse of XCCDF content and can reduce the coding needed to implement the shorthand-to-XCCDF transformation.

**5. Reuse and Implementation Considerations**

This section revisits the SSG approach covered in Section 3 from a reuse perspective. The first subsection describes how maps and other DITA features can improve upon SSG solutions for reusing XCCDF content. The second subsection discusses how the use of DITA can simplify implementation of the conversion from shorthand XML to XCCDF.
Block and Inline Content Reuse

DITA maps can improve upon the SSG “shared/xcddf” reuse mechanism. Relying on directory subtrees imposes a rigid hierarchy on reuse relationships, and is insufficient to account for all the varieties of platform fragmentation. There are a lot ways to categorize platforms. The package manager-based classification shown in Figure 5 is just one of many. Lineage is another way to characterize a platform. For example, Ubuntu Linux is derived from Debian, while RHEL is a commercial sibling of the open source Fedora. Linux distributions can also be classified according to graphical desktop environment. For example, some Linux distributions come bundled with GNOME, while others come with Enlightenment. And GNOME and Enlightenment each have variants. Linux distributions optimized for specialized hardware, increasingly common as the “Internet of Things” continues to grow, add yet another wrinkle to platform fragmentation. Raspbian, a Debian derivative tailored for Raspberry Pi hardware, is an example of a hardware-optimized Linux distribution.

Multiple DITA maps can be used to slice and dice a repository of DITA resources for creating XCCDF benchmarks covering a wide variety of Linux-based platforms. Each map can represent a different platform class. And maps can overlap with one another, or a map can reference another map. Maps can capture parent-child relationships as well as sibling relationships. Also, DITA supports conditional profiling OASIS-DITA, which allows the same content to be associated with multiple organization schemes as defined using DITA’s filtering attributes. The classification possibilities are endless. Unlike the SSG shared directory subtree, a DITA map-based approach is flexible enough to keep pace with growing platform fragmentation.

The discussion of DITA so far has focused on topics and maps, with an emphasis on reusability of topic-based information such as XCCDF rules. In addition to facilitating reuse of structural block content, DITA provides mechanisms for reusing inline content. These mechanisms can be applied to frequently repeated fragments in an XCCDF benchmark document such as the product name (“Red Hat Enterprise Linux 7” in our scenario). One such DITA capability is content key references, a method for indirectly addressing inline content locations Oxygen.
A content key and key reference can implement the same functionality as the “product-name-macro” SSG's XSLT template rule. One way to do this is to use DITA's `<keydef>` element to define a “product_name” key in a benchmark's root map as shown in the DITA benchmark document (specializing on the base “map” element type) in Figure 12. The “product_name” key can then be referenced from anywhere within the root map using the `@keyref` attribute of the DITA `<ph>` (phrase) element.

The “product_name” key can also be referenced from within any resource the root map references. For example, the referenced topic `introduction.dita` could contain the following paragraph:

```xml
<p>This document provides prescriptive guidance for establishing a secure configuration for <ph keyref="product_name"/> systems.</p>
```

**Code Reuse**

A major benefit of DITA is that it can reduce the amount of coding needed to transform a collection of resources to a desired output format. The savings result from the DITA standard's requirements for output-producing processors. A conforming DITA processor must be able to merge topics in a map as well as resolve content references, eliminating the need for custom transformation code to perform these functions. A specialization-aware DITA processor can do all the above for specialized DITA documents through inheritance of processing behavior from base types via the `@class` attribute. The DITA Open Toolkit DITA-OT, an output-producing and specialization-aware DITA processor used in Oxygen and other XML software applications, illustrates the potential benefits of DITA for transformation of shorthand XML to XCCDF.

The DITA Open Toolkit is not part of the DITA standard — they are separately managed. However, both originated from the same research and development effort at IBM Priestley S-R. The Toolkit does not support XCCDF out of the box, but has an extensible plug-in mechanism for implementing custom output formats or adding additional functionality to existing output formats. The plug-in architecture allows for developers to reuse built-in transformation code, as well as code from other plug-ins, by integrating new XSLT via “extension points.” The DITA Open Toolkit's built-in transformation code includes a preprocessing module that implements map operations and content references. The preprocessing is used in all transformations by default.

A Toolkit XCCDF plug-in would require XSLT template rules matching against `@class` attribute values to convert elements from the specialized types from Section 4 into XCCDF equivalents. For example, the template rule below transforms the “rule” element type's `<rule>` element into an `<xccdf:Rule>` element. The value of `$rule-prefix` could be passed as a parameter to the plug-in. For the RHEL7 example from Section 2, the parameter value would be “xccdf_gov.nist_rule_”.

```xml
<xccdf:Rule>
  $rule-prefix fn:rule-id
</xccdf:Rule>
```

fn:rule-id is a stylesheet-defined function that constructs a reverse-DNS syntax XCCDF identifier from `$rule-prefix` and the `<rule>` element's `@id` attribute value.
The template rule below transforms the “rule” element type's `<oval>` element into an `<xccdf:check>` element with a nested `<xccdf:check-content-ref>`. `fn:oval-def` constructs the OVAL definition's reverse-DNS name from the `$oval-def-prefix` parameter value (“oval:gov.nist.redhat_redhat_enterprise_linux_7:def:” for the RHEL7 example) and the `<oval>` element's @href attribute value. `$oval-uri` provides the value of `<xccdf:check>`'s @href attribute (“http://oval.mitre.org/XMLSchema/oval-definitions-5”). `<xccdf:check-content-ref>`'s @href attribute is assigned the value of `$oval-location` (“CIS_Red_Hat_Enterprise_Linux_7_Benchmark_v2.1.0-oval.xml”).

6. Conclusion

The case study described in this paper investigated the feasibility of DITA for creating XCCDF security configuration checklists, and whether DITA could improve upon the SCAP Security Guide project's ad hoc approach to authoring and reuse. Proof-of-concept implementations of DITA element types for rules and profiles showed that DITA specialization is both feasible and offers tangible benefits beyond the SSG shorthand XML, such as validation and an improved authoring experience. Additional analysis and examples showed how DITA topic maps can improve reuse of structural XCCDF elements, and how DITA features such as key referencing can facilitate reuse of inline fragments.

This case study has a significant limitation. Analysis was limited to a single XCCDF checklist with only four rules, two profiles, and no `<Group>` or `<Value>` XCCDF elements. The rule and profile DITA element types do not provide the full range of XCCDF's flexibility. For example, “rule” and “profile” do not consider XCCDF checklist capabilities such as automated remediation of misconfigurations. Similarly, the case study also assumed checklist authors would use OVAL for rule checking and would adopt a specific convention for using the @selected attribute. While these simplifying assumptions align with a large amount of the XCCDF content published thus far, some authors might find these assumptions limiting. Until more sophisticated checklists from a cross section of Linux platforms are studied, it would be premature to conclude definitively that DITA will revolutionize the development of XCCDF and other SCAP content. The SSG authoring
approach may have its shortcomings, but it has been successful in producing an impressive collection of widely-deployed SCAP content. Further implementation and testing are needed to determine whether the promising results presented in this paper can scale up to a collection of larger and more diverse checklists.

An open question is to what extent adoption of DITA as an alternative to the SSG's ad hoc authoring approach would simplify the SSG transformation shown in Figure 7. At the very least, assembly relationships, platform commonalities, and reusable structures and fragments would be more maintainable explicitly represented in DITA than implicitly represented in XSLT and Makefiles. Additionally, DITA's default processing of maps and key references should simplify implementation of the “Combine shorthand XML components” and “Transform shorthand to XCCDF” steps. Implementing an XCCDF plug-in would be a good way to learn how much the DITA Open Toolkit further simplifies implementation.

Author’s Note
I am grateful to Eliot Kimber for his DITA Configuration and Specialization Tutorials, which were immensely helpful when writing this paper and creating the implementation examples. I also wish to thank Charles Schmidt, co-author of the XCCDF specification, for his thoughtful and helpful review of an earlier draft.

Appendix. List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSS</td>
<td>Cascading Style Sheet</td>
</tr>
<tr>
<td>DITA</td>
<td>Darwin Information Typing Architecture</td>
</tr>
<tr>
<td>DTD</td>
<td>Document Type Definition</td>
</tr>
<tr>
<td>MAC</td>
<td>Mandatory Access Control</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology</td>
</tr>
<tr>
<td>OVAL</td>
<td>Open Vulnerability Assessment Language</td>
</tr>
<tr>
<td>RHEL</td>
<td>Red Hat Enterprise Linux</td>
</tr>
<tr>
<td>SELinux</td>
<td>Security-Enhanced Linux</td>
</tr>
<tr>
<td>SCAP</td>
<td>Security Content Automation Protocol</td>
</tr>
<tr>
<td>SSG</td>
<td>SCAP Security Guide</td>
</tr>
<tr>
<td>URI</td>
<td>Uniform Resource Identifier</td>
</tr>
<tr>
<td>XCCDF</td>
<td>Extensible Configuration Checklist Description Format</td>
</tr>
<tr>
<td>XSLT</td>
<td>Extensible Stylesheet Language Transformation</td>
</tr>
</tbody>
</table>
References


[2] Certain commercial and third-party products and services are identified in this paper to foster understanding. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.


[4] This “rule” element type assumes that an XCCDF rule is disabled by default. Thus, an XCCDF <Rule> element generated from a DITA “rule” should have @selected="false". Although more restrictive than the XCCDF specification, this assumption allows for a more simplified “rule” syntax.

[5] This shorthand syntax assumes that rules in the benchmark are disabled by default. Identifying a rule in a profile thus implies that it should be changed to a selected status (with @selected="true").
This lack of flexibility is a limitation of the case study and not of DITA. The rule and profile element types described in this paper only scratch the surface of what is possible with DITA's specialization mechanism.

*Balisage: The Markup Conference*