XML-Native Constraint Evaluation

by
M. Cokus, Dr. R. Costello, Dr. M.A. Malloy, E. Masek, D. Winkowski

The MITRE Corporation
903 Gateway Boulevard, Suite 200
Hampton, VA 23666

Abstract

This paper discusses approaches to validating XML documents for compliance to constraints. Our particular focus is on structural and content constraints that go beyond what is readily expressible in XML Schema technologies. We provide examples and solutions drawn from our specific experience building an XML-native constraint validator based on a mathematical language called *Structural Notation (SN)*. SN is used to express operational constraints as machine-processible Rules against a particular category of hierarchically structured, text-oriented military messages, called *Message Text Formats (MTFs)*, which have been migrated to a corresponding XML-based representation.

We discuss the challenges we faced in implementing this XML-native constraint evaluator. For example, we discuss how, to build a Rule validator, we found it necessary to extend the underpinnings of logical evaluation in XPath 2.0 to use *three-valued logic (3VL)* rather than two-valued logic. We detail some general principles for expressing and enforcing constraints against regularly structured text, when rendered as an XML document. We enumerate minimal capabilities needed by a constraint language and evaluator for XML documents and suggest some ways our approaches can be generalized for use in other domains. Because the need to apply constraints to incomplete or flawed documents is not unique to the military messaging world, a constraint evaluation model such as we propose, grounded in 3VL, is relevant to the XML user community at large.

Keywords: XML, XPath 2.0, XSL 2.0, Constraint, Validation, Text Messaging, Three-valued Logic, Message Text Format, Structural Notation, XML-MTF, XSN-MTF

Background

Message Text Formats (MTFs) support information exchange

To be effective, the military as well as other government and commercial organizations must share information across dissimilar, independently developed systems that involve diverse languages, cultures and command/management structures. One way of addressing these challenges is through formatted, text-oriented information exchanges. A particular standard of hierarchically structured messages, called *Message Text Formats (MTFs)*, has been used to relay battlefield information since the 1970's. The MTF standard comprises about 600 message types and over 6000 simple and complex data types, all defined, agreed upon and implemented by more than 70 nations.

Each MTF type specifies a particular grammatical construction of components, each of which may be composed of constituent components. At the lowest hierarchical level are *Fields*. A *Set* is composed of a sequence of possible Fields, and possibly a logical grouping of them called a *Field Group*. A *Segment* is composed of a sequence of Sets and possibly other Segments. The *Message* component is a special case of Segment. In addition, components can be specified as optional and/or repeatable. Because components are built using agreed structures (syntax) and vocabulary (semantics), the resulting MTFs help mitigate misunderstandings that can arise out of differences in terminology, acronyms, and so on that exist across functional and geographical barriers.

Constraints refine MTF specifications

There are basic grammatical rules governing how to assemble the constituent components of MTFs, together with specifications regarding the repeatability or optionality of those components within individual message types. But they are insufficient for capturing all the internal logical requirements that impact a message instance's compliance with the standard. Thus, the specification of each standard message type also includes Rules (i.e., *constraints*) that must be satisfied for any message instance of that type to be considered "in compliance" with the standard.

Circa 1980, MITRE designed a mathematical language called *Structural Notation (SN)* for formally encoding such operational constraints against MTFs in the form of machine-processible Rules. Government-proprietary evaluation software was developed and deployed to enforce SN Rules in MTF processing systems. We call the automation aspect of the MTF SN compliance problem *SN validation*.

Structural Notation (SN) expresses MTF constraints

Each SN Rule consists of a *Statement* and an optional *Condition*. The *Statement* is the part of a Rule that specifies some action on or requirement of any MTF to whose type the Rule is assigned. For example, the Rule might require the use of the same alphanumeric value within two components in the message, or it might specify the maximum number of times a component can be repeated. A generic example of the former kind of Rule might express a constraint like *<Some component> must have the same alphanumeric value as <some other component>*. A *Statement* such as this, without an associated *Condition*, makes an unconditional assertion about the document and applies to all instances of that type. In such cases the *Condition* can be assumed to evaluate to *true* by default.

A *Condition*, if present, specifies the circumstances under which the action or requirement part of the Rule as expressed in the *Statement* takes effect. For example, the *Statement* might apply only when two components contain identical alphanumeric values, or when some specified component is present in the message instance. Such a Rule might generically state *<Some component> must occur fewer than <N> times, if <some other component> has the alphanumeric value "XYZ."* Such a *Statement* with an associated *Condition* (i.e., the part of the Rule in the "if" clause) makes a conditional

assertion about the message; the assertion expressed in the *Statement* part of the Rule applies only when the *Condition* evaluates to *true*.

Figure 1 below synopsizes the Rule evaluation logic. We will discuss the significance of the *null* evaluation that participates in this logic a little later.

If *Condition* is *false* or *null*, then
MTF "passes" validation [by default] for Rule.

If *Condition* is *true* [by assessment or by default], then:
If *Statement* is *true*, MTF "passes" validation for Rule.
If *Statement* is *null*, MTF "passes" validation [by default] for Rule.
If *Statement* is *false*, MTF "fails" validation for Rule.

Figure 1. Rule evaluation logic

It should be noted that the constraints we are interested in are *intramessage* (or *intradocument*) constraints. That is, it is possible to examine information "within a single message [document] instance" to determine whether a constraint is satisfied. For example, the use of one particular optional component may preclude the use of some other component, but only *in the same message instance*; or the use of a particular text value may limit the values allowed elsewhere *in the same message*. We are not concerned with expressing or evaluating constraints that require simultaneously looking at the contents of multiple messages, which can be an even more complex problem!

MTFs go XML-based

The desire to leverage non-proprietary software in support of information exchange has spurred many organizations, including the military, to quicken their adoption of XML technology. The emerging body of XML standards naturally parallels the interoperability supports and concepts needed to web-enable MTFs. To better meet the needs of the 21st century warfighter, MTFs have been migrated to an XML-based representation called *XML-MTF*.

An approved XML-MTF Mapping Specification was developed to provide the XML tags needed to structure XML versions of existing MTF types. An XML-MTF Schema Derivation Specification details how to algorithmically derive XML-MTF Schemas from the standard. This enables *XML-native validation* of various syntactic, structural and value constraints applicable to XML-MTFs by viewing them as classes of document instances. Both specifications are in place and agreed by the MTF user community, so that any MTF can be equivalently expressed as an XML document.

However, to provide the same level of validation for XML-MTF as that available for MTFs – and to do so in an XML-native form – we had to devise the means to implement SN validation for XML-MTF. In other words, it was not enough to support demonstrating that an XML-MTF is well-formed and valid; the document also had to satisfy these additional SN Rules. This required re-expressing those Rules so they are

processible in an appropriate validation framework using XML technologies. We call this task *XSN-MTF*.

Challenges to SN Migration

This section summarizes the challenges we faced during the XSN-MTF effort. Our experiences provided the basis for the guiding principles for constraint expression and evaluation presented later in the paper.

Constraint ambiguities

When we inspected the existing collection of constraints expressed in SN to assess the amenability of migrating them to XML-based representations, it became clear that they were products of a by-gone era. To conserve space, the designers of the constraint language had provided many abbreviations and notational shortcuts to make the written Rules terser. The unfortunate side-effect of this economy of expression is that terseness tends to obscure the resulting Rules' meanings.

For example, we discovered that Rule designers sometimes encoded the same kinds of constraints differently into their specific SN representations. Operational personnel often had different interpretations of the same Rule's meaning. In addition, ambiguous references to message components, as expressed within the Rule syntax, made it difficult for implementers to encode appropriate validation software to locate and compare the intended data; this resulted in implementation inconsistencies. Finally, we found constraint language syntax "features" in operational use that were not documented as part of the existing constraint language specification. Other features were formally included in the specification, ostensibly for the sake of the theoretical beauty or completeness of the language, but were used seldom if at all in practice.

Technology shortfalls

We initially hoped that many of the Rules expressible in SN could be re-expressed in XML Schema constructs. This hope dimmed when we realized that many common constraints encoded relationships between non-sibling elements in the XML-MTF document. Such relationships cannot be captured easily in XML Schema.

Iteration and alignment

SN Rules can reference repeatable components in the message type to which they are assigned. For such Rules to be evaluated, all referenced component repetitions must be inspected, and appropriate pairings of the repeated components that appear as distinct operands in the same Rule must be considered. We call the inspection of the repetitions *iteration*; the formulation of appropriate pairings of repeated components is *alignment*. The challenging part of Rule evaluation in these cases is determining all the combinations of relevant data that must be evaluated to make a truth determination.

For example, what happens when two repeatable components, which occur *n* and *m* times respectively, must be compared for the Rule evaluation to take place? Should *n* 1-to-*m* comparisons take place? Or *m* 1-to-*n* comparisons? Perhaps min(*n*,*m*) 1-to-1 comparisons? Some possibilities are illustrated in Figure 2.

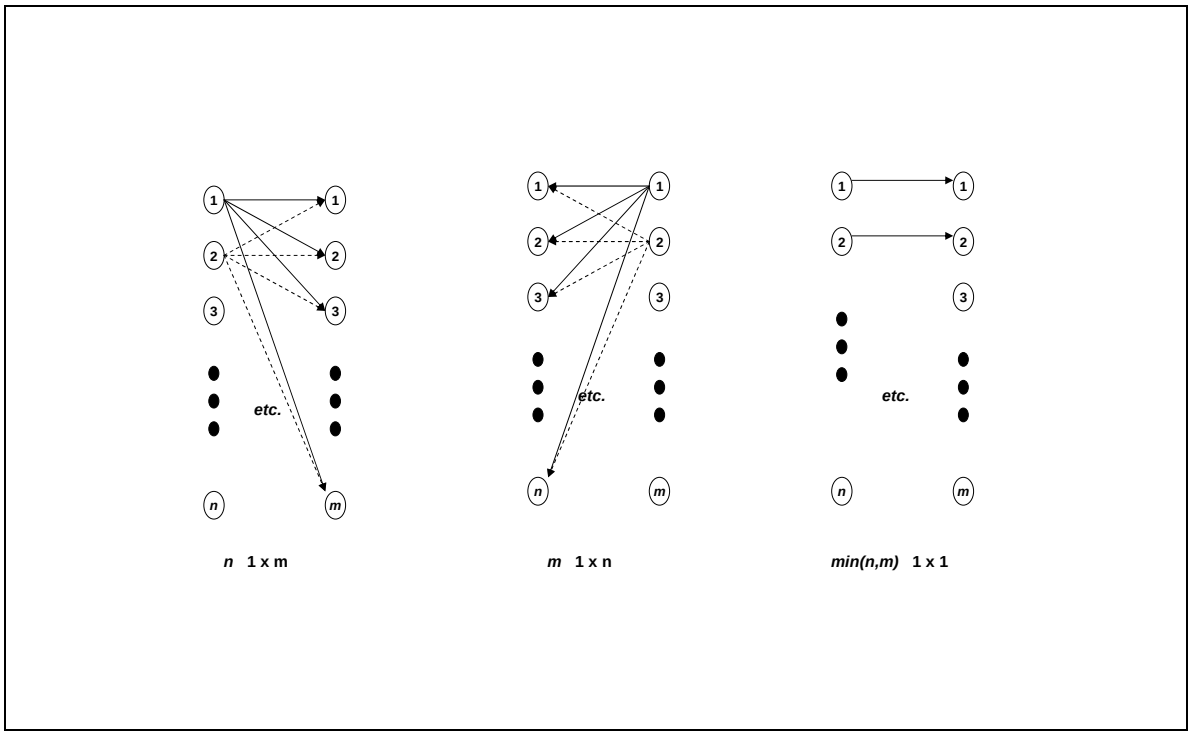

Figure 2. Possible comparison alignments

We found the encoding of many Rules confused the designers and implementers, which led to many-to-many comparisons; but often such comparisons were not intended and are not meaningful. For example, suppose a Rule were written to require that all *m* repetitions of some element must have the same value as all *n* repetitions of another element. The only way this can occur in an instance document is if all such components have the very same value! Although it is possible this is a viable constraint, it appears suspect in many situations; furthermore, simpler Rules requiring fewer comparisons could be written to enforce such a constraint. We concluded Rules needed to expose the intended alignments more succinctly when two or more repeatable components were involved to avoid encoding and implementation inconsistencies.

The iteration and alignment requirements that exist for applying such constraints to MTFs obviously exist for applying similar constraints to corresponding XML-MTF documents as well. So we recognized a Rule evaluator for XML-MTF documents would need the capability to process multiple instantiations of a constraint to determine whether that constraint is satisfied by a given document instance. This meant the processing logic to control repetition inspection (i.e., *iteration*) and repetition pairing (i.e., *alignment*) had to be supported in the validation model.

Three-valued logic (3VL) requirement

We can informally think about the loaded term *context* as "the situation that can lead to the use of a component." For a component that is *not* hierarchically subordinate to (i.e., nested within) another component of the document, its context is the document. For a component that *is* hierarchically subordinate to (i.e., nested within) another component of the document, its context is the component that contains it.

Typically, the evaluation of a logical expression results in a straightforward manner to *true* or *false*, depending on whether the circumstance that expression describes is exhibited within the context the evaluation takes place. In the constraints we are concerned with, logical expressions reference elements of the document. For example, if a component's context is the document itself, then clearly it always is possible to assess an expression as *true* or *false* when that expression questions such a component's occurrence within the document.

Now consider a slightly different situation. Suppose there is a component *C2* whose context is *not* the document, but rather it can only occur within some other component *C1* within the document. Given an expression that questions the existence of *C2*, if *C2 does* appear in the document – this only could come about if its context *C1* appears in the document – clearly the expression evaluates to *true*. Similarly, if *C1 does* appear in the document, but *C2 does not* appear within *C1*, the expression evaluates to *false*.

But what if *C1 does not* appear in the document? In a standard two-valued logic system, such as that used in XPath, an expression that attempts to examine *C2* or questions the existence of *C2* in this situation will evaluate to *false*, which in many circumstance could cause the Rule that contains that expression to evaluate to *false* also and thus fail. This presents a problem for us, because in our operational environment, a Rule should not fail if the context for evaluating it is not present in the document.

In other words, each SN Rule is written to enforce assertions against a collection of possible instantiation of the document; for some other perfectly valid and well-formed instantiations of the document, or for fragmentary instantiations, a Rule simply may *not apply* because the document does not exhibit the context for evaluating it. Thus it would be somewhat inaccurate for the expression that questions *C2's* existence simply to evaluate to *false*, because the context that *could have* led to evaluating the expression – i.e., the use of *C1 – does not* occur in the document under examination. So we needed a way of consistently addressing cases such as this.

One remedy to this situation is to AND existential tests for all relevant operands with the expression that actually evaluates the constraint; but this is unwieldy for all but the simplest cases. Our approach instead requires that such an expression evaluate to the logical value *null* since its truth *cannot be determined* based on what is currently known (i.e., what is contained in the document under examination).

A logical system in which this third logical value *null* is in effect – in addition to *true* and *false* – is a *three-valued logic (3VL) system*. We found that the concept of 3VL was

pivotal to evaluation as it originally had been specified for the constraint language SN. We concluded that 3VL had to be incorporated into the XSN-MTF validation model. We will discuss 3VL more later in the paper.

Universal and existential quantification

We had to consider how to implement two kinds of quantification for Rule evaluation for XSN-MTF. The first, *universal quantification*, is commonly called "for all." It represents the conjunction (AND-ing together) of results from evaluating an expression with different combinations of relevant data. By definition, if all the resulting expressions evaluate to true, the quantification is satisfied. The second, *existential quantification*, is commonly called "there exists." It represents the disjunction (OR-ing together) of results from evaluating an expression with different combinations of relevant data. By definition, if at least one of the resulting expressions evaluates to true, the quantification is satisfied.

The reason quantification is important for Rule evaluation is that many of the Rules applicable to documents express one kind of quantification or the other. For example, it might be necessary to require that *for all* components *C1* in the document, *C1* must contain two or more uses of component *C2*. Or for a particular document type it might be required that *there exists* at least one occurrence of a specific alphanumeric value within any instance document. Some programming environments support specific constructs that can be used to control quantification behaviors. However, at the same time the constraint language syntax itself must provide the Rule designer the ability to express the intent of the Rules involving such quantification concepts to ensure appropriate implementations of those Rules.

Evaluation Infrastructure

Figure 3 pictorially depicts the stages of transforming the constraint language into its XML-based counterpart. In the next few paragraphs, we discuss the infrastructure that we needed to develop to support this transformation.

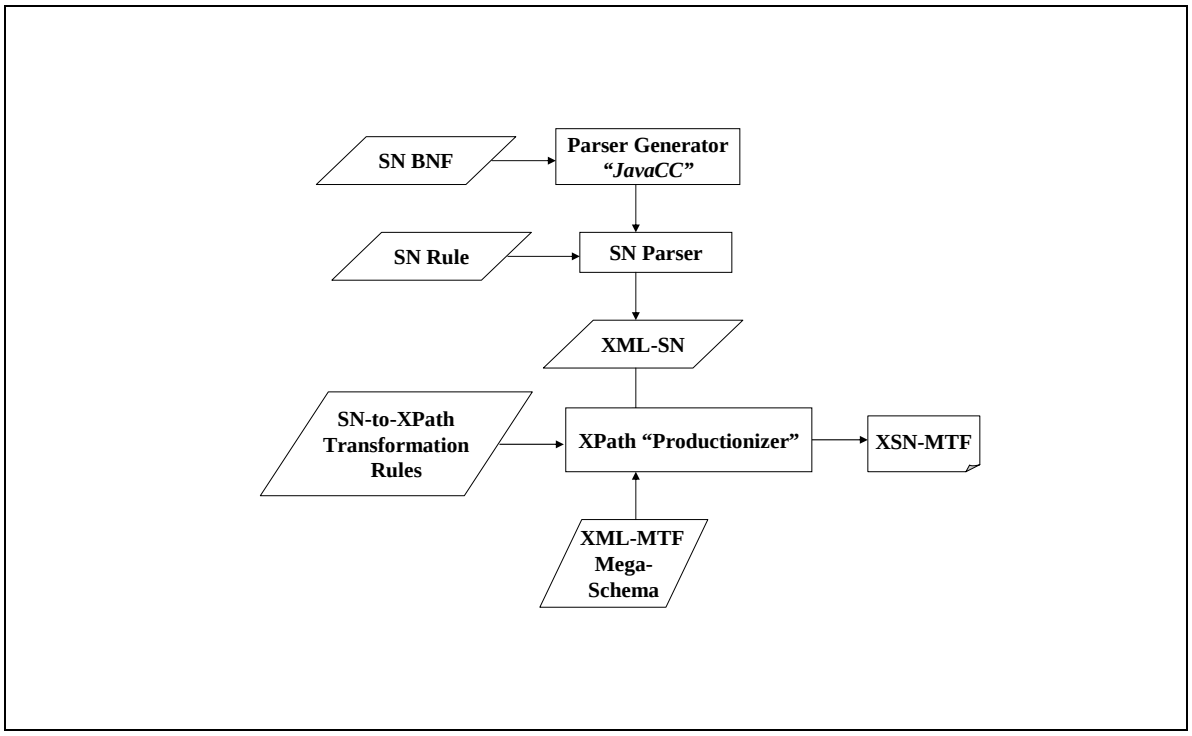

Figure 3. Transforming SN to XSN-MTF

Rendering XML-SN

An initial part of the transformation process involved the capability to parse and tag the Rule into a form more amenable to subsequent processing. This step could have been omitted had the Rules been expressed directly in XML technologies to begin with; but we were not in such a convenient situation. To ensure appropriate mappings were done, first the original constraint language had to be modified to mitigate the problems cited earlier, such as eliminating ambiguous component references and clarifying the intended iteration and alignments. This was accomplished by coordinating formal changes to the language specification through its relevant user communities.

For example, the revision of the language now requires that Rules incorporate complete and fully qualified component references, similar in detail to path names in XPath. Features of the constraint language that were redundant, out-of-use or confusing were deprecated. Subsequently we developed a BNF grammar for the improved constraint language that could serve as input to the parser generator *JavaCC*. Rules then could be fed into the resulting parser to produce a tagged representation we call XML-SN, poised for subsequent XML-based processing. We will not discuss that portion of the mapping further in this paper; please refer to [1] for amplification.

Navigating with XPath 2.0

We chose XPath as the navigational language for locating data. We had to use XPath 2.0 to get built-in constructs for expressing universal and existential quantification that we would need later in the processing. In the case of a Rule assigned to an MTF type, path

names were expressed in the original constraint language syntax in terms of integer Segment, Set, Field Group, and Field numbers, which sequentially reference the hierarchical components of the MTF structural design, the so-called *metaMTF*. This system of references is not meaningful for navigating an XML-MTF; however, a component reference that appears in a Rule assigned to a given document schema can be translated into an equivalent XPath reference using appropriate tags to navigate to the relevant data.

More to the point, to re-interpret the component references from our constraint language to useful path names for XPath, we needed to use tags referencing the XML-MTF document structure instead of relative integer positions. We wrote interfaces to the XML-MTF Schema to access the tagging metadata associated with a relevant document instance to which the Rule might be assigned. When fed a group of metadata associated with a particular component reference via the original Rule syntax, these functions generated an XPath path name tailored to navigating the XML-MTF instance document to which the Rule was assigned. For those interested in the details of this aspect of the XSN-MTF mapping, please see [2].

3VL comparison operators

The behaviors of most programming languages are built around a *two-valued logic (2VL)* paradigm. That is, expressions evaluate to *true* or *false* – period. XPath is no exception. Because 3VL was pivotal to the evaluation behaviors in our constraint language as explained earlier, we wrote an XSL 2.0 Stylesheet to implement 3VL function counterparts for a number of comparison functions we needed so they would impose 3VL-based interpretations on their operands and results. These include numeric equality, inequality, greater-than and less-than as well as lexicographic equality and inequality functions. This step was necessary to by-pass the default behaviors of XPath and incorporate 3VL-based reasoning.

Consider comparing two strings for equality. We can represent this generically using the available XPath functions as *string(left_operand) eq string(right_operand).* Now suppose the *left_operand* examines elements that do not occur in the document under examination. This means we would not be able to retrieve any nodes of the kind specified as the *left_operand*. Since the operand cites an empty node set (i.e., no exemplars of this path exist in the document), the *eq* expression evaluates to *false* in the XPath 2VL world. This is not what we wanted.

In a 3VL context, attempting a string equality comparison, when no exemplars exist per either or both operand's path name, must result in a *null* expression evaluation. This is because a meaningful context for comparing the components fails to exist. For such expressions to evaluate to *false* instead might inadvertently cause a Rule to be determined as "failing" when it should not. Thus we provided for re-expressing such comparisons, replacing the standard XPath function with our own 3VL-based function implementations using XSL 2.0. For this notional example, the resulting expression, using function prefix

notation, is the following: *eq3-s(string(left_operand),string(right_operand))*, where *eq3-s()* is our implementation of the 3VL function for string equality comparison.

3VL logical operators

We also had to implement 3VL function counterparts for logical AND, OR and NOT. The 3VL operators *and3(), or3()* and *not3()*, which we again implemented via the XSLT Stylesheet, replaced the "default" logical operations in XPath and are extensions of those operations familiar to us from 2VL. These operations are summarized in the truth tables shown in Figure 4.

and3	true	false	null
true	true	false	null
false	false	false	false
null	null	false	null

or3	true	false	null
true	true	true	true
false	true	false	null
null	true	null	null

not3	true	false	null
	false	true	null

	E2			
	→	**true**	**false**	**null**
E1	**true**	true	false	null
	false	true	true	true
	null	true	null	null

Figure 4. 3VL Tables

Notice how the *null* valuation has been incorporated in these tables. The *null* valuation addresses instances in which the logical outcome of the operation cannot be definitively decided. For example, suppose we consider two string equality expressions *E1* and *E2*. Further suppose the comparison stated in *E1* has been determined *true*, but unfortunately the comparison stated in *E2* involves examining nodes for which no exemplars exist in the document. Because we have replaced the standard string equality comparison with the 3VL function *eq3-s(), E2* will return the *null* valuation in this situation. Further, under the rules of 3VL, *and3(E1,E2)* produces *null*. The reasoning that underlies this is: if additional information were made available, *and3(E1,E2)* could have a different evaluation depending on whether we eventually find out *E2* is *true* or *E2* is *false*; but for now, we are uncertain, so the logical result must be *null*.

On the other hand, when *E1* is *true* and *E2* is *null*, the expression *or3(E1,E2)* evaluates to *true*. This is because, for a disjunction to evaluate to *true*, at least one of its operands must be *true*. We already know *E1* is *true*, so we don't even need to concern ourselves about the eventual valuation of *E2* to decide the outcome of *or3(E1,E2)*! So we use the *null* evaluation to model situations where the context fails to exist to evaluate an expression. Recall we mentioned earlier when discussing the 3VL requirement what an important consideration this is, for ensuring that Rules are not inadvertently interpreted as failing as the intermediate results that contribute to their overall evaluation are combined logically.

Mapping SN expressions to XPath/XSL

Given this basic groundwork, we established semantic and syntactic correspondences between all the operators we needed for the migrating constraint language and XPath operators and XSL functions as a prelude to translating the expressions which are the "building blocks" of the Rules. Figure 5 shows two examples of such mappings, for numeric and string equality comparisons, respectively.

The examples also suggest the correspondence of operand syntax from the original constraint language SN to XPath path names. Specifically, *[3]F2* refers to the *second component or Field (F2)* within the *third hierarchical component or Set ([3])* in the MTF structural design. As mentioned earlier, in the redefined expressions we must use tags referencing the XML-MTF document structure instead of relative integer positions. So in the figure we show the generic tags "set3" and "field2" for this purpose. Were the example expressions bound to a specific XML-MTF type, actual tag names would have been substituted from the corresponding XML-MTF Schema.

SN operator: =
Explanation: The "=" operator is for doing numeric comparison. It may be used in either the Statement part or the Condition part of a Rule.
Available XPath operator: number(left_opnd) eq number(right_opnd)
Redefined operator in XSL function prefix notation:
ex:eq3-n(left_opnd, right_opnd)
Example:
SN: [3]F2 = 3
Available XPath: number(/set3/field2) eq 3
Redefined operator in XSL function prefix Notation:
ex:eq3-n(/set3/field2, 3)
Note: "eq3-n" is the 3VL function for number equality comparisons.

SN operator: EQ
Explanation: The "EQ" operator is for doing string comparison. It may be used in either the Statement part or the Condition part of a Rule.
Available XPath operator: string(left opnd) eq string(right opnd)
Redefined operator in XSL function prefix notation:
ex:eq3-s(left opnd, right opnd)
Example:
SN: [3]F2 EQ /foo/
Available XPath: string(/set3/field2) eq "foo"
Redefined operator in XSL function prefix Notation:
ex:eq3-s(/set3/field2, "foo")
Note: "eq3-s" is the 3VL function for string equality comparisons.

Figure 5. SN to XPath/XSL Mapping Examples

A Rule as a closed-form logical expression

We noted each constraint is of the form *if Statement then Condition*, which is the formulation of logical implication (i.e., →). We observed that Rule evaluation is

equivalently stated *Condition* → *Statement* (i.e., *E1* → *E2*). The 3VL variant of the implication operator is shown in Figure 4.

It is easy to reason through this logic. The Rule evaluates to *true* and "passes" in two situations. First, the constraint is satisfied if the *Statement* evaluates to *true*. This makes sense, because if the assertion (*Statement*) is satisfied, then so is the Rule, whether it applies to the document or not! If the *Condition* evaluates to *false* (which means the Rule does not apply to the document), the Rule also evaluates to *true*. This is consistent with the reasoning underlying implication: if we start with a *false* antecedent (*Condition*), it is possible to prove any consequent (*Statement*) to be *true*, so the Rule is satisfied regardless of the *Statement*'s evaluation in this case.

The Rule evaluates to *false* and "fails" in one situation only. This occurs when the *Condition* is *true* but the *Statement* is *false*; that is, the assertion is applicable to the document but it has not been satisfied. In all other cases, the Rule evaluates to *null*. This means the context does not exist in the document instance to determine the satisfaction of the Rule, so that the Rule passes "by default" (i.e., we can ignore it).

We also observed that logical implication is equivalent to the closed-form expression *Statement OR3 NOT3(Condition)* – that is, *or3*(*Statement, not3(Condition))* using function prefix notation. The interested reader can validate this equivalence by constructing the associated truth table to ensure that it is identical to that of 3VL implication (but we promise you it is). This simple equivalence allowed us to create an XPath equivalent, closed-form logical expression for a Rule instead of complex *if-then-else* constructs, by applying the path name and operator mappings to its expression building blocks and using the prefix function notation.

The resulting closed-form expression is constructed to evaluate to *true* if the Rule is satisfied and *null* if the context for evaluating the Rule does not exist; it will evaluate to *false* if the Rule is not satisfied. We wrote a simple function *rule-satisfied()* whose role is to implement this 3VL-based evaluation behavior, summarized earlier in Figure 1, when the constraint is applied to the instance document in this form. (This function essentially maps the *null* evaluation to *true* or *false* as appropriate, so that Rules that are to be ignored will not inadvertently appear to fail in XPath's 2VL system.)

Evaluation Model

The final stage in this process involves embedding the resulting XPath expression of the Rule into suitable processing logic. The focus here is for the infrastructure to provide control of the iteration (i.e., repetition inspection) and alignment (repetition pairing) aspects of Rule validation against an instance document when repeatable elements are referenced.

An Easier Model for Illustrations

In the XML-MTF model, each message is a document type, and we express constraints for evaluation against an individual message instance. To avoid getting bogged down in understanding the XML-MTF design and tag names, for the remaining examples in this paper we instead will reference a well-known Recipe model. Cooking terms and concepts probably are familiar to more readers than the semantic details of tactical messages! In particular, we use a model similar to the example published by Møller and Schwartzbach [3]. So for the illustrative examples we will express constraints for evaluation against a recipe instance where:

- Each *Recipe* consists of a *Title*, ingredients, preparation, possibly some *Comments*, and its *Nutrition* content.

Each ingredient can be base or composite:

- A *BaseIngredient* is composed of a *Name*, an *Amount* (possibly unspecified), and a *Unit* (unless *Amount* is dimensionless).
- A *CompositeIngredient* is composed of *BaseIngredients* and *IngredientPreparation* listed as *Steps*.

The *FinalPreparation* lists the *Steps* needed to complete the *Recipe*.

An example Recipe document is shown as Figure 6.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Sample XML file generated by XMLSPY v2004 rel. 4 U (http://www.xmlspy.com)-->
<Recipe
   xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
   xsi:noNamespaceSchemaLocation="XML2004ConferenceExample02.xsd">
        <Title>Ricotta Pie</Title>
        <CompositeIngredient>
                <Name>Dough</Name>
                <BaseIngredient>
                        <Name>Flour</Name>
                        <Amount>4</Amount>
                        <Unit>cup</Unit>
                </BaseIngredient>
                <BaseIngredient>
                        <Name>Baking powder</Name>
                        <Amount>5</Amount>
                        <Unit>teaspoon</Unit>
                </BaseIngredient>
                <!-- etc • • • -->
                <IngredientPreparation>
                        <Step>Combine the flour, baking powder, and 1 cup of the sugar
                                together.</Step>
                        <Step>Cut in the shortening and mix until the mixture resembles coarse
                                crumbs.</Step>
                        <Step>Mix in 4 of the eggs and 1 teaspoon of the vanilla.</Step>
                        <Step>Divide dough into 4 balls and chill (if needed).</Step>
                </IngredientPreparation>
        </CompositeIngredient>
```

```
        <CompositeIngredient>
                <Name>Filling</Name>
                <BaseIngredient>
                        <Name>ricotta cheese</Name>
                        <Amount>3</Amount>
                        <Unit>pound</Unit>
                </BaseIngredient>
                <BaseIngredient>
                        <Name>eggs</Name>
                        <Amount>12</Amount>
                </BaseIngredient>
                <!-- etc • • • -->
                <IngredientPreparation>
                        <Step>Beat the 12 eggs, 2 cups sugar and vanilla extract together </Step>
                        <Step>Stir in the ricotta cheese and the chocolate chips.</Step>
                        <Step>Set aside.</Step>
                </IngredientPreparation>
        </CompositeIngredient>
        <BaseIngredient>
                <Name>Milk</Name>
                <Amount>Variable</Amount>
                <Unit>Text</Unit>
        </BaseIngredient>
        <FinalPreparation>
                <Step>Roll dough flat and place into a 9 greased inch pie pan ...</Step>
                <Step>Add the filling ... </Step>
                <!-- etc • • • -->
        </FinalPreparation>
        <Comment>This sounds harder than it is.  Do not be intimidated by all the steps.</Comment>
        <Nutrition>
                <Serving>6 ounces</Serving>
                <Calories>348</Calories>
                <Fat>18</Fat>
                <Carbohydrates>64</Carbohydrates>
                <Protein>18</Protein>
        </Nutrition>
</Recipe>
```

Figure 6. Recipe Example

Singular cases

In the simplest case, a Rule may reference only components that appear either once or not at all in the instance document. Or, the Rule may consider only existential assertions, where one *or more* occurrences of a component will satisfy the constraint. That is, only one inspection of data is required to decide the Rule.

Suppose the designer decided that recipes appropriate for beginner cooks should contain the word "Easy" at the beginning of the *title*, and that *composite ingredients* are too complicated for "Easy" recipes. The designer might state a Rule *If the recipe title starts with "Easy" then it must not contain any composite ingredients*. We can re-express the proposed Rule in XPath/XSL as follows:

```
rule-satisfied(or3( eq3-n(count(/Recipe/CompositeIngredient),0),
```

not3(eq3-s(/Recipe/Title,'/Easy/*'))))

This expression is built around the *or3*(*Statement, not3(Condition))* formulation of the Rule. The Statement or assertive part of the Rule is expressed in by examining the size of the node set that results from looking for any *composite ingredients*. The Condition part of the Rule is expressed using our 3VL implementation of string comparison *eq3-s()*, where the special pattern-matching syntax ′/Easy/*′ determines whether the *Title* string begins as desired. (This particular syntax was motivated to ensure compatibility with the syntax of the original constraint language.) The above expression will evaluate to *true* if the Rule is satisfied and *false* if it is not satisfied. In particular, the recipe for Ricotta Pie suggested by Figure 6 would not satisfy the Rule, because both the filling and the dough are *composite ingredients*, and the *title* fails to start with the word "Easy".

If the world of documents and constraints we wished to express were so simple, the job would nearly be done at this point, except for thinking through a few minor details. Unfortunately, document elements can be repeated, and constraints can inspect more complex document features. When a Rule's operands cite repeated elements in the document, or Rules inspect more interesting features of the document, multiple combinations of data may need to be considered to produce an overall evaluation of the Rule.

Iteration

Because MTF components can be repeatable, the SN constraint language we were migrating provides a syntax called *subscripts* to modify each repeatable component to make explicit which of it repetitions should be considered during the iteration. Similarly, when re-expressing these Rules in many cases an operand must be examined iteratively "for all" its relevant manifestations within the document instance, considering each component occurrence at each hierarchical or nested level along the operand's path.

As a starting point for discussing iteration, suppose the designer decided to enforce the constraint *Within each composite ingredient, the number of base ingredients must be fewer than the number of ingredient preparation steps*. (We will not argue whether this is a reasonable constraint.) This Rule is an unconditional assertion. That is, there is no Condition part of the Rule, so the Condition is assumed to be *true*. Thus this Rule applies to every instance of the document type to which it is assigned, with the outcome of Rule evaluation depending solely on the evaluation of the Statement part. We have pictorially illustrated the intended inspection for a notional example in Figure 7.

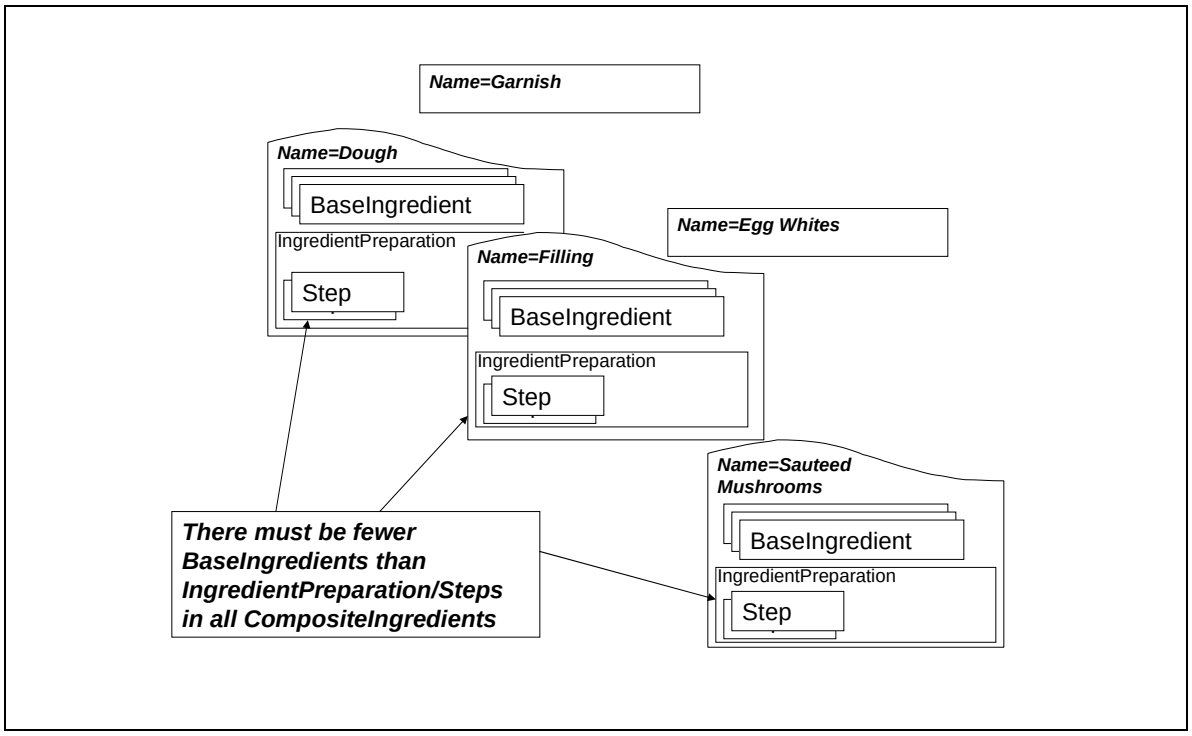

Figure 7. A simple "for all" inspection

At this point the technically-oriented reader might imagine a loop construct in a conventional programming language to guide the iterative inspection. A looping mechanism can enforce the described "for all" examination of the composite *ingredients*, and the logic within that loop can test for situations in which the constraint should *fail*. We can informally re-express the intent of the above Rule from this negative viewpoint as: *The constraint fails to be satisfied if there is any composite ingredient with the same or fewer ingredient preparation steps than base ingredients in it; otherwise the constraint is satisfied*. If the constraint fails to be satisfied on any iteration, *false* is returned and the Rule fails. If the constraint is satisfied for all iterations, *true* is returned and the Rule passes.

Let *n* denote the number of *composite ingredients*, and let *i* denote an iteration index. A pseudocode example is:

```
for i = 1 to n
        if number of BaseIngredients in i^th CompositeIngredient >=
                number of IngredientPreparation/Steps within i^th CompositeIngredient then
                        return false;
return true;
```

Thinking back to the original constraint language, repeatable components – which must be modified with an SN subscript – can be interpreted as a realization of the "for all" concept, which in turn can be expressed as an iteration construct. Additionally, iterations can be nested according to the hierarchical structure of the document, from hierarchically outermost to hierarchically innermost components, if more than one repeatable element must be iterated.

In this case the transformation to XPath equivalent forms also must set up iterations to navigate the document during evaluation when repeatable elements are involved. We need to formulate an expression that describes the conditions under which the constraint is satisfied. For this example, an XPath/XSL formulation is:

```
rule-satisfied( or3( every $i in /Recipe/CompositeIngredient satisfies
        lt3-n(count($i/BaseIngredient), count($i/Ingredient/Preparation/Step)),
        not3(true) )
```

where *lt3-n()* is our 3VL implementation for numeric less-than comparison. The "every" clause cites a collection of *composite ingredient* elements to examine. The "satisfies" clause is built using the *or3*(*Statement, not3(Condition))* formulation of the Rule. The default Condition part is represented by *true*. This is a bit superfluous, but we have included it to keep the translation consistent. The outcome of the first argument to the *or3()* function (i.e., the Statement part of the Rule) will determine the outcome of the Rule evaluation. This part checks to make sure for each *composite ingredient* iterated using *$i* that it contains fewer *base ingredients* than *ingredient preparation steps.*

The above expression evaluates to *true* if the expressed constraint is satisfied; it evaluates to *false* otherwise, which is the desired result. Thus in XPath, the *every* construct, together with its *dollar variable* index (i.e., *$i*) controls the "for all" examination of repeated components and so can be used to accomplish iteration over a collection of element repetitions. If more than one level of repetition needs to be examined, the *every* constructs can be nested and distinct index variables used.

Alignment

Alignment is a specialization of iteration; it refers to simultaneously comparing the same repetition of two or more repeatable components during evaluation. This can be stated another way as *lock-step comparison*. A goal of alignment is to facilitate pairing components in such a way that the Rule performs one-to-one comparisons. We pointed out earlier that many-to-many comparisons, although supported by the original constraint language syntax, usually are not meaningful.

We saw above how in XPath, the *every* construct can be used to control the "for all" examination of repeated components. By subscripting (e.g., *[$i]*) the relevant elements with the same iteration index (i.e., *$i*) from the *every* construct, we can accomplish aligned comparisons.

To provide an example for understanding alignment, imagine that to help ensure the logical readability of the *recipe*, the designer wants to require that for a *recipe* with *n* composite *ingredients*, there are at least *n* steps in the *recipe preparation*, with the first *n* of those *steps* using each of those *n* composite *ingredients* in turn: the first of those *steps* mentions the first composite *ingredient*; the second *step* mentions the second composite *ingredient*; and so on. (Once again, we won't dwell on the reasonableness of this requirement.)

This requirement expresses an aligned comparison, matching composite *ingredients* with *recipe preparation steps*. Not all aligned comparisons require that there is a one-to-one alignment of the two comparison node sets, but in this example, that's what the designer intended. We would express this requirement as two constraints. The first constraint enforces that the number of *final preparation steps* must equal or exceed the number of composite *ingredients*. That is, *If the recipe has composite ingredients, then there must be at least as many steps in the final preparation*. We leave it as an exercise for the reader to build this Rule. The second constraint can be expressed as the Rule: *If the recipe has composite ingredients, then there must be a corresponding final preparation step for each composite ingredient in which its name appears.* We have depicted this aligned comparison concept pictorially in Figure 8.

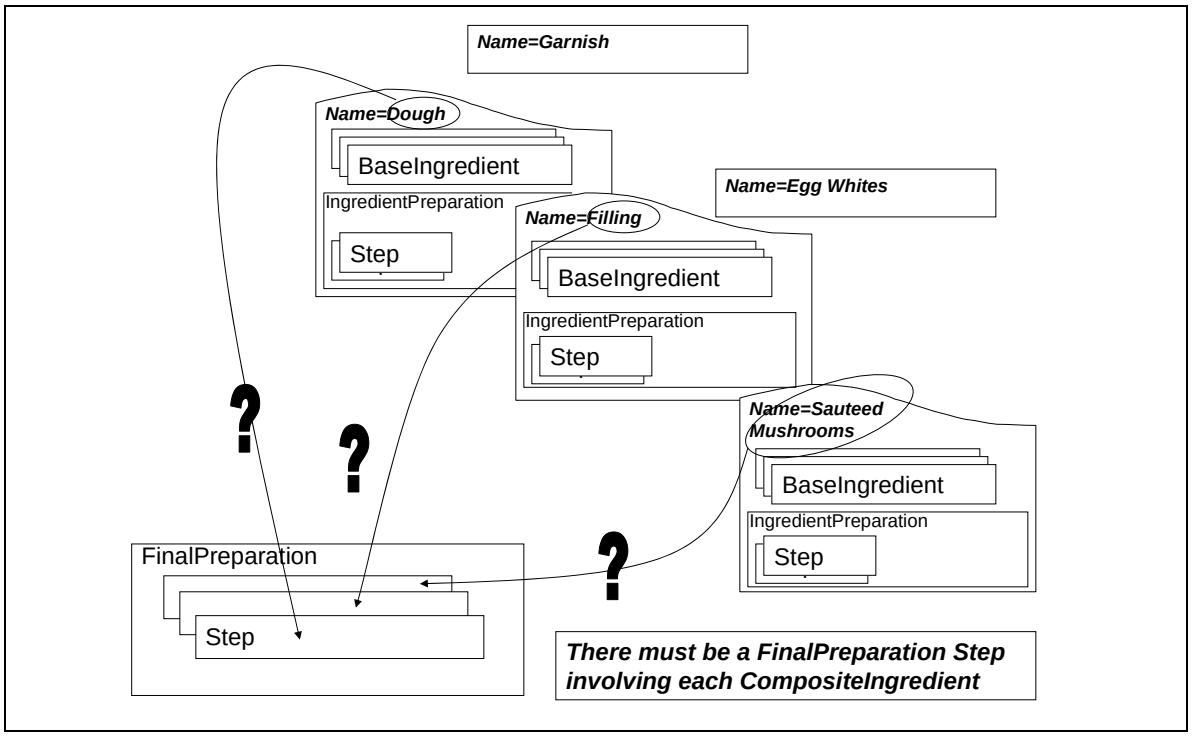

Figure 8. The aligned comparison concept

Again at this point the reader might imagine a loop construct in a conventional programming language. To compare two components in lock step, the same iteration index must be used for them. To know when *false* should be returned and the Rule should fail, we can informally re-express the intent of this constraint from a negative viewpoint as: *The constraint fails to be satisfied if the final preparation step in correspondence with any composite ingredient fails to contain that ingredient's name; otherwise the constraint is satisfied.* Let *n* denote the number of composite ingredients, and let *i* denote an iteration index. A pseudocode formulation is:

```
for i = 1 to n
        if /Recipe/FinalPreparation/Step[i] does not contain i^th CompositeIngredient's name then
                return false;
return true;
```

By logical extension, if two or more levels of paired comparisons are required, more than one looping construct and index can be encoded.

Similarly, the XPath expression of the aligned comparison concept must set up iteration in such a way that the same dollar variable is used to choose paired element repetitions "in lock step" while navigating the XML-MTF document during evaluation. If more than one unique pairing is needed, nested *every* constructs can be used.

For this example, the constraint which enforces the alignment of the *steps* to the *composite ingredients*, expresses in XPath the conditions under which the constraint is satisfied. An expression *E1* captures the Condition part of the Rule by testing whether the *recipe* actually contains *composite ingredients*:

```
gt3-n(count(/Recipe/CompositeIngredient),0)
```

where *gt3-n()* is our 3VL implementation for numeric greater-than. An expression *E2* enforces the Statement part of the Rule and is formulated as follows. Our 3VL implementation of string comparison *eq3-s()* can take a second argument with the special pattern-matching syntax *'*/<s>/*'* which determines whether the first argument contains the string <s> preceded by and followed by zero or more additional characters. (This particular syntax was motivated to ensure compatibility with the syntax of the original constraint language.) Here we use it to make sure the *composite ingredient name* appears in its corresponding *final preparation step*:

```
every $i in 1 to count(/Recipe/CompositeIngredient) satisfies
        eq3-s(/Recipe/FinalPreparation/Step[$i],
                concat('*/',/Recipe/CompositeIngredient[$i]/Name, '/*'))
```

We can combine these two ideas in the form:

rule-satisfied(or3 (*E2*, not3(*E1*)))

This expression evaluates to *true* only if the expressed constraint is satisfied for all *composite ingredients* in the *Recipe*. If the pairing is not "perfect" (e.g., maybe there are *n composite ingredients*, but only *n-1 final preparation steps*), the failure of the document to comply with the designer's intent would have been detected by the first Rule we mentioned earlier which would fail during validation under these circumstances.

Other iteration restrictions

Alignment can be viewed as a restriction on the basic concept of iteration or "for all" inspection. The constraint language we work with supports other iteration restrictions as well, all accomplished using a subscript syntax in the original language. Each has a counterpart in XPath. Some realizations are fairly simple. For example, it sometimes can be meaningful to restrict the applicability of a constraint to only the *n*th or only the *last* occurrence of an element. This can be accomplished by modifying references to that

element with the subscript *[n]* or *[last()]* respectively. For example, referring to the recipe model, the designer might require that the first *final preparation step* in a *recipe* must contain the word "Preheat" to ensure the cook gets the oven going.

Some constraints are satisfied if the assertion is satisfied for just SOME combination of relevant elements in the document: in other words, there must exist at least one situation in the examined document for which the assertion holds. An example is a concept such as *There is some repetition of this element that contains the specified value*. For example, the designer might require for each *base ingredient* in the recipe only that *There is some ingredient preparation step that contains the base ingredient's name.* Such expressions can be re-formulated using the *exists* construct in XPath. Or, a less direct re-expressions can be formulated using the negation of an expression with the *every* construct. For this example, we can loosely state that as: *NOT(every ingredient preparation step does NOT contain the base ingredient's name).* In general terms, if it is NOT the case that every element does NOT contain a specified value, then there must be SOME element that DOES contain that value.

Other restrictions have less direct counterparts too. For example, our constraint language supports expressing an assertion that is satisfied only if there is *NO* combination of relevant elements in the document that exhibit the feature being examined. With a little thought, this can be recognized as the negation of the existential case. That is, to say that "there is NO repetition of this element that contains the specified value" is the same as *NOT(There is some repetition of this element that contains the specified value).* Or, it can be re-expressed as an *every* case in which the examined feature itself is negated. For this example, we could say *every repetition of this element does NOT contain the specified value*.

Value containment is just one possible assertion that can be involved in these formulations; it was used in these examples for the sake of simplicity. Other more complex features can be enforced similarly, using some combination of comparative expressions and the *every* and *exists* constructs.

Proof-of-concept and use cases

To illustrate the practical application of the mappings and the methodologies discussed above, we set up a prototype of the workflow illustrated earlier in Figure 3. We devised a small collection of hypothetical Rules or *use cases* typical of the kinds of constraints expressed in the target operational environment and built a hypothetical XML-MTF Schema for test purposes. We wrote an XPath Generator Stylesheet to demonstrate the feasibility of automating the transformation of the XML-SN (parsed form of Rules) into XSN (XPath/XSL equivalents) bound to that hypothetical document type per the transformation principles described in this paper. We successfully validated the feasibility of the process for the use cases. The proof-of-concept validator and use cases were intended to provide a reference model for future vendor implementations.

Conclusions

To support generalizing XML-native constraint techniques such as we have described here to arbitrary discourse domains, we need to summarize what we have learned about expressing and enforcing constraints in more abstract terms.

General Principles

Based on our experiences, we generalized the following guiding principles for expressing constraints against XML documents:

- To ensure constraints are unambiguous, an accurate specification for encoding [syntax] and interpreting [semantics] constraints must exist. This will increase the likelihood that:
    - Similarly educated operational personnel will discern the same meaning [semantics] from the same Rule [syntax].
    - Similarly educated document and Rule designers will encode similar Rules [syntax] to express similar concepts.
- To appropriately and consistently enforce Rules, an accurate specification for encoding [syntax] and interpreting [semantics] constraints must exist as the basis for implementing validation software. In particular:
    - The syntax must include a construct to provide the navigational information to locate any element in a document structure. We use the term *path name* for this construct.
    - The syntax must require that the navigational information provided to locate elements within the document structure is *complete* and *fully qualified*.
        - *Complete* means the path name syntax must require citing any and all elements needed to locate the referenced element, listed in hierarchical order according to the document specification.
        - *Fully qualified* means the path name syntax must require citing specific occurrences of any repeatable elements it references.
    - For Rules involving comparisons that references repeatable components, syntax must be provided to align comparisons of those components as needed to support the intent of the Rule (i.e., 1-to-1 or *n*-to-*m*).
        - It should be noted that constraints involving *n*-to-*m* comparisons may be suspect or are reducible in complexity.

In this paper, we did not discuss the syntactic details of our original constraint language SN beyond a high level view. We did this to avoid getting the reader bogged down in understanding the tactical messaging world from which this language arose. Nonetheless, one advantage we want to point out to using this sort of "third party" constraint language as a starting point, and transforming it to XPath equivalents, is that it allows the message (i.e., document) designer to express constraints in terms [s]he understands, without having to learn the syntax of XPath. SN in fact complies with the guidance we have listed here for a useful constraint expression language.

Additionally, we generalized the following guiding principles for evaluating constraints against XML documents:

- The validation framework must support the processing logic to realize iteration and alignment when Rules reference repeatable document components.

- The validation framework must support 3VL logic to handle in a consistent way those situations in which Rules cannot be evaluated due to insufficient document content.

Minimal evaluator capabilities

We recommend the following list items as minimal capabilities needed by a constraint evaluator for XML documents.

- The evaluator shall be capable of locating elements referenced in the document instance per the path name.

- The evaluator shall be capable of inspecting an element's metadata as needed to support other functions of the validation process.

- The evaluator shall implement the 3VL logical operations and(), or() and not().

- The evaluator shall implement other functions needed to support constraints expressible in the domain, consistent with the principles of 3VL.

    - We found the following comparison operators were needed at a minimum to migrate our constraint language: numeric equality, inequality, greater-than and less-than, as well as lexicographic equality and inequality. Other functions might be useful as well, such as arithmetic operations.

- The evaluator shall provide constructs for supporting iteration over a collection of elements.

- The evaluator shall provide constructs for supporting restrictions of iteration, such as alignment.

Future Directions

The migration of the SN constraint language to XML-native equivalents will be formally embodied in the appropriate military standards that specify XML-MTFs. Due to project resource constraints, we automated the XPath generation aspect of the XSN-MTF transformation process for only those Rules that involve no iteration restrictions. That is, we implemented "for all" case generation only; we manually generated iterative/alignment examples to bench check them. So further automation work can be accomplished in this area. In addition, the processes need to be generalized beyond the specifics of the XML-MTF domain in which we work; however, we believe the conceptual building blocks for doing that have been well laid out in our work and provide a basis for extension by others.

The migration path we chose for XML-enabling constraint validation allowed us to leverage a 30-year community investment in the definition and management of text-oriented tactical messages and the kinds of information they contain. When migrating the message standard to an XML form, we found the existing notions of document well-formedness and validity with respect to a Schema insufficient for capturing all the internal logical requirements that impact assessing an XML-MTF instance's compliance with the underlying MTF standard. We had to incorporate SN Rule validation to supplement this process.

The need to apply constraints to structured text when rendered as an XML document, including incomplete or flawed documents, is not unique to the military messaging world. XML-native techniques for expressing and enforcing constraints such as we have described here can be generalized to other discourse domains. We found there could be advantages in allowing designers to continue expressing constraints in a legacy language whose terminologies are intuitively meaningful to them, to avoid their having to learn the complexities of XPath syntax, provided a methodology can be established for mapping from the legacy language to XML technologies. This is something to consider when looking at this same problem in other contexts. The principles and capabilities we have laid out as a model for XML-based constraint expression and evaluation, grounded in 3VL, are relevant and useful for the XML community at large.

Acknowledgements

The work described in this paper was performed by The MITRE Corporation under the sponsorship of the *Air Force Command and Control, Intelligence, Surveillance, and Reconnaissance Center (AC2ISRC) Information Management Branch (SCG)* at Langley Air Force Base, VA.

References

[1] Cokus, M. et. al., *Transforming Military Command and Control Information Exchange*, XML 2004 Conference Proceedings, November 2004.

[2] Malloy, M. A., *XML-MTF Derivation Procedures for Message Text Format (MTF) Structural Notation (SN) version 1.3 [draft]*, August 2004, The MITRE Corporation.

[3] Møller, Anders and Schwartzbach, Michael A., *The XML Revolution – Technologies for the Future Web*, http://www.brics.db/~amoeller/XML/xml/example.html.