A Modular Integrated Syntactic/Semantic XML Data Validation Solution

By
Christian Martinez, B.A., M.S.

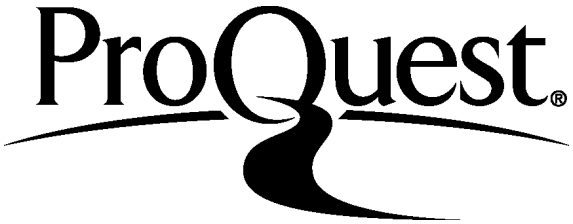
Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Professional Studies
in Computing

at

School of Computer Science and Information Systems

Pace University

May 2016



We hereby certify that this dissertation, submitted by **Christian Martinez** satisfies the dissertation requirements for the degree of Doctor of Professional Studies in Computing and has been approved.

Lixin Tao — 5/20/16

Dr. Lixin Tao
Chairperson of Dissertation Committee

Date

Ronald Frank — 5/20/16

Dr. Ronald Frank
Dissertation Committee Member

Date

Meikang Qiu — 5/20/16

Dr. Meikang Qiu
Dissertation Committee Member

Date

Seidenberg School of Computer Science and Information Systems

Pace University

Abstract

A Modular Integrated Syntactic/Semantic XML Data Validation Solution

by
Christian Martinez

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Professional Studies
in Computing

May 2016

Data integration between disparate systems can be difficult when there are distinct data formats and constraints from a syntactic and semantic perspective. Such differences can be the source of miscommunication that can lead to incorrect data interpretations. What we propose is to leverage XML as means to define not only syntactic constraints, but also semantic constraints.

XML having been widely adopted across multiple industries, is heavily used as a data protocol. However, commonly used XML parsers have only embedded syntactic validation. In other words, if semantic constraints are needed, these come into play after a parser has validated the XML message. Furthermore, semantic constraints tend to be declared inside the client system, either in the code itself or in some other form of persistent storage such as a database. Our solution to this problem is to integrate the syntactic and semantic validation phases into a single parser. This way, all syntactic and semantic rules can be configured outside the client system. For our purposes, semantic rules are defined as co-constraints. Co-constraints are when the value, presence or absence of an element or attribute is dependent on the value, presence or absence of another element or attribute in the same XML message. Using this same concept, we have also built a parser that, based on co-constraints, can express business constraints that transcend the message definition. Our research provides a reusable modular middleware solution that integrates syntactic and semantic validation. We also demonstrate how the same semantic validating parser can be used to execute business rules triggered by semantic rules.

Combining semantic and syntactic validation in the same XML parser or interpreter is a powerful solution to quick integration between disparate systems. A key of our proposal is also to have the syntax definition and semantic definitions separate, allowing them to evolve independently. One can imagine how syntax might not change between systems, but the semantic constraints can differ between message consumers.

Acknowledgements

First and foremost, I want to thank my wife and kids for putting up with my time spent on this dissertation. I can still hear the complaints from them pointing out another weekend thrown away (in their eyes). Secondly I want to thank Dr. Lixin Tao. His patience and guidance were the light that shone my path. Also, all of my friends and family who kept asking me when the day would come, well, it is finally here.

Trademarks

All terms mentioned in this dissertation that are known to be trademarks have been appropriately capitalized. However, the author cannot guarantee the accuracy of this information. A list of these trademarks is given below (It is not exhaustive):

W3C, XML, XPath, XSL, and XSLT are Registered Trademarks of World Wide Web Consortium (W3C).

Apache, Maven, Xerces and Xalan are Registered Trademarks of Apache Software Foundation.

Saxon is Registered Trademarks of Sourceforge.net.

Table of Contents

Listings

List of Figures

Chapter 1

Introduction

When thinking about messaging, we need to focus on a protocol and by extension on a message grammar. In order to communicate between separate systems, a pre-defined grammar has to be established. In addition to grammar, there is the need to define application level rules. These rules we refer to as semantic constraints. In an effort to simplify the development of messages between clients, we feel that the inclusion of semantic constraints can further solidify the expected behavior in either side of the transaction flow. A similar but broader scoped architecture has been introduced by standard bodies such as the World Wide Web Consortium (W3C) [1][2][3]. In that design, semantic constraints are managed and executed in the "Semantic Mediation" phase. The objective is to analyze the messages and produce semantic validation results for higher-level applications. In our solution, these rules can be defined and executed during the message validation phase. The results of the validation are communicated back to the user of the library.

Our research is an extension of an existing integrated parser [4] that focuses on reusability, ease of use and integration. On testament to us having achieved goal is that, prior to the defense of this research, our solution started to be used in other research projects. In this chapter we compare different syntactic definition specifications that have attempted to merge syntax and semantic expressions into a single definition. We

feel this is suboptimal and demonstrate that advantages of using our parser. This section explains the constructs involved in defining an XML based grammar and semantic constraints. One way to think about XML is as a meta language used to define business or system specific protocols. These protocols are backed by a syntax defined in one of the many syntax definition languages we will discuss later. To extend the power of the protocol, we need to add semantic constraints that are easily validated as part of the syntactic validation phase. We leverage Schematron as our grammar for semantic co-constraint definitions. Our solution encapsulates both strong syntactic constraints and very flexible expressive semantic constraints based on Schematron in a single parser. This is our main contribution in this research.

1.1 Importance of XML Syntax and Semantic Validation

As a pre-requisite to any XML based message, the message needs to be syntactically valid. A valid document implies a well-formed document. To briefly describe the distinction, a well-formed document only needs to follow the basic constraints provided by the XML specification. A valid document however, needs to not only abide to the XML specification, but also needs to follow the grammar-based rules defined in the syntactic constraint or grammar definition document [5]. As part of this research, a third phase is introduced to the validation process, namely the semantic rule validation phase.

As far as this research is concerned, semantic rules are defined as co-occurrence constraints (co-constraints). What this means is that the presence, sequence or value(s) of certain elements are dependent on the presence, sequence or value(s) of other elements [6][7]. Schematron has been used as the base for semantic rule definitions. One of our

goals was to implement an XML parser that can incorporate the use of XML Schema with Schematron providing a single application-programming interface (API) that would combine syntactic and semantic validation. Although there already exists such a parser [4], our contribution has been focused on adding support for the Schematron ISO version. In doing so, we have also redesigned the parser's in-memory representation of Schematron. With a middle-ware focus, the API for the parser produced by this research, has been created to serve as a library dependency on a larger system. One of our goals was to produce an easy to use reusable library that could easily be integrated with existing enterprise services.

1.2 Challenges to Current XML Validation Methods and Techniques

When it comes to XML schema grammars, we have discussed DTD and W3C XML Schema. These two grammars provide syntax to describe every element and attribute in an XML document. RELAX NG another grammar, works by identifying patterns between elements, text and attributes. Patterns allow for ambiguous element definitions. There can be a child element with the same name as the parent but with different attributes. In DTD and W3C Schema grammars, such ambiguity is not allowed as every element and attribute combination must be defined. Finally, there is Schematron. Schematron is a rule-based grammar. Validity is determined by the success or failure of rules that help identify relationships between elements and attributes of an XML document. Each grammar by itself seems to fall short of providing a complete solution to syntactic and semantic validation [8]. What becomes apparent, and a driver of this research, is that a combination of XML structure definition and rule based grammar definition is ideal [9]. The result is a combined grammar that allows for broader

relationship definitions that can encapsulate the syntactic and semantic aspects of the parser brought about by this research. The parser is a reusable middleware component that can be combined with any XML schema language to create a complete syntactic and semantic validating solution.

With the introduction of the Java XML Validation API [10] the ability to select any XML grammar for syntactic validation has been built into the java validation API. Our focus has been to allow a user the seamlessly integrate semantic validation into their syntactic validating parser.

Research by Steven Golikov [4] rendered an integrated syntactic and semantic parser that improved on the default Schematron parser implementation by providing an in-memory representation of the Schematron tree structure. This parser does not however implement features in the ISO version of Schematron. Our research aims to extend Steven's research and create an ISO compliant Schematron parser with syntactic validation. Our goal is to focus on delivering reusable component that can be integrated into an existing XML message producing and consuming system.

1.2.1 Separated Syntax and Semantic Validation

As mentioned above, current parsers tend to specialize on a particular aspect of XML content validation. If we look at RELAX NG, DTD and W3C Schema parsers, it is clear that the focus is mainly on syntactic validation. Although there has been research to integrate syntactic and semantic validations, the suggested solutions have been to combine the syntax and semantic rules into a single document. This is where we differ in that we suggest separating the two as we feel the two can evolve at different rates.

For example, imagine a shipping system that would allow for free shipping if the total order amount exceeded a specific number and the customer paid for premier membership. If we take a W3C Schema, we would have to introduce a conditional element. Since we cannot do this we can accomplish this task by defining an extension that could omit the presence of a shipping cost element or attribute. Below is the actual schema excerpt.

```
<xsd:complexType name="NoShippingPurchaseOrder">
    <xsd:sequence>
        <xsd:element minOccurs="1" maxOccurs="unbounded" ref="ex:inventoryItem" />
    </xsd:sequence>
    <xsd:attribute name="orderId" type="xsd:string" use="required" />
</xsd:complexType>

<xsd:complexType name="PurchaseOrderWithShipping">
    <xsd:complexContent>
        <xsd:extension base="ex:NoShippingPurchaseOrder">
            <xsd:sequence>
                <xsd:element name="shippingCost" minOccurs="1" maxOccurs="1"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<xsd:element name="inventoryItem" abstract="true">
    <xsd:complexType>
        <xsd:attribute name="inventoryId" type="xsd:string" use="required" />
        <xsd:attribute name="price" type="xsd:double" use="required" />
        <xsd:attribute name="qty" type="xsd:int" use="required" />
    </xsd:complexType>
</xsd:element>
```

Figure 1 Example of W3C Schema Inventory

In **Figure 1**, we see a W3C Schema definition for a purchase order coming out of a warehouse. The rule determining which version of the shipping related element to use lies within the application that processes the schema. In this scenario changes to the rule, such as allowing all customers not just premier members access to free shipping would require an application level change. That is not to say that there can be multiple ways of implementing a rule validation system, but having the rule defined outside the markup inevitably adds complexity to how such rules get managed. Below is a figure demonstrating how such a rule can be implemented using Schematron. One thing to keep in mind is that the syntactic and semantic rules will be fully separated in terms of their definition, but from a transactional perspective, the semantics are treated as an extension to the syntactic rules. **Figure 2** is a representation of an Schematron schema excerpt that would handle the co-constraint validation regarding shipping cost. The excerpt uses some of the syntax introduced by the Schematron ISO version. In essence, two global variables are defined and used during the rule evaluation. First is a threshold that can be used to set a minimum amount constraint on whether the order qualifies for free shipping or not. The second variable is a Boolean which checks if the user is a premier member and if the total amount exceeds the minimum amount defined by the previous variable. If and when the rules change, the expression in these two variables is what needs changing. Besides the basic knowledge of XPath, the changes are very simple and isolated when thinking about changing a more complex W3C Schema file. Also, the separation allows for rules to be defined and executed while parsing. Rules like the ones defined in Schematron are not possible using W3C Schema. Not only does separating the rules from the syntax allow for greater flexibility and simplicity, it also simplifies the schema

definition. I think this is one important contribution, that by supporting the modularity of the Schematron ISO version, we can simplify the management of Schematron rules, and we can also simplify the XML's grammar because element extensions denoting special circumstances, like the *NoShippingPurchaseOrder* element above, are no longer needed. Eliminating redundant markup will simplify the maintenance of the grammar as well.

```
<sch:let name="freeShippingAmountMinimum" value="100"/>
<sch:let name="hasFreeShipping"
         value="child::customer/@isMember and child::order/@total > $freeShippingAmountMinimum"/>

<sch:phase id="invoice">
    <sch:active pattern="orderValidation"/>
</sch:phase>

<sch:pattern id="orderValidation">
    <sch:rule context="order">
        <sch:assert test="$hasFreeShipping and @shippingCost=0">
            The customer should not have received a shipping cost.
        </sch:assert>
    </sch:rule>
</sch:pattern>
```

Figure 2 Schematron excerpt

There has been research proposing a merger of grammars [7]. Schemapath was proposed as an extension to the existing W3C Schema grammar. As previously demonstrated, the W3C Schema grammar lacks that ability to support co-constraint expressions. Schemapath proposes adding a new element to the W3C Schema syntax that would allow for co-constraint expression to be injected. Although this is an elegant solution that combines syntactic and semantic validation, there are some limitations that our research overcomes, most notably the co-location of syntactic and semantic grammar. For this first part, a user of Schemapath would not be able to choose a different semantic grammar as Schemapath is an extension of W3C Schema and as such every Schemapath document

must be a valid W3C Schema document. Second, it would seem counter intuitive to have rules that can access a random document element. How would a rule know that an element it depends on has passed the syntactic validation if the rule is executed prior to validating the entire XML document? This limitation would make rules focus on syntactic validity and not on overall semantic constraints.

As shown above, Schematron offers us the power of validating distinct parts of an XML document. Rules can look at any part of a document. This is however a dangerous feature. Looking into all parts of an XML document assumes the syntax of the document is valid and that all element values and attribute values are accurate. Our research solves this problem by allowing for the integration of syntactic validation. With the ISO version of Schematron, showing in **Figure 2**, new powerful constructs have been introduced which add even more flexibility to a Schematron definition document. In the coming sections we will dive into further details on such extensions.

1.2.2 Lack of Support for ISO Version of Schematron

As mentioned, our research aims to create an Schematron middleware parser that supports the changes and extension introduced in the ISO version of Schematron. The research done by Steven Glikov [4] introduced a java parser for Schematron that integrated syntactic and semantic validation. However, that parser supported Schematron 1.5. It lacked features that make creating complex Schematron documents today much simpler. Below is a brief description of what these features are; Chapter 2 will dive into more detail.

In the ISO version of Schematron, attention has been paid to the modularization of a definition document. Fragments of Schematron grammar, or modules, can be created and injected, reused as needed. An ***include*** element has been introduced which allows a user to reference on of these fragments. This element can be nested in the ***schema, diagnostics, phase, pattern*** and ***rule*** elements. This feature greatly improves the structure of an Schematron document. In version 1.5, there was no way to reuse markup using pure markup.

The next feature introduced is the ***let*** element. This element allows for the definition of variables. Original Schematron parsers were XSL based parsers. An XSL transformation would be done using the Schematron document as the source of the resulting markup. The ability to inject variables using XSL made if very friendly to reporting user readable output. In the ISO version, variables can now be expressed and reused in the Schematron schema. Variables can be declared at different levels and contexts. The elements ***schema, pattern, phase*** and ***rule*** can all contain variables. The scope of a variable depends on where it was declared, more on this in Chapter 2.

Finally, there is the ***param*** element. This element is the result of allowing patterns to be abstract. In version 1.5 of Schematron, only ***rules*** could be abstract. In the ISO version, this ability has been extended to the ***pattern*** element. The best way to explain the additional expressiveness added by this extension is the think of an abstract pattern as a reusable function, much like a function in a programming language such as java. A parameter is a variable passed in to a ***pattern***. Only ***pattern*** elements that extend an abstract ***pattern*** can have parameters. Chapter 2 will go into further detail on this topic.

1.2.3 Complex Non-Standardized Processing of Schematron Documents

Being based on Steven Glikov's [4] research, we aim to extend and improve the parser produced by that research. One of the complexities involving the existing parser is the proprietary in-memory representation of an Schematorn document. The parser builds an object hierarchy, which is a one to one mapping of the Schematron grammar. In our approach, we did not want to duplicate the grammar in-memory. Instead, we wanted to encapsulate access to the Schematron grammar. The parser is based on XPath based expressions mapping to fragments of an Schematron document. The fragments are encapsulated and accessed by fragment readers. Fragment readers have a root XPath expression set which delimits the Schematron fragment they are responsible for accessing. This way the parser's implementation is greatly simplified. The complexity of accessing and dealing with XPath is encapsulated in the Java XML API. In summary, the in-memory representation of Schematron is the Document Object Model representing the Schematron grammar.

1.2.4 Difficult to Run as Reusable Middleware

The existing parser was designed to run as a sub-system with integration capabilities at different levels. The system can be interacted with either directly by a user or automatically by peer systems. The difficulties come in when certain network or system architectures do not allow for simple intra-system deployments and integrations. Our proposed parser was developed as a library that can be customized by a development team as they see fit. It can run inside an existing system exactly as an integrated library would, or it can be exposed as a service in a service oriented architecture fashion inside

an enterprise [11]. By focusing on granular integration, we have maximized flexibility for integration.

1.3 Problem Statement

The benefits of an integrated parser that can do syntactic and semantic validation speak for themselves. Our integrated parser is based on a modular design allowing users to inject their own syntactic validating technology into our single validation phase. By default, we support the W3C XML Schema grammar for syntactic validation. To support semantic rules, we have implemented the ISO version of Schematron. We have leveraged the Document Object Model (DOM) in order to avoid creating customized data structures to handle the semantic grammar during validation. We have also implemented a semantic rules pre-processor using macro-expansions to inject external and/or abstract content into a single DOM structure. The result is a very light weight module that can be integrated into any type of system, from an enterprise wide system to a simple validation service.

The proposed integrated parser enables users to separate syntactic grammar from semantic grammar. The syntactic validation phase is optional, in certain scenarios, users might want to improve on performance by only focusing on the semantic validation phase. All of the Schematron ISO feature have been implemented and are fully supported.

1.4 Solution Methodology

1.4.1 Macro-Expansions

With the new ISO features that allow the external injection of fragments through the ***include*** element and the added support for abstract element inheritance, we decided to introduce macro-expansion [11] so that the entire abstract and external markup was pre-processed during initialization. This approach greatly simplifies how the parser validates. Because the entire markup has been pre-processed and included in the state of the parser, there is no need to fetch values and interpret abstraction while validation is taking place. This greatly simplified our algorithm during validation.

1.4.2 Standardize In-Memory Representation

Another strong improvement done on top of the existing integrated parser is the use of the Document Object Model (DOM) to represent the schema in memory [13]. We decided to leverage a standardized, well-understood model to represent the schema. This would simplify again how we interacted with the different elements of the schema while validation is taking place. Macro-expansion was a definite pre-requisite for adding the DOM.

1.4.3 XPath as Our Internal Query Language

Leveraging the DOM gave the ability to use XPath as the language of choice for our querying engine. During validation, internally the parser is constantly accessing the Schema to read-in and execute the ***assert*** or ***report*** expressions. In addition to the expressions, the ISO has added support for variables declarations that have their values loaded at runtime during validation. XPath has been very instrumental in our research.

1.4.4 Modular Application Programming Interface

To facilitate integration and adaptation, we focused on delivering an easy to use API that can be treated as any other dependent library. The parser is meant to be used as part of a larger system; it was not designed to run independently. This decision we feel made our research architecture agnostic. We hope that the ease of integration will give us some future proof benefits. As architectures evolve, we did not want to tied to a specific implementation.

1.5 Research Contributions

Our research has yielded contributions in the architecture an integrated parser and in it applicability of that parser. In addition, it is an integrated parser supporting the Schematron ISO version. Below is a detailed list of all the contributions.

- Provides improvements on the design and integration from the previous parser. Leveraging standardized XML based data structures coupled with a modular architecture opens up the parser for extensions.
- The use of macro-expansions as an implementation solution to the integration of the Schematron ISO features is also a contribution. Macro-expansions are a clever solution to the injection of markup at runtime.
- The ability to selectively enable or disable syntactic validation. Syntactic validation is somewhat expensive to execute on every XML message. Being this the case, it is not very practical to always syntactically validate message that might for the most part remain static in terms of their XML structure.

- Easy integration with existing systems. The parser can be treated as a separate component that is mostly controlled by the XML and Schematron schemas.
- Can be thought of as an abstract architecture for a rules based processing API. The functionality present in Schematron's ISO version is very powerful. Abstraction and insertions are just two features that allow for the maintainability of enormous schemas. One can maintain the design and replace the rules and syntax definition with other non-XML based protocols.

1.6 Dissertation Road Map

In this section we give an outlook as to what to expect in the rest of this dissertation.

Chapter 2 entitled 'Review of Current XML Validation Approached' explains current XML technologies from the syntactic validation and the semantic validation viewpoints. In the chapter we begin by explaining alternative technologies that can help in defining the grammar for an XML message. Our parser can leverage any of these technologies as the syntactic and semantic validation phases can execute separately. Following the syntactic introduction, we explain existing technologies that merge both syntactic and semantic validation. We complete the chapter by comparing our solution versus the existing solutions. We explain why our approach provides advantages over the existing approaches.

Chapter 3 entitled 'A Simplified Design for Integrating Syntax/Semantic Validation Using Schematron' In this chapter we begin explaining in detail how our validator works. We explain the architecture behind the validator and how it supports the non-ISO Schematron features. The validator maps each Schematron feature and is able to, at runtime, access any part of the Schema by leveraging the standardized Document Object Model (DOM) and XPath. The chapter ends with an explanation of how validation results are handled and notified.

Chapter 4 entitled 'Adding Support For ISO Schematron Features' is an extension of the previous Chapter. This chapter focuses on how we have implemented the new Schematron ISO features in the validator. A main focus is to demonstrate the advantages to schema maintenance and development the new features bring. The ISO allows schema developers to componentize and modularize their schemas. A concept of inheritance at multiple levels and the inclusion of external schema fragments are a clear advantage over the pre-ISO Schematron features.

Chapter 5 entitled 'Experimental Validation' focuses on examples that demonstrate how the validator works. We demonstrate via examples how to use the syntactic and semantic capabilities of the validator. We explain in detail the grammar definition as well as the Schematron schema used in every example. In this chapter we also demonstrate how to use the validator in a stand-alone format.

1.7 Conclusion

In this chapter we explained the necessity for an integrated semantic and syntactic XML parser. We also touched on the benefits of using Schematron as the syntax for semantic validation. Our parser's implementation introduces new approaches that leverage standardized technologies to aid in the XML semantic and syntactic validation process. Our parser also presents itself as a reusable middleware component. This allows it to be suitable for a wider array of applications. The following chapter will deal begin to further our approach at an integrated syntactic and semantic validating parser.

Chapter 2

Review of Current XML Validation Approaches

In this research, we will define XML validation as being composed of two phases. The phases are organized in a waterfall manner, where the success of the first phase impacts the execution of the second. We recognize that each phase can be executed independently, but for clarity and integration of our validator, we choose to integrate them.

The first phase has to do with syntactic validation. In our examples, XML is used to encapsulate markup backed by a grammatical definition. The grammar identifies the constructs of the markup, from typing, cardinality and ancestry, to default values for optional content. The second phase is focused on co-constraint validation. The co-constraints are defined as XPath rules. It is because of these rules that we prefer to integrate the syntactic validation phase with the semantic one. The rules must have the full grammar available to them; in addition, there must be a guarantee that the XML instance abides to the grammar as well so that the rules can be executed.

In the sections below, we dive into more detail on each phase. At the end of this section, we also explain our integrated phase solution and compare it with existing solutions.

2.1 XML Syntax Validation

Created as a solution for the representation of dynamic content on the web, the Extensible Markup Language (XML) has grown beyond use on the web alone. Standards such as FIXML, an XML representation of the FIX protocol, are heavily used in the financial

industry [14] as an inter-system data transfer format. XML has been able to survive and adapt to different environment because of its flexibility when it comes to grammar definition. XML makes a distinction between well-formed documents and valid documents. By extension, a valid document must be well formed, but not the other way around. For more information on the rules governing a well-formed XML document, please refer to the website of the W3C [15]. They are the consortium responsible for the maintenance and publication of the XML specification.

As mentioned, one of the key aspects of XML that have allowed surviving for so long, is the flexibility in grammar definitions. Over the past couple of years, a number of grammar defining specifications have been developed and adopted by the industry. Later in this chapter we will dive into detail on some of the most popular ones. When it comes to explaining syntax validation, it is very important to understand that an XML document is said to be valid if it complies with the grammatical rules defined for that document. Below we begin discussing the different specifications generally used to define grammatical constraints on XML.

2.1.1 W3C XML Schema Language

A Schema definition contains the structure an XML document must adhere to in order to be considered valid. It defines constraints on the composition of its elements, what data types are supported by each element's content and an attribute's value as well as the existence of default values used when attributes are left out of the instance document [15]. The latest version of the W3C XML Schema has also incorporated the ability to evaluate co-occurrence constraints. The constraints are defined under a new ***assert*** element type. It stills holds that coupling the syntactic and semantic validation forces

both sets of validations to be executed at the same time. Maintainability of co-occurrence constraints is more complex and might affect the validity of the schema document.

2.1.2 Document Type Definitions (DTD)

When thinking about the W3C's DTD specification, the first thing that comes to mind is, why is it used instead of the W3C XML Schema (XSD) specification? One reason could be complexity [16]. DTD's do not provide the level of sophistication and complexity, as do XSD's. But, with complexity come features and extensions. One very important feature is the ability to define data types, extensions and cardinality. With a DTD a user cannot enforce that an element of type *Price* exist as the first child of an element named *Order*. In an XSD, this would be rather simple since the concept of strongly typed elements is built into the specification. In addition, when it comes to cardinality, DTD's are limited to zero or one, zero or more or one or more. With XSD's, an element or attribute can have *maxOccurs* and *minOccurs* attributes to define any cardinality constraints [17].

However, DTD's do allow for the definition of *ENTITY*'s. These constructs can be thought of as variables to be referenced within the DTD definition. XSD's do not replace this construct. As becomes apparent, even when it comes to XML document grammar definitions, there is no silver bullet. The use of DTD's versus XSD's comes down to personal choice as well as complexity of the instance document. If the grammar is for a document that defines a very simple syntax without the need to data types and complex cardinality constraints, then DTD's might prove to be very appropriate. Simplicity can go a long way.

2.1.3 Relax NG

Designed by OASIS (Organization for the Advancement of Structured Information Standards), a standards body similar to the W3C (World Wide Consortium), Relax NG proves to be a simpler lighter weight solution to both DTD's and XSD's. When it comes to grammar definitions, Relax NG supports a compact syntax that makes writing such a grammar a bit more intuitive, as show in **Figure 3**. Being based on a concept of pattern definition, grammars defined using Relax NG can consist of complex constraints without a need for cumbersome syntax. For example, in **Figure 3**, we can see how an attribute has been made optional while still enforcing that one of them be present. In the excerpt, we can see two Relax NG features being used. The first, delimited by a parenthesis prior to the declaration of the *inventoryId* attribute, is the group feature. Putting element or attribute declarations within parenthesis creates a grouping to which specific constraints can be applied [18]. In our example, we used the *choice* feature that enforces one but not both attributes are present. This constraint would not be possible with DTD or XSD because of the way in which they treat attributes. The only constraints that could be applied would be to make each attribute optional. This would incorrectly validate an *inventoryItem* element that did not have either the *inventoryId* or the *externalId* present. Aside from its simplicity, Relax NG also adds more expressiveness to XML grammar definitions.

```
element purchaseOrder {
    element inventoryItem {
        ( attribute inventoryId { text },
            | attribute externalId { text } ),
        attribute price { xsd:decimal },
        attribute qty { xsd:decimal }
    }*,
    attribute orderId { text }
}
```

Figure 3 Relax NG Excerpt

2.2 XML Semantic Validation

The idea of representing semantic information with XML has been around for a very long time. It can be said that one of the primary incentives behind the invention of XML was to allow various systems or businesses to exchange information with semantics [19]. In 1998 the ISO started the Basic Semantics Register (BSR), a registry of XML grammars spanning different industries. The goal was to define the syntax and semantics for XML message across various industries [20][21]. Today, there are numerous XML grammars defined across various industries. Early in this dissertation we referenced FPML (Financial Products Markup Language), a markup used by financial industries to communicate data on swaps, derivatives and structured products. As is obviously apparent, XML has been successfully used to communicate messages between distributed business partners. However, since XML does not natively support semantic definitions it has been very difficult for it to be universally accepted for certain forms of transactions [22]. To address this limitation, as we have discussed, there have been multiple solutions that integrate semantic rules and validation into a tradition XML processing system. In its latest Schema specification, the W3C has added support for semantic rule definitions.

Below we go into further detail on existing XML semantic validating technologies with a simple modification. Because the W3C specification defines co-constraint support in the same way as SchemaPath, in this section we will discuss them together as a unit. Following that discussion, I will introduce Schematron and explain its advantages and disadvantages over the W3C Schema and SchemaPath. As an added note, RELAX NG also supports semantic rules in the form of Schematron rules. For this writing, I will focus on Schematron alone.

2.2.1 W3C Schema 1.1 And SchemaPath

SchemaPath is a simple extension to W3C Schema. What this means is that SchemaPath supports the entire features of the W3C Schema 1.0. We specify the 1.0 version since in version 1.1, the W3C has introduced a solution very similar to that introduced by SchemaPath. Next we discuss the SchemaPath's solution and use that as an introduction to the W3C's solution.

A valid SchemaPath document is by default a valid W3C Schema 1.0 document. To support semantics, SchemaPath has introduced a new element <*xsd:alt/*> and a new type <*xsd:error*> [23]. A schema author can use the *alt* element to control the type of an element or an attribute. The type *error* is used to communicate the failure of an alternative or *alt* condition. As seen in **Figure 4** taken from the introduction to SchemaPath paper [23] an *alt* element's *cond* attribute holds an XPath expression that is used to validate that alternative type. The *type* attribute defines what the type of the element or attribute will be based on the validity of the *cond* expression. As can also be seen, if a condition evaluates true when it should not have, the *error* type can be used to notify the SchemaPath engine of an error.

```
<xsd:element name="description">
  <xsd:complexType>
     <xsd:attribute name="print" type="PrintCodeType"/>
     <xsd:attribute name="color">
        <xsd:alt priority="0"     type="PantoneCodeType"/>
        <xsd:alt cond="../@print" type="xsd:error" />
     </xsd:attribute>
  </xsd:complexType>
</xsd:element>
```

Figure 4 SchemaPath Excerpt

In **Figure 4** we also see the use of the *priority* attribute of the *alt* element. The attribute gives priority in descending order to conditions when multiple conditions evaluate true at the same time. To summarize the example in **Figure 4**, a description cannot have a print attribute when a color attribute is present. The types of the print and color attribute are also different. Next let's discuss how the W3C Schema, a much more widely used XML technology, has added co-constraint capabilities to its syntax. However, a major difference is that the W3C's support for co-constraints does not alter the type of the encapsulating element. The focus is not a type validation or modification, but more on type instance structure as defined by its syntactic definition.

The W3C's Schema Definition Language (XSD) has gained enormous support and popularity in recent years. Because of its popularity, the W3C began to work on improving the usability and extensibility of the XSD. In 2005 they began working on the 1.1 version of the XSD that promised to eliminate some of the pain points expressed by developers. Some of these pain points related to type inheritance, the use of wild cards for complex types and the lack of support for co-constraints [15][19]. As mentioned, the support added for co-constraint validation resembles a bit the solution provided by SchemaPath. Built in to the language is now an *assert* element who's *test* attribute holds

an XPath 2.0 expression which validates the co-constraint. In comparing it with SchemaPath, an *alt* element was added with a *cond* attribute that encapsulated the rule definition. *Assert* elements can be included in XSD *complexType* element and *restriction* elements inside a *simpleType* element. As with SchemaPath, they don't provide a way of customizing error messages and are only ran against the encapsulating element [25]. That is, the element it is defined for provides the context for the rule. In **Figure 5** we look back at the SchemaPath example using the XSD syntax.

```
<xsd:element name="description">
    <xsd:complexType>
        <xsd:attribute name="print" type="PrintCodeType"/>
        <xsd:attribute name="color" type="PantoneCodeType" >
            <xsd:assertion test="../@print" />
        </xsd:attribute>
    </xsd:complexType>
</xsd:element>
```

Figure 5 W3C Schema With SchemaPath Assertion Excerpt

When a schema author starts thinking about defining co-constraints using SchemaPath or the XSD 1.1, he is forced to focus and couple the syntactic definition with the co-constraint. This might be a limitation when semantically speaking, a rule spans across multiple elements. In such a scenario, the thought process might be beyond that of a syntactic nature and more about business rules or semantics. This is a significant limitation when comparing these technologies with Schematron. In Schematron, the schema author's focus is much broader. By grouping rules into patterns, the authors through process can be more global in nature. Looking at the data as a whole unit rather than focusing on particular elements. In the next section, we discuss Schematron in more detail.

2.2.2 Schematron

Schematron is a rule based schema definition language. The rules are organized into *patterns*. A *pattern* can be thought of as a logical grouping of cohesive rules. With Schematron the schema author is given the opportunity to think about what the data relationships and semantics are within the XML message. For instance, in the example below displayed in **Figure 6**, we show two distinct orders. The first order was done by a customer who also happens to be a member of the website, the second order done by a non-member customer. The structure of the order element is quite distinct. When developing a grammar-based schema for this message, a schema author would have to think about the optional elements that can be contained inside the customer element. It would be impossible to express the required existence of the contact and payment child elements of a customer when that customer is not a member, and the absolute prohibition of them when the customer is a member. With a rule based schema definition language like Schematron, a schema author can think about the fact that non-member customers do not have pre-existing payment and contact details. This fact translates to the required existence of the contact and payment elements. The semantics of what it means to be a member translate into rules in the schema.

```
<orders>
    <order total="100" shippingCost="0.0">
        <customer isAMember="true" memberId="member000"/>
        <item id="item1" price="50" qty="1"/>
        <item id="item2" price="50" qty="1"/>
    </order>
    <order total="100" shippingCost="14.34">
        <customer isAMember="false">
            <contact>
                <address>123 Elm St.</address>
                <city>New Hampshire</city>
                <postal-code>34587</postal-code>
                <email>john.doe@nowhere.com</email>
            </contact>
            <payment type="amex" number="xxxxxxxxxx" secId="4532"/>
        </customer>
        <item id="item11" price="50" qty="1"/>
        <item id="item22" price="50" qty="1"/>
    </order>
</orders>
```

Figure 6 Co-Constraint Instance Examples

In Schematron, rules are defined inside *pattern* elements. A *rule* element's *context* attribute contains an XPath expression used to set the context for the different assertions and reports to execute. *Assertion* and *Report* elements define their expressions in the *test* attribute. When and *assertion* evaluates to false, then the rules fails. When a *report* evaluates to true, then the report's content is executed [26]. Below in **Figure 7** is the Schematron schema used to evaluate the order message in **Figure 6**. The Schematron example in **Figure 7** uses an abstract pattern to define validation rules for both conditions where the customer is a member and when the customer is not a member. The ability to support abstract patterns was introduced in the ISO version of Schematron. The rule with the context variable name $**nonMemberCustomerContext** checks that all non-member customers have contact and payment child elements. The use of an explicit variable name that indicated a non-member validation rule was done for the convenience of expressing the XPath constraints.

```
<sch:pattern abstract="true" id="customerValidation"
        xmlns:sch="http://purl.oclc.org/dsdl/schematron">
    <sch:rule context="$orderContext">
        <sch:assert test="@quantity > 0"> Orders must have a quantity </sch:assert>
    </sch:rule>
    <sch:rule context="$customerContext">
        <sch:report test="$isAMember and *">
            A member should not contain any additional elements.
        </sch:report>
        <sch:assert test="$isAMember and $memberId">
            If the customer is a member it must contain a memberId.
        </sch:assert>
    </sch:rule>
    <sch:rule context="$nonMemberCustomerContext">
        <sch:assert test="child::contact and child::payment">
            If the customer is not a member it must contain a contact and payment
            child elements.
        </sch:assert>
    </sch:rule>
</sch:pattern>
```

Figure 7 Customer Validation Rules

2.3 Integrated XML Syntax/Semantic Validation

With the exception of Schematron, in the previous section we described existing popular technologies that have in their own ways integrated semantic and syntactic validation. The major difference between those semantic validating technologies and Schematron is the scope of semantic rules [27]. In other semantic validators, semantic rules have been restricted to execute within the context of the element actively being syntactically validated. This limits the expressiveness and breadth of a co-constraint definition. A schema author will tend to define co-constraints within the context of a single entity rather than co-constraints across entities in distinct parts of a document. With Schematron, a schema author has the freedom to express co-constraints binding elements that might not have syntactic constraints between them. An example of a shift in approach when defining co-constraints using Schematron, is the use of the *pattern* element. A pattern can be thought of as logically grouping of co-constraints and not one

based on syntactic constraints or limitations. Our goal is to bring the best of both worlds to schema authors.

This research is an extension of the research done by Doctor Steven Golikov [4] that defined an integrated syntactic and semantic XML validator. One key distinction is that in this research we have made the semantic validation phase a pluggable component. What this means is that executing the syntactic validation phase is optional. This ability can prove useful when working in a highly automated environment where the XML producer and consumer systems are very familiar. In this type of environment, limiting the execution of the syntactic validation phase can prove to have significant performance benefits. Below we go into detail on comparing and contrasting our solution to existing approaches.

2.3.1 Analysis of Existing Integrated Parser

As discussed earlier, the existing parser was designed as a sub-component inside an existing enterprise system. Integration can be done by inline or in process or via a web server and a web services application programming interface (api). The parser encapsulates syntactic and semantic validations. A custom syntactic validator has been implemented which encapsulates the loading and validation of an XML instance document. The semantic validator leverages the syntactic validation code to load its XML instance document during validation and also has a separate instance that helps construct its custom Schematron data-structure. Two clear obstacles for wide spread reusability and integration are the custom syntactic validator and the customer Schematron data-structure. Our point of view has been to use standardized popular XML technologies as part of our foundational boilerplate code. Creating custom solutions for

common operations such as XML syntax validation can prove to be difficult to propagate for wide spread use. A greater obstacle for wide spread use, in our opinion, is the coupling of the semantic validator with the custom syntax validator. Below we go into more detail on issues we feel limit the current parser's integration and adoptability potential.

2.3.1.1 Lack of Support for ISO Features

The existing parser aimed at partially supporting Schematron ISO features. The parser does not offer support for variable declarations or the use of abstract patterns. These are two of the most powerful additions to Schematron. Variables have an invaluable use case as lookup or reference fields. By reusing the value, a rule or pattern can be simplified. Referencing variables also has the side effect of allowing a rule to look at values beyond its context. When it comes to abstract patterns, they provide an invaluable tool for organizing and composing complex schemas. Adding support for the full ISO specification can only improve the reusability and adaptability of a parser.

2.3.1.2 Non-Standardized Processing of Schematron Documents

An important aspect of our approach taken when coming up with the design for our integrated parser was to leverage existing widely used technologies as much as possible. This is another point of divergence between our approach and that taken in the existing integrated parser. When it comes to representing an Schematron document in memory, the existing parser has developed a custom data-structure. The data-structure is a tree-based structure that follows the Schematron tree hierarchy. In essence, our approach ends with the same representation, the main difference is that in our approach we did not create a customized object hierarchy, what we did was leverage the DOM's

representation of the a Schematron document. Instead of managing collections of nodes, we manage XPath expressions that allow for the navigation of the Schematron tree-based representation.

2.3.1.3 Difficult Reusability as Middle-Ware

When taking into account the customizations built around the existing syntactic and semantic integrated parser, reusability might prove to be difficult. The parser constructs and maintains its own version of the XML instance document. There is not a way to pass in a pre-existing, pre-validated (syntactically) document for semantic validation. Although the document created by the custom syntactic validator is available via a getter method, assuming that other parts of a system might not have other uses for the DOM structure is far fetched. Another distinction between our parser and that already developed is the scope the solution is meant to fit in. To clarify, the current parser has a broader scope it has been built as an end-to-end solution to parsing. In our parser, we have taken a very granular approach. Our parser is meant to be used as a Schematron api inside a system. We do not aim to replace syntactic validators or to provide an end-to-end solution. We aim to be used as you would a DOM parser, as one more tool in your arsenal.

2.4 Other Solutions

As we have already touched upon, there are other solutions that have provided integrated parsers. However, the main distinctions between those solutions and the one we propose are the ability to optimize the use of syntactic validation and the ability to support co-constraints across all parts of an XML instance document. Other parsers allow for co-

constraint expression within the context of the element currently being syntactically validated.

2.4.1 Limited Expressiveness of Existing Solutions

SchemaPath, W3C Schema and RelaxNG support co-constraint expressions. RelaxNG even can integrate Schematron expressions into its schema definition. However, all of these parsers put constraints on the scope of co-constraint expressions. A co-constraint expression is limited by the context of the element being syntactically validated. **Figure 4** and **Figure 5** are examples of SchemaPath and W3C Schema co-constrain expressions. All of the co-constraint expressions are within the context of the surrounding element declaration. Schematron provides a liberated perspective at co-constraint rule definition. With feature such as variable definitions and abstract patterns, schema authors are liberated to seeing the document as a whole and not at an element-by-element basis.

2.4.2 Focus On Integration of Co-Constraint Definitions with Syntactic Definitions

Existing solutions tend to couple the syntactic definition with the semantic definition. Although you can argue a case for this approach being simpler and self contained, when it comes to scalability and change impact, having a single large document for both syntax and semantics can prove to be a performance and maintenance issue. By separating semantics from syntax, we allow schema authors to focus on the semantics of the document, and to update only the semantic expressions without affecting the syntactic definitions. Schematron ISO has provided tools to allow simpler document management and maintainability. With our parser, you can use any of your desired syntactic validators, with the added benefit of delegating all semantic co-constraint validations to Schematron.

2.5 Chapter Summary

In this chapter we have compared existing syntactic and semantic integrated parsers including one implemented by Steven Golikov [4] which is also an integrated validator supporting Schematron 1.5 and parts of the ISO Schematron. We have expressed the shortcomings of each of these validators. Other validators except for Steven's have combined syntactic definitions witch semantic ones. We have explained that by doing so, limitations as to the semantic expressions have been introduced. In these parsers semantic expressions are given the context of their parent elements. When it comes to comparing with Steven's parser, we have aimed at providing a validator that leveraged standard technologies such as the DOM and that separated syntactic from semantic validations. In the coming chapters we go into more detail on the workings the validator.

Chapter 3

A Simplified Design for Integrating Syntax/Semantic Validation Using Schematron

In order to achieve a simple and easy to use syntactic and semantic parser, we felt that it was imperative to add support for the new Schematron ISO feature set [28]. In order to support the new ISO features, we took an approach of macro-expanding a Schematron schema, giving us a complete flat structure. For example, when using the include functionality, an Schematron schema declares an *include* element that points to the document to include. Our validator will load this included file and append it to the Schematron schema's Document Object Model (DOM) representation. The contents of the *include* element is replaced by the contents in the external file. As part of this expansion, the validator makes sure that the contents of the external file is allowed to be included in the section where it was declared to be included. Continuing with leveraging the DOM representation of a schema, each component of an Schematron schema using during validation is encapsulated in a document fragment reader that leverages XPath to read and interpret the schema's data during validation. This decision allows us to avoid building custom data structures to represent the schema internally. In the coming sections we will go into further detail on the design and implementation of the validator. This section explains how the Schematron ISO features are supported.

3.1 ISO Schematron New Features for Semantic Constraint Specification

As previously mentioned, the ISO version of Schematron introduced very powerful features into the Schematron grammar. Features that simplify the maintenance of the

grammar by introducing the ability to declare and reuse grammar fragments via the ***include*** element. These fragments are complemented by the support for variable declarations. As can be imagined, supporting fragments implies the support for abstractions and inheritance. As such, prior to the ISO version, ***rule*** elements where the only ones allowed being abstract. With the ISO, ***pattern*** elements also support abstractions. Abstract patterns leverage a new element called ***param*** that is how concrete instances have a context to execute upon. Another great new addition, something present in the original Schematron XSTL based parser, the support for variables. The ***let*** element was introduced to allow a parser to share state during the validation process by treating all ***let*** declarations as variables. Below, I will go into further detail on each of these new features.

3.1.1 Support for Abstract Patterns

Managing a set of complex rules can be very challenging. At its core, as stated on the Schematron website, Schematron is "a language for making assertions about patterns found in XML documents"[29]. Prior to the ISO version of Schematron, the only element allowed to be abstract was the ***rule*** element. However, being that Schematron is focused on asserting patterns in a document, having only rules be abstract meant duplicating patterns or pattern definitions. With the ability to predefine a pattern, organizing, maintaining and scaling Schematron rules is much simpler. It might help to think about it as storing a function for later use. The function itself does not have any state it only defines actions. Rules can be thought up of actions to be taken on an XML pattern.

Having the ability to predefine patterns for later inclusion would not make much of an impression if we could not extract those abstractions into separate files or modules. In the next section we discuss the new ***include*** element. This element is the missing link to making Schematron ISO fully modular.

3.1.2 Including Schema Fragments Via Include

Include elements are a powerful dynamic construct when it comes to traditional XML validation. The idea of dynamically updating reusable semantic rule fragments during validation is quite innovative and popular on the web [19]. Normally one can think of XML validation as a very static process. From a traditional pure validation sense, static syntax rules are what govern XML validation. Rules that focus on cardinality, default value initialization, data type validation and markup sequence. From a semantic perspective, we have dealt with abilities to define static co-constraints. These co-constraints are pre-determined with hardcoded values. Prior to the ISO version of Schematron, users did not have the ability to modularize their rule definitions and executions. A parser had the ability to execute different patterns based on data values, but the rules that were activated as part of a pattern had hardcoded values to interpret. With include elements; this is no longer a limitation.

Include elements allow for the modularization of code (XML semantic rules) and execution. A simple yet accurate analogy is that of a function or method in the Java language, or any functional language that achieves a modular design for an application [31]. Large complex programs benefit from modularity by having methods contain a granular focused piece of code. When the methods are combined during execution, what the program does becomes apparent as a result of its execution. In the same fashion,

include elements allow for the externalization of complex rules, rules that can evolve separately without affecting the main file where the sequence of rule executions is defined. Included fragments can contain patterns (abstract or not), phases, rules, report, assert, let and diagnostic elements. In essence anything executable can be part of an included fragment. The only rule is that the markup within the fragment follows the syntactic rules set by the parent element owning the fragment.

Include elements are a significant improvement to how rule implementers think about Schematron documents. The approach at organizing a document assimilates more towards functions or methods. But, this is only half the picture. The other half is the ability to allow the fragments to dynamically change the data values used during validation. This is possible because of the ***let*** element introduced in the ISO version of Schematron. Previous XSLT based parsers supported variables through XSLT extensions, in the ISO version, variable use and declaration has been integrated into the markup.

3.1.3 Adding Support for Dynamic Variables Using Let

As previously explained, ***let*** elements are used to declare variables used during co-constraint validation. The let element was introduced in the ISO version of Schematron. Using the let element, variables can be declared for different contexts. If declared as a direct descendant of the Schematron root element, then the context of the variable is global. As such, it can be referenced from any part of the document. Variable references are identified by dollar sign prefixes. A variable can be referenced by itself or as part of an XPath expression.

As mentioned, a variable can belong to different contexts. The contexts are, global, at the pattern, phase and rule levels. The variable is visible and available for the descendants of the root element in the context where the let element has been nested. Later on we will dive into detail on how the resolution and visibility of a variable is managed.

3.1.4 Abstract Constraint Definitions

As with modern object oriented programming (OOP) languages, Schematron has introduced tools that allow developers to encapsulate, modularize and reuse common functionality [32]. Unlike OOP, we feel that Schematron takes a more functional perspective on what abstractions are. In Schematron, the focus is not on creating is-a relationships, but more about can-do relationships. By can-do we mean that a schema can validate or support actions defined by reusable rules. For instance, in our order management system example, we can extract into a fragment the ability to validate order state. Any schema that imports the fragment can be thought of as can-do order state validation. Remember that a fragment declares abstractions that can be leveraged by other concrete validation documents. Further on in this chapter we will look at this example in more detail.

3.1.5 Using Abstractions to Simplify Schematron Documents

As discussed above, abstractions can encapsulate can-do relationships to be reused by other validation documents. In this section we look at an example in detail as we explain the different levels of reuse and abstraction that can be accomplished with Schematron. At the end of this chapter, we will have explained how powerful the ability to modularize

can-do functions can be. They will become building blocks for more complex schema definitions.

```
<sch:pattern abstract="true" id="shippingDateValidation"
             xmlns:sch="http://purl.oclc.org/dsdl/schematron">
    <sch:let name="currentDateTime" value="current-dateTime()"/>
    <sch:rule context="$dateTime">
        <sch:assert
                test="(hours-from-dateTime($currentDateTime)
                    - hours-from-dateTime($dateTime)) > $hourMax">
            The time the order was placed is too far in the past.
        </sch:assert>
    </sch:rule>
</sch:pattern>
```

Figure 8 Date Validation Function

Figure 8 displays a document fragment of a reusable date range calculation. The fragment is used by incorporating it into a Schematron schema using the include tag. To put it in terms, a schema that includes the fragment can-do date validations. What it is validating is that the time passed between a dated event marked by the *$dateTime* parameter, and now is not greater than the value in parameter *$hourMax*. Taking the ordering system example discussed in previous chapters, we would say that an order that has not been processed for more than an amount of hours, should fail validation.

3.1.6 Defining Reusable Abstract Patterns and Rules

In the previous section we saw an example of a functional use of an abstract pattern. By functional we mean a reusable somewhat generic piece of validation logic. In comparison to any other schema definition language, this ability is unique to Schematron. In Listing 4-1 we can see a clear example of the can-do relationship Schematron features can take on. In contrast to W3C Schema or any other is-a relationship definition language [33], any validation rule that needed to provide date time range validation could

do so by just calling the pattern. But, what is missing in Schematron would be a simple way to guarantee that the data being validated by the abstract pattern is-a date-time type. This is-a relationship needs to go hand in hand with the can-do relationship. This is the whole idea behind the joined semantic and syntactic validator.

3.1.7 Componentizing Schematron Documents Via the Include Element

In the previous section we touched upon the ability to reuse abstract pattern and rule definitions by leveraging the ability to include document fragments into an Schematron schema definition. The ability to include fragments has been also leveraged by other schema definition languages because of the benefit of simplifying and isolating instance specific schemata. **Figure 9** and **Figure 10** are a comparison of the difference between the markup needed to write rules that can be defined and included in a fragment with the much simpler markup of including and using those rules.

```
<sch:pattern abstract="true" id="customerValidation"
             xmlns:sch="http://purl.oclc.org/dsdl/schematron">
    <sch:rule context="$orderContext">
        <sch:assert test="@quantity > 0">
            Orders must have a quantity
        </sch:assert>
    </sch:rule>
    <sch:rule context="$customerContext">
        <sch:report test="$isAMember and *">
            A member should not contain any additional elements.
        </sch:report>
        <sch:assert test="$isAMember and $memberId">
            If the customer is a member it must contain a memberId.
        </sch:assert>
    </sch:rule>
    <sch:rule context="$nonMemberCustomerContext">
        <sch:assert test="child::contact and child::payment">
            If the customer is not a member it must
            contain a contact and paymentchild elements.
        </sch:assert>
    </sch:rule>
</sch:pattern>

<sch:pattern abstract="true" id="shippingDateValidation"
             xmlns:sch="http://purl.oclc.org/dsdl/schematron">
    <sch:let name="currentDateTime" value="current-dateTime()"/>
    <sch:rule context="$dateTime">
        <sch:assert
                test="(hours-from-dateTime($currentDateTime)
                    - hours-from-dateTime($dateTime)) > $hourMax">
            The time the order was placed is too far in the past.
        </sch:assert>
    </sch:rule>
</sch:pattern>
```

Figure 9 Expanded Rules

Below is a much simpler and concise form at expressing the same set of rules but leveraging the *include* element. It is very clear that the contents of **Figure 10** is preferred when reusing the same rule definitions than having to repeat what is in **Figure 9** every time these rules need to be reused.

```
<sch:include href="validation-fragment.sch"/>
<sch:include href="date-validation-fragment.sch"/>

<sch:pattern id="fullValidation" is-a="customerValidation">
    <sch:param name="orderContext" value="order"/>
    <sch:param name="customerContext" value="/order/customer"/>
    <sch:param name="isAMember" value="@isAMember"/>
    <sch:param name="memberId" value="@memberId"/>
</sch:pattern>

<sch:pattern id="orderDateValidator" is-a="shippingDateValidation">
    <sch:param name="dateTime" value="order/@createdTime"/>
    <sch:param name="hourMax" value="2"/>
</sch:pattern>
```

Figure 10 Include Elements

3.1.8 Using Schematron Fragments Containing Abstractions

As shown above, there is a huge benefit in terms of code maintainability and simplicity when using fragments. The idea of can-do relationships can be extended to document fragments. Schema authors can organize executable, reusable behavior into document fragments that can serve as building blocks for more complex schema definitions. Ideally a schema definition should focus on rules that are too specific to the structure of the XML message being validated to be reusable.

3.1.9 The Use of Dynamic Variables

Something that was only possible via the XSLT processor has now been added to the Schematron syntax. The ***let*** declaration allows the processor to read in dynamically assigned variables [23]. In listing 3-4 we can see a great use of this ability. In order for the time window validation to take place, the rule needs to know the current date-time. By leveraging the ***let*** element, the rule asserting the time window will always grab a snapshot of the current date-time. This is in contrast to a default declaration in W3C Schema definitions where default values can be assigned to attributes making them always available unless overridden. Dynamic variables give rules the needed context to stay relevant across different XML instances of the same messages.

3.2 Macro-Expansion Based Algorithm

The parser leverages the Document Object Model (DOM) as its data structure to represent a Schematron schema. To simplify and optimize the validation stage, we have used a macro-expansion algorithm that include into the original DOM structure the

concrete values for abstract *patterns*, *rules* and also the externally defined contents of an *include* element.

To optimize the pre-processing phase where we use macro-expansion, the parser incorporates all of the reference data pointed to by abstractions, variables and includes during the initial startup phase. Once incorporated, the DOM will contain the entire necessary markup needed for validation.

The following sections go into further detail on the implementation of the parser. The goal is to identify the parts of the parser that have made it successful in the parsing and validation of Schematron schemas conforming to the ISO version of Schematron.

3.2.1 Benefits of Using Macro Expansions Versus XML Document Transformations

With macro-expansion, the parser is able to pre-populate schema elements by including referenced markup. For instance, in the case of and ***include*** element, the macro expansion phase will pre-load the included fragment into the active document. During validation, the parser will not have to reference an external file to read in an external fragment; instead the fragment is added to the active DOM tree representing the full Schematron schema. As shown in **Figure 11**, the include fragment is entirely incorporated into the Schematron in-memory representation. From the parser's perspective, the DOM always was a single document.

```
// Before pre-processing
<sch:let name="freeShippingAmountMinimum" value="100"/>
<sch:let name="hasFreeShipping"
        value="child::customer/@isAMember and order/@total >
            $freeShippingAmountMinimum"/>
<sch:include href="date-validation-fragment.sch"/>

// After pre-processing
<sch:let name="freeShippingAmountMinimum" value="100"/>
<sch:let name="hasFreeShipping"
        value="child::customer/@isAMember and order/@total >
            $freeShippingAmountMinimum"/>
    <!-- This is the schematron fragment to include -->
<sch:pattern abstract="true" id="shippingDateValidation"
             xmlns:sch="http://purl.oclc.org/dsdl/schematron">
    <sch:let name="currentDateTime" value="current-dateTime()"/>
    <sch:rule context="$dateTime">
        <sch:assert
                test="(hours-from-dateTime($currentDateTime) -
                hours-from-dateTime($dateTime)) > $hourMax">
            The time the order was placed is too far in the past.
        </sch:assert>
    </sch:rule>
</sch:pattern>
```

Figure 11 Include feature examples before and after applying the pre-processor.

3.2.2 Macro-Expansion support for include Schematron elements

As demonstrated in **Figure 11**, the macro-expanded DOM contains the content of the externally defined Schematron grammar. In **Figure 12** we see a basic tree diagram showing what happens when we pre-process the contents of an include element.

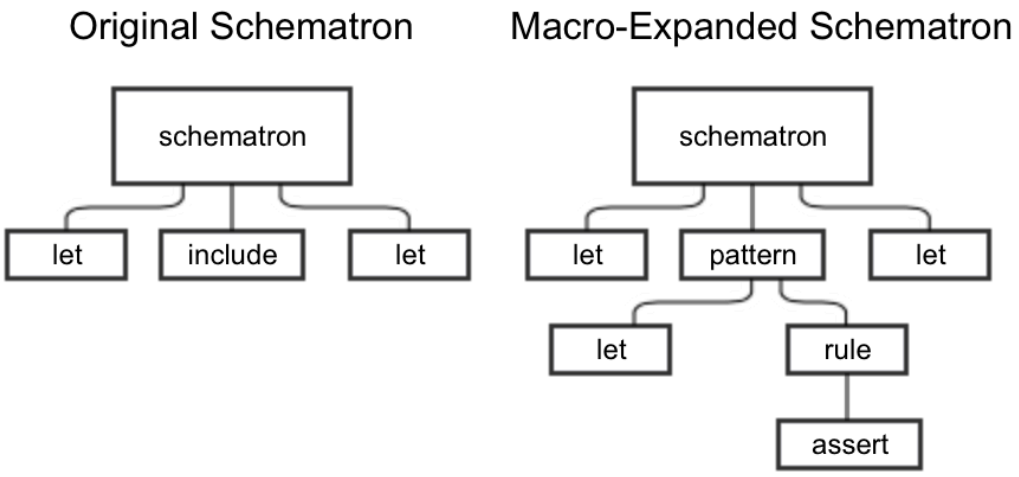


Figure 12 DOM Macro-Expansion for *insert* elements

The initial document contains the declaration of the *include* element. The pre-processor reads the location of the DOM fragment from the element's *href* attribute and loads it into a separate DOM tree instance. The next step is the macro-expansion step. The pre-processor will get a handle to the *include* element and insert the *pattern* element found in the content to include as its child. In **Figure 12**, the right part is what the new Schematron DOM looks like.

3.2.3 Macro-Expansion support of abstract elements

Abstractions are key when it comes to maintainability and reusability of Schematron grammar. *Pattern* and *rule* abstraction declarations can be found in the original Schematron schema or in a fragment loaded into the processor via an *include* declaration. What the pre-processor does when it finds an abstract declaration is to look for all variable pointers, recognizable by a dollar sign ($) that precedes its name. Once these variables have been found, it searches for extending elements and replaces the $ value with the value declared as a parameter by the extending child. In **Figure 13** we can see the macro-expansion process of an abstract *pattern*.

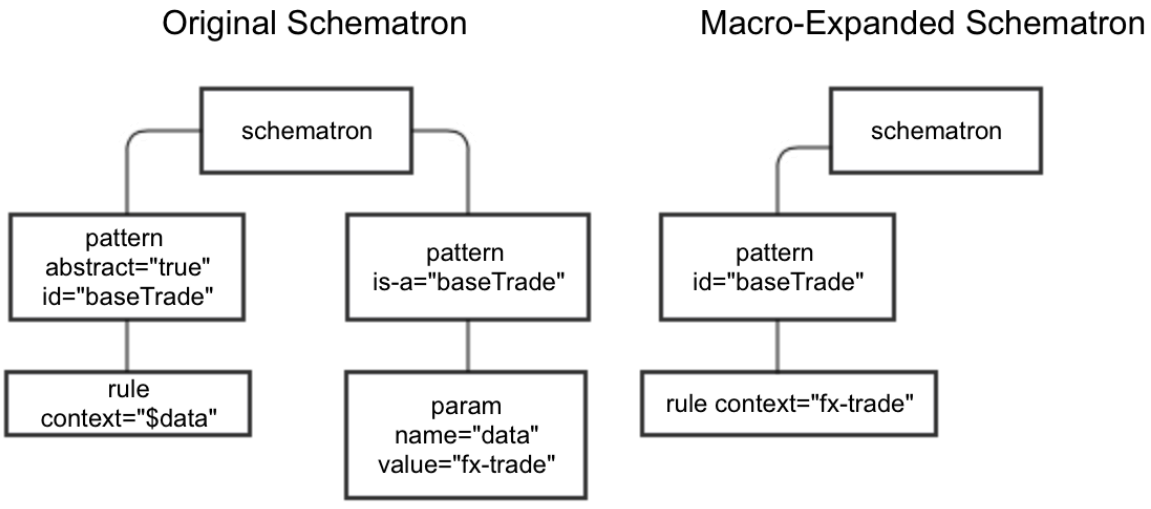


Figure 13 Macro-Expansion of an abstract *pattern*

In the image above, the abstract *pattern* declares a parameter for its only rule named $data. The extending *pattern* declares a parameter with the value fx-trade. After the macro-expansion process, the original *pattern* element has its parameter value replaced by fx-trade. The post pre-processing phase DOM tree no longer has an extending element and the abstract element has been expanded to include the parameter value.

For all of the parser's interaction with the DOM, specially during the pre-processing phase, we use an XPath based querying mechanism that simplifies and standardizes how we interact with Schematron during the parser's lifetime. As is apparent, the parser does numerous DOM manipulations that require us to quickly and efficiently find the node that needs to be pre-processed. As will be explained later, the parser uses and utility class that further simplifies the interaction between XPath and the DOM.

3.3 Using XPath as the mechanism to access Schematron values

After the pre-processing phase, the parser creates a series of XPath based accessors to the different parts of the Schematron schema. These accessors or readers encapsulate XPath expressions and a context node from which the expressions are mapped. The role of a reader is to expose access to XPath querying capabilities. The context node is the Schematron element the parser holds a reference to. The query's passed in are all relative XPath queries, relative to the context node of the reader. In addition, there us an XPath expression factory class that encapsulates the building of these XPath queries. This factory takes in the values used in the query and returns the XPath query. The classes are, the reader is the SchemaDocumentFragmentReader class and the factory is the

SchematronISOXPathSupport class. These classes are the heart of the parser as they are used to navigate the Schematron schema during validation.

3.3.1 Identify the Mapping Rules for ISO Schematron New Features

As mentioned above, at its core, the parser leverages an XPath expression builder that allows for the access of Schematron elements during validation. The base class for a builder is the SchematronXPathSupport class. By default, the class provides methods for building XPath expressions used for mapping Schematron core elements. A core element is defined as any element that remained unchanged since Schematron 1.5. Although not a goal, this parser should be backwards compatible with Schematron 1.5. For future expansion, the class defines four abstract methods, getNsResolver, getNamespaceURI, getNamespacePrefix and getNodeNameToInterpreter. The first three methods provide access to Schematron's namespace associated with the given implementation version. The last method provides a mapping to an interface that can be used to encapsulate any custom behavior defined by Schematron elements. An example would be the optional value-of attribute for the ***name*** element. A ***name*** element's value-of attribute is optional; the presence of the attributes will eventually change the behavior of the name element.

When extending the parser to a new version of Schematron, the base SchematronXPathSupport needs to be extended to support not only the new namespace definitions, but also any new elements introduced into the Schematron syntax. For the ISO compliant version, we have created a class SchematronISOXPathSupport that defines the ISO specific XPath expressions and namespace definitions. In addition to the base Schematron elements, the ISO version introduces two new methods that build the expressions mapping the new Schematron elements introduced in the ISO version. The

methods are getAbsoluteIncludeNodesOffTheRootSchemaNode and getRelativeLetElementNodesXPath. The first method provides an XPath expression used access ***include*** elements while the second provides a relative expression used to access ***let*** elements. In the following chapters, we will detail how these expressions are used during validation.

3.3.2 Access the In-Memory Expanded Schema Using XPath, a Standardized XML Query Language

Prior to the start of the validation phase for the parser, there is a normalization stage that prepares the Schematron instance document. This phase will pre-process any include, abstract pattern and abstract rule elements by expanding them into the in-memory DOM representation of the Schematron document. After the macro expansion, XPath expressions can now be used to access the normalized expanded version of the Schematron document. The macro expansion phase will insert the included or abstract XML fragment into the main DOM tree as show in **Figure 14**.

```
// Before pre-processing
<sch:let name="freeShippingAmountMinimum" value="100"/>
<sch:let name="hasFreeShipping"
        value="child::customer/@isAMember and order/@total >
            $freeShippingAmountMinimum"/>
<sch:include href="date-validation-fragment.sch"/>

// After pre-processing
<sch:let name="freeShippingAmountMinimum" value="100"/>
<sch:let name="hasFreeShipping"
        value="child::customer/@isAMember and order/@total >
            $freeShippingAmountMinimum"/>
    <!-- This is the schematron fragment to include -->
<sch:pattern abstract="true" id="shippingDateValidation"
             xmlns:sch="http://purl.oclc.org/dsdl/schematron">
    <sch:let name="currentDateTime" value="current-dateTime()"/>
    <sch:rule context="$dateTime">
        <sch:assert
                test="(hours-from-dateTime($currentDateTime) -
                hours-from-dateTime($dateTime)) > $hourMax">
            The time the order was placed is too far in the past.
        </sch:assert>
    </sch:rule>
</sch:pattern>
```

Figure 14 Before and After Processing of an Schematron Fragment

3.4 Next Generation of Integrated Syntactic and Semantic Parser

By leveraging a modular design [35] and popular standardized technologies we have implemented a very lightweight semantic validating parser. **Figure 15** is conceptual diagram of the different high-level modules in the parser. The parser is packaged into a library that is easily integrated into a pre-existing semantic validating parser. What we have done is to separate the syntactic from the semantic phases of validation. Our goal is not to replace existing battle tested syntactic validators; instead we aim to integrate with such validators. This decouples us from a user's XML schema definition of choice. The integration is as simple as passing a post-syntactic validation reference of the document node to the semantic validator. The only constraint at this moment is the use of a DOM implementation for as the wrappers around the XML tree. The semantic validator leverages the DOM for its navigation and modification capabilities. The queries are incorporated into the Schematron document itself in the form of XPath queries.

As discussed, the parser makes heavy use of the DOM and XPath expressions. Avoiding custom in-memory representations of Schematron was possible by the use of the DOM. Also, using the DOM allowed us to avoid building a custom solution to access Schematron elements during validation. XPath and the parser's expression builder module handle all of this access.

Previous syntactic and semantic parsers have integrated the two steps into a single point of entry. An important differentiating factor that distinguishes us from other integrated parsers is the decoupling of XML grammar definition language from the parser's

syntactic phase. For instance, SchemPath, an integrated parser, and by integrated I mean a parser that does both syntactic and semantic validations, is based on an extension to the W3C Schema grammar. The extension is based on an element to add conditional constraints or expressions and a new type to identify exception states [23]. The element is the ***alt*** element and the type is xsd:error. This integrated parser adds its semantic support by extending the W3C Schema and hence coupling the semantic portion of a validation to the syntactic grammar definition. Another integrated parser is Steven Golikov's [4] implementation. This parser encapsulated the syntactic phase resulting in a coupled syntactic parser that forces the user to abide by the version of the grammar supported by the parser. Our implementation adds flexibility and scalability by focusing on semantic validation leaving hooks for syntactic validation process integration. As our research is based off of Steven's, we will mainly focus on describing improvements to that research introduced by our implementation of an integrated parser, starting with the fact that our parser has a potential for integration rather than coupled to a specific syntactic validation implementation.

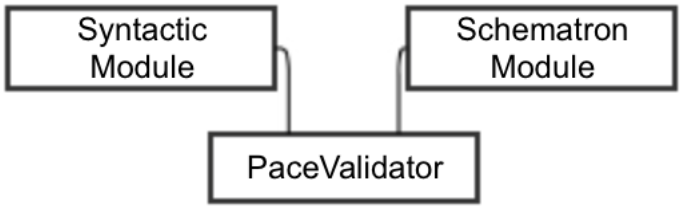


Figure 15 Modular Concept

3.4.1 Clean Modular Design of an Integrated Parser

Our parser is composed on a set of collaborating modules that facilitate integration with pre-existing syntactic parsers and also allows for extensibility by adding support for

future versions of Schematron. Modules represent different parts of the application programming interface (API) that each deal with granular behavior. They are classes that work together to provide behavior specific to a function of the parser. The granular functions are macro-expansion and XPath expression support.

Although not explicit, integration of the semantic validation workflow with a syntactic workflow is very simple. The parser provides a validate method which takes as a parameter a DOM document representing the XML instance being validated. When supporting syntactic validation, the implementer should syntactically validate the XML instance prior to passing it to the semantic parser.

3.4.2 Extending the scope of new features in ISO Schematron

Adding support for the ISO features in an integrated parser broadens the usability of the parser. The include capabilities allow for the separation of business specific rules or fragments from a base document validating set of rules. In addition, because of its ease of integration, this parser can easily become part of an existing mature XML based messaging system. The use of standards and macro-expansion also has a positive performance impact which enables the possibility of this parser to be used in a real-time dynamic system. Further in this document we will dive into details on the implementation.

3.5 Advantages of Proposed Solution

Our proposed solution has a number of advantages over the existing solution. We have focused on providing modularity around co-constraint validations hence allowing users to control the selection of syntactic parsers. With modularity in mind, full support for the

ISO Schematron standard has been implemented. With the ISO version, Schematron now supports inheritance, a feature that maximizes reusability and maintainability. Our parser has also been built with integration into existing systems in mind. A call to our parser takes an existing DOM representation of the document to validate. We do not reload the document for validation. Also, this new parser supports multiple validations on the same instance of the Schematron parser. Below we go into more details on each of our advantages.

3.5.1 Separation Of Syntactic And Semantic Definitions

Syntactic definitions can at times, for good reasons, get complex and extensive. As with computer programming, complex large pieces of code written in a procedural manner can be extremely difficult to update and maintain. We can think of a syntactic XML definitions as following procedural style of programming because both have a waterfall approach. In procedural programming style a programmer defines functionality in a set of procedures that carry out some behavior. In syntactic definitions, we define XML metadata that has a single purpose of describing the contents of an XML element. The parser then interprets the metadata and validates the XML document following the same order of steps defined in the metadata. With procedural programming languages, it is highly recommended to keep modularity in mind as to avoid coding complex rigid procedures. Following this mindset, we felt it would give syntax authors more flexibility if we separated the semantic declarations from semantic ones. In the end, both do provide metadata definitions, but for completely different purposes. To further stress the importance of separating syntax from semantics, we will look at an example where co-constraints that can be expressed in Schematron cannot be expressed in the other

integrated XML definition languages such as the W3C Schema, Schemapath or RelaxNG. **Figure 16** depicts a hypothetical sales report containing the orders and customers that made those orders. For the sake of not duplicating customer information, the order definitions and customer definitions have been separated in to sibling elements. Orders have a customerId attribute that links the order to the customer definition.

```
<sales-report>
    <orders>
        <order totalCost="100" shippingCost="14.34" customerId="123" >
            <item id="item11" price="50" qty="1"/>
            <item id="item22" price="50" qty="1"/>
        </order>
    </orders>
    <customers>
        <customer id="123" isAMember="false">
            <address>123 Elm St.</address>
            <city>New Hampshire</city>
            <postal-code>34587</postal-code>
            <email>john.doe@nowhere.com</email>
            <payment type="amex" number="xxxxxxxxxx" secId="4532"/>
        </customer>
    </customers>
</sales-report>
```

Figure 16 Sales Reports XML

To follow previous examples, we want the ability to simply introduce rules that semantically change the report. We want to validate that if a customer is a member and has an order total cost at one hundred or more dollars, then the sipping costs should be zero. With other integrated validators, expressing this rule is not possible. For other integrated validators, co-occurrence expressions are defined within the context of a parent element. In our example depicted in **Figure 16**, the customer element is a sibling of the order element. Because of the context constraint, we cannot skip forward in our document to check the validity of the shippingCost attribute. In **Figure 17** we can see a Schema for the document above. As you can see, the definition is a little verbose. We do

not argue the necessity or usefulness of a Schema, what we argue is the need to add co-constraints to them. With Schematron, expression the co-constraint that validates the shipping cost of members is trivial.

```
<xsd:element name="sales-report" type="ex:reportType"/>
    <xsd:complexType name="reportType">
        <xsd:sequence>
            <xsd:element name="orders" type="ex:ordersType"
                    minOccurs="1" maxOccurs="1"/>
            <xsd:element name="customers" type="ex:customersType"
                    minOccurs="1" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="ordersType">
        <xsd:sequence>
            <xsd:element name="order">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="item" minOccurs="1">
                            <xsd:complexType>
                                <xsd:attribute name="id" type="xsd:string"/>
                                <xsd:attribute name="price" type="xsd:double"/>
                                <xsd:attribute name="qty" type="xsd:integer"/>
                            </xsd:complexType>
                        </xsd:element>
                    </xsd:sequence>
                    <xsd:attribute name="totalCost" type="xsd:double"/>
                    <xsd:attribute name="shippingCost" type="xsd:double"/>
                    <xsd:attribute name="customerId" type="xsd:string"/>
                </xsd:complexType>
            </xsd:element>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="customersType" >
        <xsd:sequence>
            <xsd:element name="address" minOccurs="1" maxOccurs="1"/>
            <xsd:element name="city" minOccurs="1" maxOccurs="1"/>
            <xsd:element name="postal-code" minOccurs="1" maxOccurs="1"/>
            <xsd:element name="email" minOccurs="1" maxOccurs="1"/>
            <xsd:element name="payment" minOccurs="1" maxOccurs="1">
                <xsd:complexType>
                    <xsd:attribute name="type" type="xsd:string"/>
                    <xsd:attribute name="number" type="xsd:string"/>
                    <xsd:attribute name="secId" type="xsd:integer"/>
                </xsd:complexType>
            </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:string"/>
        <xsd:attribute name="isAMember" type="xsd:boolean"/>
    </xsd:complexType>
```

Figure 17 W3C Schema Definitions For The Sales Report

Although the schema is somewhat simple, it still has details necessary to ensure the syntactic validity of a sales report. In **Figure 18** we can see a Schematron document expressing the rule checking shipping costs. I would like to point out how simple and

focused the Schematron document is. This modularity and simplicity is key for our version of an integrated parser.

```
<sch:let name="freeShippingAmountMinimum" value="100"/>
<sch:pattern id="shippingCostSpecial">
    <sch:rule context="//orders">
        <sch:assert
                test="count( order[ @customerId = //customer[ @isAMember='true' ]/@id
                                    and @shippingCost > $freeShippingAmountMinimum
                                    and @totalCost >= 100.0 ] ) = 0.0" >
            The report contains incorrect shipping costs.
            Members are being charged for closing costs.
        </sch:assert>
    </sch:rule>
</sch:pattern>
```

Figure 18 Schematron Rule Definitions

A single XPath expression spanning across siblings can check for the shipping cost constraint for members. The concise small Schematron document is also a great example of modularity. If this rule changes, there is not need to scan the whole syntax Schema but only the smaller Schematron definition. The figure also demonstrates the use of variables, an ISO Schematron feature. Variables are a straightforward way to encapsulate dynamic values. As a contrast to the existing integrated Schematron validator, this new validator adds full ISO support and also decouples from syntactic validation giving users the freedom to choose the definition language of choice.

3.5.2 New ISO Schematron Features And Their Applications

As touched upon in the last section, our validator fully supports Schematron's ISO feature set. We have implemented these new features following a standardized approach. We have leveraged the Document Object Model (DOM) and XML Namespaces to support variable declarations and element abstractions. In chapter three, we will detail the design approach of our parser.

As we have previously mentioned but would like to stress, the ISO Schematron features enable users to divide and conquer complex Schemas. The use of variables as parameters to abstract patterns helps eliminate boilerplate code that might need to be duplicated had there not been a way to natively support dynamic content. Adding the include feature, Schematron segments can be physically separated and logically organized. Our parser uses the concept of micro-expansion to pre-parse and create the necessary data structures prior to beginning validation.

3.5.3 Designed with Reusability In Mind

By focusing on the lowest level of granularity, our parser can be used in various environments. We have developed components that can be instantiated and used within different parts of a system. When it comes designing the core parts of the parser, we have encapsulated the base behavior of each component in an effort to maximize customization, reusability and scalability. In chapter three we explain the strengths of our design and implementation.

3.6 Interacting with the parser

We have provided a series of unit tests that can be executed to see how the different stages of the parser are executed. The parser has been configured as an Apache Maven project. In the test resources folder, you will a large number of Schematron and XML documents that can be used to test the parser. For specific ISO test cases, you can look at the test files under the ISOVersion folder. In order to run external files against the parser's default main edu.pace.PaceValidator, from the command line, you need to pass

four program properties, instance, syntax, semantic and baseuri. Below is an example of the command to execute.

```
java -cp dps-parser-0.9-SchematronParser.jar edu.pace.PaceValidator \
   -instance FXTradesBad.xml \
   -syntax FXTradeSchema.xsd \
   -semantic SmartTradeRouter.sch \
   -baseuri <Parser Home Directory>/ISOVersion/includenode/
```

Listing 1 Command Line Argument

Building the parser using Maven allows us to include all necessary dependencies within the final artifact dp-parser-0.9-SchematronParser.jar. The main class to call is the edu.pace.PaceValidator.

3.6.1 Instance Property

This parameter points to the XML message file. This is the data that needs to be validated against a syntactic definition and a Schematron schema. Validating the instance document syntactically is optional.

3.6.2 Syntax Property

An optional property that points the parser to the syntactic definition document. For this implementation of the parser, it needs to be a W3C schema document.

3.6.3 Semantic Property

A required property that points to the Schematron schema location. As will be explained in the next section, any *include* elements need to have either relative or absolute

Universal Resource Identifiers (URI). The absence or presence of the baseuri property determines if the URI is absolute or relative.

3.6.4 Baseuri Property

This parameter determines whether the paths to the previous properties are absolute or relative. It also extends to the location of a fragment declared inside the Schematron schema. For the parser to be able to load all files, it is preferably to keep all files in the same directory and configure that directory's absolute path using the baseuri property. This way all other properties, including any *include* elements in the Schematron schema, can be in a relative path to the baseuri.

3.7 Chapter Summary

In this chapter we put together the ISO features that make Schematron extremely powerful. Supporting abstractions of rules and patterns through document fragments is a huge benefit when it comes to schema simplification and scalability. On the same note, let elements allow schema definitions to declare real-time variable pointers. These features begin to morph Schematron into a sort of meta-programming language where you have dynamically assigned content and reusable function calls. We feel these features are stepping stone into rethinking how XML semantics are used and defined.

Chapter 4

Adding Support for ISO Schematron Features

As previously mentioned, an important contribution of our parser implementation is the support for new Schematron ISO features. In our opinion, the most important addition presented by the ISO version of Schematron is the ability to support abstractions and modularity at different levels. A close second is the ability to dynamically assign values to variables. This allows semantic rules to be fully dynamic, something that was only accomplished by the use of XSLT variables in previous versions. In this chapter we go into detail on the design and implementation of the parser.

4.1 Parser Design Overview

In **Figure 19** we can see an overview of the core components used during initialization. The initial stages of the parser are focused around the pre-processor stage. The SchematronPreProcessor is responsible for the macro-expansion phase prior to beginning validation. Further in this chapter we go into further details on the pre-processor. For now, we just need to understand that the pre-processor leverages the SchematronXPathSupport object, an object that encapsulates XPath queries specific to the Schematron document. These queries allow the pre-processor to get access to the relevant DOM nodes for expansion. Once the expansion phase is complete, a list of SchematronFragementReader classes get instantiated. Each reader represents the entry point into an active pattern. The design principle behind the multiple SchematronFragmentReader classes comes from the delegation pattern. It is an

alternative to inheritance as the validator delegates the searching of the DOM to each reader [36].

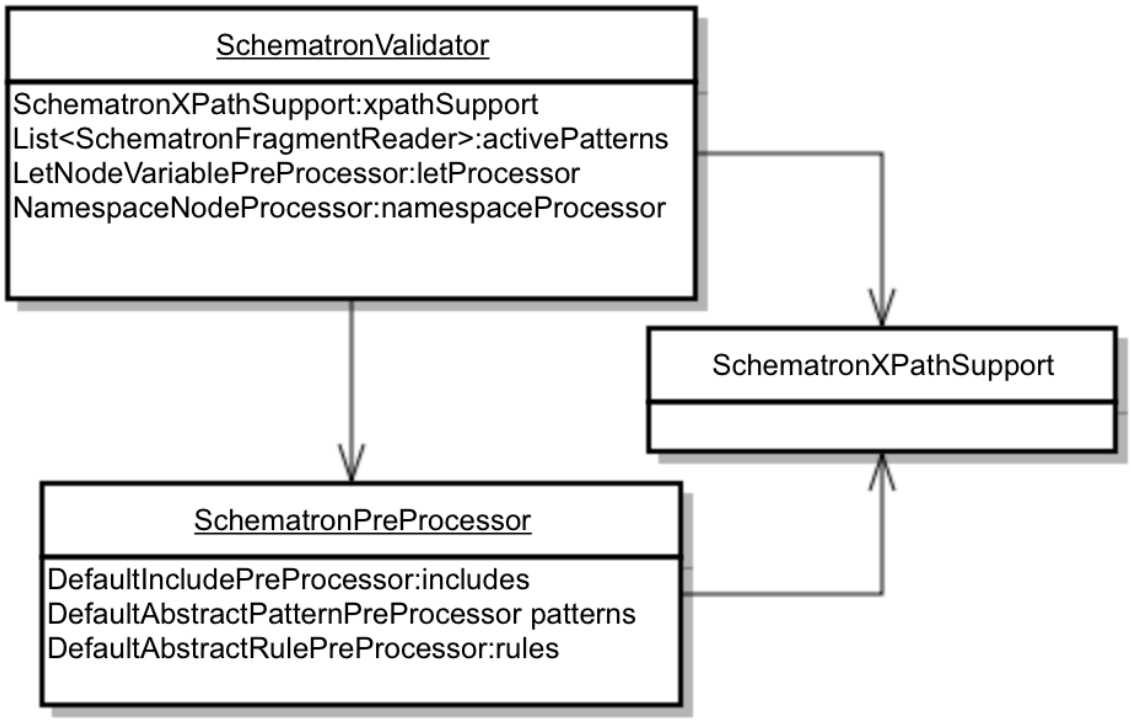


Figure 19 Object Diagram of the SchematronValidator

When validation begins, the parser starts iterating through its collection of fragment readers. A reader provides access to the Schematron nodes where the rules are, defined.

The parser interprets the rules defined and uses its XPath compiler to execute the rules against the incoming XML message. In **Figure 20** we see the relationship between the validator and the reader. The validator delegates access to the Schematron rules and expressions to the fragment reader.

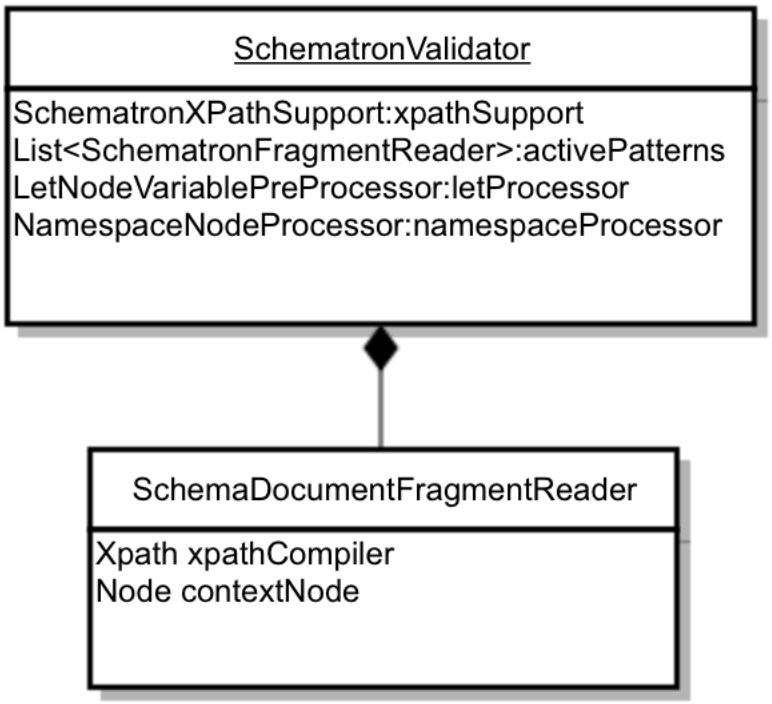


Figure 20 Fragment Reader

4.2 Validator Implementation Details On the Initialization Phase

A key component of the parser is how it leverages the DOM during the validation phase. During initialization, mappings are created and maintained that serve as pointers to sections of the DOM that contain the relevant schema definitions fragments. While validation is in progress, these mappings are leveraged to retrieve the relevant semantic expressions so that they can be evaluated against the active XML instance document. This chapter begins with an explanation of the validator initialization flow. The construction of the parser creates pointers to the active patterns that need to be validated. Once these pointers to the active patterns have been created, then the parser is capable of executing any pre-processing steps. The sections below go into further detail o the flow described in **Figure 21**.

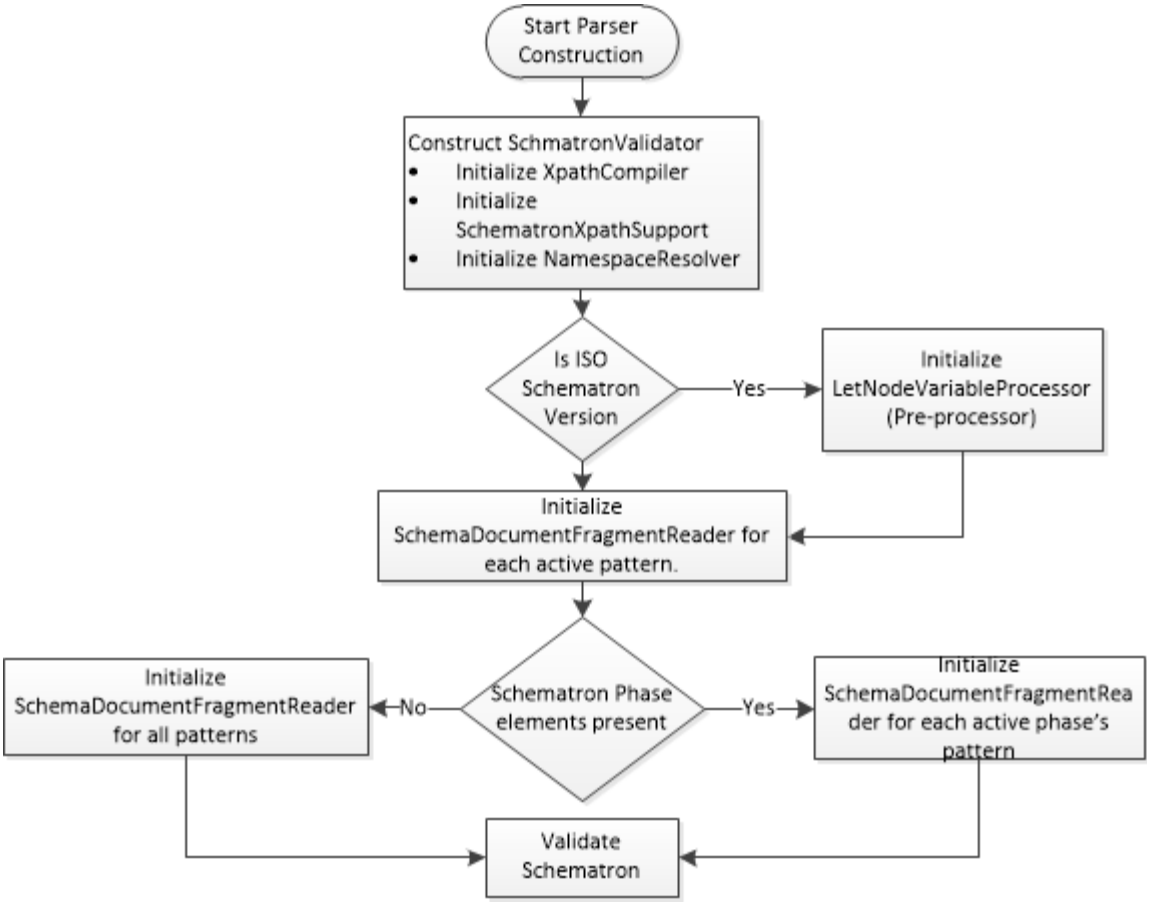


Figure 21 Validator Initialization Flow

4.2.1 Identifying Schema Mappings

Each Schematron element used during validation is mapped to an instance of SchemaDocumentFragmentReader (fragment reader). **Figure 21** shows a flow diagram [37] of the initialization phase, this is where the fragment readers for the active patterns are created. All of the necessary XPath and Namespace support object instances are leveraged by the fragment reader. In the diagram, we also see the initialization of a pre-processor needed to support the *let* Schematron ISO feature. The fragment readers that get instantiated are pointers to the patterns defined in the Schematron schema the validator will use. The patterns can be identified by either the configuration of an active

phase element or by just the definition of said elements. The purpose of the fragment readers is to encapsulate access to Schematron elements during validation. Instead of having a custom class instance and interaction for each Schematron element, a fragment reader can expose the necessary dependencies of particular node using a common interface. For instance, if we are validating a *pattern* element, the *pattern's* fragment reader provides access to the *pattern's* children by exposing an executeQuery method that takes in an XPath expression in string format. This expression is executed returning a list of *rule* nodes that can be wrapped to new fragment reader instances. What this gives us a common interface to access any element of the Schematron schema. **Listing 2** is a code snippet of how *rule* fragment readers are dynamically created from the active pattern node during validation.

Listing 2 Creating SchematronFragmenReader Snippet

```
// Create a fragment reader for a child rule element of the active pattern
final SchemaDocumentFragmentReader ruleElement = new SchemaDocumentFragmentReader(
activePatternNode,
        xpathSupport.getNsResolver(),
        xpathSupport.getRelativeRuleElemXPath() );
NodeList rules = ruleElement.getContextNodes();
for ( int ruleIndex = 0; ruleIndex < rules.getLength(); ruleIndex++ ) {
    final Node ruleNode = rules.item( ruleIndex );
    if ( letPreProcessor != null ) {
        if ( hasLoadedLetElementsForRules.compareAndSet( false, true ) ) {
            LOG.info( "Loading let variable declarations for rule nodes." );
            letPreProcessor.registerVariables( ruleNode );
        }
    }
//            final Node ruleNode = rules.item( 0 );
    // Load the rule's assert elements using the fragment
    final NodeList assertElements = ( NodeList ) ruleElement.executeQuery(
xpathSupport.getRelativeAssertElemXPath( ruleIndex + 1 ), NODESET );
    final String contextValue = ruleNode.getAttributes().getNamedItem(
SchematronConstants.RULE_CONTEXT_ATT ).getNodeValue();
    if ( assertElements.getLength() > 0 ) {
        failedAsserts.addAll( validateAssertElements( xmlInstance, assertElements,
xpathEvaluator, contextValue ) );
    }
    // Load sibling report elements using the fragment reader
    final NodeList reportElements = ( NodeList ) ruleElement.executeQuery(
xpathSupport.getRelativeReportElemXPath(), NODESET );
    if ( reportElements.getLength() > 0 ) {
        passedReports.addAll( validateReportElements( xmlInstance, reportElements,
xpathEvaluator, contextValue ) );
    }
```

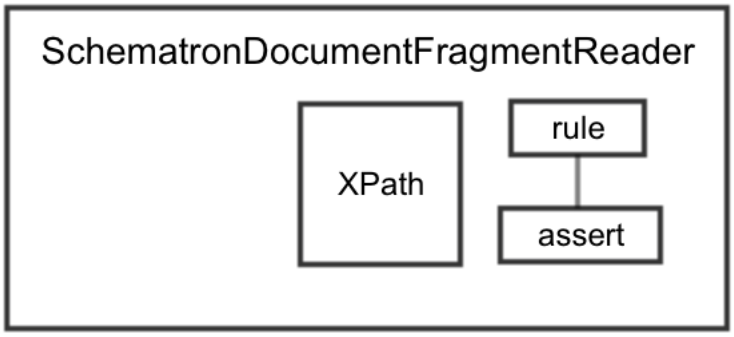


Figure 22 Logical view of SchemaDocumentFragmentReader

In the snippet we see how we use the SchematronDocumentFragmentReader (reader) instance to load all of the *assert* and *report* elements for the given *rule*. **Figure 22** represents a logical view of the reader. A reader uses an instance of an XPath compiler to access the context node it was given. In the snippet, we give the rule reader a reference to the active pattern's DOM node, it then uses this node to load all associated rules and by extension their *assert* and *report* elements. We can also see in the snippet how we handle *let* nodes. The letPreProcessor is leveraged to map the variables to their relative locations, this way we can easily access the variable's expressions during validaiton. To give the full picture, prior to the snippet, the parser is iterating through all of its active patterns. Each active pattern in itself is an instance of a reader which we use to access the pattern's DOM node.

4.3 Validator Implementation Details On the Pre-Processing Phase

As has been previously mentioned, our research leverages macro-expansions to normalize the representation of a Schematron schema during validation. In order to

continue to support the use of the DOM as a means to representing the schema in memory, we pre-process each Schematron element that leverages inheritance or an element that gets included via the ISO *include* element.

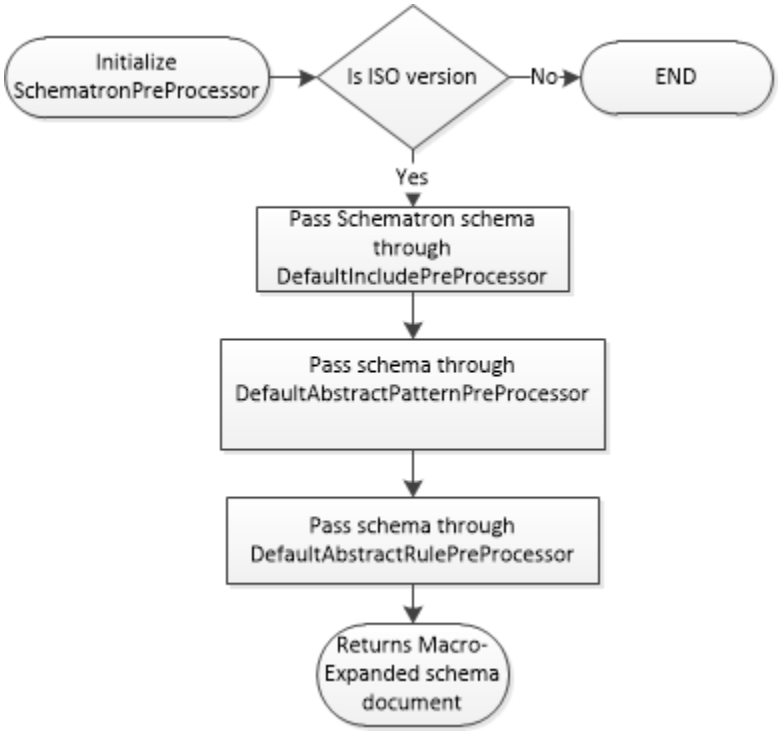


Figure 23 Validator Macro-Expansion Phase

In the pre-processing phase, the Schematron schema is loaded and passed through each of the pre-processors. A pre-processor will read in its referenced XML content and modify the original schema DOM. The result is a DOM representation of a fully expanded Schematron schema, as can bee seen in **Figure 23**. Include element and any abstractions are replaced by the content they referenced. Further in this chapter we provide examples of how the input document compares to the macro-expanded version.

4.3.1 Building modular interpreters based on the schema mappings

At the heart of our Schematron schema (schema) interpreter are XPath based mappings to the various schema elements needed during the pre-processing phase as well as the validation phase. In extrapolating the mappings, we have isolated the dynamic parts of interpreting a schema. Once we boil it down, the behavior needed to access and interpret the contents of the schema should be the same across the interpreter's implementation. What we were looking for was a simple pluggable approach at accessing and interpreting different parts of the schema. This goal is what leads to the creation of the pre-processing phase handlers.

At the core of the expression mappings is the SchematronXPathSupport class. It provides an encapsulation of the base schema elements. This makes it simpler to add pre-ISO schema support; the SchematronISOXPathSupport class, a descendant of the SchematronXPathSupport class, handles current ISO support. The support class provides access to two types of XPath expressions, absolute or relative. Absolute expressions are heavily used during the pre-processing phase. During this phase the interpreter is flattening the document by expanding nodes with references to other markup, such as *include* nodes. Relative expressions are heavily used during validation. When accessing the rules of an active *pattern*, we can use dynamic relative mappings to load of the *rule* or *assert* elements for that pattern. **Figure 24** is a snippet of the SchematronISOXPathSupport. The image points out the use of access methods for absolute and relative XPath expressions. In the figure, we can also see how the different schema element names are handled. All relevant schema element names are declared as constants in the SchematronConstants class. Although pulling the names of the elements

to a constant does not provide any additional functionality, it does however facilitate the maintenance and readability of the code.

```
public String getAbsoluteIncludeNodesOffTheRootSchemaNode() {
    return "//" + getNamespacePrefix() + ":" + SchematronConstants.INCLUDE_ELEM_NAME;
}

public String getRelativeLetElementNodesXPath() {
    return getNamespacePrefix() + ":" + SchematronConstants.LET_ELEM_NAME;
}
```

Figure 24 XPath Mappings Snippet

4.4 Provided Implementation

As previously mentioned, we wanted to provide a default implement of the Schematron parser that leveraged standard and widely used application programming interfaces (api's) and data structures. The parser needed to create a standardized reusable data structure that could be easily accessed during validation. We wanted to validate multiple documents using the same Schematron in-memory structure. Our approach was to support macro expansions so that the entire schema was available in-memory as a DOM data structure. Having the entire schema as a DOM structure in-memory allowed us to simplify access to the schema during validation. As touched upon in the previous section, the schema access is encapsulated in instances of the SchemaDocumentFragmentReader class. The fragment reader instance holds a reference to a node in the DOM. This node is then used as the basis for relative expressions giving better performance than if we started access from the root every time. Results from the validation are returned to the user as instances of ValidationResult classes. Schema validation messages and results are communicated via this object. The following sections

will go into more detail on the implementation of this design as demonstrated in **Figure 23**.

4.4.1 Pre-Processing Schema Phase (Macro Expansions)

The ISO version of Schematron provides features that allow schema authors to reference external reusable content. The contents can be categorized into two main categories. First is the modularized schema content that can include other referential contents, and second is the inheritance focused schema contents where abstractions can be defined. The *include* element is the only element that falls into the modularized schema content category. It allows any valid Schematron markup to be inserted as long as the excerpt is allowed and valid to be inserted where the element is declared. The *pattern* and *rule* elements both have an abstract attribute which if set to true means they are abstract. Abstract elements can define reusable markup by allowing input as to what the context will be. Each type of referential contents has an associated pre-processor that understands how to expand and insert the markup into the schema. For *include* elements, the pre-processor also validates that the included markup is valid with the Schematron ISO syntax. The SchematronPreProcessor class coordinates all of the pre-processing. The use of the class only happens once while the parser is building its representation of the schema. Below we dive into more detail on each one of the pre-processors.

4.4.2 DefaultIncludePreProcessor Implementation

The DefaultIncludePreProcessor is responsible for replacing any *include* elements with the fragment being included. In addition to the manipulation of the schema DOM structure, the pre-processor is also responsible of validating the fragment can be included in the location of the *include* element. The first step is to leverage the

SchematronISOXPathSupport object to retrieve pointers to all of the *include* elements in the schema. If there are none, then we continue and the pre-processing phase is done. **Figure 25** is a snippet showing the method that is called to check for *include* elements and expand them.

```
/**
 * This is the method which begins the pre-processing stage.
 */
public void preProcessIncludes() {
    try {
        final NodeList includeNodesToProcess = findIncludeNodesToProcess();
        for (int nodeIndex = 0; nodeIndex < includeNodesToProcess.getLength(); nodeIndex++) {
            final Node includeNode = includeNodesToProcess.item(nodeIndex);
            final Document loadedDocument = loadIncludedFragment(includeNode);
            if (!checkIncludeFragment(includeNode, loadedDocument)) {
                throw new IllegalArgumentException("The included document is not valid, will not continue.");
            }
            mergeIncludedFragment(includeNode, loadedDocument);
        }
    } catch (XPathExpressionException | IOException |
            ParserConfigurationException | SAXException |
            URISyntaxException e) {
        LOG.error("Unable to load include nodes: " + e.getMessage(), e);
    }
}
```

Figure 25 Method That Pre-Processes Include Elements

As shown in the snippet, the pre-processor searches the whole schema for any *include* elements. The search is guided by an SchematronISOXPathSupport instance. As discussed above, objects of type SchematronXPathSupport encapsulate all access to the schema. If *include* elements are found, we then proceed to search and merge the fragments reference by the *include* element. The mergeIncludeFragment method is responsible for modifying the schema referenced passed in at construction time. **Figure 26** is an example of a schema with *includes* elements. **Figure 27** is an example of the expanded schema.

```
<sch:schema xmlns:sch="http://purl.oclc.org/dsdl/schematron"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://purl.oclc.org/dsdl/schematron ../schematron-iso.xsd"
        queryBinding='xslt2'
        schemaVersion="ISO19757-3">
    <sch:pattern abstract="true" id="table">
        <sch:rule context="$table">
            <sch:assert test="$row">
                The element
                <name/>
                is a table. Tables contain rows.
            </sch:assert>
        </sch:rule>
        <sch:include href="IncludeRuleFragment.sch"/>
    </sch:pattern>
    <sch:include href="IncludeAbstractPatternFragment.sch"/>
</sch:schema>
```

Figure 26 Schematron Schema Excerpt With Includes

In **Figure 26** we have two *include* elements nested under different parent elements. What we are trying to show here is that during the pre-processing phase, we have to validate that any elements added by the first *include* are valid elements under a *pattern* element. For instance, adding a *phase* element to the fragment would cause a validation error to be thrown. *Phase* elements cannot be nested inside a *pattern* element. Such an error would impede the parser from continuing, as it would be treated as a syntactic error. **Figure 27** is the output of the pre-processing phase.

```
<sch:schema xmlns:sch="http://purl.oclc.org/dsdl/schematron"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    queryBinding="xslt2"
    schemaVersion="ISO19757-3"
    xsi:schemaLocation="http://purl.oclc.org/dsdl/schematron ../schematron-iso.xsd">
    <sch:pattern abstract="true" id="table">
        <sch:rule context="$table">
            <sch:assert test="$row">
                The element
                <name/>
                is a table. Tables contain rows.
            </sch:assert>
        </sch:rule>
        <!-- This is the IncludeRuleFragment.sch fragment section -->
        <sch:rule context="$row">
            <sch:assert test="$entry">
                The element
                <sch:name/>
                is a table row. Rows contain entries.
            </sch:assert>
        </sch:rule>
        <!-- End include -->
    </sch:pattern>
    <!-- This is the IncludeAbstractPatternFragment.sch fragment section -->
    <sch:pattern id="HTML_Table" is-a="table">
        <sch:param name="table" value="table"/>
        <sch:param name="row" value="tr"/>
        <sch:param name="entry" value="td|th"/>
    </sch:pattern>
    <!-- End include -->
</sch:schema>
```

Figure 27 Expanded *Include* Elements

The fragments are surrounded by comments pointing to the file names in the href attribute of the *include* element in **Figure 26**. The pre-processor is capable of reading in include fragments either via http or local disk. Handling *include* fragments is as complicated as pre-processing gets. The reason is not because we merge two independent DOM trees, but around the validation of the DOM fragment and how that validation changes based on where the *include* element is declared.

4.4.3 DefaultAbstractPatternPreProcessor Implementation

In Schematron ISO, support for pattern abstractions has empowered schema authors with the ability to modularize and reuse schema markup by writing pluggable excerpts. Similar pattern validations can be grouped and used by schemas for distinct XML documents. In keeping with the approach of expanding an in-memory schema, the

contents of a references abstract *pattern* have replaced the contents of the descendant *pattern*. **Figure 28** represents an abstract *pattern* element and the descendant *pattern*. The *pattern* with the id LAYERED_TABLE extends the layered_table *pattern* and passes the parameters that corresponding to variables declared by the abstraction. What the pre-processor will do is take a copy of the abstract *pattern's* child elements and include them in the descendant *pattern*. As explained above, during validation this expansion simplifies the XPath expressions needed during validation in order to access the parent's content.

```
<sch:pattern abstract="true" id="layered-table">
    <sch:rule abstract="true" id="table-div">
        <sch:assert test="../div">
            The <sch:name/> element is a child of
            schematron-output.
        </sch:assert>
    </sch:rule>
    <sch:rule context="$table">
        <sch:assert test="count( /$table/$row ) &gt; 0">
            The element <name/> is a table.
            Tables must have a row
            <sch:value-of select="$row /name()"/>.
        </sch:assert>
    </sch:rule>
    <sch:rule context="$table">
        <sch:extends rule="table-div"/>
    </sch:rule>
</sch:pattern>
<sch:pattern is-a="layered-table" id="LAYERED_TABLE">
    <sch:param name="table" value="table"/>
    <sch:param name="row"
               value="trow"/>
</sch:pattern>
```

Figure 28 Abstract Pattern Elements

In **Figure 29** we can see what the expanded *pattern* element looks like. As you can notice, access to the abstraction has been greatly simplified. Instead of have XPath expressions to two different elements, we can now just have a single expression to access to the contents.

```
<sch:pattern xmlns:sch="http://purl.oclc.org/dsdl/schematron"
             id="LAYERED_TABLE">
    <sch:rule abstract="true" id="table-div">
        <sch:assert test="../div">
            The <sch:name/> element is a child of
            schematron-output.
        </sch:assert>
    </sch:rule>
    <sch:rule context="table">
        <sch:assert test="count( /$table/$row ) &gt; 0">
            The element <name/> is a table.
            Tables must have a row
            <sch:value-of select="$row /name()"/>.
        </sch:assert>
    </sch:rule>
    <sch:rule context="table">
        <sch:extends rule="table-div"/>
    </sch:rule>
</sch:pattern>
```

Figure 29 Expanded Abstract Pattern Elements

4.4.4 DefaultAbstractRulePreProcessor Implementation

Processing abstract rules is done in the same way as abstract patterns. The inputs and outputs are the same with the only difference that when processing abstract rules, the pre-processor search for rule elements with the value of the abstract attribute set to true. In the example we are about to show in **Figure 30**, you will see the declaration of the abstract *rule* and then what the expansion looks like after passing through the pre-processor. The motivation is the same, to simplify and standardize access to all part of the schema during validation. Rule extensions in Schematron can be defined within the pattern where the abstract rule is defined. In our example you will see a short excerpt defining the implementing rule, and then a larger excerpt showing the abstract definition as well as the implementation.

```
<sch:rule context="svrl:text">
    <sch:extends rule="childless"/>
    <sch:extends rule="second-level"/>
</sch:rule>
```

Figure 30 Simple abstract rule instances

The *extends* element inside this rule is how an abstract *rule* can be extended. Since abstract *rules* do not define a context, the context is taken from the extending rule. As with other pre-processors, the abstract *rule* pre-processor will replace the *extends* element with the contents of the abstract *rule* it is referencing or extending. In **Figure 31**we can see more details on the results of the pre-processor.

```
<sch:rule context="svrl:text">
    <sch:assert test="count(*)=0">
        The <sch:name/> element should not
        contain any elements.
    </sch:assert>
    <sch:assert test="../svrl:schematron-output">
        The <sch:name/> element is a child of
        schematron-output.
    </sch:assert>
</sch:rule>
```

Figure 31 Post Abstract Rule Expansion Examples

In **Figure 31** notice how the *extends* elements have been replaced by the actual content of the abstract *rule.* The ability to replace markup with the intended markup is what makes this parser unique when compared against its predecessor. Having a consolidated DOM structure during validation simplifies and optimizes the function of the parser by allowing a single DOM structure to represent the entire schema.

4.4.5 Maintaining an In-Memory XML Structure

As mentioned above, an important feature of this parser is to maintain a single XML structure of the entire schema. To accomplish this, we leveraged macro-expansions to eliminate any externally referenced or co-located semantic markup. The benefits of this approach extend to the real-time access to schema elements needed during validation. By simplifying and standardizing the XML representation of the schema, we are able to standardize how we access all elements of the schema prior to or during validation. The

SchematronXPathSupport class is how we have leveraged the XPath api in combination with the DOM to access markup at all stages of validation or pre-validation.

4.4.6 Encapsulates Features Sets Based On Mapping Rules

As touched upon in previous sections, all of the Schematron features including the new ISO extensions are accessible via instances of SchematronDocumentFragmentReader. This class encapsulates access to the single DOM structure. A context node is used to determine the starting point of the queries. After a reader has been created for a given context, associated mapping rules are used to traverse the DOM and retrieve the data on request. This is how we can avoid retaining custom data structures to access the schema during validation.

4.5 Validator Implementation Overview

Having a consolidated approach for our schema management is the basis for our implicit support for the different Schematron ISO and pre-ISO features. The parser, after the macro-expansion phase is complete, begins by processing all variables for the top-level elements. These are the *schema*, *phase* or *pattern* elements. Global variables can be declared as children of the *schema* element. Variables that have a narrower scope can be declared in the *phase* or *pattern* elements. After the variables have been processed, the parser looks for the active *pattern* or *phase* element. If none, then it means that the parser will process all top-level elements. When interpreting a *pattern*, the processor will create a fragment reader for all *rule* elements inside a given pattern. With this reader, the parser will load all of the rules and assertions needed for validation. When validation is

complete, ValidationResults are returned informing the caller of success or failure. **Figure 32** is a flowchart of the explained validation process.

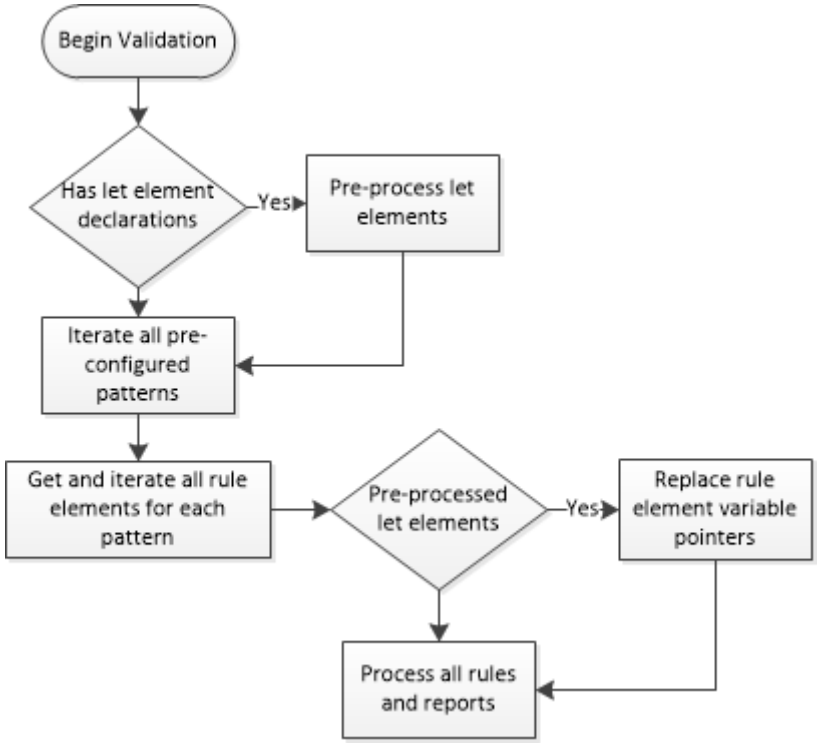


Figure 32 Validation Phase

4.5.1 XPath Mappings

The first thing the parser interprets is the version of Schematron it will deal with. This enables different XPath expressions to be supported if a newer version of Schematron is produced. The Schematron version syntax is encapsulated by an instance of a SchematronXPathSupport class. This class represents an implentation of the factory pattern [38].

4.5.2 Validation Results

Validation results are captured in list containing instances of ValidationResult. The ValidationResult class encapsulates the message if any reported by Schematron, the expression or XPath used during the validation, the id of the node that was being validated and finally the result. At the lowest level, the nodes that will evaluate to a pass or fail result are either *assert* elements or *report* elements. A major difference between these two elements is that *assert* elements always have to evaluate to true, if not, the contents of the element gets processed. For *report* elements however, the opposite is true. In the SchematronValidator, you will see a method for each type of element that sets the correct value need to pass or fail the rule.

4.6 Chapter Summary

In this chapter we saw an overview of the benefits of our version of the Schematron parser. The leveraging of micro-expansions and standardized data-structures really gives this parser a flexible and scalable architecture. We have also introduced the default implementation of the Schematron features within the parser. In the following chapter we go into more detail on the benefits around supporting the new Schematron ISO features.

Chapter 5

Experimental Validation

In this chapter we demonstrate different use cases our parser can be used for. We begin with very basic examples that focus on particular feature of Schematron and finally end with a more complex use case that takes advantages of the new Schematorn ISO feature set. The example we will introduce exploits the fact that Schematron can also be thought of as a rule-validating engine. We have build a smart trade router using Schematron as the routing rule definition language. We hope to show that our parser, because of its ease of integration, can be used as an integral part of a robust rules engine.

5.1 Syntactic Definition and Validation

To begin, we focus on basic syntactic and semantic validation. We are building a service or system that understands how to validate and process three types of foreign exchange (FX) trade messages [39]. In the schema definition you can see we have spot trades, forward trades and swap trades. For the scope of this example, the main difference between these types of trades is that swap trades encompass two legs rather than a single leg. A leg element holds the details of the trade. Those details are the currency pair, the side of the trade and the amount traded. **Figure 33** describes the grammar for a trade message. As defined in the schema, a trade message can have multiple trade entries. Depending on the type of trade, the leg child elements changes. Going back to a point we made earlier, expressing the co-constraint of a trade containing two legs when the type of the trade is FX_SWAP is not possible by using only a W3C Schema definition. For that we need Schematron's expressive rule based schema.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="trades">
        <xs:complexType>
            <xs:sequence>
                <xs:element minOccurs="1" maxOccurs="unbounded" ref="trade"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="trade">
        <xs:complexType>
            <xs:sequence>
                <xs:element minOccurs="1" maxOccurs="1" ref="legs"/>
            </xs:sequence>
            <xs:attribute name="id" type="xs:string" use="required"/>
            <xs:attribute name="type" type="tradeTypes" use="required"/>
        </xs:complexType>
    </xs:element>
    <xs:element name="legs">
        <xs:complexType>
            <xs:sequence>
                <xs:element minOccurs="1" maxOccurs="2" name="leg">
                    <xs:complexType>
                        <xs:attribute name="ccy" type="xs:string" use="required"/>
                        <xs:attribute name="price" type="xs:double" use="required"/>
                        <xs:attribute name="side" type="sidesType" use="required"/>
                        <xs:attribute name="amount" type="xs:double" use="required"/>
                        <xs:attribute name="type" type="legType" use="optional"/>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:simpleType name="tradeTypes">
        <xs:restriction base="xs:string">
            <xs:enumeration value="FX_SPOT"/>
            <xs:enumeration value="FX_FWD"/>
            <xs:enumeration value="FX_SWAP"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="sidesType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="BUY"/>
            <xs:enumeration value="SELL"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="legType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="SPOT"/>
            <xs:enumeration value="FWD"/>
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

Figure 33 Trade Message Grammar Using W3C Schemas

Having the full expressiveness of a grammar defining language such as W3C Schema combined with a rule based co-constraint definition language such as Schematron

provides the best of both worlds. In the following sections we dive into the Schematron portion of the example.

5.2 Semantic Co-Constraint Definition and Validation

The semantic rules revolve around checking the number of legs in a trade. According to the grammatical syntax, a trade can have 1 or more leg elements nested under a single *legs* parent element. What we cannot explain using the grammar is that FX_SWAP trades have to contain two legs while other types of trades can only contain a single leg. However, we have also used the semantic definition syntax to check that a trade can only contain 1 or 2 legs maximum. This type of syntactic constraint is much clearer if defined using the W3C Schema syntax. In figure **Figure 33** we can see the minOccurs and maxOccurs constraint on the *leg* element definition. In our opinion such a definition reads much better than the Schematron syntax.

```
<schema xmlns="http://www.ascc.net/xml/schematron">
    <title>FX Trade Rules</title>
    <pattern name="Trade Type">
        <rule context="trades/trade">
            <assert test="count(legs/leg) &gt; 0 and count(legs/leg) &lt; 3"
                    id="tradeLegCount">
                Trades have to have legs.
            </assert>
        </rule>
    </pattern>
    <pattern name="Swap Trade Type">
        <rule context="trades/trade[@type='FX_SWAP']">
            <assert test="count(legs/leg) = 2" id="swapTradeLegCount">
                A Swap has to have two legs, one for Spot and one for Forward trades.
            </assert>
        </rule>
        <rule context="trades/trade[@type!='FX_SWAP']">
            <assert test="count(legs/leg) = 1" id="nonSwapTradeLegCount">
                Non-Swap trades have one leg declared.
            </assert>
        </rule>
    </pattern>
</schema>
```

Figure 34 Trade Message Semantic Co-Constraint Definition

In **Figure 34** we see the Schematron schema used to semantically validate a trade message. The first pattern named *Trade Type* defines a single rule discussed above. This rule ensures that a trade can have between 1 and 2 legs. As discussed, such syntactic constraints can be more efficiently defined using a syntactic language such as the W3C Schema. The second pattern defined *Swap Trade Type* is where we can see the semantic constraints that cannot be defined using just a grammar definition document. The first rule in the pattern checks that all trades of type FX_SWAP contain only two legs. The second ensures that all non-swap trades contain only a single leg. Such semantic constraints create an ability to capture business specific constraints and automate their validation. In the next section we take a look at two XML messages received. Both are syntactically correct, something verified by the XML Schema using standardized XML application programming interfaces, but the second message, although syntactically correct, is semantically incorrect.

5.3 XML Message Syntactic and Semantic Validation

As explained, our example use of our semantic validator focuses on the validation of trade messages. These messages are sent to our system each containing a collection of trades for processing. According to the syntactic definition, each trade is required to have at least one and at most two legs. A leg defines the financial details of the trade. Trades tend to be much more complex than this, but to maintain clarity and simplicity in this example, we have chosen to only focus on some of the financial details regarding trade data. As mentioned above, in our first message example we have a total of three trades. Each type is syntactically and semantically correct. When passed through the validator,

we don't see any of the assertion error messages seen in the Schematron schema. We can see this message in **Figure 35**.

```
<trades>
    <trade id="TRN-1234" type="FX_SPOT">
        <legs>
            <leg ccy="USD/CAD" price="100.33" side="BUY" amount="1000000"/>
        </legs>
    </trade>
    <trade id="TRN-3456" type="FX_FWD">
        <legs>
            <leg ccy="USD/CAD" price="101.33" side="SELL" amount="1000000"/>
        </legs>
    </trade>
    <trade id="TRN-5678" type="FX_SWAP">
        <legs>
            <leg ccy="USD/CAD" price="100.33" side="BUY" amount="1000000" type="SPOT"/>
            <leg ccy="USD/CAD" price="100.34" side="SELL" amount="1000000" type="FWD"/>
        </legs>
    </trade>
</trades>
```

Figure 35 A Correct Trade Message

If we go back and look at the syntactic and semantic rules, we can see that the message above is perfectly acceptable. However, the next message has an incorrect entry. The FX_SWAP trade has a single leg and the FX_SPOT trade has two legs on the same side. From a syntactic perspective, this message is acceptable. However, according to our semantic constraints, the message in **Figure 36** is semantically incorrect because it violates the trade type and leg count constraints. The validator's output can be seen in **Figure 37**. The validation message prints the assertion failures for the assertion with identifier *swapTradeLegCount* and the assertion with identifier *nonSwapTradeLegCount*. This completes an example of the pre-existing Schematron features. Our next example extends the functionality of our validator by enabling smart trade routing functionality. The goal is to route trades based on their semantic categorization. In this extension we will demonstrate the use of some of the Schematron ISO features that demonstrate the

clear advantages in terms of schema evolution and maintenance over the previous Schematron versions.

```
<trades>
    <trade id="TRN-1234" type="FX_SPOT">
        <legs>
            <leg ccy="USD/CAD" price="100.33" side="BUY" amount="1000000"/>
            <leg ccy="USD/CAD" price="100.33" side="BUY" amount="1000000"/>
        </legs>
    </trade>
    <trade id="TRN-1234" type="FX_FWD">
        <legs>
            <leg ccy="USD/CAD" price="101.33" side="SELL" amount="1000000"/>
        </legs>
    </trade>
    <trade id="TRN-1234" type="FX_SWAP">
        <legs>
            <leg ccy="USD/CAD" price="100.33" side="BUY" amount="1000000" type="SPOT"/>
        </legs>
    </trade>
</trades>
```

Figure 36 Semantically Incorrect Messages

```
{assertion_results=
    [Message:[ A Swap has to have two legs, one for Spot and one for Forward trades.
            ]     IsValid:[false] ValidationExpression:[count(legs/leg) = 2]
            ] ValidationNodeId:[swapTradeLegCount];,
    Message:[ Non-Swap trades have one leg declared.
            ]     IsValid:[false] ValidationExpression:[count(legs/leg) = 1]
            ] ValidationNodeId:[nonSwapTradeLegCount];]}
```

Figure 37 Validator Output As Printed By The System

5.4 Smart Trade Routing Using Semantic Constraints

To show the versatility of the parser, this next example uses features introduced in the ISO version of Schematron to extend the previous semantic co-constraints with business specific rules. In our example, if a trade has passed all co-constraint validation, then it is routed according to its type. To begin, let's take a look at the co-constraint fragment that is to be included into the document that defines the rules that have to do with message routing. The point to take from how we have used the *include* element is that we were

able to separate business routing rules from co-constraints. This allows for the reusability of the co-constraints and also allows for easy extensibility of the either the trade co-constraints or the routing rules. **Figure 38** depicts the fragment to be included in the business routing rules schema. As before, if any of the co-constraints fail, we get to see an exception that explains the semantic constraints that failed. Later on in this section we will have another figure to show the error message.

```
<pattern id="TradeTypeCoConstraints">
    <rule context="trades/trade">
        <assert test="count(legs/leg) &gt; 0 and count(legs/leg) &lt; 3"
                id="tradeLegCount">
            Trades have to have legs.
        </assert>
    </rule>
    <rule context="trades/trade[@type='FX_SWAP']">
        <assert test="count(legs/leg) = 2" id="swapTradeLegCount">
            A Swap has to have two legs, one for Spot and one
            for Forward trades.
        </assert>
    </rule>
    <rule context="trades/trade[@type!='FX_SWAP']">
        <assert test="count(legs/leg) = 1" id="nonSwapTradeLegCount">
            Non-Swap trades have one leg declared.
        </assert>
    </rule>
</pattern>
```

Figure 38 Co-Constraint Fragments

As can be imagined, the fragment above can be inserted into any schema. In our next figure, we see how the *include* element is used. One thing to keep in mind is that the elements included, have to fit in the correct section of the parent schema. As see in **Figure 38**, the element to include is a *pattern*. As is visible in **Figure 39**, the routing schema uses the *defaultPhase* attribute in the *schema* element. To integrate the included element, we added an *active* element in our default phase that points to the

"TradeTypeCoConstraints" pattern. A small extension we did to the include handler was to support a classpath URL prefix. This feature enables the parser to look for the file to include in its system classpath.

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://purl.oclc.org/dsdl/schematron schematron-iso.xsd"
        queryBinding='xslt2'
        schemaVersion="ISO19757-3"
        defaultPhase="routes">

    <title>Trade routing rules.</title>

    <phase id="routes">
        <active pattern="TradeTypeCoConstraints"/>
        <active pattern="routeTrades"/>
    </phase>

    <include href="classpath:definitions/FXTradeSyntaxRuleFragment.sch"/>

    <pattern id="routeTrades">
        <title>Smart Trade Type Router</title>
        <rule context="//trade">
            <report test="@type='FX_SPOT'">
                {"id":"<value-of select="@id"/>","type":"FX_SPOT"}
            </report>
        </rule>
    </pattern>
</schema>
```

Figure 39 Routing Rule Schematron Schema Definitions

In the next section we explain the example code in detail. When the routing rules fire, it selects the correct function that would route the trade to the correct subscriber. As an added benefit of the report element, we have included a Java Script Object Notation (JSON) message as the body of the report [40]. What that allows the code to do is transform the text to a map with an id entry and a type entry. The value of the type is how the router understands where to route the message. With the id field, we can access the full trade from the accessible DOM structure.

5.5 Explanation of Smart Trade Routing Code

The routing application is based on event-based programming. All messages are handled by implementation of the IMessageEventHandler interface. **Figure 40** depicts a snippet from the TradeMessageEvent (event handler) class. This class is the default implementation of the IMessageEventHandler interface. Being a modular design, the programmer can inject an instance of IMessageEventValidator. The event handler's role is to normalize the external form of the incoming message so that the validator can consume it. Notice the implementation of the handleMessage method below. In it we can see how a generic stream is converted into an XML document along with the syntax definition for the incoming message. The schemaInstance variable is a byte buffer holding the schema in memory for later use.

```
@Override
public IMessageEventHandler withEventValidator( IMessageEventValidator validator ) {
    this.validator = validator;
    return this;
}

@Override
public void handleMessage( InputStream xmlMessage ) {
    validator.validate( XMLSupport.loadXMLMessage( new InputSource( xmlMessage ),
            new StreamSource( new ByteArrayInputStream( schemaInstance.array() ) ) ) );
}
```

Figure 40 Default Implementation of the IMessageEventHandlerHandler Inteface

As mentioned, an instance of IMessageEventValidator can be injected using the withEventValidator method. Implementations of this interface are responsible to encapsulate the syntactic and semantic validators. These implementations will execute the validation and also whatever behavior is required post validation. **Figure 41** depicts how we initialize the SchematronValidator. The important thing to notice is that the validator loads the ISO version of Schematron's schema. Not only do we syntactically

validate the incoming XML message, but we also validate that the Schematron schema we are about to use for our semantic rules also abides by the syntax defined by Schematron's ISO version.

```
protected final SchematronValidator validator;

AbstractMessageEventValidator( String schematronSchemaUri ) {
    final Document schematron = XMLSupport.loadXMLMessage(
            new InputSource( ClassLoader.getSystemClassLoader().getResourceAsStream( schematronSchemaUri ) ),
            new StreamSource( ClassLoader.getSystemClassLoader().getResourceAsStream( SCHEMATRON_ISO_SCHEMA ) ) );
    validator = new SchematronValidator(
            schematron,
            SchematronValidator.DEFAULT_SELECTOR
    );
}
```

Figure 41 Abstract Constructor Used By Implementations Of The IMessageEventValidator Interface

When validating, all that is needed is to call the validate method of our SchematronValidator instance. This in turn returns the validation results. One thing to note about this use of the validator is the reusability of the validator instance. Once wired up, we can reuse the instance for multiple messages events.

```
@Override
public void validate( Node message ) {
    Map< String, List< ValidationResult > > result = validator.validate( message );
    if ( result == null || result.isEmpty() ) {
        LOG.info( "None of the routing rules kicked off." );
        return;
    }

    if (result.containsKey( "assertion_results" )) {
        LOG.error( extractError( result ) );
        return;
    }
```

Figure 42 Snippet Showing How To Use The Validator

In **Figure 42** we can see how simple the validator is to use. The resulting map contains all of the results from all of the assertions executed. If you notice, in **Figure 42**, there is a check for assertion errors that might have occurred. If the result contains a value "assertion_results", this means that some of the co-constraints failed. At that point, this application only logs the error and returns without routing any trades. **Figure 43** depicts how we can use the contents of a Schematron report to help route the trade messages. Here we see the use of an ObjectReader class to parse the JSON message into a Map. We can then see how we extract the trade's id from the map and also from the current document being validated. The type attribute found in the map determines the route the trade takes.

```
for ( List< ValidationResult > validationResults : result.values() ) {
    for ( ValidationResult validationResult : validationResults ) {
        if ( !validationResult.isValid() ) {
            continue;
        }
        final String report = validationResult.getMessage().trim();
        try {
            final Map< String, String > reportMap = objectReader.readValue( report );
            final Node trade = ( Node ) xpath.evaluate( "//trade[@id='" + reportMap.get( "id" ) + "']", message, NODE );
            TradeType.valueOf( reportMap.get( "type" ) ).route( trade );
        } catch ( XPathExpressionException | IOException e ) {
            LOG.error( "Unable to retrieve trade for report: " + report );
        }
    }
}
```

Figure 43 Handling The Validation Results

5.6 Conclusion

In this chapter we have successfully used the SchematronValidator in different contexts. We have demonstrated how we can build rule engines by leveraging Schematron's expressiveness. In addition, we have also show a potential extension for this research. JSON is become more and more popular. We can implement a JSON version of

Schematron. JSON schemas can be used to define syntactic constraints for JSON messages, by the same extension, we can use JSON schema's to then handle the assertion and report definitions.

Chapter 6

Research Conclusion

6.1 Major Achievements and Research Contributions

In this research we have implemented a Schematron ISO parser in Java. In addition to semantic validation, the parser has the ability to execute syntactic validation in the same pass. The combination of both types of validation cover a wide spectrum of validation requirements making the parser a versatile research or business instrument.

Contributions can also be seen in the form in which the Schematorn ISO features were implemented. Below is a bulleted list of these contributions:

- Leveraged standardized data structures, namely the Document Object Model (DOM). The DOM is used to represent the various Schematron components in memory.
- Use of macro-expansions to pre-process some of the Schematron ISO features. With the introduction of inheritance and script fragmentation, the parser optimizes by unraveling all of the necessary markup into a single DOM representation. During validation, there is not need to insert fragments or inherited elements as this phase happens during initialization of the parser.
- Built as a fully independent and reusable library that can easily get integrated into a wider application. The ease of use of the API facilitates integration. A single parser instance can be used numerous times.

- Demonstrated the parser can be used beyond syntactic and semantic validation. By injecting business rules in reports, the parser can be used to tie business rules with semantic constraints.

6.2 Future Work

The parser can still be improved to allow it to stay relevant with the changing technology trends. Below is a list of potential future work examples.

- Replacement of XML Schema with JSON Schema.
- Integration with technologies that can provide support for ontologies. This would extend the parser to support not just co-constraint validations but also natural language rules.
- Add the support to handle multiple Schematron documents at the same time.

References

[1] A. Duke, “Use Case: B2B Integration with Semantic Mediation,” *Semantic Web Use Cases and Case Studies*, Mar-2007. [Online]. Available: http://www.w3.org/2001/sw/sweo/public/UseCases/BT/. [Accessed: 03-Oct-2015].

[2] T. Berners-Lee, “Axioms of Web architecture,” 27-Aug-2009. [Online]. Available: https://www.w3.org/DesignIssues/XML-Semantics.html. [Accessed: 03-Apr-2016].

[3] “A Gentle Introduction to XML - The TEI Guidelines.” [Online]. Available: http://www.tei-c.org/release/doc/tei-p5-doc/en/html/SG.html. [Accessed: 08-Apr-2016].

[4] S. Golikov, “Integrated Syntactic/Semantic XML Data Validation with a Reusable Software Component,” Doctorate, Pace University, 2012.

[5] J. Cleary, “XML & DTDs :: Jacob Cleary.” [Online]. Available: https://www.ischool.utexas.edu/technology/tutorials/webdev/xml_dtds/04_xml.html. [Accessed: 06-Apr-2016].

[6] “Co-occurrence constraints - W3C Wiki.” [Online]. Available: https://www.w3.org/wiki/Co-occurrence_constraints. [Accessed: 06-Apr-2016].

[7] E. Van der Vlist, *RELAX NG*, 1st ed. Sebastopol, CA: O’Reilly, 2004.

[8] D. Lee and W. W. Chu, “Comparative Analysis of Six XML Schema Languages,” *Dep. Comput. Sci. Univ. Calif. Los Angel.*, no. ACM SIGMOD Record, 29(3), Sep. 2000.

[9] “XML Schemas: Best Practices.” [Online]. Available: http://www.xfront.com/ExtendingSchemas.html. [Accessed: 06-Apr-2016].

[10] E. R. Harold, “The Java XML Validation API,” 10-Feb-2010. [Online]. Available: http://www.ibm.com/developerworks/library/x-javaxmlvalidapi/. [Accessed: 27-Jul-2014].

[11] D. Kaye, *Loosely coupled: the missing pieces of Web services*. Marin County, Calif: RDS Press, 2003.

[12] "Macro Expansion - The GNU C Preprocessor Internals." [Online]. Available: https://gcc.gnu.org/onlinedocs/cppinternals/Macro-Expansion.html. [Accessed: 06-Apr-2016].

[13] "What is the Document Object Model?" [Online]. Available: https://www.w3.org/TR/DOM-Level-2-Core/introduction.html. [Accessed: 11-Apr-2016].

[14] "XML - Usage of XML in Developing Forex and Finance Applications." [Online]. Available: http://www.totalxml.net/forex_xml.php. [Accessed: 06-Apr-2016].

[15] W3C, "W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures." [Online]. Available: http://www.w3.org/TR/xmlschema11-1/#intro. [Accessed: 19-Jul-2014].

[16] W. Martens, F. Neven, T. Schwentick, and G. J. Bex, "Expressiveness and complexity of XML Schema," *ACM Trans Database Syst*, vol. 31, pp. 770–813, 2006.

[17] R. A. Wyke and A. Watt, *XML Schema Essentials*. New York: John Wiley, 2002.

[18] "RELAX NG Tutorial." [Online]. Available: http://relaxng.org/tutorial-20011203.html. [Accessed: 11-Apr-2016].

[19] "RDF vs. XML | Cambridge Semantics." [Online]. Available: http://www.cambridgesemantics.com/semantic-university/rdf-vs-xml. [Accessed: 11-Apr-2016].

[20] U. Ogbuji, "Thinking XML: XML meets semantics, Part 1," 01-Feb-2001. [Online]. Available: http://www.ibm.com/developerworks/library/x-think1.html. [Accessed: 02-Aug-2014].

[21] "ebXML - Wikipedia, the free encyclopedia." [Online]. Available: https://en.wikipedia.org/wiki/EbXML. [Accessed: 08-Apr-2016].

[22] C. Vergara-Niedermayr, F. Wang, T. Pan, T. Kurc, and J. Saltz, "SEMANTICALLY INTEROPERABLE XML DATA," *Int. J. Semantic Comput.*, vol. 07, no. 03, pp. 237–255, Sep. 2013.

[23] C. S. Coen, P. Marinelli, and F. Vitali, "Schemapath, a minimal extension to XML Schema for conditional constraints," in *Proceedings*

of the 13th international conference on World Wide Web, 2004, pp. 164–174.

[24] "Schema - W3C." [Online]. Available: https://www.w3.org/standards/xml/schema. [Accessed: 08-Apr-2016].

[25] N. Delima, S. Gao, M. Glavassevich, and K. Noaman, "XML Schema 1.1, Part 2: An introduction to XML Schema 1.1," 13-Jan-2009. [Online]. Available: http://www.ibm.com/developerworks/xml/library/x-xml11pt2/index.html. [Accessed: 31-Aug-2014].

[26] "Schematron tutorial." [Online]. Available: http://dh.obdurodon.org/schematron-intro.xhtml. [Accessed: 11-Apr-2016].

[27] "XML.com: Schemarama." [Online]. Available: http://www.xml.com/lpt/a/727. [Accessed: 11-Apr-2016].

[28] E. Van der Vlist, *Schematron*. Sebastopol, Calif.: O'Reilly, 2007.

[29] Ri. Jeliffe, "The top three mistakes in Schematron - O'Reilly Broadcast," 07-Jun-2009. [Online]. Available: http://broadcast.oreilly.com/2009/06/the-top-three-mistakes-in-sche.html. [Accessed: 01-May-2014].

[30] "Analyzing critical rendering path performance | Web Fundamentals - Google Developers." [Online]. Available: https://developers.google.com/web/fundamentals/performance/critical-rendering-path/analyzing-crp?hl=en. [Accessed: 06-Apr-2016].

[31] "4.1 Modular Design Review." [Online]. Available: https://www.mcs.anl.gov/~itf/dbpp/text/node40.html. [Accessed: 06-Apr-2016].

[32] B. Meyer, *Object-oriented software construction*, 2nd ed. Upper Saddle River, N.J: Prentice Hall PTR, 1997.

[33] D. Taniar and J. W. Rahayu, Eds., *Web information systems*. Hershey: Idea Group Pub, 2004.

[34] "XML.com: An Introduction to Schematron." [Online]. Available: http://www.xml.com/lpt/a/1318. [Accessed: 08-Apr-2016].

[35] "Modular programming - Wikipedia, the free encyclopedia." [Online]. Available: https://en.wikipedia.org/wiki/Modular_programming. [Accessed: 08-Apr-2016].

[36] "Delegation Pattern," in *Learn Objective-C for Java Developers*, Berkeley, CA: Apress, 2009, pp. 315–323.

[37] "What is a Process Flowchart? Process Flow Diagrams | ASQ." [Online]. Available: http://asq.org/learn-about-quality/process-analysis-tools/overview/flowchart.html. [Accessed: 08-Apr-2016].

[38] "Design Pattern: factory patterns - Coding Geek." [Online]. Available: http://coding-geek.com/design-pattern-factory-patterns/. [Accessed: 08-Apr-2016].

[39] "Foreign exchange market - Wikipedia, the free encyclopedia." [Online]. Available: https://en.wikipedia.org/wiki/Foreign_exchange_market. [Accessed: 08-Apr-2016].

[40] B. Smith, *Beginning JSON*. Berkeley, CA: Apress, 2015.

Glossary

B2B	Business to business
DOM	Document Object Model
DTD	Document Type Definition
ISO	International Organization for Standards
RELAX NG	REgular LAnguage for XML Next Generation
W3C	World Wide Web Consortium
XML	eXtensible Markup Langauge
XPATH	XML Path Language
XSD	W3C XML Schema
XSL	Extensible style sheet language formatting
XSLT	eXtensible Stylesheet Language Transformations

Appendix A : Building and Installation for the Schematron Parser

Here we describe how to get the parser up and running in your development environment. The steps will be described as a sequence; any assumptions will be explained.

1 Setting up your environment. First off, the parser has been developed using Java 7. Installing Java is very straight forward, instructions can be followed here https://www.java.com/en/download/help/download_options.xml.

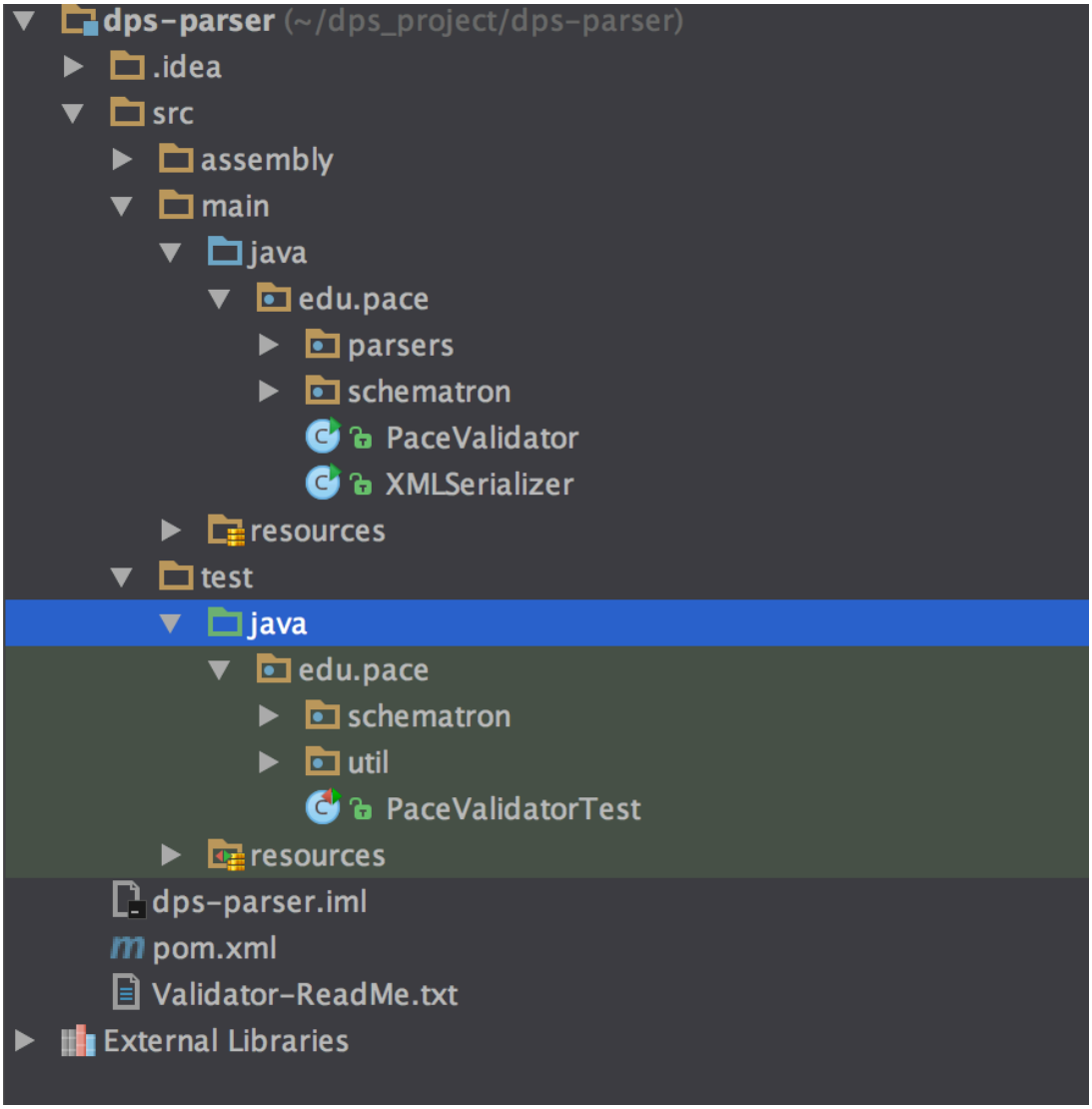
2 Next, Apache Maven is required to build the parser. Maven is used to manage the parser's dependencies as well as for building the parser. Maven is a widely used technology, installation instructions can be found here https://maven.apache.org/install.html.

3 During development we used IntelliJ IDEA Community Edition. This was our editor of choice. The community edition is free and can be downloaded from the JetBrains website. Instructions for downloading and installation can be found here https://www.jetbrains.com/idea/help/installing-and-launching.html.

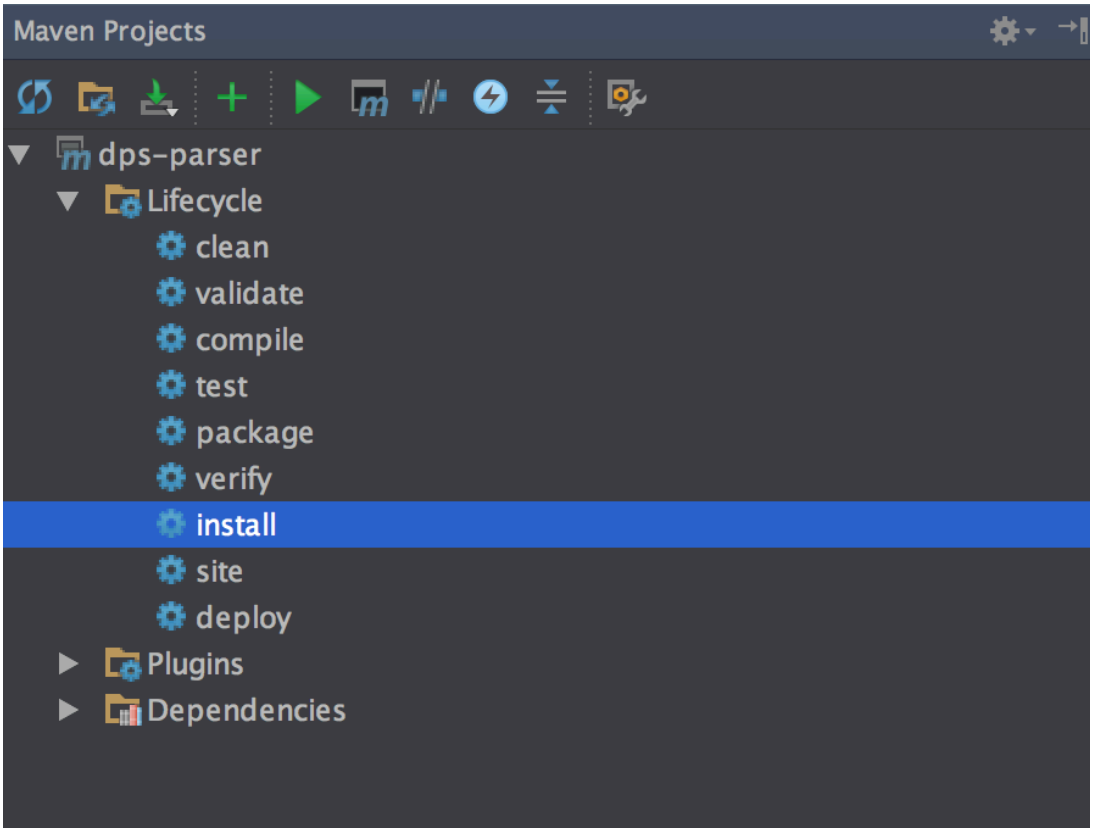
4 To install the code attached with this dissertation, you must unzip the tar file into your directory of choice. Assuming you have installed Java, Maven and IntelliJ, you can open the project into IntelliJ by opening the pom.xml file contained in the root of the directory where you unzipped the project tar. So, if my tar is called parser_project.tar.gz and I unzip it into a directory called dps-parser, the pom.xml will be found in the dps-parser folder. After launching IntelliJ, click in the Open link as show in the image below.

In the following dialog, navigate the pom.xml file and click open. IntelliJ will automatically configure Maven and will setup the project for you.

5 Once the project is loaded, your project navigation area in IntelliJ will look like the image below.



6 The project is a standard Maven project. All the source files are found under **src/main/java**, and all the tests are found under **src/test/java**. You can use IntelliJ to view the code and execute all tests. To build a copy of the jar, you can open a command line, go to the root directory where the pom.xml was opened from and execute **mvn clean install**. Alternatively, you can call on the install life cycle link on the Maven plugin section of your IntelliJ editor. Below is an image.



7 After the build is complete, the final build artifact can be found under **target** in the root directory where the pom.xml is found. The final build is called dps-parser-0.9.1-SchematronParser.jar. This file will contain all of the necessary classes needed to use the parser as a dependency within your project.