Toward the Discovery and Extraction of Money Laundering Evidence from Arbitrary Data Formats using Combinatory Reductions

Alonza Mumford, Duminda Wijesekera
George Mason University
amumford@gmu.edu, dwijesek@gmu.edu

***Abstract*—The evidence of money laundering schemes exist undetected in the electronic files of banks and insurance firms scattered around the world. Intelligence and law enforcement analysts, impelled by the duty to discover connections to drug cartels and other participants in these criminal activities, require the information to be searchable and extractable from all types of data formats. In this overview paper, we articulate an approach — a capability that uses a *data description language* called Data Format Description Language (DFDL) extended with higher-order functions as a host language to XML Linking (XLink) and XML Pointer (XPointer) languages in order to link, discover and extract *financial data fragments* from raw-data stores not co-located with each other —see figure 1. The strength of the approach is grounded in the specification of a declarative compiler for our concrete language using a higher-order rewriting system with binders called Combinatory Reduction Systems Extended (CRSX). By leveraging CRSX, we anticipate formal operational semantics of our language and significant optimization of the compiler.**

***Index Terms*—Semantic Web, Data models, Functional programming, Data processing, Formal languages, Law enforcement**

I. Introduction

The approach leverages emerging developments in *data description languages* such as Data Format Description Language (DFDL) [1] for providing efficient representations of dense binary and textual data formats through vendor-neutral mechanisms. A DFDL schema allows raw data to be read from its native data format as an instance of a DFDL data model, and equivalently, composed to raw data from an instance of a DFDL data model. Within the context of this application, a DFDL schema represents a data repository containing any data format because the schema can refer to the local storage of the data it describes and provide instructions as to how that data may be read or written in its native form (e.g., bits, formats). Further, the concept outlines the addition of new abstractions to DFDL for defining the relationship and linkage between data fragments corresponding to different data files as well as for functions for extracting data fragments. The strength of this approach is grounded in the specification of a declarative parser-generator for these DFDL extensions using CRSX, which implements Klop's Combinatory Reduction Systems (CRS) with extensions to support the writing of compilers [2].

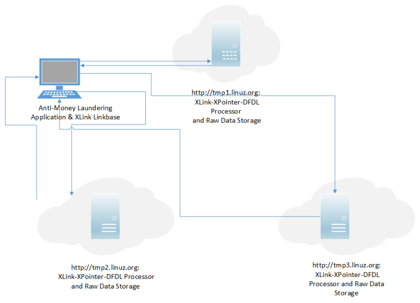

Fig. 1. An illustration of an anti-money laundering application that connects to multiple data storage sites. In this case, the native data format at each site differs, and a *data description language* extended with higher-order functions and linking/pointing abstractions are used to extract *data fragments* based on their ontological meaning.

II. Related Work

This work presents a multifaceted challenge that primarily breaks out into two areas. The first challenge is to provide a mechanism that can be used to describe and access any number of data formats. A class of data parsing languages commonly referred to as *data description languages* to include PADS[3] and DATASCRIPT have demonstrated this potential. This capability is not the same as offered by prescriptive data format languages such as JSON or even JSON-LD [4], which require compliance to a pre-specified structure and physical format. Descriptive languages have the advantage of being able to describe a data model's logical representation, which defines the semantics of the data, as well as its physical representation, which defines the methods by which its stored, without having to alter the target data from its initial format.

The second challenge is to combine the former capability with a lightweight-mechanism that supports metadata-based discovery and extraction of arbitrary data fragments from raw data stores without the system development and maintenance costs associated with major data conversion, and database storage and indexing. While popular data storage and extraction schemes such as Apache's Hadoop/MapReduce [5] and Accumulo/Big Table [6] provide a rich software-framework, they typically require data conversion for querying data fragments.

Unlike its *data description language* cohorts, DFDL extends a subset of XML Schema Description Language (XSDL), and augments the inherit logical model of the schema with DFDL annotations that are used to describe the physical representation of the data. In the same manner, our approach further extends DFDL's logical model with annotations for semantic-based traversal between local and remote resources that can be used to facilitate distributed discovery, parsing and extraction of raw data fragments. In addition to being interpretable by external XLink-XPointer processors, these semantic annotations also serve as instructions to the DFDL compiler for parser generation.

III. Methodology

The high-level methodology for this research proposal has been decomposed into four components. First, a plausible money laundering scheme is provided, and some inferences regarding how that scenario may appear in financial information is conceived. Second, the data records that are the focus of the money laundering investigation are examined for their format and subsequently, the data is modeled to understand its logical and physical specification. Third, the information within the data is conceptualized based on the "money scheme" and modeled for its semantic relationships using linking and pointing abstractions. Fourth, parts of the specification for the parser-generator are provided. Further details are provided in each respective section of the paper.

IV. Analysis of a Money Laundering Scheme

Consider an example of a money laundering scheme that could be used by any analyst to drive the discovery and extraction of data with evidentiary value from electronic insurance records. In the example scheme, the early cancellation of insurance policies with *return of premiums* has been used to launder money. Based on an assessment from the law enforcement community, this kind of insurance scheme has occurred where there have been [7]:

- A number of policies entered into by the same insurer (i.e. a person or company that underwrites an insurance risk) for small amounts and then canceled at the same time
- Return premium being credited to an account different from the original account, and
- Requests for return premiums in currencies different to the original premium.

V. Data Modeling

A. Inspection of the Insurance Account Records

The information referenced in the *return premium* scheme is made available in three electronic insurance records: the *policyAccountRecord*, *cancelRequestRecord*, and *creditRequestRecord*. At this step, the analyst's objective is to inspect the data such that he may describe a data model for each format type, which consist of a logical structure and physical representation.

In figure 2(a.1), the logical structure of the *policy AccountRecord* is a sequence complexType named by the identifier *PLCYACCT*. The *policyAccountRecord* type can be viewed as a data structure, where its value contains other values and its definition contains a datatype and identifier for each field. For example, *INSUR* is the identifier for a simpleType field named *Insurer*. The physical representation of the *policyAccountRecord* type is also describable. Delimiters, which are a sequence of characters, are used to specify the boundary between separate, independent areas in the text representation. For example, "/" is an infix separator between an identifier and value such as *PAYCUR* and *Peso (ARG)*, and white-space is a initiator and "//" is a terminator for each field. Also, the character-encoding scheme for the text representation is identified as ASCII. In figure 2(b.1), the *creditRequestRecord* type is given using some peculiar characters for separating the fields in the record. The *cancelRequestRecord* type uses the standardized JSON format (not shown). At this stage, the analyst discovers that each of the record types do not share a common format type such as XML or JSON.

B. DFDL-based Data Modeling for Parsing

At this step, the analyst models the logical model in the sequential order of the data file using the "logical datatypes & constraints" such as those in listed in figure 3. Then, the analyst maps the physical representation of each data file to its logical model, using the "physical representation properties" like those also shown in figure 3.

At compilation, a *DFDL schema* model generates a "program," which is essentially a parser and unparser. Upon parsing, if the input *policyAccountRecord* data file (ref: 2(a.1)), for example, satisfies all the constraints specified by the *policyAccountRecord.dfdl* schema file (ref: figure 4), it is considered to be valid according to the schema. More importantly, the DFDL parser generates a logical representation of that input data file, shown in figure 2(a.2), called the DFDL information set (infoset) or data model. In return, the DFDL information set can be used to unparse or generate a data file.

VI. Ontological Modeling

A. Inspection of the Application Domain Context

At this point, the analyst will apply his analytical reasoning to define the concepts that are relevant to the money laundering domain. In *ontological engineering*, a *concept* definition conveys the name of an evidentiary fact and its value data type. In figure 5, the generic kinds of conceptual abstractions are given along with corresponding examples of how those abstractions are applied within the example domain. In figure 6, a conceptual ontology of the "anti-money laundering" domain is given to show the kinds of classes and properties used in the domain. Generally, classes are identified by nodes and properties are identified by directed paths or *arcs*. This figure illustrates, for example, how these conceptual labels such as *creditRequestRecord* are structured taxonomically by composition (i.e., *hasPart*) and equivalence (i.e., *sameAs*) relations.

DFDL Logical Datatypes & Constraints (a)	**Context Free Grammar (CFG) example**	**Higher-Order Abstract Syntax (HOAS) example**
Structures (xs:complexType)	2 <dcl_xs> ::= "< " XS_COMPONENT <stmt>* "</" XS_COMPONENT ">" 11 XS_COMPONENT ::= "xs:complexType"	XsComponent[ComplexType]
Atomic data values (xs:simpleType)	11 XS_COMPONENT ::= "xs:simpleType"	XsComponent[SimpleType]
Ordering (xs:sequence or xs:choice)	11 XS_COMPONENT ::= "xs:sequence" | "xs:choice"	XsComponent[Sequence], XsComponent[Choice]
Occurences (xs:minOccurs or xs:maxOccurs)	5 <stmt> ::= XS_ATTRIBUTE "=" <xs_attribute_value> 16 XS_ATTRIBUTE ::= "xs:minOccurs" | "xs:maxOccurs" 8 <xs_attribute_value> ::= <xs_number>	ComponentAttribute[MinOccurs], ComponentAttribute[MaxOccurs]
DFDL Physical Representation Properties (b)		
Physical types (dfdl:representation)	5 <stmt> ::= DFDL_ATTRIBUTE "=" <dfdl_attribute_value> 18 DFDL_ATTRIBUTE ::= "representation" 7 <dfdl_attribute_value> ::= <dfdl_enum_value>	FormatProperty[Represetation]
Delimiters (dfdl:initiator, dfdl:separator, dfdl:terminator)	18 DFDL_ATTRIBUTE ::= "initiator" 7 <dfdl_attribute_value> ::= <dfdl_string_value> | <reg_exp_value>	FormatProperty[Initator], FormatProperty[Separator], FormatProperty[Terminator
Extraction of elements (dfdl:lengthKind)	18 DFDL_ATTRIBUTE ::= "lengthKind" 7 <dfdl_attribute_value> ::= <dfdl_enum_value>	FormatProperty[LengthKind]
Points of uncertainty (dfdl:discriminator)	3 <dcl_dfdl> ::= "<" DFDL_ADMIN <stmt>* ">" <dcl_dfdl> "</" DFDL_ADMIN ">" 15 DFDL_ADMIN ::= "dfdl:discriminator"	DfdlValidation[Discriminator]
Detecting occurrences (dfdl:occursCount)	18 DFDL_ATTRIBUTE ::= "occursCount" 7 <dfdl_attribute_value> ::= <non_neg_int_value> | <dfdl_exp_value>	FormatProperty[OccursCount]
XLink Properties (c)		
XLink type and label attributes (xlink:type or xlink:label)	5 <stmt> ::= XLK_ATTRIBUTE "=" <xlk_attribute value> 23 XLK_ATTRIBUTE ::= "xlink:type" | "xlink:label" 89 <xlk_attribute_value> ::= <xlktype_enum>	XlinkAttribute[XlkType], XlinkAttribute[XlkLabel]
DFDL Higher-Order Functions (HOF) (d)		
DFDL hof (dfdl_ext:filter or dfdl_ext:contains)	19 DFDL_HOF ::= "dfld_ext:filter" | "dfdl_ext:contains"	DfdlHof[Filter], DfdlHof[Contains]

Fig. 3. A specification of a DFDL compiler using CRSX performs *stepwise* transformations from the DFDL *concrete syntax* (shown in left column)) to an equivalent *higher-order abstract syntax* (HOAS) intermediate language (in right column). This transformation to the target language matches a context-free grammar (CFG) syntactic rule (in center column) to each unit of DFDL syntax and uses CRS-based *rewrite rules* to address semantic and optimization concerns.

```
<?xml version="1.0" encoding="UTF−8"?>
<xs:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:dfdl="http://www.ogf.org/dfdl/dfld−1.0/"
     xmlns:xlink="http://www.w3.org/1999/xlink" xmlns:dfdl_ext="http://linuz.org/dfdl_ext" xlink:type="
     extended">
   ...
   <xs:element name="policyAccountRecord" minOccurs="0" maxOccurs="unbounded" dfdl:lengthKind="implicit
        " xlink:type="resource">
      <xs:complexType>
            <xs:sequence dfdl:sequenceKind="ordered">
               <annotation>
                  <xs:appinfo source="http://www.ogf.org/dfdl/v1.0">
                     <dfdl:element representation="text" encoding="ascii" lengthKind="delimited"
                          sequenceKind="ordered" initiator="//" separator="/" separatorPosition="infix"
                          separatorPolicy="required"/>
                     <dfdl_ext:filter>
                        <dfdl_ext:param def=[function_definition]/>
                     </dfdl_ext:filter>
                  </xs:appinfo>
               </annotation>
               <xs:element name="policyAccountIdentifier" type="xs:string" dfdl:lengthKind="explicit"
                     dfdl:length="20" xlink:type="resource" xlink:href="http://linuz1/policyAccountSchema
                     .dfdl#xpointer(///policyAccountIdentifier[@xs:string=value])"/>
               <xs:element name="policyStartDate" ... xlink:label="policyStartDate" xlink:type="resource
                     " xlink:href="http://linuz1/policyAccountSchema.dfdl#xpointer(///policyStartDate[
                     @xs:date=value])"/>
               <xs:element name="policyHolder" ... xlink:label="policyHolder" xlink:type="resource"
                     xlink:href="http://linuz1/policyAccountSchema.dfdl#xpointer(///policyHolder[
                     @xs:string=value])"/>
               <xs:element name="policyInsurer" ... xlink:label="policyInsurer" xlink:type="resource"
                     xlink:href="http://linux1/policyAccountSchema.dfdl#xpointer(///policyInsurer[
                     @xs:string=value])"/>
               <xs:element name="payerName" ... xlink:label="payerName" xlink:type="resource" xlink:href
                     ="http://linuz1/policyAccountSchema.dfdl#xpointer(///payerName[@xs:string=value])"/>
               <xs:element name="payerCurrency" ... xlink:label="payerCurrency" xlink:type="resource"
                     xlink:href="http://linuz1/policyAccountSchema.dfdl#xpointer(///payerCurrency[
                     @xs:string=value])"/>
               <xs:element name="premiumAmount" ... xlink:label="premiumAmount" xlink:type="resource"
                     xlink:href="http://linuz1/policyAccountSchema.dfdl#xpointer(///premiumAmount[
                     @xs:string=value])"/>
            </xs:sequence>
         </xs:complexType>
     </xs:element>
      ...
</xs:schema>
```

Fig. 4. The *Policy Account Record* DFDL schema illustrates attributes and elements belonging to the XLink (e.g., *xlink:type*) and extended DFDL (e.g., *dfdl_ext:filter*) namespaces. Note that the DFDL *name* attribute (e.g., *name="premiumAmount"*) is a named reference type to the *data model* context, while the *xlink:label* attribute is a named reference type to the *ontological model* context as defined through XLink and anti-money laundering application domain. Therefore, if the application domain shifts to a new domain of inquiry, then a new schema with the same data model but different conceptual labels can be devised.

B. XLink-XPointer-based Concept Modeling for Data Linking, Addressing & Extraction

XLinks can be embedded within a XML document that contains links between other XML or non-XML documents. Since any DFDL schema is also an XML document, XLinks can be placed within DFDL schemas. In figures 4 and 7, both *policyAccountRecord.dfdl* and *moneyLaunderLinkbase.xml* specify an *extended link*, which defines a collection of resources and a collection of arcs between resources. Not only does each resource point to a *financial data fragment*, it also represents a *concept* within the *anti-money laundering ontology* shown in figure 6. In figure 4, *resource* attributes mark local resources. In this case, a local resource is equivalent to a locally-stored data fragment that can be read from or written to by an associated DFDL schema. Each remote DFDL schema contains multiple XLink annotations in order to identify the respective data fragment's location.

In order to create a connection between two resources and define the *meaning* of the relationship between them, an *anti-money laundering* linkbase is devised —see figure 7. A linkbase [8] provides the *location* and *label* information for each *financial data* resource. A linkbase describes links between resources by providing an *arc* defintion or traveral instruction. On line 12, for example, the traversal from the source resource, *linkbase*, to the destination resource, *http://tmp1.linuz.org/policyAccountRecord.dfdl* is defined. The XLink syntax grants for a number of attributes in the XLink namespace, as shown in figure 8.

For data fragment addressing and selecting, XPointer expressions are applied in the xlink:href attributes of various

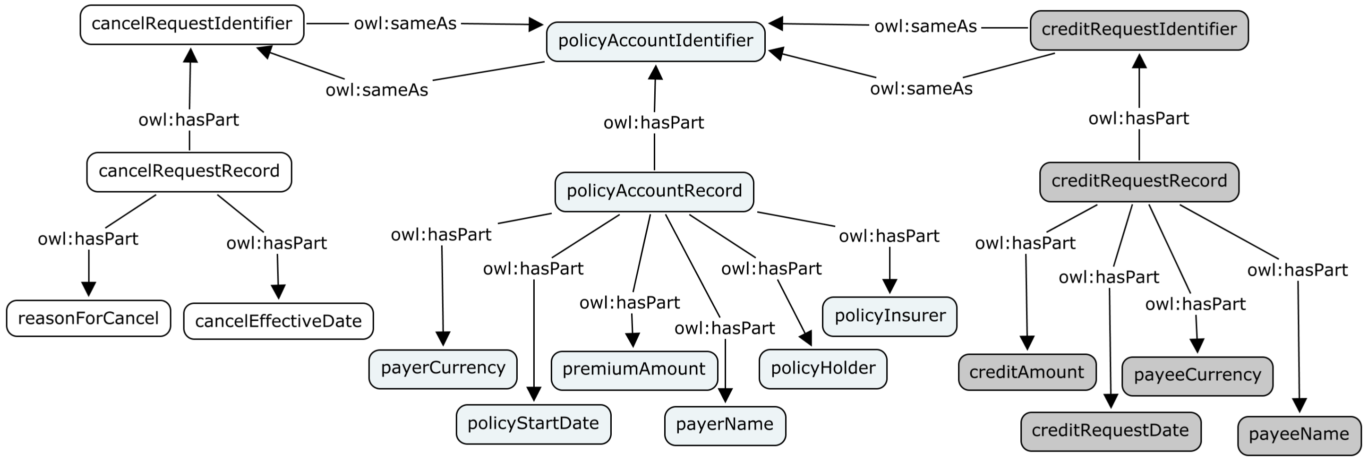

Fig. 6. The semantic relationships between the various concepts (data fragments) involved in a money laundering scheme may be illustrated in our ontological model of the domain.

elements in order to point to the data fields of the three data record types. In figure 4 on line 6, for example, the link address the *reasonForCancel* element with *xs:string* value is provided. In the example, XPointer depends on XPath expressions to point to resources. This *data linking* architecture is defined by a linkbase as well as corresponding XLink/XPointer-extended DFDL schemas for each record type and remote data stores.

VII. Extensions for Higher-Order Functions

This section focus on the utility of higher-order functions (HOFs), accompanied by XLink/XPointer constructs, in facilitating data extraction from native data repositories. In figure 4, an example of a *dfdl_ext:filter* function is illustrated. This higher-order function takes a *predicate* function and list, and returns to the *money laundering application* the list of elements that satisfy the predicate. Note another HOF construct is given in figure 3. To facilitate the data extraction, each DFDL processor is pre-complied, and the *function definition* is passed to the DFDL parser's runtime environment by way of a XLink arc traversal. As a result, the analyst is able to ignore the details of the data model and URI-based location of data fragment, and implement functional queries based on the conceptual modeling of the money laundering domain.

VIII. Compiler Specification for Data Description Language extended with HyperLink Structures and Higher-Order Functions (HOF)

This section addresses the initial specification of a DFDL compiler using CRSX to perform syntactic analysis, semantic analysis and transformation of a DFDL instance into a higher-order abstract syntax (HOAS) intermediate language. As a mathematical rewriting method, CRSX is used to formalize a *stepwise* transformation and evaluation of the concrete DFDL language into a highly optimized parsing application or raw data writer. By way of a recursive tree traversal over a DFDL schema instance, the CRSX-based compiler steps through each of the following compilation phases:

A. Syntax to CFG Production Rule for DFDL

First, the names for all components of the DFDL schema language are specified. In this case, a *component* is anything that can be defined or declared in the DFDL vocabulary (e.g., an element, a simple type or a complex type). As in figure 3, more than four hundred components belonging to the DFDL namespace were specified. Second, for each specified name in the language, explicit transformation mappings were made from the DFDL schema syntax onto context-free grammar (CFG) structures.

A DFDL schema fragment is illustrated in figure 3, by the xs:complexType declaration example. This particular code fragment is related to defining a complexType element that contains other elements such as choice and sequence. As the CFG for complexType is used to guide the parse into each fragment of the DFDL schema syntax, essentially an invocation of a unit of instruction is sent to the DFDL compiler to be executed. The XML markup in the DFDL schema express structure. The process of parsing a DFDL schema identifies elements and attributes, and creates an abstract image (i.e., the DFDL data model) that corresponds to the DFDL schema structure.

Consider, for example in figure 4, one of the DFDL element declarations. On "line 9", it declares to be an instance of a 'dfdl:element' element type. The DFDL element must comply with the structure and attribute constraints stated by the element type in order to qualify as an instance. In this case, membership of an instance of an DFDL element or attribute to a type is determined by validation of a DFDL processor that is tasked to accept or reject DFDL instances as well as data mapped to those instances. In the case of the syntax for XLink, XPointer and higher-order functions (ref: figure 3) used in the extended DFDL schema, CFG production rules are also prescribed in a similar manner.

Further, a parallel exits between the DFDL schema and the data file which validated against the DFDL schema. Use of a CFG is the approach taken by the compiler for providing

```
<?xml version="1.0" encoding="UTF-8"?>
<linkbase xmlns:xlink="http://www.w3.org/1999/xlink" xlink:linkbase="http://www.w3.org/1999/xlink/
    properties/linkbase">
  <link xlink:type="extended" xlink:title="moneyLaunderLinkbase">
    <!-- Linkbase loads on extraction request. -->
    <basesloaded>
    <startrsrc xlink:label="filter_spec" xlink:href="/local/filter_spec.xml#params" />
    <linkbase xlink:label="linkbase" xlink:href="/local/linkbase.xml" />
    <load xlink:from="spec" xlink:to="linkbase" actuate="onRequest" />
    </basesloaded>

    <!-- Arcs between linkbase and DFDL-data stores. -->
    <invokeStoreArc xlink:type="arc" xlink:arcrole="linkbase" xlink:from="linkbase" xlink:from="
        PolicyAccountRecord"/>
    <invokeStoreArc xlink:type="arc" xlink:arcrole="linkbase" xlink:from="linkbase" xlink:from="
        CancelRequestRecord"/>
    <invokeStoreArc xlink:type="arc" xlink:arcrole="linkbase" xlink:from="linkbase" xlink:from="
        RefundRequestRecord"/>

    <!-- Locator elements. -->
    <loc xlink:type="locator" xlink:label="PolicyAccountRecord" xlink:href="http://tmp1.linuz.org/
        policyAccountSchema.dfdl"/>
    <loc xlink:type="locator" xlink:label="PolicyAccountIdentifier" xlink:href="http://tmp1.linuz.
        orglinuz1/policyAccountSchema.dfdl#xpointer(////policyAccountIdentifier[@xs:string=value])"/>
    <loc xlink:type="locator" xlink:label="PolicyStartDate" xlink:href="http://tmp1.linuz.org/
        policyAccountSchema.dfdl#xpointer(////policyStartDate[@xs:date=value])"/>
    <loc xlink:type="locator" xlink:label="PolicyHolder" xlink:href="http://tmp1.linuz.org/
        policyAccountSchema.dfdl#xpointer(////policyHolder[@xs:string=value])"/>
    <loc xlink:type="locator" xlink:label="PolicyInsurer" xlink:href="http://tmp1.linux.org/
        policyAccountSchema.dfdl#xpointer(////policyInsurer[@xs:string=value])"/>
    <loc xlink:type="locator" xlink:label="PayerName" xlink:href="http://tmp1.linuz.org/
        policyAccountSchema.dfdl#xpointer(////payerName[@xs:string=value])"/>
    <loc xlink:type="locator" xlink:label="PayerCurrency" xlink:href="http://tmp1.linuz.org/
        policyAccountSchema.dfdl#xpointer(////payerCurrency[@xs:string=value])"/>
    <loc xlink:type="locator" xlink:label="PremiumAmount" xlink:href="http://tmp1.linuz.org/
        policyAccountSchema.dfdl#xpointer(////premiumAmount[@xs:string=value])"/>
    ...

    <!-- Relationship between policy account, cancel request and refund request identifiers. -->
    <invokeIdArc xlink:type="arc" xlink:arcrole="owl:sameAs" xlink:from="PolicyAccountIdentifier"
        xlink:to="CancelRequestIdentifier"/>
    <invokeIdArc xlink:type="arc" xlink:arcrole="owl:sameAs" xlink:from="PolicyAccountIdentifier"
        xlink:to="RefundRequestIdentifier"/>
    <invokeIdArc xlink:type="arc" xlink:arcrole="owl:sameAs" xlink:from="CancelRequestIdentifier"
        xlink:to="PolicyAccountIdentifier"/>
    <invokeIdArc xlink:type="arc" xlink:arcrole="owl:sameAs" xlink:from="CancelRequestIdentifier"
        xlink:to="RefundRequestIdentifier"/>
    <invokIdArc xlink:type="arc" xlink:arcrole="owl:sameAs" xlink:from="RefundRequestIdentifier"
        xlink:to="PolicyAccountIdentifier"/>
    <invokeIdArc xlink:type="arc" xlink:arcrole="owl:sameAs" xlink:from="RefundRequestIdentifier"
        xlink:to="CancelRequestIdentifier"/>

    <!-- Relationship between policy account record and its parts. -->
    <invokeParArc xlink:type="arc" xlink:arcrole="owl:hasPart" xlink:from="PolicyAccountRecord"
        xlink:to= "PolicyAccountIdentifier" />
    <invokeParArc xlink:type="arc" xlink:arcrole="owl:hasPart" xlink:from="PolicyAccountRecord"
        xlink:to= "PolicyStartDate" />
    <invokeParArc xlink:type="arc" xlink:arcrole="owl:hasPart" xlink:from="PolicyAccountRecord"
        xlink:to= "PolicyHolder" />
    <invokeParArc xlink:type="arc" xlink:arcrole="owl:hasPart" xlink:from="PolicyAccountRecord"
        xlink:to= "PolicyInsurer" />
    <invokeParArc xlink:type="arc" xlink:arcrole="owl:hasPart" xlink:from="PolicyAccountRecord"
        xlink:to= "PayerCurrency" />
    <invokeParArc xlink:type="arc" xlink:arcrole="owl:hasPart" xlink:from="PolicyAccountRecord"
        xlink:to= "PayerName" />
    <invokeParArc xlink:type="arc" xlink:arcrole="owl:hasPart" xlink:from="PolicyAccountRecord"
        xlink:to= "PremiumAmount" />
    ...
  </link>
</linkbase>
```

Fig. 7. The money laundering linkbase.

```
(a.1) Input "policy account record" data:

PLCYACC/741032−1071//
DATE/2013−09−28//
PLCYHLD/Allegier, Cox & Associates, Inc.//
INSUR/ALI Corp.//
PAYER/Grupo Palermo S.A.//
PAYCUR/Peso (ARG)//
PRMAMT/42004.98//

(a.2) DFDL generated XML model:

<policyAccountRecord>
    <policyAccountIdentifier>741032−1071</
          policyAccountIdentifier>
    <policyStartDate>2013−09−28</policyStartDate>
    <policyHolder>Allegier, Cox & Associates Inc.
         </policyHolder>
    <policyInsurer>ALI Corp.</policyInsurer>
    <payerName>Grupo Palermo S.A.</payerName>
    <payerCurrency>Peso (Argentine)</
          payerCurrency>
    <premiumAmount>42004.98</premiumAmount>
</policyAccountRecord>

(b.1) Input "credit request record" data:

[[[[[[[CRDREQ%]741032−1071%]
CRDATE%]2013−11−02%]
PAYEE%]Allegier, Cox & Associates, Inc.%]
PAYCUR%]USD%]
CRDAMT%]5000.00%]

(b.2) DFDL generated XML model:

<creditRequestRecord>
    <creditRequestIdentifier>741032−1071</
          creditRequestIdentifier>
    <creditRequestDate>2013−11−02</
          creditRequestDate>
    <payeeName>Allegier, Cox & Associates, Inc.</
          payeeName>
    <payeeCurrency>USD</payeeCurrency>
    <creditAmount>5000.00</creditAmount>
</creditRequestRecord>
```

Fig. 2. *Policy Account* and *Credit Request* records. A DFDL *parser* accepts raw data (e.g., in (b.1)) and generates a DFDL data model (in (b.2)). Symmetrically, a DFDL *unparser* uses a DFDL data model generate equivalent raw data.

syntactic checking. Ultimately, the aggregate of four hundred or so CFG production rules will partition any DFDL schema into a set of components, where each component can match against an unique fragment of a DFDL schema. The CFG has been designed to ensure that any DFDL schema be reduced to its normal form in order to provide a specific name for each component of the DFDL schema specification.

B. CFG Production Rule to HOAS for DFDL

Next, rules for transformation of the CFG into the HOAS intermediate language are prescribed in the DFDL compiler implementation. Note in figures 3 and 4, a HOAS *constructor* name is shown for each provided CFG. A HOAS representation is the equivalent to an abstract syntax tree (AST), and it serves as the intermediate representation for further transformation and optimizations of a DFDL schema

Concept	Example
Classes	propertyAccountRecord, cancelRequestRecord and creditRequestRecord (ref: figs. 5 and 8)
Instances	An instance of a propertyAccountRecord is one bearing "741032-1071" as the policyAccountIdentifier (ref: fig 2, a.1).
Relations: hasPart, sameAs	The three properties, policyAccountIdentifier, cancelRequestIdentifier, and creditRequestIdentifier are equivalent (sameAs) (ref. figs. 8 and 9).
Properties	policyAccountIdentifier, policyStartDate, policyHolder, policyInsurer are properties of a policyAccountRecord (ref: figs. 5, 8 and 9).
Values	"USD" and "5000.00" are the values of payeeCurrency and premiumAmount respectively for a particular instance of a creditRequestRecord (ref: fig. 2, b.2).
Rules	The three properties, policyAccountIdentifier, cancelRequestIdentifier, and creditRequestIdentifier are equivalent (sameAs) if they evaluate to the same value, for example, "741032-1071".

Fig. 5. This table gives an explanation of the anti-money laundering conceptual ontology and illustrates where these concepts are defined within XLink-extended DFDL schemas and linkbase. [2]

Attribute	Value	Description
xlink:type	extended	Parent element, which defines a complex link in which multiple links can be combined based on other attributes.
	resource	Child element of extended-Type element, which provides a local resource to associate with the link.
	locator	Child element of extended-Type element, which specifies the location of a remote resource associated with the link.
	arc	Child element of extended-Type element, which define traversal rules between the link's associated resources.
xlink:label	NCName	Traversal attribute of extended-, resource-Type elements, which provides a reference (of itself) to arc-Type in composing a traversal arc.
xlink:from, xlink:to	NCName	Traversal attributes of arc-Type element, which define the source and target resources of the arc link.
xlink:href	URI	Attribute of locator-Type element, which provides the data that helps an XLink application to locate a remote resource .
xlink:role	URI	Semantic attribute of extended-, resource-Type elements, which indicates a property of the resourcein a computer readable-form.
xlink:arcrole	URI, linkbase	Semantic attribute of arc-Type element, which coincides with the [RDF] view of a property, where the role can be understood as HAS relationship between the starting-resource and the ending-resource.
#xpointer		Creates XPointer fragment links with syntax: #xpointer(id("<value>"))

Fig. 8. XLink elements and attributes used in the anti-money laundering application. [4]

instance. As illustrated in the HOAS column of figure 3, a HOAS surmounts the difficulty of having to define name binding constructs in the abstract syntax [9]. For example, *XsComponentType* is a syntactic category and *ComplexType*, which is a name of a type that has membership to that category, is bound using the [ ] syntax. All the DFDL types

```
TERM ::=(
  Let[ VALUE, TYPE, x::VALUE . TERM ];
  Lam[ VALUE, TYPE, x::VALUE . TERM ];
  Context[ ];
  Element[KIND, $List[ATTRIBUTE], $List[
      DFDL_PROPERTY], $List[XLP_ATTRIBUTE], TERM
      ];
  Pair[ TERM, TERM ];
  Nil;
  T;
  T-Attribute
  T-BuildSchema
  T-BuildElement
  XML-Visit[ XLink-XPointer ]
);
```

Fig. 9. Consider our top level terms for the DFDL CRSX system after normalization. The terms are written in the form of a higher-order abstract syntax.

(including the new extensions) are derived from *syntactic sorts* or *syntactic categories*, which are normalized into the top level terms of the DFDL HOAS shown in figure 9. The objective is to convert all CFG derivations into the syntactic sorts. For example, in the normalization process, the "XsComponentType[ComplexType]" would be transformed to the term: "Element[XsComponentType[ComplexType]], ...]".

C. HOAS to CRSX Rewrite Rule for DFDL

CRSX *rewrite rules* are specified to address the semantic and optimization transformation and evaluation of the DFDL compiler. An example *rewrite rule* is given (in figure 10) that defines explicit scoping of XS COMPONENT ATTRIBUTE(s) in the DFDL specification. The meaning of the *rewrite rule* syntax is provided in [10]:

(a) A *rewrite rule* takes the form:

$$name[options] : pattern \to contraction$$

, where name should be a *constructor* and the pattern and contraction should be *terms*; where

(b) *XsComponent-Attribute* and *Copy* are constructors, which take an optional ordered or positional *parameter* list in immediately following [ ]s, where each parameter is itself a *term*, and called a *subterm*, uncapitalized words (e.g. x and foo) denote *variables*; and where

(c) a *Lambda-construction* with a single subterm binds the variable x (before the .) and contains a single construction with two subterms that both are occurrences of x.

In CRSX, we model this as Let[E1, x.E2], i.e. let all *occurrences* of x in the body of function E2 be replaced or substituted by x, where x := E1. This allows explicit scoping. The entire compiler is specified as rule system is written as a sequence of rules each terminated by ; semicolon;

IX. Conclusion and Future Work

In this paper, an approach is given for a lightweight-capability that supports metadata-based discovery and extraction of *informational* fragments from raw data stores without having to alter the information's native data format.

```
XsComponent-Attribute[Copy[#QName]]
:
{#Env; #QName: ComponentAttribute[#kind]}
XML-Attribute[ #prefix, #QName, #Value, ok.#
    Continuation[ok]]
->
{#Env}
 Let[ #Value, a.{#Env} AddXsAssoc[#prefix, #
     QName, a, ok.#Continuation[ok]]
;
```

Fig. 10. Illustrates a CRSX *rewrite rule* that defines explicit scoping of XSD attributes. CRSX *rewrite rules* are an equivalent form of a programming language's *operational semantics*. In this case, we define the operational semantics from the perspective of compilation.

This approach offers an advantage over popular data extraction schemes such as Apache Hadoop platform that require the conversion of data into a *prescriptive* data format. The approach extends an existing *data description language* with linking/-pointing constructs and higher-order functions. An overview of the DFDL compilation is provided using concepts from programming language design and formal rewriting systems. The future work includes: specifying the transformation and evaluation of the DFDL/XLink/HOF specification into parser combinator form; investigating the operational semantics of the higher-order function (HOF) and linking abstractions in order to optimize distributed data extraction; and generating comparative performance metrics.

Acknowledgment

The authors would like to thank Dr. Kristoffer Rose for his valuable guidance on Higher-order Rewriting for Compiler Specifications and the Combinatory Reduction System with Extensions.

References

[1] O. D. WG, S. M. Hanson, and A. W. Powell, "Data format description language (dfdl) v1. 0 specification."

[2] K. H. Rose, "Crsx–an open source platform for experiments with higher order rewriting," *HOR 2007*, p. 31, 2007.

[3] K. Fisher and R. Gruber, "Pads: a domain-specific language for processing ad hoc data," in *ACM Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 295–304.

[4] M. Sporny, G. Kellogg, and M. Lanthaler, "Json-ld 1.0-a json-based serialization for linked data," *W3C Working Draft*, 2013.

[5] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[7] I. A. of Insurance Supervisors, *Examples of Money Laundering and Suspicious Transactions Involving Insurance*. International Association of Insurance Supervisors, 2004. [Online]. Available: http://books.google.com/books?id=bSvoHAAACAAJ

[8] S. DeRose, E. Maler, D. Orchard, and N. Walsh, "Xml linking language (xlink) version 1.1, w3c recommendation 06 may 2010," 2010.

[9] F. Pfenning and C. Elliot, "Higher-order abstract syntax," in *ACM SIGPLAN Notices*, vol. 23, no. 7. ACM, 1988, pp. 199–208.

[10] K. H. Rose, "Crsx-combinatory reduction systems with extensions," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 10. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2011.