

XML Framework for Decision diagrams

1. Introduction

The purpose of this paper is to describe XML based framework for manipulation of Decision diagrams.

Decision diagram is a data structure that permits efficient, in terms of space and time, representation of discrete functions [1]. Most widely used kinds of Decision diagrams are Binary Decision diagrams (BDDs) used for representation of binary-valued logic functions [2]. They are nowadays a standard part of many CAD systems for digital systems design and an efficient tool in many signal processing tasks where manipulation and calculations with functions of a large number of variables are required.

There are standard methods for programming of Decision diagrams. For example, CUDD library is used in many applications as a standard in this area [3]. However, most of these software solutions use Decision diagrams only internally and do not offer a possibility of storing the actual Decision diagram structure for possible further use or exchange with other similar software. Furthermore, even if there is an option to store more information about generated Decision diagram it is mostly on rudimentary level. Currently there is no standard for storing and exchange of Decision diagrams.

In our view this state of affairs is severely limiting the interoperability and comparison of operation of these software solutions. This article is meant as the first step towards solution of this problem.

The possibilities for practical applications of such standardized framework are numerous. For instance, such a solution would permit exchange of generated diagrams between various applications. Another possible application of such framework would be to convert some of optimal forms of Decision diagrams of the standard benchmark functions in XML (Extensible Markup Language) based format [4]. This would make the task of evaluation of results of various applications easier and more precise.

Furthermore, if a logic function can be in the minimal form represented with its optimal Decision diagram only that Decision diagram needs to be stored and transmitted. This feature of Decision diagrams can yet prove to be of great value for future applications.

XML (Extensible Markup Language) is a special-purpose markup language created for the task of describing various forms of data structures. XML provides a convenient way of describing flexible and robust formats for storage and handling of various kinds of data with special emphasis on types of data with complex internal structure.

As Decision diagrams in essence represent exactly such kind of data structure the XML represents a natural choice.

Inherent properties of XML permits us to develop the framework flexible enough to encapsulate many various existing forms of Decision diagrams [5]. It also provides us with possibility to further extend this standard in future to accommodate possible new forms of Decision diagrams.

The existence of XML parsers for all major programming platforms would make the implementation of this framework for any purpose extremely easy.

The XML SOAP (Simple Object Access Protocol) technology aimed exactly at handling data with tree like hierarchical structure provides many ready made functions for manipulation and handling of such data.

This solution could be linked on various levels with other members of broad XML family such as SVG and MathML which would provide various new possibilities interesting primarily for scientific workers in this field, as instant rendering of a Decision diagram in SVG vector format for example.

1.2 XML Basic Concepts and the Framework

As we have already stated, XML or Extensible Markup Language represents a special purpose markup language. Its main purpose is description of formats for recording of various kinds of data with complex internal structure and possibly mixed information content.

Rather than trying to specify a rigid data structure for each possible kind of data and each particular application, XML specifies a simple set of rules that each XML based file format should use in order to be understood by XML enabled software.

Each XML document is a structured text, while the information it contains is presented in form of hierarchically arranged elements. The type of elements and their internal structure is specified separately according to XML rules.

XML 1.1 the current version of XML standard, is a result of work of XML Core workgroup at W3 consortium.

Although the structure of XML documents is usually very intuitive and self documenting, thus, understandable to humans, XML documents are primarily designed to be processed by software. XML parser represents a software solution capable of reading and understanding XML documents. The aim of XML parser is to extract the data from XML document following its internal structure and to transfer that data into data structures specific for the particular application.

XML parsers usually come in form of programming libraries as a part of software development environments. XML parsers exist for all currently relevant platforms and programming environments. In this work, we have been using Microsoft XML parser which is a part of Microsoft .NET framework.

Closely associated with XML document, we find the term of well formed document. Well formed XML document is a document that consists entirely of well formed XML elements. In terms of XML syntax, a well formed element is some data encapsulated by XML specific brackets (i.e. <element_name> start bracket and </element_name> end bracket), where the name of the element is defined by user.

From the point of view of XML parser, nothing more is required for a particular XML document to be readable.

For the sake of standardization, some additional specifications are usually required.

An XML Schema represents the specification of the internal structure of an XML document designed for a particular application. An XML Schema specifies elements of the XML document, their internal structure, attributes associated with them, the type of data each element stores, and internal hierarchy of elements.

If one particular XML document conforms to description provided in a particular XML Schema, it is said to be valid. Validity of the XML document can be confirmed by XML parsers either at reading operation or when the XML file is being generated by parser. The confirmation of validity of XML documents is one more tool to ensure proper exchange of data between various software systems.

XML Schema, XML documents that conform to it, and XML parser, represent three necessary components of each XML framework.

The XML framework proposed in this paper consists of several separate components. We will list and briefly describe the purpose of each one of them. The structure of some of the components is explained in detail in further chapters. Examples of code are provided in the appendix section of this document.

The main components of the XML framework introduced are:

1. XML Schema document of Decision diagrams designed specifically for this purpose,
2. XML document representing the Decision diagram, created based on the XML Schema for Decision diagram. Several examples of such documents are provided in the appendix.
3. An example application for handling XML documents containing Decision diagram developed in the Microsoft .NET development environment, constructed on the base of Microsoft XML parser as an illustration of how one such application should be constructed. how such an application should be constructed.

2. Basic architecture

There are at least two possible ways of approaching the problem of XML representation of Decision diagrams, both of them having its advantages and weaknesses.

The first approach would be to follow the intuitive way of representing Decision diagram in a hierarchical tree-like structure as they are usually represented on paper. This would be a human oriented approach to the problem. However this kind of organization of data structure would permit us to store Decision diagrams in their complete form, that is, in the form where each node in the tree has exactly one parent. It is obvious that in this way we cannot represent Decision diagram in their optimal/minimal form. A possibly significant overhead of the storage space exists because identical sub graph elements need to be stored multiple times. **Chapter 3.** deals with this approach.

The second approach would be to mimic the way Decision diagrams are usually represented by the software aimed at working with them and just give the firm standard based on the XML for this representation. This would be the programming oriented approach. Although slightly less intuitive, this architecture of data structure would permit us to store Decision diagrams in their optimal form and, thus eliminate overhead of multiple stored sub-graphs. Minor modifications of existing software would be needed to implement this XML framework, since data structures would be compatible. **Chapter 4.** deals with this approach.

3. XML framework for simple BDD representation

In this chapter, we will describe a simple way to implement basic Binary Decision diagram tree structure using standard XML. For the sake of simplicity, we shall limit the discussion in this chapter to Binary Decision diagrams only. Further chapters will deal with more complex forms of Decision diagrams.

The basic concept of nested objects native to XML corresponds extremely well to *parent -> child* architecture of BDD. We will make an extensive use of this fact.

We can view each BDD tree as an element connecting two sub-graph elements, where each sub-graph element itself consists of two similar sub-graph elements and so on until we reach the level of terminal nodes.

This recursive structure renders well in XML.

We define the complex element type in the following way (an extract from the corresponding XML schema):

```
<xsd:complexType name="nodeType" mixed="true">
  <xsd:sequence>
    <xsd:element name="left" type="dd:nodeType" minOccurs="0" maxOccurs="1"/>
    <xsd:element name="right" type="dd:nodeType" minOccurs="0" maxOccurs="1"/>
    <xsd:element name="terminal" type="xsd:integer" minOccurs="0" maxOccurs="1"/>
  </xsd:sequence>
  <xsd:attribute name="ID" type="xsd:integer" use="required"/>
  <xsd:attribute name="level" type="xsd:integer" use="required"/>
</xsd:complexType>
```

This definition is recursive in nature, each element of this data type can have up to two sub-elements of the same type, representing its sub-graphs. In turn they consist of two additional sub-graph elements and so on to the terminal node level. One or both of these sub-graph elements can be omitted if structure of BDD demands so. Elements containing no sub-elements represent the terminal nodes.

The additional information that each node must contain can be stored in attributes of the XML element or in additional sub-elements.

This is a very simple and elegant architecture very intuitive in its nature. Furthermore, it is well adopted to tools aimed at processing and parsing XML documents, namely XmlSerializer library in Microsoft .NET framework making it easy to use the standard functions for accessing the stored data implemented in this and similar libraries.

The resulting XML code is self-documenting and easy to read even to human observer. (Appendix 2.)

However this architecture is very limited in many ways.

First of all it imposes a very rigid rule in terms of the number of descendants of each node. We have here explained the architecture on the example of a BDD. It can easily be extended to Multi-valued Decision diagrams [1], the maximal number of descendant nodes must be however specified in advance and be the same for each node of the graph.

This disadvantage can be overcome by more or less difficulties by introduction of additional XML elements.

The other main disadvantage is that this architecture is strictly three based, that is, it assumes that each sub-graph/node element has exactly one parent. This may not seem as a disadvantage at first, but we must

remember that in their optimal form Decision diagram break this rule. In optimal form of Decision diagrams, all identical sub-graphs are stored only once with multiple links pointing at them and, thus, with multiple parents. This is the essence of using Decision diagrams as means of efficient representation of binary functions. We address this limitation in the next chapter.

4. XML framework for optimal Decision diagram representation

In many cases, Decision diagrams consist of identical sub-tree segments. In order to utilize in a proper way an expressive power of Decision diagrams to represent logical functions in optimal form, we must make use of this fact. Identical sub-tree segments must be stored only once in our XML document. This would imply that one such segment would have more than one path pointing to it from more than one parent node. It is clear that this violates the basic assumption of a tree structure, on which the idea of nested objects is founded. As we have seen, the concept of nested elements is native to XML, and it was very easy for us to exploit this advantage as described in chapter 1.

However, the implementation of this modified concept will require a slightly different approach. In its optimal form, a Decision diagram is represented by, from the point of data structure theory, a more complex structure of the ordered graph.

As the number of parent nodes pointing to each specific node is not known in advance, we cannot try to impose any rigid structure for our data structure, in sense of limiting the number of incoming pointers. Since we already have to implement such a solution for parent nodes, it is easy to use the same solution for descendant nodes of the node in question, that is, for outgoing links, thus removing the limit on the number of descendants as well. An architecture that would facilitate such capabilities will permit us to store any kind of Decision diagram, i.e. BDD, MDD, etc.

To accommodate these demands, we propose the following data structure architecture.

Information about parents and descendants of each node is stored in form of two separate, structurally identical, linked lists. Each element of such linked list would consist of two fields; one a pointer to parent/child node of Decision diagram would represent real link existing in the Decision diagram, the other pointer to another field in the linked list would serve an internal function of navigating through the list.

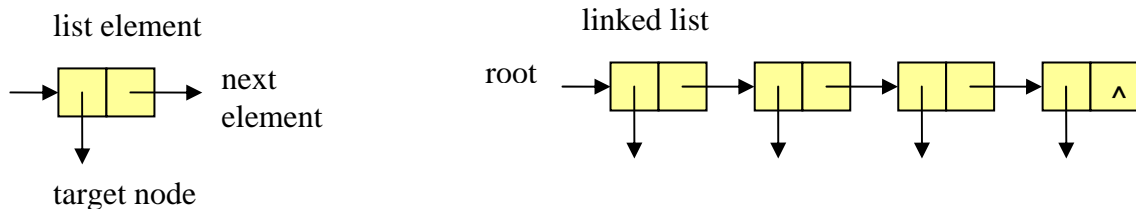


Fig 1. Aan element in parents/children linked list and the list itself.

Two such linked lists are created and attached to each node in a Decision diagram, representing parents and descendants of that particular node. Each node element in our architecture contains two pointers to these lists.

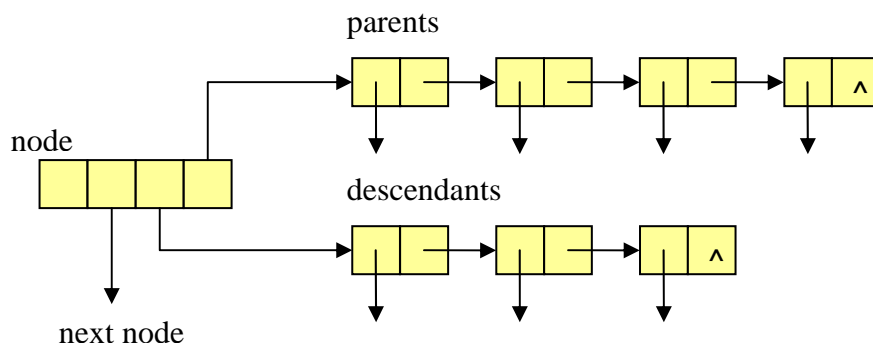


Fig 2. Node element and lists of parents and descendants associated with it.

The nodes themselves are stored in the form of linked lists also; where links between elements of these lists do not have any semantic meaning in the context of the Decision diagram itself, but represent just the means of accessing the elements in data structure. Fig 3. gives a better view of data structures discussed.

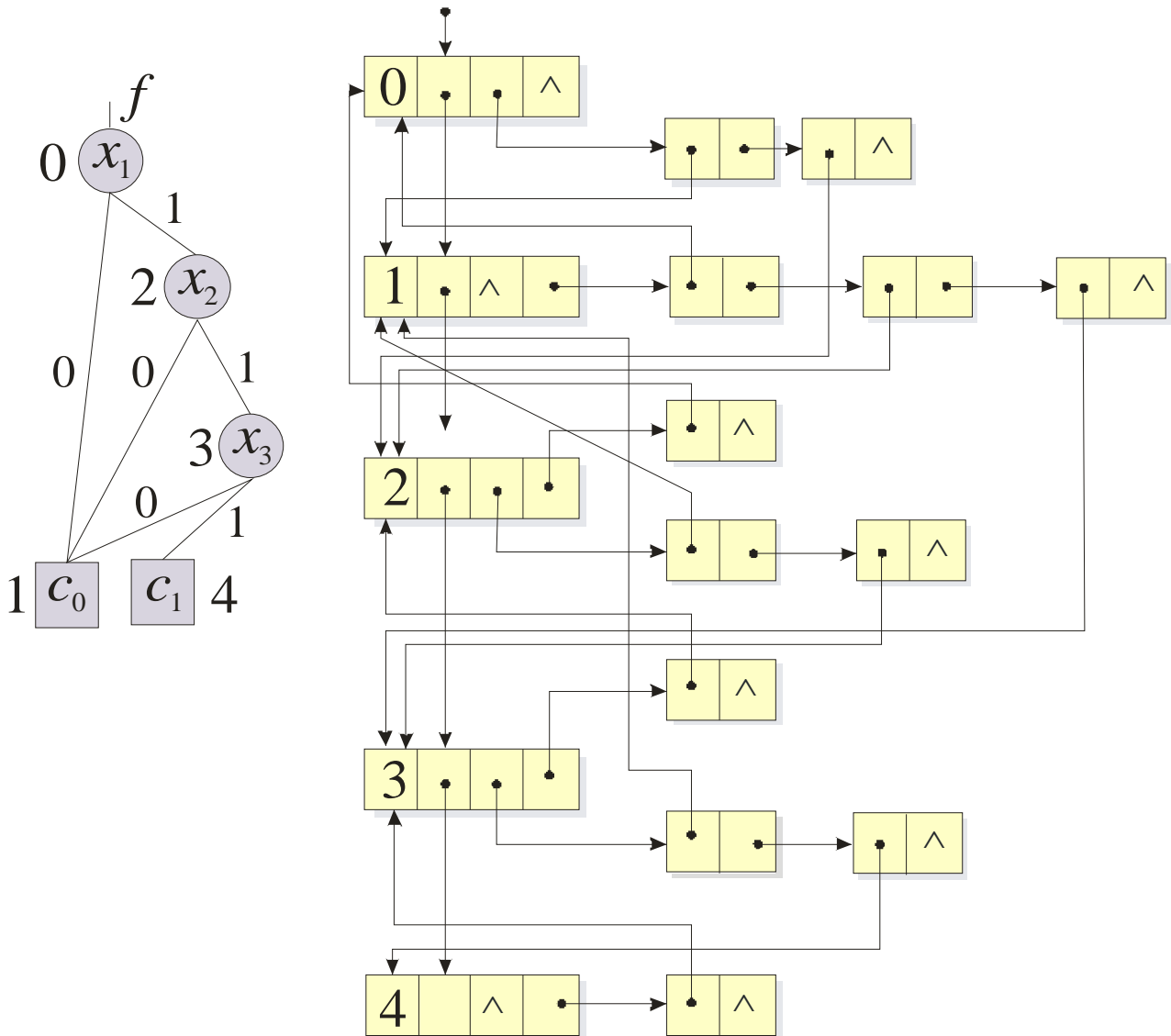


Fig 3. BDD and the corresponding data structure.

All the real links, the links that have some semantic meaning in the context of Decision diagrams, are contained in the linked lists of nodes (Fig. 4).

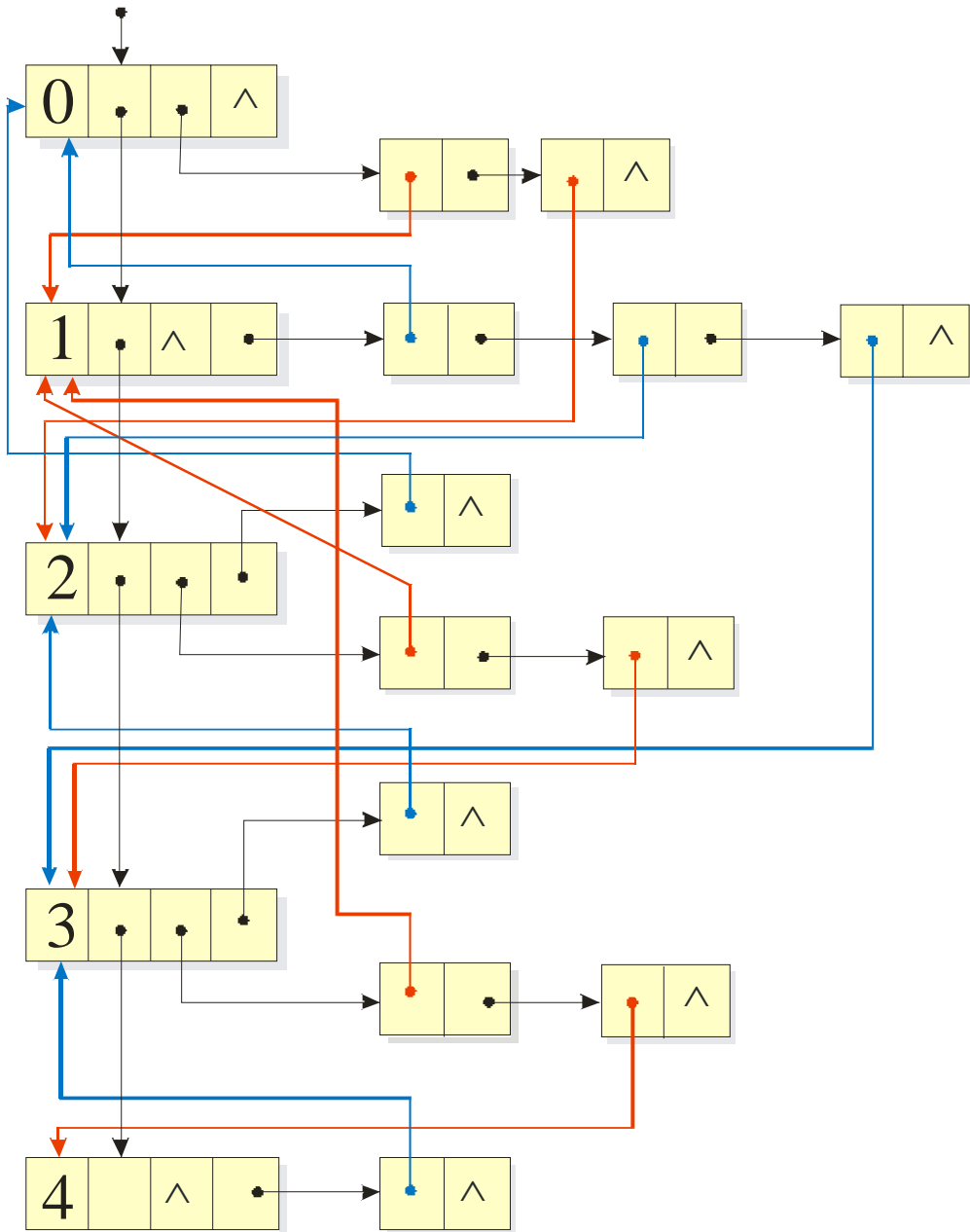


fig 4. Parent(blue) and Child(red) links in data structure

An additional attribute field of each node element is dedicated to storing any additional information, i.e., a numerical value, matrix, function, etc., that we want to associate with the node.

This architecture is identical to the standard way graphs are usually described in data structure theory. This permits us to use the standard, well adapted and efficient algorithms for addition and removing of nodes, accessing elements of the graph, etc.

It is also consistent with the usual way Decision diagram have been represented in various previous applications aimed at dealing with them [6].

4.0 XML implementation

Although the implementation of the above described in XML is not so intuitive like implementation of simple tree structure, XML still provides efficient ways of accomplishing this task.

We define basic complex element type NodeType in the following way:

```
<xsd:complexType name="NodeType" >
  <xsd:sequence>
    <xsd:element name="next" type="NodeType" minOccurs="0" maxOccurs="1"/>
    <xsd:element name="children" type="PointType" minOccurs="0" maxOccurs="1"/>
    <xsd:element name="parents" type="PointType" minOccurs="0" maxOccurs="1"/>
  </xsd:sequence>
  <xsd:attribute name="ID" type="xsd:integer"/>
  <xsd:attribute name="Level" type="xsd:integer"/>
  <xsd:attribute name="Value" type="xsd:integer"/>
</xsd:complexType>
```

The linked list of nodes is represented by the use of recursive declaration of this element type where each NodeType element has one nested element of the same type in its structure, i.e., the following element in the list of nodes.

In much the same way we implement elements of simpler linked lists representing parents and descendants of each node.

```
<xsd:complexType name="PointType">
  <xsd:sequence>
    <xsd:element name="next" type="PointType" minOccurs="0" maxOccurs="1"/>
  </xsd:sequence>
  <xsd:attribute name="point" type="xsd:integer"/>
</xsd:complexType>
```

It should be noted that we use the same mechanism of nested elements to represent links in the linked lists.

Each PointType element of a linked list has one element of the same type as a nested object. Each NodeType on the other hand contains two PointType elements as its sub-elements, representing the roots of parents and children linked lists.

This way of representing element links and hierarchy is native to XML and it is understood and supported by XmlSerializer of Microsoft .NET framework permitting us to use this mechanism to automatically generate corresponding data structures based on the XML document.

However, these links represent only the links that are part of a structure we designed to represent a Decision diagram and are not inherit to the Decision diagram itself. The possible complexity of node connections in the Decision diagram forces us to use another method. Each PointType element contains a field carrying the ID of actual node to which the actual Decision diagram link should point to.

These connections cannot be automatically stored or generated in the XML and need to be reestablished after an XML document has been parsed. The main reason for this is a possibility of establishing a loop in the graph which would pose an impassable obstacle for any XML parser. Although close loops are not permitted in Decision diagram, there is no simple way of specifying this in terms of data structure itself as for it there is no strict distinction between an ordered and unordered graph.

The control of this must be implemented separately during the process of creation of the Decision diagram. The functions for reestablishing of the real node connections in the graph must also be implemented separately from the XML parser.

The <treeType> element represents a wrapper element which task is to encapsulate the Decision diagram structure itself and store the additional information regarding the nature of the Decision diagram, i.e., the type of the Decision diagram, number of variables, nodes, levels, paths etc. This element stores additional information either in the form of XML attributes or additional elements.

Furthermore, not all of these features need to be specified in advance. Additional storage elements can be declared and included in each separate XML file as XML itself provides facilities for handling the elements that are not previously specified.

Extensions to the Decision diagram XML Schema can also be included. In this way, the XML framework can be specifically tailored for every application.

5. XML Parser

We present a small application for the purpose of creating and processing XML documents. The application was developed based on the Microsoft XML library the part of .NET framework.

The core of the application represents Microsoft XML Serializer class. This class represents an implementation of the full XML parser capable of reading and writing XML documents, their validation according to specified schemes, and transferring of data from an XML document to software specific internal data structures.

The process of XML parsing requires a set of internal software specific structures which will accommodate the data extracted from the XML document. The structure of an XML document in some degree dictates the architecture of internal data structures. Each element defined in the XML Scheme for a particular type of documents should have a corresponding class. We define a series of classes to serve this purpose.

The class NodeType corresponds to the similarly named element type in our XML Schema. It represents the core element of our Decision diagram structure. The nested objects of NodeType and PointType types are implemented to provide links to children and parents linked lists that will be associated with each node and links to the next node in the Decision diagram structure. The integer values ID, Value, Level, are intended to receive the XML attributes associated with an XML element.

```
public __gc class NodeType
{
public:
    NodeType();

    NodeType *next;
    PointType *children;
    PointType *parents;

    [XmlAttributeAttribute(Namespace=S"http://www.mycity.co.yu")]
    int ID;

    [XmlAttributeAttribute(Namespace=S"http://www.mycity.co.yu")]
    int Value;

    [XmlAttributeAttribute(Namespace=S"http://www.mycity.co.yu")]
    int Level;

};
```

Class PointType represents an element of parents/children linked list. Each of these elements consists of one link to the next element in the list and of a pointer to the NodeType object. These pointers serve the purpose of node links that exist in real Decision diagram structure in mathematical sense, that is, have some semantic value.

```
public __gc class PointType
{
public:
    PointType *next;

    [XmlAttributeAttribute(Namespace=S"http://www.mycity.co.yu")]
    int point;

    NodeType *where;
```

```
};
```

Class tree defines the wrapper object intended to encapsulate the whole structure and corresponds to the wrapper object in an XML document. It contains one NodeType element that represents the root element of Decision diagram.

```
public __gc class tree
{
public:
    tree();

    NodeType *root;

    [XmlAttributeAttribute(Namespace=S"http://www.mycity.co.yu")]
    String *TType;

    [XmlAttributeAttribute(Namespace=S"http://www.w3.org/2001/XMLSchema-
instance")]
    String *schemaLocation;
};
```

It is important to mention that internal data structures should only resemble as close as possible the structures described in the XML Schema, but they do not need to be identical. The task of XML parsing as implemented in C++ represents a compromise between flexibility of a markup language such as XML and the strong data type concept that is an imperative of C++. In a general case, the structure of an XML document is not strict, thus XML parser will try to extract the data from the XML document and store it in prepared textures in a best possible way. It will not however report an error if a unexpected well formed XML element occurs in the XML document. The XML .NET framework offers mechanisms to accommodate such an event so no data is lost. This feature is especially important if we have in view possible extensions of the XML Decision diagram specification and once again proves that XML is a right choice for this particular task.

As mentioned earlier the proposed architecture is not completely compatible with the basic XML concepts. Main problem we encounter is the graph representation of a Decision diagram. A graph in general is not a hierarchical structure with possible closed loop paths which cannot be directly transferred to XML. The XML parser would report an error if such an attempt will be made. Although Decision diagrams represent an ordered graph structure without closed loop paths, the XML parser has no way of knowing this. The problem occurs with the pointers to node objects stored in linked lists of parents and descendants.

This problem can be solved by some additional processing of our data structures before each writing attempt and after each read operation.

Instead of storing of proper pointers in the XML document, we shall store only ID values of descendant and parent nodes in <PointerType> elements.

Function `void LinkNodes(tree *mytree)` traverses the structure after each read attempt and reconstructs the links in the PointerType class elements, thus, restoring the full functionality of the Decision diagram structure.

Similar `void DelinkNodes(tree *mytree)` function performs the opposite task of preparing the structure for writing of the XML document. It disconnects the nodes setting the pointers in elements of linked lists to the NULL value. The ID values of nodes which pointers should point to are at all times stored in the pointer property of the PointerType class objects.

These structures conclude the procedures needed for storing and manipulation of Decision diagrams in the XML based framework. The data structures proposed here are compatible with standard Decision diagram manipulation procedures, such as procedures for node addition and removal, Decision diagram reduction, etc.

Appendix 1.1. XML Schema – simple BDD tree version

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:dd="http://www.mycity.co.yu" targetNamespace="http://www.mycity.co.yu"
elementFormDefault="qualified" attributeFormDefault="qualified">

  <xsd:element name="tree" type="treeType"/>

  <xsd:complexType name="treeType">
    <xsd:all>
      <xsd:element name="root" type="dd:nodeType" minOccurs="1"/>
    </xsd:all>
    <xsd:attribute name="myType" type="xsd:string" use="required"/>
  </xsd:complexType>

  <xsd:complexType name="nodeType" mixed="true">
    <xsd:sequence>
      <xsd:element name="left" type="dd:nodeType" minOccurs="0" maxOccurs="1"/>
      <xsd:element name="right" type="dd:nodeType" minOccurs="0" maxOccurs="1"/>
      <xsd:element name="terminal" type="xsd:integer" minOccurs="0" maxOccurs="1"/>
    </xsd:sequence>
    <xsd:attribute name="ID" type="xsd:integer" use="required"/>
    <xsd:attribute name="level" type="xsd:integer" use="required"/>
  </xsd:complexType>

</xsd:schema>
```

Appendix 1.2. XML Schema – flexible graph structure

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:dd="http://www.mycity.co.yu" targetNamespace="http://www.mycity.co.yu"
elementFormDefault="qualified" attributeFormDefault="unqualified">

  <xsd:element name="tree" type="TreeType"/>

  <xsd:complexType name="TreeType">
    <xsd:all>
      <xsd:element name="root" type="NodeType" minOccurs="1" maxOccurs="1"/>
    </xsd:all>
    <xsd:attribute name="TType" type="xsd:string"/>
  </xsd:complexType>

  <xsd:complexType name="NodeType">
    <xsd:sequence>
      <xsd:element name="next" type="NodeType" minOccurs="0" maxOccurs="1"
nillable="true"/>
      <xsd:element name="children" type="PointType" minOccurs="0" maxOccurs="1"
nillable="true"/>
      <xsd:element name="parents" type="PointType" minOccurs="0" maxOccurs="1"
nillable="true"/>
    </xsd:sequence>
    <xsd:attribute name="ID" type="xsd:integer"/>
    <xsd:attribute name="Level" type="xsd:integer"/>
    <xsd:attribute name="Value" type="xsd:integer"/>
  </xsd:complexType>

  <xsd:complexType name="PointType">
    <xsd:sequence>
      <xsd:element name="next" type="PointType" minOccurs="0" maxOccurs="1"
nillable="true"/>
    </xsd:sequence>
    <xsd:attribute name="point" type="xsd:integer"/>
  </xsd:complexType>
</xsd:schema>
```

Appendix 2. XML document containing Decision diagram (example)

```
<?xml version='1.0' ?>
<dd:tree TType='BDD' xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
xmlns:dd='http://www.mycity.co.yu' xsi:schemaLocation='http://www.mycity.co.yu
file://c:\Documents%20and%20Settings\stane\My%20Documents\Work\XML\xml%20optimal
.xsd'>
  <dd:root ID='0' Value='0' Level='0'>
    <dd:next ID='1' Value='0' Level='1'>
      <dd:next ID='2' Value='0' Level='1'>
        <dd:next ID='3' Value='0' Level='2'>
          <dd:next ID='4' Value='0' Level='2'>
            <dd:next ID='5' Value='0' Level='3'>
              <dd:next ID='6' Value='1' Level='3'>
                <dd:parents point='1'>
                  <dd:next point='3'>
                    <dd:next point='4' />
                  </dd:next>
                </dd:parents>
              </dd:next>
            </dd:next>
          </dd:next>
        <dd:parents point='1'>
          <dd:next point='3'>
            <dd:next point='4' />
          </dd:next>
        </dd:parents>
      </dd:next>
    <dd:children point='5'>
      <dd:next point='6' />
    </dd:children>
    <dd:parents point='2' />
  </dd:next>
  <dd:children point='3'>
    <dd:next point='4' />
  </dd:children>
  <dd:parents point='0' />
</dd:next>
<dd:children point='5'>
  <dd:next point='6' />
</dd:children>
<dd:parents point='0' />
</dd:next>
<dd:children point='1'>
  <dd:next point='2' />
</dd:children>
</dd:root>
</dd:tree>
```

References

- [1] Sasao, T., Fujita, M., (ed.), *Representations of Discrete Functions*, Kluwer Academic Publishers, 1996.
- [2] Bryant, R.E., "Graph-based algorithms for Boolean functions manipulation", *IEEE Trans. Comput.*, Vol. C-35, No. 8, 1986, 667-691.
- [3] Somenzi, F., "CUDD Decision diagram Package",
<http://bessie.colorado.edu/~fabio/CUDD>
- [4] "W3 XML Primer", www.w3.org
- [5] Stankovic, R.S., Astola, J.T., *Spectral Interpretation of Decision diagrams*, Springer, 2003.
- [6] Somenzi, F., "Efficient manipulation of Decision diagrams", *Int. Journal on Software Tools for Technology Transfer*, 3, 2001, 171-181.