4.5 RelMDD - Design Specification

This section will give a detailed documentation of the structure, implementation and technical detail of the RelMDD system. We will give the technical details of the system as well as platform and system requirements. We shall also look at the other packages needed in order to use RelMDD. We shall give an account of the system architecture, the software components, the user interface, and syntax of files containing a basis.

4.5.1 Overview of RelMDD

As mentioned above, RelMDD is a C-Library that implements arbitrary heterogeneous relation algebras using the matrix algebra approach represented by ROMDDs. RelMDD is a library written in the programming language C. It is a package that can be imported by other programs and/or languages such as Java and Haskell when programming or manipulating arbitrary relations.

The implementation is currently restricted to the basic operations of relation algebras, i.e., union, intersection, composition, converse, and complement as well as three special relations known as the identity relation, empty relation and universal relation. By design, the package is capable of manipulating relations from both classes of models, standard models and non-standard models of relation algebras. The main purpose of this project is to:

1. Implement heterogeneous relation algebras using matrix algebra.
2. Understand how relations represented as matrices could be implemented using MDDs.
3. Make available a library to aid researchers to manipulate relations in their own programs.

In RelMDD system, there are two sets of functions, namely, internal and external functions. The internal functions consist of functions that manipulate decision diagrams whiles the external functions does the manipulation of relations. Users of RelMDD will normally and frequently use the external functions to manipulate relations.

There are three major generic internal functions that are used to manipulate relations represented as decision diagrams. These functions are imported from CUDD [46]. They are `Cudd_addApply()`, `Cudd_addMonadicApply()`, `Cudd_addMatrixMult-iply()`. `Cudd_addApply()` is used for manipulating operations including meet and join, `Cudd_addMonadicApply()` is used to manipulate converse and complement where as `Cudd_addMatrixMultiply()` is used to compute the products of relations. Relation in RelMDD is a C structure consisting of a matrix (relation), list of source and target objects, the size of the matrix and the decision diagram representing this relation. RelMDD is basically a set of C files, each containing several functions packaged into a static library. Basically, the system reads relations from a file represented as matrices and list of objects, and represents them as decisions diagrams internally. It then does manipulations using decision diagrams and then output the result of the relations as a list. So the user does not interact with the decision diagrams since they represent relations internally. At input the matrices are stored in Harwell-Boeing Exchange Format.

4.5.2 Language and Operating System Platforms

We chose to implement RelMDD in C programing language (ANSI standard C) because we needed a programming language which is very fast in terms of execution of programs and C is one of the fastest so far, hence C. Moreover, most of the packages such as CUDD and libxml used in RelMDD are all implemented in C, hence it will be easier if we also implement RelMDD in C. RelMDD is currently targeted to Linux operating system platforms which made it easier to utilize other packages in our system.

4.5.3 RelMDD naming convention

In RelMDD, to name the package, files, variables, methods, and structures we have followed ANSI standard C. By convention, identifiers are mostly in lowercase letters.

4.6 Components and Technical Details of RelMDD

In this section we shall discuss the various components and the technical details of RelMDD. These components include structure of basis File, XML parsing, the Harwell-Boeing exchange format of a matrix, RelMDD operations, internal and external functions.

4.6.1 Structure of Basis File

As in ReAlM [54], basis for relation algebras in RelMDD is stored in XML format for flexibility and easy accessibility. Without a basis file, RelMDD cannot be used, since the result of operations depends on values provided by the basis file. If the program starts, the content of basis file are read into memory and then used for computations. The contents are stored in C structures using dynamic memory allocations. A basis file is associated with XML schema definitions (.xsd). The XML schema file is used to validate the contents of a basis file. This ensures that, data stored in the XML files are stored in a specific format to be recognized by the system. One should call the XML schema validation functions to validate the XML file before parsing it. RelMDD includes a schema file, which serves as the default schema file within the system.

The content of a basis file consist of several XML tags. The root element "RelationBasis" which is the starting point of the basis file, specifies the type of relations in its attribute name called "TypeOfRelation". `Example: <RelationBasis TypeOfRelation ="FiniteRelAlg">`. The next tag is the relation tag `<relation>... </relation>`. This tag contains all other tags within a basis and differentiates one basis from the other. So it is possible to have more than one basis in one file. The next tag is `<comment> ...</comment>`. This tag can be used to provide a brief description of the basis. Example: `<comment> this is a basis file for a sample relation algebra between two objects </comment>`. The tag label by "object" lists the set of objects under consideration. `Example: <objects>A , B </objects>.` The relations tag is used to specify the number of relations between two objects. An example might be like: `<relations source="A" target="A" number="4"/>`. This tag indicates that there are four relations between the objects A and A. The identity tag is used to store the identity of an object. `Example: <identity object="A" relation="1"/>` indicates that, the identity relation of an object A is represented by value 1. The bottom tag is used to store the bottom relations between objects. An example is `<bottom source="A" target="A" relation="4"/>` indicates that bottom relation of an object A is represented by value 4. The top tag is used to store the top relations between objects. `Example: <top source="A" target="A" relation="0"/>` indicates that the top relation of an object A is represented by value 0.

The later half of the basis file defines operations between pair of objects for all possible combinations. Basis file only defines the standard operations. These operations are defined in union, intersection, composition_Union, composition_Intersection, transposition and complement tags. For example, union can be used in the following way to define a union operation between two objects A and B.

```
<union source="A" target= "B">
0,0,0;
```

```
0,1,1;
0,2,2;
0,3,3;
1,0,1;
1,1,1;
1,2,3;
1,3,3;
2,0,0;
2,1,3;
2,2,2;
2,3,3;
3,0,3;
3,1,3;
3,2,3;
3,3,3
</union>
```

The union tag uses ";" as a delimiter among different entries in the operation table. The above union tag specifies sixteen different union operations. The first operation "0, 0, 0" implies that the union operation of "0" and "0" with "*A*" and "*B*" as the source and target objects respectively is "0". For a detailed structure of the basis and XML schema file we refer to the appendix.

4.6.2 XML Parsing

To manipulate relation using RelMDD, a basis file must be loaded into the system by calling the function `RelMDDParseXmlFile(const char *xmlFilename)` which will initialize all XML functions. However, `RelMDDValidateXmlFile(const char *xmlP-ath)` should be used to validate the document before parsing. RelMDD uses a hash map technique to map the relations to the objects. The system has several functions for processing XML content.

These functions are divided into two parts. The first one is used by the system internally and are located in a C file named `RelMDDXmlParser.c`. They are used to retrieve and store the content of the XML files. We adopted the usual technique in which the content of the XML file is traversed from the root node to the child node until we fetch the data we want. The parsing and processing of XML data was made easier by using a well known library called libxml2. See below for details on libxml2

[61]. The other set of functions are defined in the file `RelXmlRead.c` and are available for the users to use to process basis file content when using RelMDD. Table 4.4, Table 4.5 and Table 4.7 show various functions and structures in RelMDD.

RelMDDXmlParser.c:-Internal Functions
contains all the internal functions that are use to process XML file internally

xmlChar *relXmlGetComments(xmlDocPtr doc, xmlNodePtr cur);
void relXmlDisplayComments(xmlDocPtr doc, xmlNodePtr root);
xmlChar *relXmlGetListObject(xmlDocPtr doc, xmlNodePtr cur);
relationPtr relxmlReadrelations(xmlNode *root, xmlDocPtr doc);
char **relXmlGetSourceRelation(xmlChar *pt, int i);
char **relXmlGetTargetRelation(xmlChar *pt, int i);
char **relXmlGetNumberRelation(xmlChar *pt, int i);
identityPtr relxmlReadIdentity(xmlNode *root, xmlDocPtr doc);
char **relXmlGetObjectIdentity(xmlChar *pt, int i);
double *relXmlGetRelationIdentity (xmlChar *pt, int i);
topPtr relxmlReadTop(xmlNode *root, xmlDocPtr doc);
bottomPtr relxmlReadBottom(xmlNode *root, xmlDocPtr doc);
xmlChar *relXmlGetTopSource(xmlChar *pt, int i);
xmlChar *relXmlGetTopTarget(xmlChar *pt, int i);
xmlChar *relXmlGetTopRelation(xmlChar *pt, int i);
unionsPtr relxmlReadUnion(xmlNode *root, xmlDocPtr doc);
char **relXmlGetSource(xmlChar *pt, int i);
char **relXmlGetTarget(xmlChar *pt, int i);
intersectionPtr relxmlReadIntersection(
xmlNode *root, xmlDocPtr doc);
compoUnionPtr relxmlReadCompostion_Union(xmlNode *root, xmlDocPtr doc);
compoInterPtr relxmlReadCompostion_Inter(xmlNode *root, xmlDocPtr doc);
conversePtr relxmlReadTransposition(xmlNode *root, xmlDocPtr doc);
complementPtr relxmlReadComplement(xmlNode *root, xmlDocPtr doc);
complementPtr complIni();
intersectionPtr interIni();
unionsPtr unionIni();
xmlDocPtr getdoc();
compoUnionPtr compoUionIni();
conversePtr coverselIni();
compoInterPtr compoInterIni();
double *readString(unsigned char *s);

Table 4.4: RelMDDXmlParser.c

RelMDDXmlParser.c:-Structure Types
Structures used to store the content of the xml files
typedef struct relation{ int *number; char **source; char ** target; };
typedef struct identity{ char **object; double *rel ; int len; };
typedef struct bottom{ char **source; char ** target; double *rel; int len; };
typedef struct top{ char **source; char ** target; double *rel; int len; };
typedef struct unions{ double *content; char **source; char ** target; int len; int *len_Content; };
typedef struct intersection{ double *content; char **source; char ** target; int len; int *len_Content; };
typedef struct compoUnion{ double *content; char **source; char **target; int len; int *len_Content; };
typedef struct compoInter{ double *content; char **source; char **target; int len; int *len_Content; };
typedef struct converse{ double *content; char **source; char **target; int len; int *len_Content; };
typedef struct complement{ double *content; char **source; char **target; int len; int *len_Content; };

Table 4.5: RelMDDXmlParser.c

RelMDDXmlReader.c:-External Functions
Contains all the external functions that the user can use to process XML files

static xmlDocPtr xmlDocument;
extern int RelMDDValidateXmlFile(const char *xmlPath);
extern int RelMDDParseXmlFile(const char *xmlFilename);
extern unionsPtr RelXmlGetUnion();
extern intersectionPtr RelXmlGetIntersection();
extern complementPtr RelXmlGetComplement();
extern compoUnionPtr RelXmlGetComposition_Union();
extern compoInterPtr RelXmlGetComposition_Intersection();
extern conversePtr RelXmlGetConverse();
extern topPtr RelXmlGetTop();
extern bottomPtr RelXmlGetBottom();
extern identityPtr RelXmlGetIdentity();
extern relationPtr RelXmlGetRelation();
extern void relXmlDisplayComments();
extern char ** relXmlGetObjects();

Table 4.6: RelMDDXmlReader.c

4.6.3 lixml2

In parsing XML files, we used a C parser and toolkit called Libxml2 developed for the Gnome project. It can also be used outside the Gnome platform. This software is free and available under the MIT License. Libxml2 includes complete XPath, XPointer and XInclude implementations. We used several functions from this library in our work which made it very easy for us to process XML files. For detailed descriptions of this package please refer to [61].

4.6.4 CUDD

CUDD (Colorado University Decision Diagram) is a package that provides functions to manipulate Binary Decision Diagrams (BDDs), Algebraic Decision Diagrams (ADDs), and Zero-suppressed Binary Decision Diagrams (ZDDs), CUDD is written in C by Fabio Somenzi at the Department of Electrical and Computer Engineering, University of Colorado at Boulder. RelMDD is built on CUDD version 2.4.2. It provides a large set of operations on BDDs, ADDs, and ZDDs, and a large assortment of variable reordering methods. In fact it is one of the best and well known packages that implements decision diagrams more efficiently by incorporating several optimization techniques. For detailed overview of this system please refer to [46, 13].

4.6.5 RelMDD Operations and Special Relations

Internally, RelMDD uses three generic functions imported from the CUDD package for manipulation of decision diagrams. `Cudd_addApply()` accepts a binary operator such as meet or join with two ADDs representing two matrices. Hence, we used this function to tackle the problem of meet and join. `Cudd_addMonadicApply()` accepts a unitary operator and one ADD representing a matrix. We used this function to implement operations such as the converse, and the complement of relations. However, in the case of converse we had to swap variables of the ADD before using `addMonadicApply()`. Lastly, we used `Cudd_addMatrixMultiply()` to accomplish composition by modifying it slightly to suite our purpose. In the following paragraphs, we will describe various functions that can be used in RelMDD to compute relations. Table 4.7 list the functions in RelMDD for manipulating relations. For detailed description of how to use the package please refer to the user manual accompanying the RelMDD package.

Union: `RelMDDUnionPtr RelMDDUnionOperation(...);` The function above is used to for calculating the union of two relations. It accepts six parameters as input. The first two parameters are the size of rows and columns of the matrices respectively. The next two parameters are the list of source and target objects of both matrices respectively. It should be noted that, to compute the union of two relation, the source and target objects of the both matrices are the same. This function when called returns a structure type containing the list of source and target objects, and the relations.

Intersection: `RelMDDIntstersectionPtr RelMDDIntersectionOperation(...);` The function above is used to find the meet of two relations. It behaves similarly to the join operation described above.

Compostion: To find the composition of two matrices, we use the function `RelMDDC-ompostionPtr RelMDDCompositionOperation(...);`. This function takes the size of the rows and columns of the first matrix as the first two parameters and then the size of the columns of the second matrix, followed by the list of source and target objects of the first matrix and then the target objects of the second matrix. The last two parameters are the two matrices that we want to find their product. The function returns details of the result of computations in a C structure. Note that, the source objects of the first matrix is always the same as the target objects of the second matrix.

Converse: `RelMDDConversePtr RelMDDConverse_Operation(...);` is use to accomplish the converse operation. It accepts parameters similarly to the union and intersection operations described above except that it takes only one matrix.

Complement: `RelMDDComplementPtr RelMDDComplement_Operation(...);` is used for the computation of the complement of a relation. It behaves similarly to the converse relation describe above.

Identity: To find the identity element of a given list of objects we use the function `RelMDDIdentityPtr RelMDDIdentityRelation(...);` The first two parameters are the of size of the rows and columns of the matrix we want to find its identity. This is followed by the list of objects. Here the source and target objects are the same. This function returns a structure made up of the identity relation and the list of objects.

Zero Element: `RelMDDBottomPrt RelMDDBottomRelation(...);` is used for the derivation of the zero element of a given set of objects. As usual the first two parameters are the size of the rows and columns of the list of source and target objects respectively. It returns a structure that consist of the bottom element and the list of objects.

Universal Relation: `RelMDDTopPtr RelMDDTopRelation(...);` It behaves similarly to the zero element function described above.

RelMDD.c: Internal and external functions
Contains functions for manipulating relations
extern DdNode *Transpose(DdManager * dd, DdNode * f);
extern CUDD_VALUE_TYPE transposeRelation(CUDD_VALUE_TYPE F);
extern DdNode *RelMDD_SwapVariables(DdManager * dd , int roww, int collh);
extern DdNode *Complement(DdManager * dd, DdNode * f);
extern CUDD_VALUE_TYPE complementRelation(CUDD_VALUE_TYPE F);
extern void RelMDDComplement_Operation(int tra_rn, int tra_col, char **source_objects, char **target_objects, double *matrimx);
extern void RelMDDConverse_Operation(int tra_rn, int tra_col, char **source_objects, char **target_objects, double *matrimx) ;
extern fileInforPtr RelMdd_ReadFile(char *filename);
extern double setSequence(double value);
extern DdNode *RelMDD_RenameRelation(DdManager * dd, int roww ,int collh);
extern CUDD_VALUE_TYPE complementRelation(CUDD_VALUE_TYPE F);
extern int *RelMDD_ListSequence(DdNode *E, int roww, int coll);
static FILE *open_file (char *filename, const char *mode);
extern DdNode *Cudd_addUnion_Meet(DdManager * dd, DdNode ** f,DdNode ** g);
extern DdNode *Cudd_addIntersection(DdManager * dd,DdNode ** f,DdNode ** g);
extern RelMDDIntersectionPtr RelMDDIntersectionOperation(int no_rows, int no_cols,char **source_objects , char **target_objects, double *matrix1, double *matrix2);
extern RelMDDUnionPtr RelMDDUnionOperation(int no_rows, int no_cols, char **source_objects, char **target_objects, double *matrix1, double *matrix2);
extern DdNode * node_Complement(DdNode * mat);
extern CUDD_VALUE_TYPE identityRelation(CUDD_VALUE_TYPE F);
extern RellMDDIdentityPtr RelMDDIdentityRelation(int row, int col, char **objects);
extern RellMDDBottomPtr RelMDDBottomRelation(int row, int col, char **source_objects , char **target_objects);
extern CUDD_VALUE_TYPE BottomRelation(CUDD_VALUE_TYPE F);
extern RellMDDTopPtr RelMDDTopRelation(int row, int col, char **source_objects , char **target_objects);
extern CUDD_VALUE_TYPE topRelation(CUDD_VALUE_TYPE F);
extern int RelMdd_CreateRelation(...);
extern RelMDDCompostionPtr RelMDDCompositionOperation(int no_rows, int no_cols, char **sourceA_objects,char **targetA_objects,char **targetB_objects, double *matrix1, double *matrix2);

Table 4.7: RelMDD.c

4.6.6 Loading a Matrix

Since relations are represented as matrices, RelMDD has a special format to read in matrices from files. This format adhere to the format of Harwell-Boeing benchmark suite. This format specifies how matrices should be stored and read from a file. In our case the first item on the file specifies the number of rows and column of the matrix. The second and third items on the file specify the list of source and target objects respectively. This is followed by the matrix in which the row and column indices of each element of the matrix is also specified. Refer to the appendix for sample file.

Chapter 5

Conclusion and Future Work

In this thesis, we reviewed various properties of heterogeneous relation algebras. We went ahead to introduce a system called RelView which only works in the standard model of relations algebra. This is because the underlying structure of this system is based on Boolean matrix which restricts its ability to be used outside the class of standard models.

Now, it has been proved in [49] that for every relation algebra $\mathcal{R}$ with relational sums and subobjects, it is possible to characterize a full subalgebra $\mathcal{B}$ called the basis of $\mathcal{R}$, such that the matrix algebra $\mathcal{B}^+$ with the coefficients from $\mathcal{B}$ is equivalent to $\mathcal{R}$. Based on the above theorem, we developed a system called RelMDD which works within both the standard and the non-standard models of relation algebras using the matrix approach. In order to do this, we used an advanced and more efficient data structure called multiple valued decision diagrams (MDDs) which is similar to the data structure used by RelView system in the Boolean matrix case.

This implementation combines two major advantages over a regular array implementation of matrices. Both RelView and RelMDD systems have proven that an implementation of relations using decision diagrams is of great benefit. The RelMDD system is a library written in C which can be imported by other languages such as Java or Haskell.

The package implements all standard operations on relations. A future project will add further operations such as sums and splittings. The latter will then also allow to compute relational powers and so-called weak relational products [52]. Another project will be a suitable module for the programming language Haskell that makes the RelMDD package available in this language. An advanced project will be to extend the current system into a programming language which will allow programming using relations. One can also work on integrating the package into the RelView system.

Appendix A

Appendix

A.1 XML schema file

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault =
"qualified">

<!-- definition of simple elements -->
<xs:element name="comment" type="xs:string"/>
<xs:element name="objects"/>


<!-- definition of attributes -->
<xs:attribute name="TypeOfRelation" type="xs:string"/>
<xs:attribute name="source"  type="xs:NCName"/>
<xs:attribute name="target"  type="xs:NCName"/>
<xs:attribute name="number"  type ="xs:integer"/>
<xs:attribute name="object"  type="xs:NCName"/>
<xs:attribute name="relation"  type="xs:integer"/>

<!-- definition of complex elements -->
<xs:element name="relations">
  <xs:complexType mixed="true">
  <xs:attribute ref="source"  use="required"/>
  <xs:attribute ref="target"  use="required"/>
  <xs:attribute ref="number"  use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="identity">
  <xs:complexType mixed="true">
  <xs:attribute ref="object"  use="required"/>
  <xs:attribute ref="relation"  use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="bottom">
  <xs:complexType mixed="true">
  <xs:attribute ref="source"  use="required"/>
  <xs:attribute ref="target"  use="required"/>
  <xs:attribute ref="relation"  use="required"/>
  </xs:complexType>
</xs:element>

 <xs:element name="top">
  <xs:complexType mixed="true">
  <xs:attribute ref="source"  use="required"/>
  <xs:attribute ref="target"  use="required"/>
  <xs:attribute ref="relation"  use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="union">
  <xs:complexType mixed="true">
  <xs:attribute ref="source"  use="required"/>
  <xs:attribute ref="target"  use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="intersection">
  <xs:complexType mixed="true">
     <xs:attribute ref="source"  use="required"/>
     <xs:attribute ref="target"  use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="complement">
  <xs:complexType mixed="true">
     <xs:attribute ref="source"  use="required"/>
     <xs:attribute ref="target"  use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="composition_Union">
  <xs:complexType mixed="true">
  <xs:attribute ref="source"  use="required"/>
  <xs:attribute ref="target"  use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="composition_Intersection">
  <xs:complexType mixed="true">
  <xs:attribute ref="source"  use="required"/>
  <xs:attribute ref="target"  use="required"/>
  </xs:complexType>
</xs:element>
<xs:element name="transpose">
  <xs:complexType mixed="true">
     <xs:attribute ref="source"  use="required"/>
     <xs:attribute ref="target"  use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="relation">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="comment"/>
      <xs:element ref="objects"/>
      <xs:element ref="relations" maxOccurs="unbounded"/>
       <xs:element ref="identity" maxOccurs="unbounded"/>
       <xs:element ref="bottom" maxOccurs="unbounded"/>
        <xs:element ref="top" maxOccurs="unbounded"/>
      <xs:element ref="union" maxOccurs="unbounded"/>
      <xs:element ref="intersection" maxOccurs="unbounded"/>
       <xs:element ref="composition_Union" maxOccurs="unbounded"/>
        <xs:element ref="composition_Intersection" maxOccurs="unbounded"/>
      <xs:element ref="transpose" maxOccurs="unbounded"/>
       <xs:element ref="complement" maxOccurs="unbounded"/>
    </xs:sequence>

  </xs:complexType>
</xs:element>

<xs:element name="RelationBasis" >
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="relation" maxOccurs="unbounded"/>
      </xs:sequence>
   <xs:attribute ref="TypeOfRelation" use="required"/>
     </xs:complexType>
</xs:element>


</xs:schema>
```

A.2 Sample Basis File

```
<?xml version="1.0" encoding="utf-8"?>
<FiniteRelAlg name ="two_object_basis">

<comment>
Our very first example.
</comment>
```

```
<objects> A,B </objects>

<relations source="A" target="A" number="4"/>
<relations source="A" target="B" number="2"/>
<relations source="B" target="B" number="4"/>
<relations source="B" target="A" number="2"/>



<identity object="A" relation="1"/>
<identity object="B" relation="1"/>

<bottom source="A" target="A" relation="0"/>
<bottom source="A" target="B" relation="0"/>
<bottom source="B" target="A" relation="0"/>
<bottom source="B" target="B" relation="0"/>

<top source="A" target="A" relation="3"/>
<top source="A" target="B" relation="1"/>
<top source="B" target="A" relation="1"/>
<top source="B" target="B" relation="3"/>


<union source="A" target= "B">
       0,1,1;
       1,0,1;
       0,0,0;
       1,1,1
</union>


<union source="B" target= "A">
       0,1,1;
       1,0,1;
       0,0,0;
       1,1,1
</union>

<union source="A" target= "A">
       0,0,0;
       0,1,1;
       0,2,2;
       0,3,3;

       1,0,1;
       1,1,1;
       1,2,3;
       1,3,3;

       2,0,0;
       2,1,3;
       2,2,2;
       2,3,3;

       3,0,3;
       3,1,3;
       3,2,3;
       3,3,3

</union>


<union source="B" target= "B">
       0,0,0;
       0,1,1;
       0,2,2;
       0,3,3;

       1,0,1;
       1,1,1;
       1,2,3;
       1,3,3;

       2,0,0;
       2,1,3;
       2,2,2;
       2,3,3;

       3,0,3;
       3,1,3;
       3,2,3;
       3,3,3

</union>



<intersection source="A" target= "B">
       0,1,0;
       1,0,0;
       0,0,0;
       1,1,1
</intersection>


<intersection source="B" target= "A">
       0,1,0;
       1,0,0;
       0,0,0;
       1,1,1
</intersection>


<intersection source="A" target= "A">
       0,0,0;
       0,1,0;
       0,2,0;
       0,3,0;

       1,0,0;
       1,1,1;
       1,2,0;
       1,3,1;

       2,0,0;
       2,1,0;
       2,2,2;
       2,3,2;

       3,0,0;
       3,1,1;
       3,2,2;
       3,3,3

</intersection>

<intersection source="B" target= "B">
       0,0,0;
       0,1,0;
       0,2,0;
       0,3,0;

       1,0,0;
       1,1,1;
       1,2,0;
       1,3,1;

       2,0,0;
       2,1,0;
       2,2,2;
       2,3,2;

       3,0,0;
       3,1,1;
       3,2,2;
       3,3,3

</intersection>



<composition_Union source="A"  target="B">
        0,0,0;
        0,1,0;
        1,0,0;
        1,1,1
</composition_Union >

<composition_Union  source="B" target="A">
        0,0,0;
        0,1,0;
        1,0,0;
        1,1,1
</composition_Union >


<composition_Union  source="A" target="A">
       0,0,0;
       0,1,0;
       0,2,0;
       0,3,0;

       1,0,0;
       1,1,1;
       1,2,2;
       1,3,3;

       2,0,0;
       2,1,2;
       2,2,1;
       2,3,3;

       3,0,0;
       3,1,3;
       3,2,3;
       3,3,3
</composition_Union >

<composition_Union  source="B" target="B">
        0,0,0;
        0,1,0;
        1,0,0;
        1,1,1
</composition_Union >


<composition_Intersection source="A"  target="B">
        0,0,0;
        0,1,0;
        1,0,0;
        1,1,1
</composition_Intersection>

<composition_Intersection source="B" target="A">
        0,0,0;
        0,1,0;
        1,0,0;
        1,1,1
</composition_Intersection>


<composition_Intersection source="A" " target="A">
        0,0,0;
        0,1,0;
        1,0,0;
```