Notes on Relationship Operations

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
0.1	June 6, 2017	Initial draft.	GAM
1.0	July 6, 2017	First release.	GAM

Contents

List of Figures

List of Tables

Problem Statement

This paper explores the problem of how to formulate model level operations to create, delete and update relationship instances in Executable UML (xUML). There are two primary goals:

1. Provide a consistent, well defined vocabulary to describe the operations.
2. Insure the relationship operations can be translated onto all potential target platforms.

Current xUML action languages use a notion of "relate" and "unrelate" or "link" and "unlink" to express the relationship operations. These operations are formulated using instance references. For example:

- Let `xref` be a variable whose value is a reference to an instance of some class X.
- Let `yref` be a variable whose value is a reference to an instance of some class Y.
- Let `R1` be an association between classes X and Y. For simplicity, assume `R1` is a simple many-to-one unconditional association between X and Y. A class diagram fragment showing this might appear as follows.

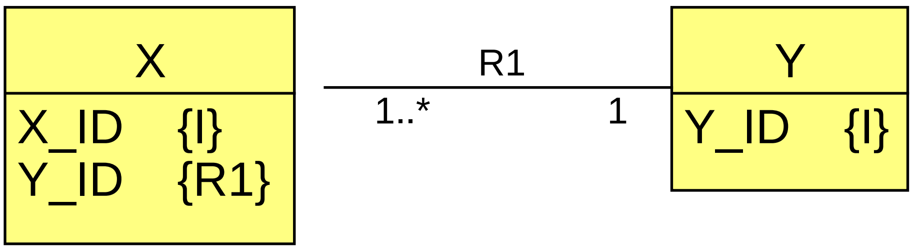


Figure 1: Example R1 Association

Most action languages have statements to create an instance of the R1 association between X and Y instances similar to:

```
relate xref to yref across R1
```

Complementary constructs are used to delete a relationship instance.

```
unrelate xref from yref across R1
```

The important point here is *not* the syntax or keywords used, but rather the implied operation and its arguments.

Comparing the diagram to the previous `relate` statement, the statement implies that an instance of X may be created (and a instance reference value to the created instance stored in the `xref` variable) without supplying a value for the `Y_ID` attribute. Otherwise, it would not be necessary to invoke the `relate` operation and supply the `yref` value. This is problematic if we insist, as we discuss later, that attributes must have a valid value at all times.

Similarly, the `unrelate` statement requires that a reference to the related Y instance be supplied as an argument. This arrangement has several awkward aspects:

1. The diagram implies that deleting an instance of X deletes the instance of R1 because the `Y_ID` attribute which manifests the relationship is no longer defined and it's not possible for the Y instance to be related to the non-existent X instance.
2. If the activity invoking the `unrelate` operation must supply an instance reference to the related instance of Y (the `yref` value in this case) then it is forced to navigate the association to obtain the instance reference. Yet, the model meta-data contains the knowledge of the **R1** association and therefore we know that some Y instance is related because that is given by the value of `Y_ID` of the referenced X instance (the X instance referenced by the `xref` value in this case).

3. What is the behavior of the `unrelate` statement if the value of `yref` does not refer to the instance that is actually related to `xref` or if it refers to an instance of some class other than Y? Are we exposing a potential for undefined behavior that must be detected at run-time and are therefore placing subtle requirements on the model execution run-time?

These action language constructs equate the idea of an instance of the relationship with that of a “link” between the two instances. Because most of these operations were formulated with conventional, static-typed implementation languages in mind, they have underlying assumptions implying that operations on relationships involve pointers. That is *not* the case for all translation targets and the concept of relationship linkage insinuates a correspondence to pointers that is implementation dependent and therefore undesirable. This is compounded by the notion of *object ids* found in object-oriented programming languages. Object ids are usually a thin, compiler-constructed veneer on a memory address.

In this paper, we will *not* suggest any action language syntax for the proposed operations. Rather, we stay focused on the semantics of the operations and the required parameters. Action languages should develop convenient syntax to support the proper underlying operations.

Proposed Operations

The following logical operations on relationships between instances in xUML are proposed.

1. The instance creation operation for a class which defines referential attributes also creates a instances of all the relationships corresponding to the referential attributes. Referential attributes must be conceptually initialized to a value when a class instance is created. However, we propose referential attribute values be supplied indirectly by specifying both the relationship number and an instance reference to the related class instance satisfying the relationship. An instance reference and the number of the relationship is sufficient[1] to determine the values of the referential attributes. This operation also applies to class-based associations where instance references to both relationship participants must be supplied.
2. Deleting a class instance where the class contains referential attributes also deletes the relationship instance to which the referential attribute pertain. Note that we do not imply that any related class instance is to be deleted as part of some *cascading delete* operation. Deletion protocols in xUML are complicated enough, especially when class lifecycles are involved, that responsibility for consistency of the class model must be accomplished by direct model actions.
3. For association type relationships, a further operation, named **reference**, must be provided to modify the relationship instance and consequently change the class instance referred to in the association. This is accomplished by supplying an instance reference to the newly referenced class instance. The **reference** operation works in lieu of directly updating the values of referential attributes. We go one step farther and say that in xUML activities, updating (but not reading) the value of a referential attribute is forbidden.
4. For generalization type relationships, a **reclassify** operation must be provided to change the subclass of the referring subclass class instance. Formally, the system performs a deletion of the current referring instance, and creates a new instance of the desired type. Any attribute values of the new subtype instance must be supplied as part of the operation to insure all attributes of the newly created subtype have valid values.

Fundamentals

Sadly, there is very little commonly accepted and shared outlook or knowledge in software engineering and this situation also extends to the xUML world. To support this proposal, we feel compelled to enumerate our basic assumptions.

- A software system is made up of one or more interacting *domains*. A domain is part of a subject-matter decomposition of the system functional requirements. Domain interactions are governed by bridges which resolve the requirements and dependencies of one domain onto the services provided by other domains.
- A domain captures the requirements allocated to it using three facets:

[1] except in the reflexive relationship case, discussed later

1. A data facet that defines entities of concern to the domain. The data facet model is based on the well established principles of the relational model of data and abstract data types.
2. A dynamics facet which describes execution sequencing and synchronization. The dynamics facet of a domain model is based on finite state automata.
3. A processing facet determines the algorithmic computations performed. The processing facet of a domain model is based on the data flow model of execution.

This paper is concerned with defining operations on relationships. So, our goal is to determine the semantics of the operations used in the processing facet to create and manipulate relationships that are defined in the data facet. We call the domain which manages data and execution on a target platform the Model Execution (MX) domain. We want to develop the relationship operations with several distinct translation realms in mind.

xUML Class Model
We consider xUML class models as an *application* of the relational model model of data. Thus, we need to have a direct mapping for all xUML data operations back to relational algebra. Ultimately, all operations on the data facet must be expressed in relational algebraic terms.

Implementations Based on Relational Data Management
One important class of MX domains manages its data using implementation components that support relational concepts (*e.g.* a Relational Database Management System). The relationship operations must be able to be readily translated onto MX domain implementations that use relationally based data management techniques.

Implementations Based on Statically Typed Programming Languages
Another very important MX domain target uses conventional programming languages that are static-typed and hold all class data in primary memory. In this case, the behavior of xUML data model operations must be simulated without having explicit relational data management facilities available.

xUML Class Model Fundamentals

Since we consider an xUML class model to be based on the relational model of data, in this section we enumerate the correspondences between xUML terms and relational model terms. We reiterate that xUML class models are an application of the relational theory of data and not a direct usage of relational concepts.

- An xUML class model is a normalized relational schema of at least Boyce-Codd Normal Form (BCNF). xUML modelers rarely use normalization in the same manner as used for database design, despite sharing the same foundation for organizing data. This arises from the different approaches taken in the two fields. Database design often starts with an arbitrary collection of data which is then analyzed for its functional dependencies. xUML analysis starts from the prospective of identifying and abstracting real-world entities and their associations according to a set of modeling rules. The result is that xUML class models tend to be formulated in normalized form from the outset since the modeling rules guide the analysis toward that end. In this case, the normalization rules are used as a check on the coherency of the analysis.
- A populated xUML class model is assumed to follow the *closed world assumption* as it is typically applied to relational schema.
- An xUML *class* corresponds to a *relvar* (relation variable). Specifically, an xUML class defines the heading of a relation value that can be stored in a relation variable of the same type. In particular, note that an xUML class does *not* correspond to an object oriented programming (OOP) language class nor does it correspond to an abstract data type as might be used as an attribute data type. An xUML class represents a logical predicate about the subject matter of the domain and its attributes define a characterization of a real-world entity with respect to the domain subject matter. This follows from the equivalency of relational algebra and first-order predicate logic.
- A class *attribute* corresponds to an attribute of the relation header for the relation value held in the relvar corresponding to a class. Each attribute has a defined data type. Such data types may be *scalar* or *non-scalar*. Data typing is an orthogonal concept to xUML classes. We will not discuss data typing here and note that this discussion depends only upon the existence of an equality operation for an attribute data type.

- A class *instance* is defined to be a tuple of a relation value stored in a relvar. Thus the cardinality of the relation value stored in a relvar is the number of instances of a class. Since we intend an xUML class model to have an implementation on a computer, we implicitly assume that the number of class instances is finite.
- At all times, every attribute of every class instance must have a value selected from the set of values defined by the attribute data type. Consequently, there is no **NULL** value and no attribute may be assigned the value of **NULL**. There has been much written and discussed about **NULL** values in the relational model and we will not repeat it here. We do note that there is much confusion over the concept of **NULL** as might be used in a modeling context and the use of `NULL` or `nil` values in a programming language context. They are not the same thing and we see no benefit to using **NULL** in a modeling context despite the usefulness of `NULL` or `nil` in a programming context. Null values are excluded on two accounts:
    1. They are unnecessary adding nothing to the expressiveness of the modeling constructs.
    2. They introduce the need for three-valued logic which is a major, unwelcome complication.
- Each xUML class must have at least one identifier (this follows from the normalization requirements). An identifier is a set of class attributes whose values, as a set, must be unique for all instances of the class. An identifier corresponds to a *candidate key* in the relational model. Consequently, the set of attributes that constitute an identifier may *not* be a subset, proper or improper, of any other identifier's attribute set (this also follows from the normalization requirements). However, it is not uncommon that the intersection of the attribute sets of the class identfiers be non-empty.
- An *instance reference* is a means to refer to one or more class instances. It is defined as a relation value whose heading is the projection of one of the identifiers of the class to which it refers. Since the values contained in such a projection are functionally related to the values of the other attributes of the instance, it is possible to obtain the set of instances to which the instance reference refers by performing a `semijoin` operation between the instance reference and the class relvar. Note that we make no distinction between an instance reference that refers to multiple (or even zero) instances and one that references only a single instance. The set aspects of a relation value handle all cases.

Relationships in xUML

In xUML, relationships model real-world associations between classes. The semantics of the relationship are indicated by verb phrases attributed to each side and the phrases appear on the class diagram. A relationship is *manifested* by a referential integrity constraint. The integrity constraint is formed by requiring *referential attributes* in one class to have the same values as *identifying attributes* in another class (possibly the same, in which case the relationship is *reflexive*).

Relationships are used for two distinct purposes:

1. Constraining the membership of the participating classes.
2. *Navigating* from instances of one class to instances of a related class.

As we discuss below, relationships can be viewed as functions or partial functions. The functional view coincides well with the use of verb phrases to describe the relationship semantics since we usually associate functions with some sort of action. Navigating a relationship is logically equivalent to invoking a function on a subset of the function's domain to obtain the image of that function in the codomain. From a relational algebra point of view, relationship navigation is accomplished by the `semijoin` operation (or the `semiminus` operation if the intent is to find the unrelated instances). In relational algebra, we do not have the restriction of only navigating along the defined referential constraints. We may perform a `semijoin` between any two relvars and get some result as long as the join attributes have the same data type, even if we are compelled to rename attributes. However, the result could be meaningless if the join attributes have no logical correlation. Navigating using the referential attributes of a relationship insures both the data type requirements and maintains logical meaning.

There are two distinct types of relationships:

1. Associations are mappings between the instances in one class to those in another. An association can be viewed as a binary relation that is a subset of the Cartesian product of a projection of the identifiers of the participating classes.
2. Generalizations represent a set partition of a superclass into a set of subclasses. The set of subclasses of the generalization are disjoint and the union of the subclasses is equal to the superclass (specifically, the intersection of the projection of the subclass identifiers is empty and the union of the projection of the subclass identifiers equals the projection of the superclass identifier).

We describe each type of relationship in separate sections.

Association Relationships

As stated previously, association relationships can be viewed as a binary relation that enumerates the instances of the two classes that are related. A class that serves the role of enumerating the mapping is called an **associator** class. An associator class also allows for additional attributes that can be used to abstract properties of the association itself (*e.g.* the time at which the association instance was formed).

Association relationships are also characterized by their *conditionality* and *multiplicity*. Conditionality determines if all the instances of a class must participate in the relationship. Multiplicity determines if a class instance may participate in the relationship more than one time. The various combinations of multiplicity and conditionality give rise to additional classifications for associations.

Simple associations
: are those associations that are singular and unconditional on at least one side. They are considered *simple* in the since that it is possible, under certain circumstances, to eliminate the associator class that manifests the association mapping.

Partial function associations
: are those associations which have conditionality on at least one side. In this case, navigating the association in the direction of the conditionality can result in no instances being found.

Many valued associations
: are those association where there may be multiple mappings between the same instances. These types of associations have, of necessity, at least one additional identifying attribute in the associative class.

There are 10 distinct combinations of multiplicity and conditionality for associations. In this section, we show each case and discuss the characteristics of the association. Each is shown in a diagram. Each diagram contains a set-oriented graphic emphasizing the association as a function between two sets. We also show the same information using UML class diagram notation. The purpose is to emphasize the set-oriented aspects of associations as well as provide a precise meaning to the UML class diagram usage.

Relationships and their correspondence to functions are discussed and their properties are characterized. However, it is notable that our intent here is different than that usually associated with mathematical discussions of functions. In mathematics, the sets over which the function operates are usually given and it is the properties of the function that are of interest. For example, we may say that $f(x) = 2x + 1$ is a function from $\mathbb{R} \to \mathbb{R}$ and then discuss the properties of $f(x)$ (in this case it is injective). From a modeling perspective, we tend to take the opposite point of view. Namely, we fix the properties of the functions since those properties relate to the problem semantics. Since the membership of the sets is **not** constant and changes over the execution time of the domain, the relationships then serve to constrain the set membership to insure the *a priori* function properties are **not** violated. For example, in a one-to-one unconditional association, creating an instance in one participant of the association requires creating a related instance in order to maintain properties of the function that underlies the association. Failure by the model processing to maintain the set membership to match the association function properties results in a failed transaction on the data model.

Because the class instance sets are finite, we can describe the association function by enumerating a binary relation containing the mapping elements (as opposed to using a formula to define the function). From the relational point of view, we define a relvar whose attributes refer to the identifying attributes of the association participants. Each tuple in the associative relvar then represents an instance of the association itself. We start by formulating all associations in this manner and then show how some associations *under certain conditions* may be simplified in a way that eliminates the need for the associative relvar.

The first four sections below describe the simple association cases. Next come the partial function associations and finally we show the many valued associations. Once we have seen the characteristics of all 10 cases, we will be in a position to discuss operations on associations and the case-by-case discussion will allow us to verify and exercise those operations.

One to One Association

The following diagram shows the properties of a one-to-one association.

The top of the diagram shows the mapping of instances of X to instances of Y in a conventional set oriented graphic. The relationship, **R**, is a function taking elements of the **X** set into elements of the **Y** set, *i.e.* $\mathbf{R} : \mathbf{X} \to \mathbf{Y}$. **X** is the *domain* of **R**, and **Y** is the *codomain* of **R**. **R** is a bijective function.

The middle section of the diagram shows the same information in UML class diagram notation. Here the notion of class identifiers has been made explicit. For simplicity but without the loss of any generality, we use identifiers that have only a single attribute. The **R Assoc** class is a relvar of degree two containing the identifiers of the participating **X** and **Y** classes. Note that either the **X** identifier or the **Y** identifier may serve as the identifier for **R Assoc**.

Because of the multiplicity and conditionality, we know that the number of instances of **R Assoc** must equal both the number of instances of **X** and the number of instances of **Y**. Under the special circumstances that **R Assoc** has no additional attributes, *i.e.* the abstraction associated with **R** only characterizes the functional mapping between **X** and **Y**, then the association may be simplified by eliminating the **R Assoc** class and including a referential attribute in either the **X** or **Y** class. This is shown at the bottom of the diagram. Either representation may be chosen and xUML models tend to make the choice based on clarifying semantic intent.

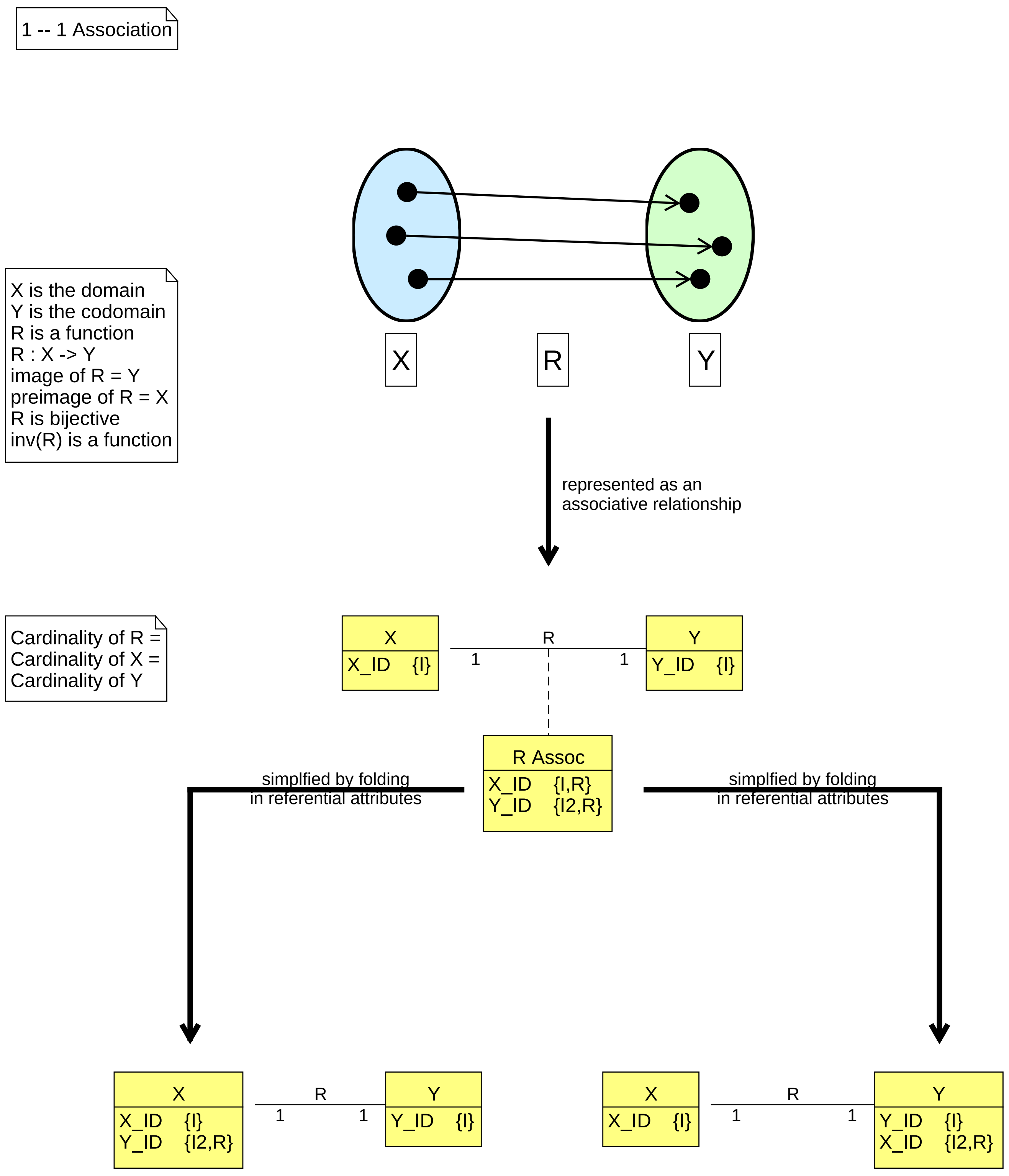


Figure 2: One-to-one Association

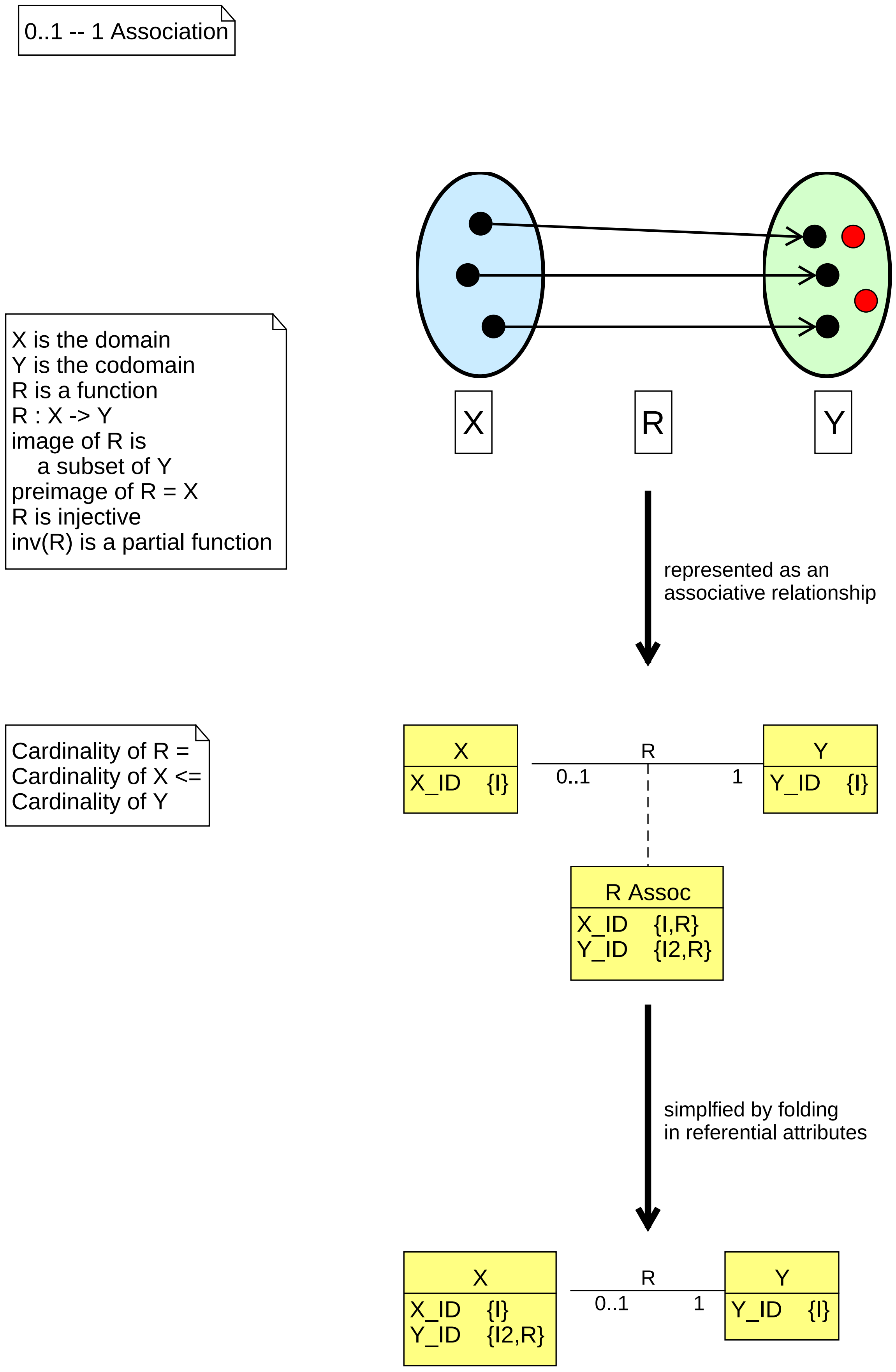
At Most One to One Association

The next case is an at-most-one to one association. The following diagram shows characteristics of associations of this multiplicity and conditionality.

The depiction of **R** in set-based notation shows that some elements of **Y** (*i.e.* those colored red) do not participate in the association. **R** is still a function, but in this case it is an injective function. Consequently, the image of **R** under **X** is a subset of **Y** and the inverse of **R** is a partial function.

The implications of this arrangement are that for some elements of **Y**, the inverse of **R** is undefined. Thus it is possible when navigating from **Y** to **X** across **R**, to obtain an empty result. From a relational algebra point of view, it is possible to select a subset of **Y** such that the semijoin of the subset to the relation value held in the **X** relvar yields an empty relation with the same heading as **X**. It holds in general, that conditionality on one side of an association implies that an empty traversal is possible and activity code must take into account that no instances will have been found in the association traversal.

The UML notation in the middle of the diagram shows **R Assoc** as the associative class that manifests the functional mapping of **R**. Note that the cardinality of **R Assoc** is equal to the cardinality of **X** but may be less than the cardinality of **Y**. Thus when the simple association conditions hold, **R Assoc** may be eliminated, but the referential attribute must be placed in **X**.


0..1 -- 1 Association

X is the domain
Y is the codomain
R is a function
R : X -> Y
image of R is
a subset of Y
preimage of R = X
R is injective
inv(R) is a partial function

Cardinality of R =
Cardinality of X <=
Cardinality of Y

Figure 3: At most one-to-one Association

At Least One to One Association

The diagram below shows an at-least-one to one association.

The depiction of **R** in set-based notation shows that multiple elements of **X** may map to the same element of **Y**. **R** is a surjective function and its inverse is multi-valued.

The UML representation shows, the **R Assoc** class is identified by X_ID. This is because the number of instances of **R Assoc** equals the number of instance of **X**. This is another way of stating that the preimage of **R** is equal to **X**. Again, under the simplifying conditions, **R Assoc** may be eliminated by placing the Y_ID referential attribute in **X**.

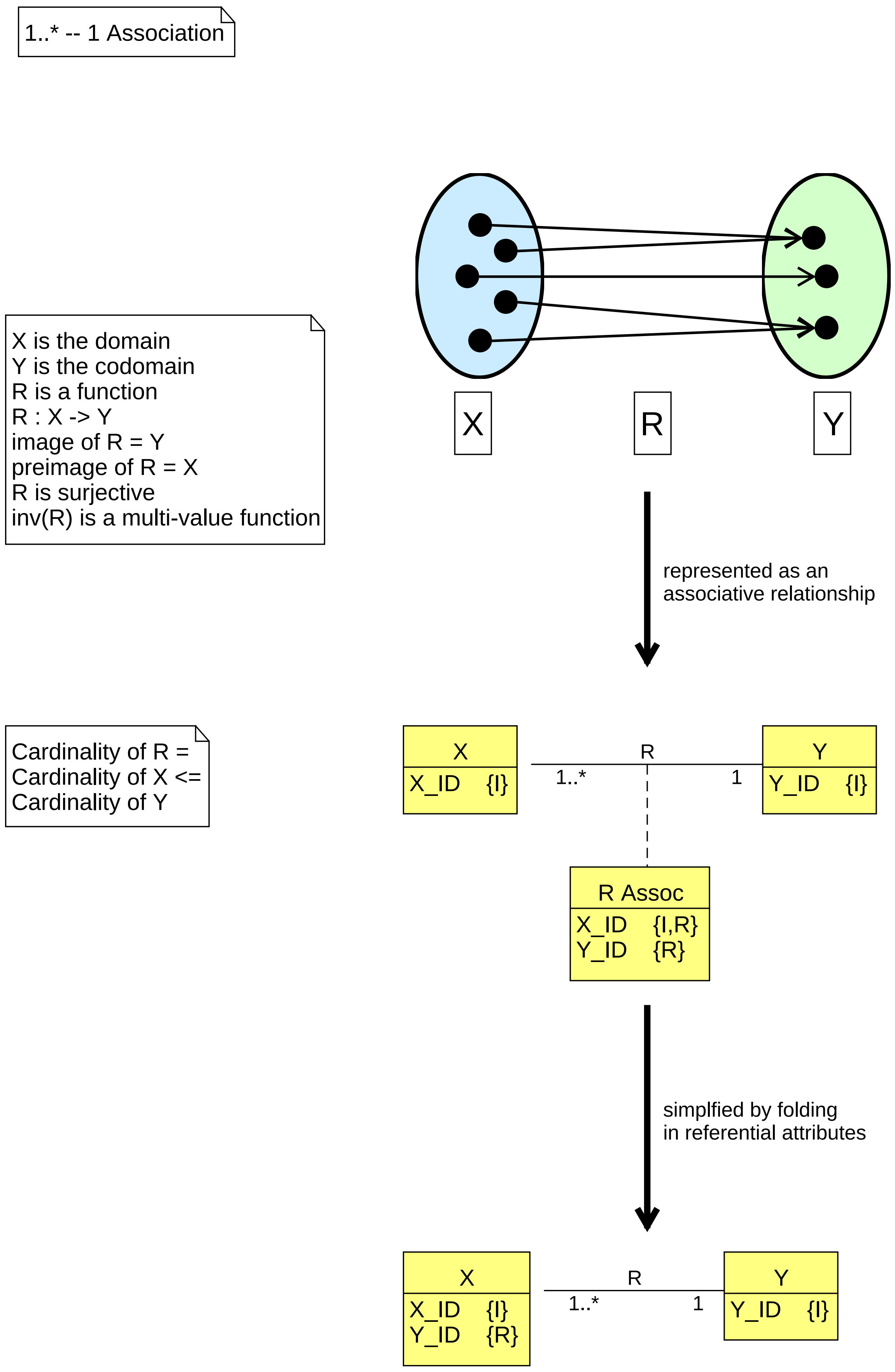


Figure 4: At least one-to-one Association

Any to One Association

The last case of simple associations is the any-to-one association as shown in the following figure.

The set-based graphic shows that some instances of **Y** are not mapped by **R** (shown by the red circles in **Y**). **R** is a function, but it is neither injective nor surjective. The inverse of **R** is both partial and multi-valued.

The UML diagram shows that **R Assoc** is identified by `X_ID` since the number of instances of **R Assoc** equals that of **X**. This association may also be simplified to eliminate the **R Assoc** class, including the `Y_ID` referential attribute in **X**.

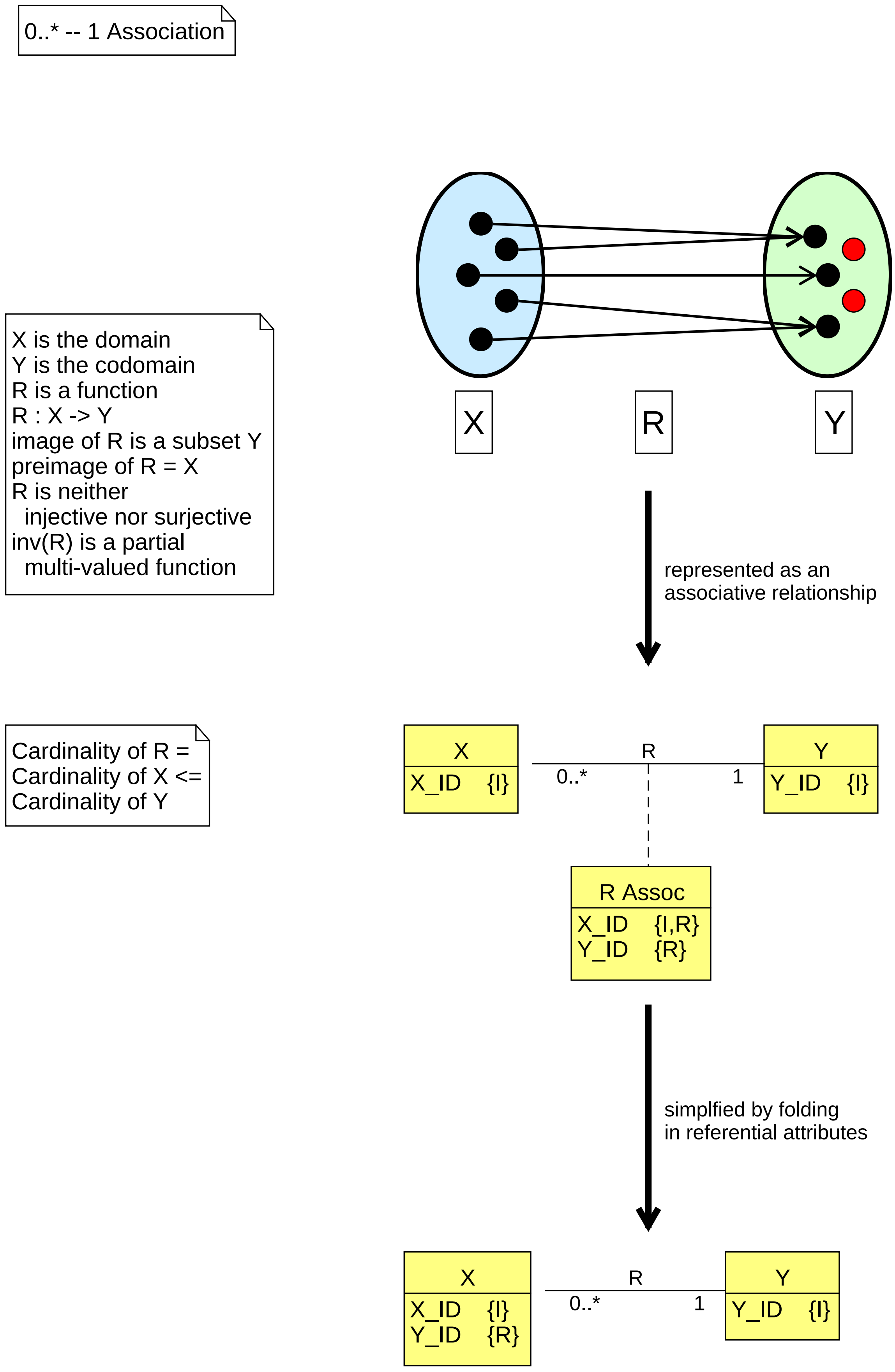


Figure 5: Any-to-one Association

At Most One to At Most One Association

The remaining permutations of multiplicity and conditionality are such that he simplification we were able to make in the previous four cases is not available. In the next three association types, the simplification is not available because of the conditionality of the association. For the three association types after that, the simplification is not available because of the multiplicity of the association.

The figure below shows a conditional one-to-one association. As before, **X** is the domain and **Y** is the codomain. In this case however, **R** is a partial function, *i.e.* not all elements of **X** have a mapping under **R**. It is also the case that not all elements of **Y** are mapped. Consequently, the image of **R** over **X** is a subset of **Y**, and the preimage is a subset of **X**. Note, that for some elements of **X**, **R** is undefined and, in the inverse case, for some elements of **Y**, the inverse of **R** is undefined.

When represented in UML notation, it is necessary to have an associative class (**R Assoc** in the figure), which explicitly enumerates all the instances of **R**. Because of the singular nature of the association, both the **X** identifier and the **Y** identifier are identifiers of **R Assoc**.

Past formulations of xUML have advised simplifying this type of association by folding an identifying attribute into **X** or **Y** as we had done in the previous cases. This is problematic because such a simplification cannot represent the conditionality without resorting to the use of **NULL** or some other *special value* for the referential attribute. Since we have required that every attribute contain a value from it defined data type and since **NULL** is not a value of any data type, a referential attribute value cannot indicate the conditionality of the association. Using a special value of for the attribute to indicate that the instance is not related is also not acceptable. Since data types define all the allowed values, there are no special ones in the set. Often, special values are chosen as being a value in some underlying system supplied data type that happens not to be used in the subset of the system type used by the attribute. For example, we might choose -1 as a special value for an attribute implemented in a system supplied integer type knowing that only non-negative values of the system type are used as attribute values. But this simply conflates unwelcome implementation constructs into the model. Special values also imply special processing in the activities of the model to recognize the conditionality of the association. Recognizing that the instances of the association class have a one-to-one correspondence to the instances of the relationship saves all the special cases of **NULL** and special values. We can recognize that the relationship is conditional when navigating from one participant to another yields the empty relation for the destination class. Some analysts object to adding another class to the class model. We fail to understand such reluctance and think the association classes convey significant meaning about the real-world association being modeled. Translation mechanisms may choose any convenient means at their disposal if they wish to optimize away the association class for this type of association.

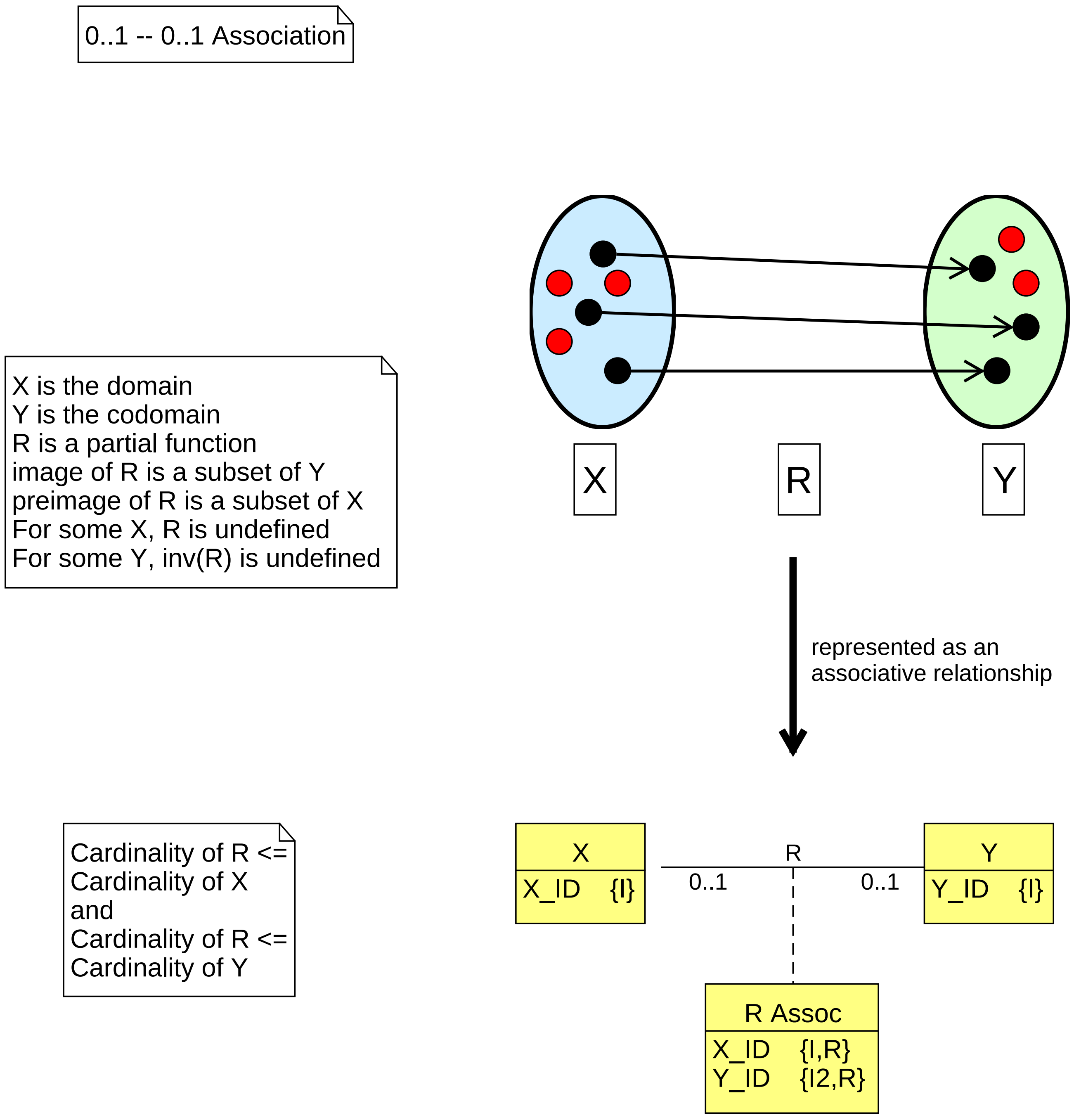

0..1 -- 0..1 Association
X is the domain
Y is the codomain
R is a partial function
image of R is a subset of Y
preimage of R is a subset of X
For some X, R is undefined
For some Y, inv(R) is undefined
X
R
Y
represented as an associative relationship
Cardinality of R <= Cardinality of X and Cardinality of R <= Cardinality of Y
X
X_ID {I}
R
0..1
0..1
Y
Y_ID {I}
R Assoc
X_ID {I,R}
Y_ID {I2,R}

Figure 6: At most one-to-at most one Association

At Least One to At Most One Association

The figure below shows a conditional many-to-one association. In this case, all instances of **Y** are mapped, but some instances of **X** (*i.e.* those shown in red) do not participate in the function. Thus, **R** is a partial function. Since the image of **X** under **R** is **Y**, we can think of **R** as begin a partially injective function. The preimage of **R** is a subset of **X** and the inverse of **R** is a multi-value function.

When represented in UML class diagram notation, the **R Assoc** class is the associative class. Instances of **R Assoc** represent instances of the class. Note that the identifier of **X** is sufficient to identify the **R Assoc** instances.

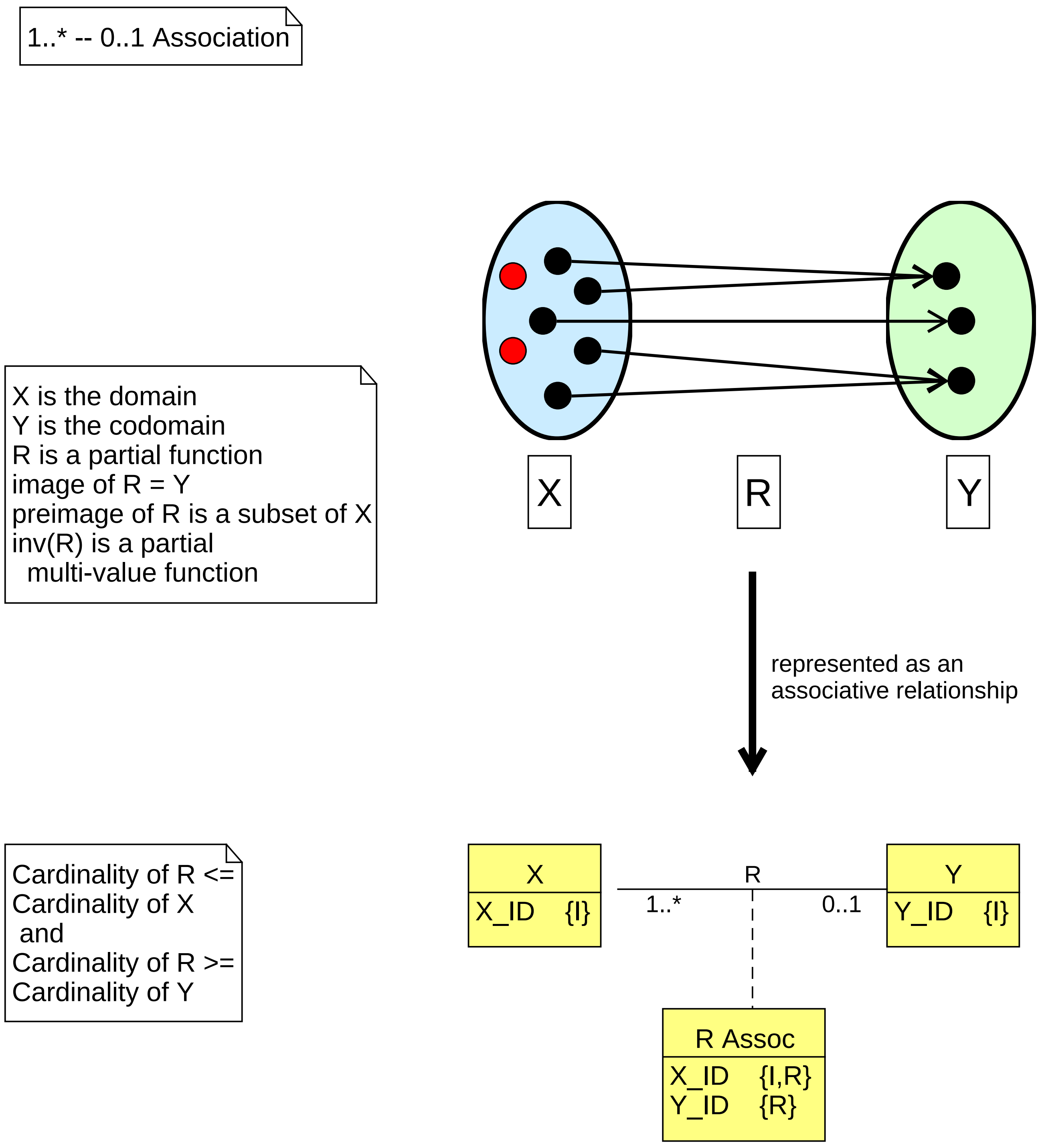

1..* -- 0..1 Association
X is the domain
Y is the codomain
R is a partial function
image of R = Y
preimage of R is a subset of X
inv(R) is a partial multi-value function
X
R
Y
represented as an associative relationship
Cardinality of R <= Cardinality of X and Cardinality of R >= Cardinality of Y
X
X_ID {I}
R
1..*
0..1
Y
Y_ID {I}
R Assoc
X_ID {I,R}
Y_ID {R}

Figure 7: At least one-to-at most one Association

Any to At Most One Association

The many-to-one biconditional association is shown in the following figure. This association allows for instances in **X** not to participate in **R**. So not only is **R** a partial function so is the inverse of **R**. When defined, the inverse is also multi-valued.

With this multiplicity and conditionality, the cardinality of **R Assoc** is less than or equal to the cardinality of both **X** and **Y**. This implies that there are no simplifications available to dispense with the **R Assoc** class. Note that the identifier of **X** (as also a referential attribute) serves as the identifier of **R Assoc**.

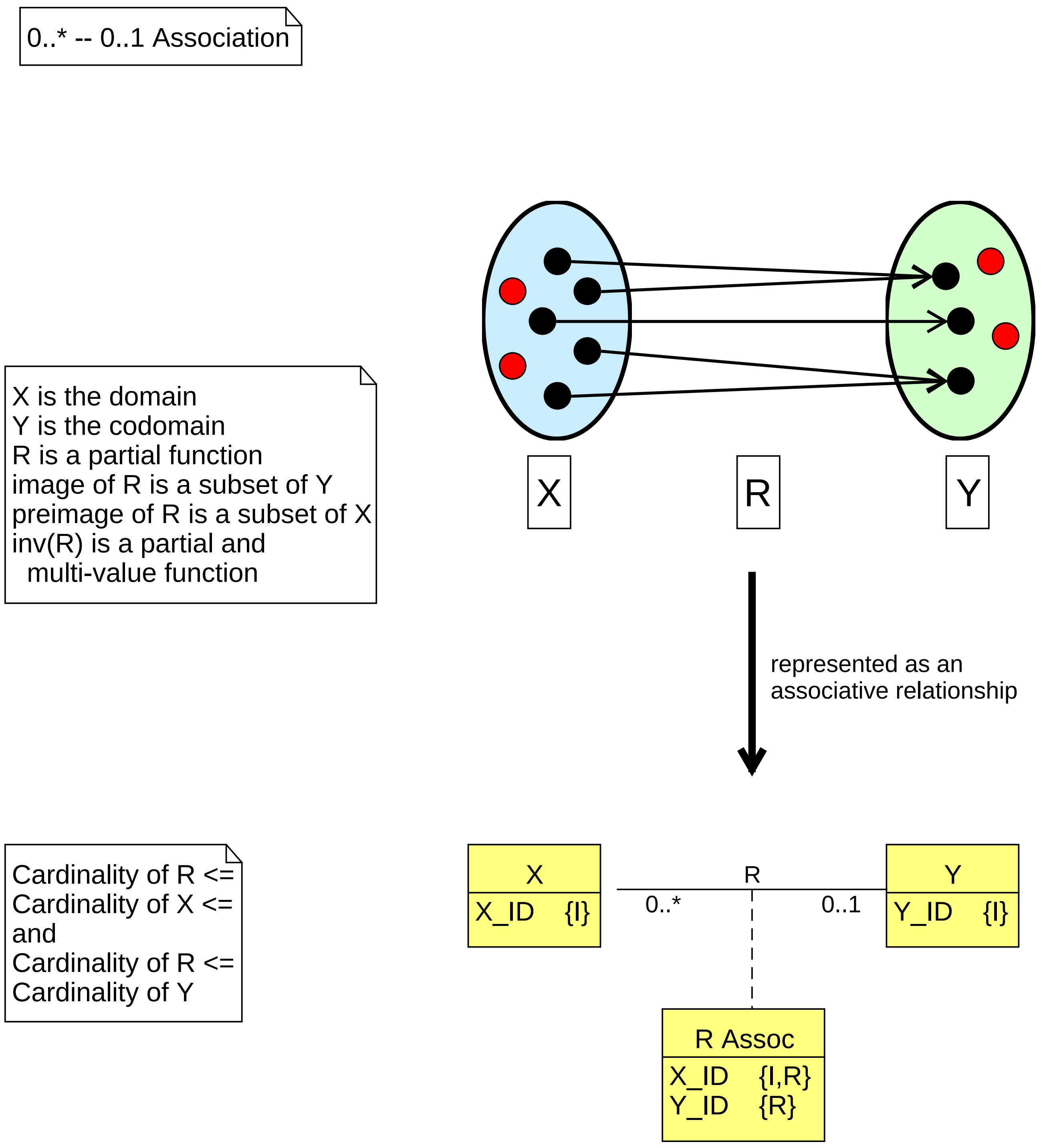


Figure 8: Any-to-at most one Association

At Least One to At Least One Association

The last three cases involve many-to-many associations. In all these cases, the associative class is a subset of the Cartesian product of the two participant classes.

The following figure shows the many-to-many unconditional case. The set functions are multi-valued. The cardinality of **R Assoc** is greater than or equal to the cardinality of both **X** and **Y**. So, to identify an instance of **R Assoc** requires both the **X** and **Y** identifiers (which are also used referentially).

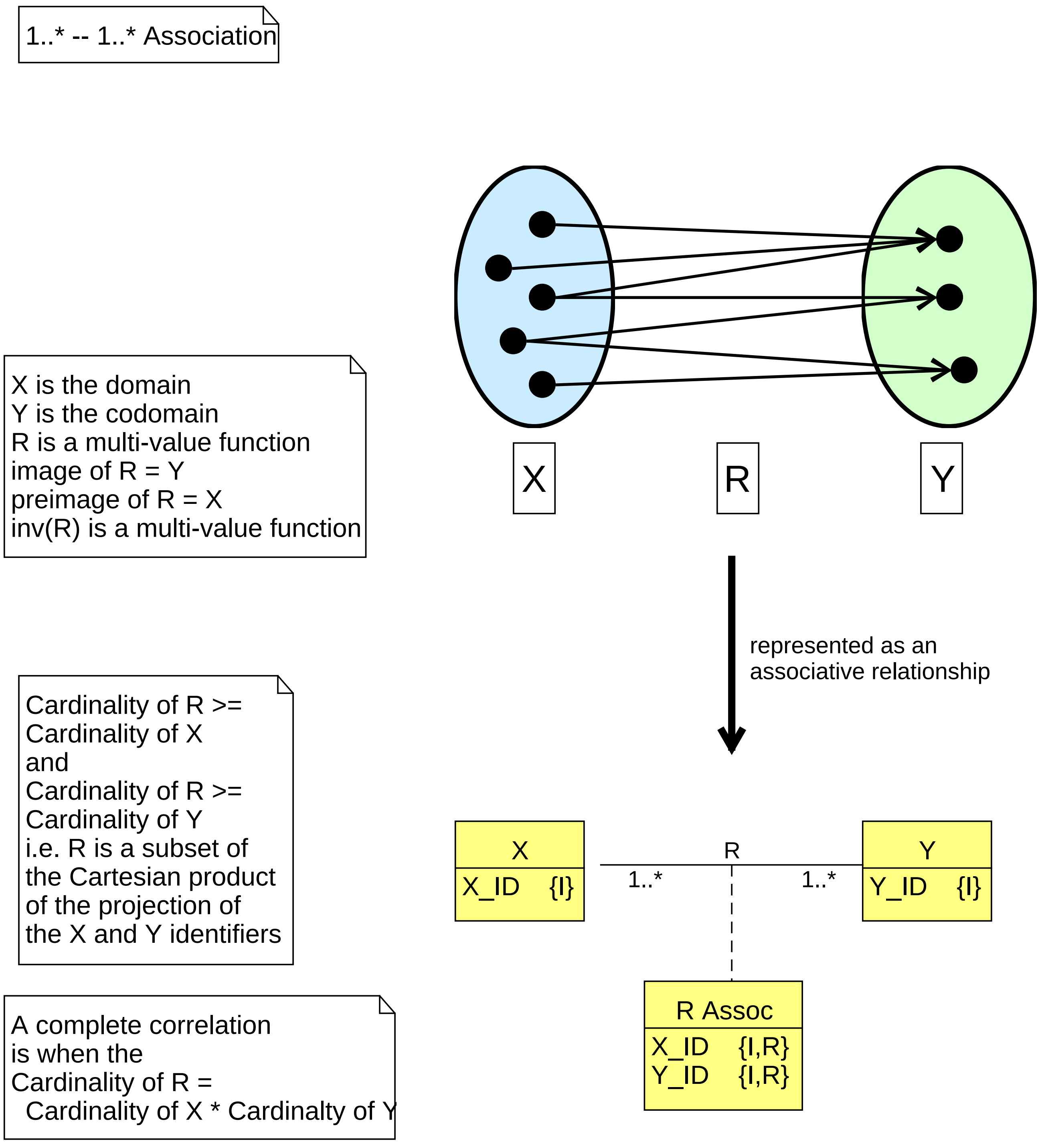

1..* -- 1..* Association
X is the domain
Y is the codomain
R is a multi-value function
image of R = Y
preimage of R = X
inv(R) is a multi-value function
X
R
Y
represented as an associative relationship
Cardinality of R >= Cardinality of X and Cardinality of R >= Cardinality of Y i.e. R is a subset of the Cartesian product of the projection of the X and Y identifiers
A complete correlation is when the Cardinality of R = Cardinality of X * Cardinalty of Y
X
X_ID {I}
R
1..*
1..*
Y
Y_ID {I}
R Assoc
X_ID {I,R}
Y_ID {I,R}

Figure 9: At least one-to-at least one Association

At Least One to Any Association

The many-to-many and conditional on one side association allows instances of **X** (as shown in red in the following figure) not to be related to any instance of **Y**.

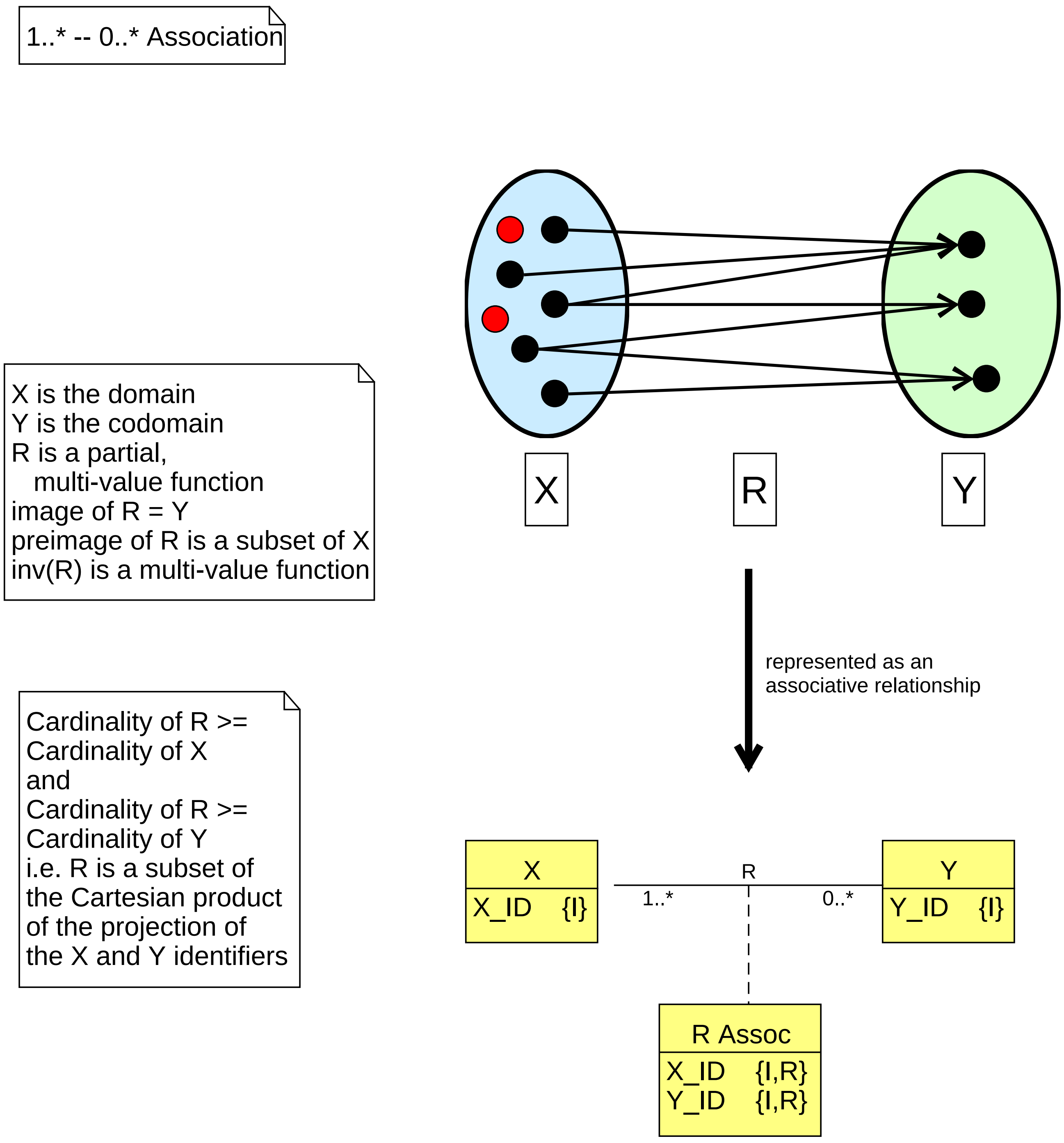


Figure 10: At least one-to-any Association

Any to Any Association

The many-to-many biconditional case allows instances from both **X** and **Y** to not be related as shown in the following figure.

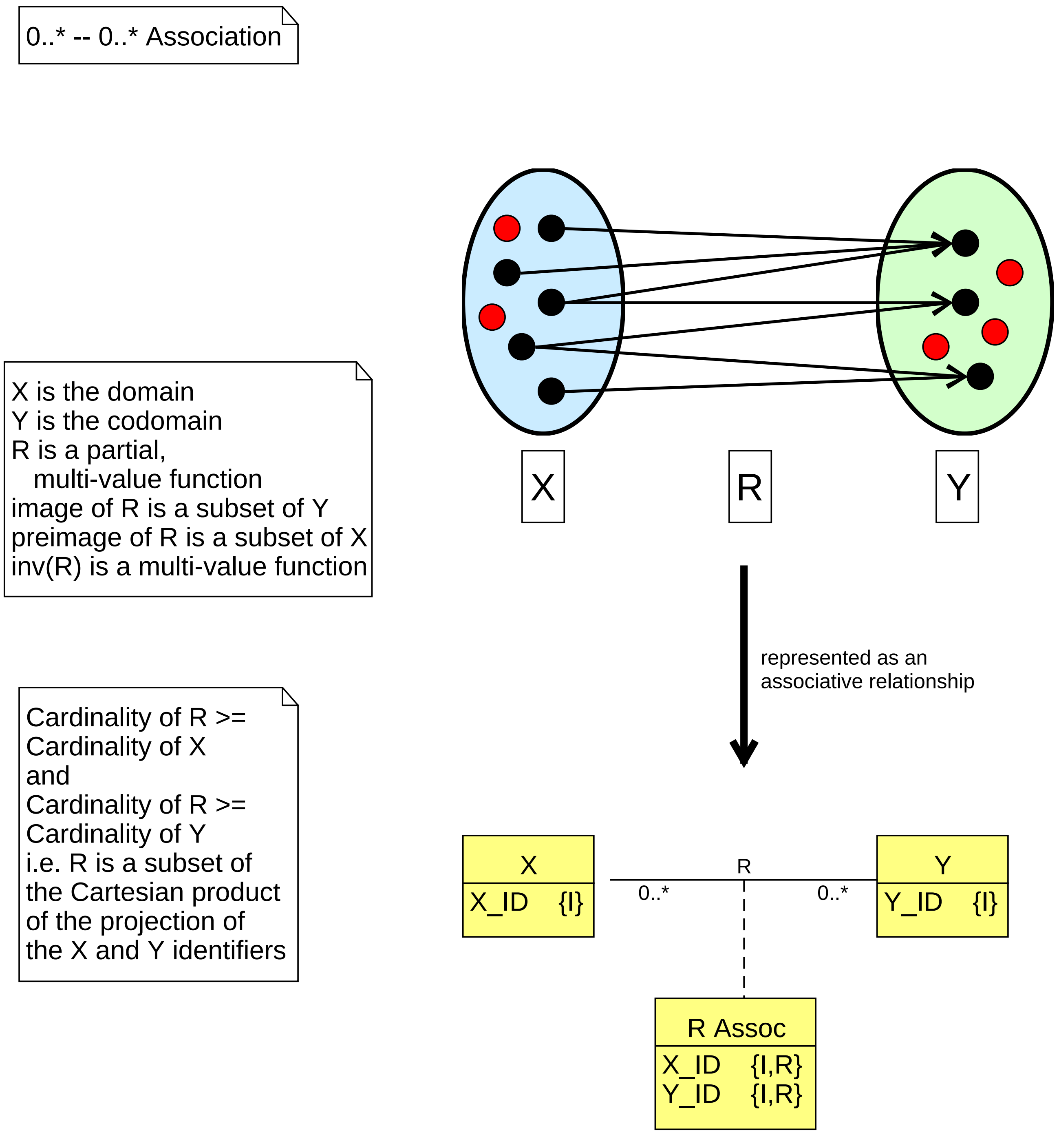


Figure 11: Any-to-any Association

Reflexive Association Relationships

In the previous examples, we have assumed that the **X** and **Y** classes are distinct. Reflexive associations are also possible where the participant classes are the same. Since the association is between members of the same set, it is not possible to have the conditionality of the association different on the two ends. This reduces the number of distinct combinations for reflexive associations to six, namely:

- One *to* one (1 *to* 1)
- At least one *to* one (1..* *to* 1)
- At least one *to* at least one (1..* *to* 1..*)
- At most one *to* at most one (0..1 *to* 0..1)
- Any *to* at most one (0..* *to* 0..1)
- Any *to* any (0..* *to* 0..*)

We do not provide any diagrams for the reflexive case since they are a simple variation on the diagrams provided previously.

Ambiguity in Reflexive Associations

Since a reflexive association relates instances of the same class, anytime that an associator class is used to manifest a reflexive association, there is an ambiguity when specifying how the instances are related. For the non-reflexive case, the class name along with an instance reference is sufficient to determine the referential attribute values for the associator class. For the reflexive case, the class name is the same and we recommend translation mechanisms allow an arbitrary direction be placed on the association. Then the terms *forward* and *backward* can be used to disambiguate the participating instances. This is also in keeping with the notion that the forward direction of a relationships is from the instance with referential attributes to referenced instance.

Typically, xUML class models contain semantic phrases attached to each end of an association. Some action languages have used these phrases for specifying the associations for reflexive instances. We avoid that technique on the basis that model level notation is just that, notation.

Summary of Association Relationships

Some insight can be gained by putting the conditionality and multiplicity combinations into tabular form[2]. The cells of the matrix contain an encoding of the properties of the association. The key to the encoding is given following the matrix.

Table 1: Matrix of Association Relationships

	1	**0..1**	**1..***	**0..***
1	F R			
0..1	F	P R		
1..*	F R	P	R	
0..*	F	P R	-	R

F ⇒
: Association is a function. Associations that are functions may use the simplified form for that case where there are no descriptive attributes of the association itself.

P ⇒
: Association is a partial function.

R ⇒
: Association may be reflexive.

The matrix shows that associations that are functions are the only ones for which the simplified form (*i.e.* eliminating the association class) are allowed. This is not surprising given the special nature of functions in the larger world of mathematics.

[2] The idea of using a matrix to show the conditionality and multiplicity comes from a private communication with Paul Higham.

xUML Operations on Association Relationships

We specifically want to develop xUML operations associated with relationship instantiation that do not involve directly supplying the values of referential attributes. This may seem strange, but eliminating the specification of referential attribute values as arguments to the operations eases the translation to architectures that do not use an underlying relational data management scheme without compromising the relational foundations of model. As long as we are still able to formulate the operations in relational algebraic terms, we are still on sound ground and have a more useful formulation of the relational principles in xUML. Using instance references to the participating classes, a Model Execution (MX) domain can determine the values of the identifiers of the participants and thereby know the values to use for the referential attributes. This view is also in keeping with our stance that xUML data operations are an application of relational theory and not a direct clone of it.

Creating Class Based Association Instances

For class based associations an instance of the association is created when an instance of the associator class is created. From a relational point of view, we must supply a value for every attribute whenever we create an instance and that includes the referential attributes. So we can formulate a creation rule for xUML that says:

> Creating an associator class requires supplying instance references to the two participating instances in addition to the values of any other attributes.

By knowing instance references to the participating classes, an architecture can determine the values of the identifiers of the participants and thereby know the values to use for the referential attributes.

Creating Simple Associations

For the case of simple associations, the association instance is created whenever an instance of the S class is created. So, the same rule applies. If a class is the referring source class for a simple association, then the creation operation must be supplied with an instance reference to the related instance.

Deleting Association Instances

Again from the relational, an instance of a class based association is deleted whenever the associator instance is deleted. The same is true for deleting a source instance in a simple association. The instance is gone, the referential attributes are not accessible and whether or not that leaves the data in a consistent state is determined at the end of the data transaction.

Updating Association Instances

From the relational point of view, we can update an association by updating the values of the referential attributes. Since we formulate the xUML operation using referential attributes, we supply the values of referential attributes indirectly using an instance reference. So an association instance can be updated by supplying an instance reference to the new instance. The MX domain can determine the values of the identifiers of the instance and use those values to update the referential attributes.

Implications for Statically Typed Implementations

When translating to a computing target with a conventional programming language, MX domains that hold all their data in primary memory usually choose to construct an identifier for instances to be the memory address of the object and to use that constructed identifier to implement the referential aspects of the data model for both consistency and navigation. Indeed part of the reason to formulate the association operations for xUML without referential attributes is to support this type of translation. The advantage of this implementation technique is just so great it cannot be ignored. MX domains constructed in this way use pointers in place of referential attributes. In all the cases we discuss, a single, non-NULL, address will implement the referring nature implied by the set of referential attributes that realizes the relationship at the model level.

However, to navigate the relationships in the opposite direction (*i.e.* the direction from the referenced instance to the referring instance), pointer based MX domains invariably insert additional pointers into the instance structures. These pointers alleviate the need to perform a search of the pool of instances in order to go *backward* in a relationship navigation[3]. Strictly speaking, the added pointers are redundant. Backward navigation could be performed via a search at some additional computational expense. We are usually unwilling to suffer the added computation and accept the space / speed trade-off of consuming more memory to achieve faster navigation. Because the backward links are redundant, it is the responsibility of the MX domain to deal with them in terms of keeping them up to date and consistent when created or modified. These pointers form a set of *back links*.

Generalization Relationships

A generalization relationship is complementary to an association relationship. Where associations are based on mapping between instance sets, generalizations are based on discriminating between instance sets.

The following figure depicts a generalization relationship in both set terms and in UML notation.

[3] Here, backwards navigation is from, in model level terms, the referenced instance to the referring instance.

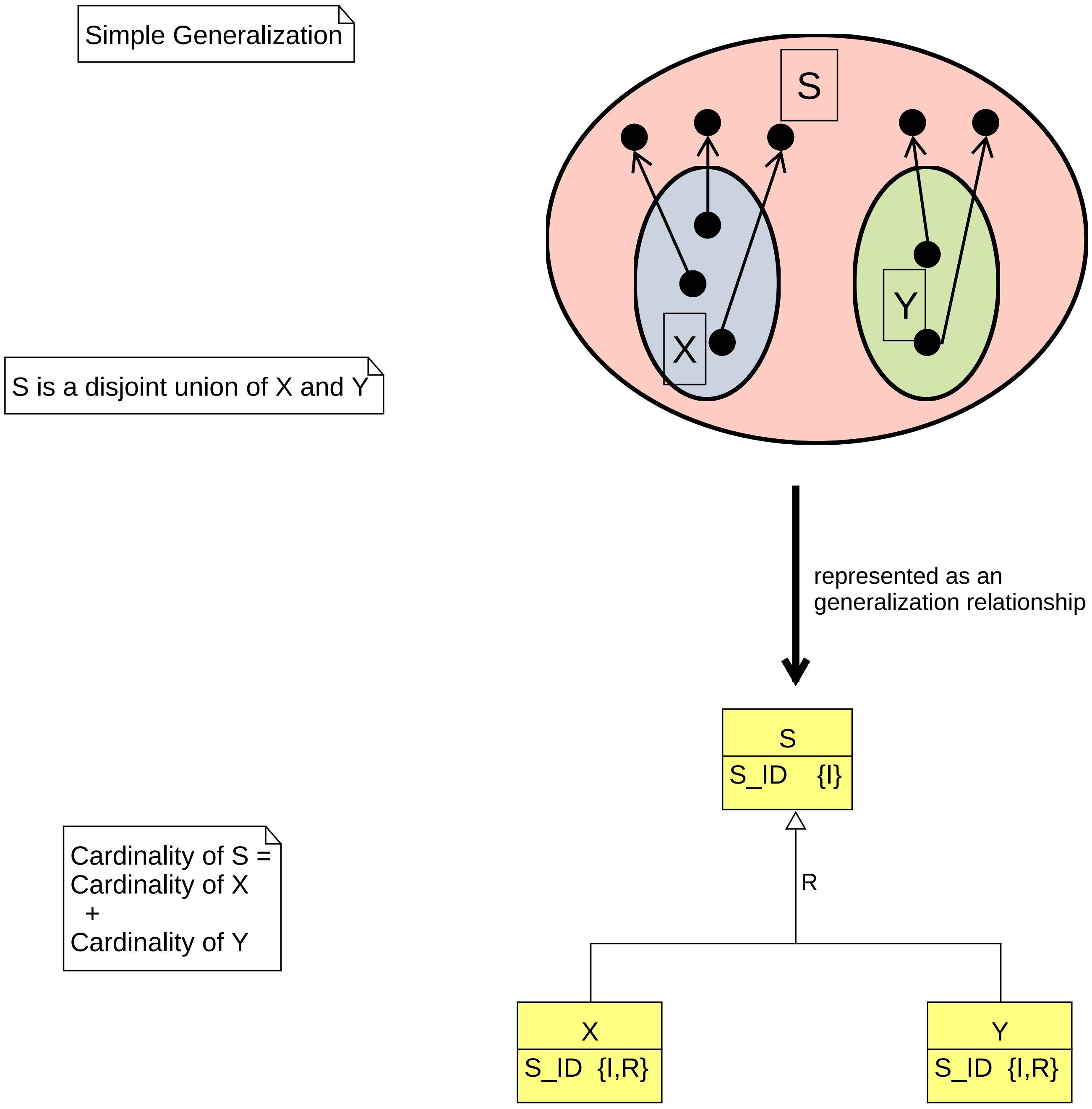

Simple Generalization
S is a disjoint union of X and Y
Cardinality of S = Cardinality of X + Cardinality of Y
S
X
Y
represented as an generalization relationship
S
S_ID {I}
R
X
S_ID {I,R}
Y
S_ID {I,R}

Figure 12: Simple Generalization Relationships

In the figure, **S** is the *superclass* and **X** and **Y** are *subclasses*. For simplicity, we show only two subclasses, but there may be an arbitrary number of subclasses. For a generalization, the instances of **X** and **Y** form a disjoint union of instances of **S** [4]. This definition has a number of implications:

- Each subclass instance is related and refers to exactly one superclass instance.
- Each superclass instance is related to exactly one subclass instance from among all the participant subclasses.

[4] More precisely, the projection of the identifier for the subclasses is a disjoint union of the projection of the identifier of the superclass. An equivalent statement is that union of the projection of the identifier for the subclass instances is equal to the projection of the identifier for the superclass and that the intersection of the projection of the identifier for the subclass instances is empty.

- An identifier of the superclass servers as an identifier of the subclass. The subclass identifier attributes also serve as referential attributes.
- The subclasses of a generalization are equivalence classes and thus form a partition of the superclass.

The UML notation we use to represent a generalization relationship is less restrictive than our definition. To be strictly correct, we need to annotate the UML diagram with the comment that **R** is *disjoint* and *complete*. We have dispensed with the notation as diagram clutter since we are not interested in generalizations that are not a disjoint union.

Note also that no notion of inheritance is implied here, despite the usual interpretation of UML generalizations in object oriented programming language constructs. We do not use inheritance in our model level constructs. However, inheritance might be used when translating model to particular programming languages. Failing to keep the strict separation between the model and the implementation is a source of constant confusion in the use of this particular UML notation.

Composition of Generalizations

Generalization relationships may be composed together in several ways.

A *repeated specialization* is the case where a class serves the role of subclass in one generalization and superclass in another generalization. The following figure shows an example of repeated specialization.

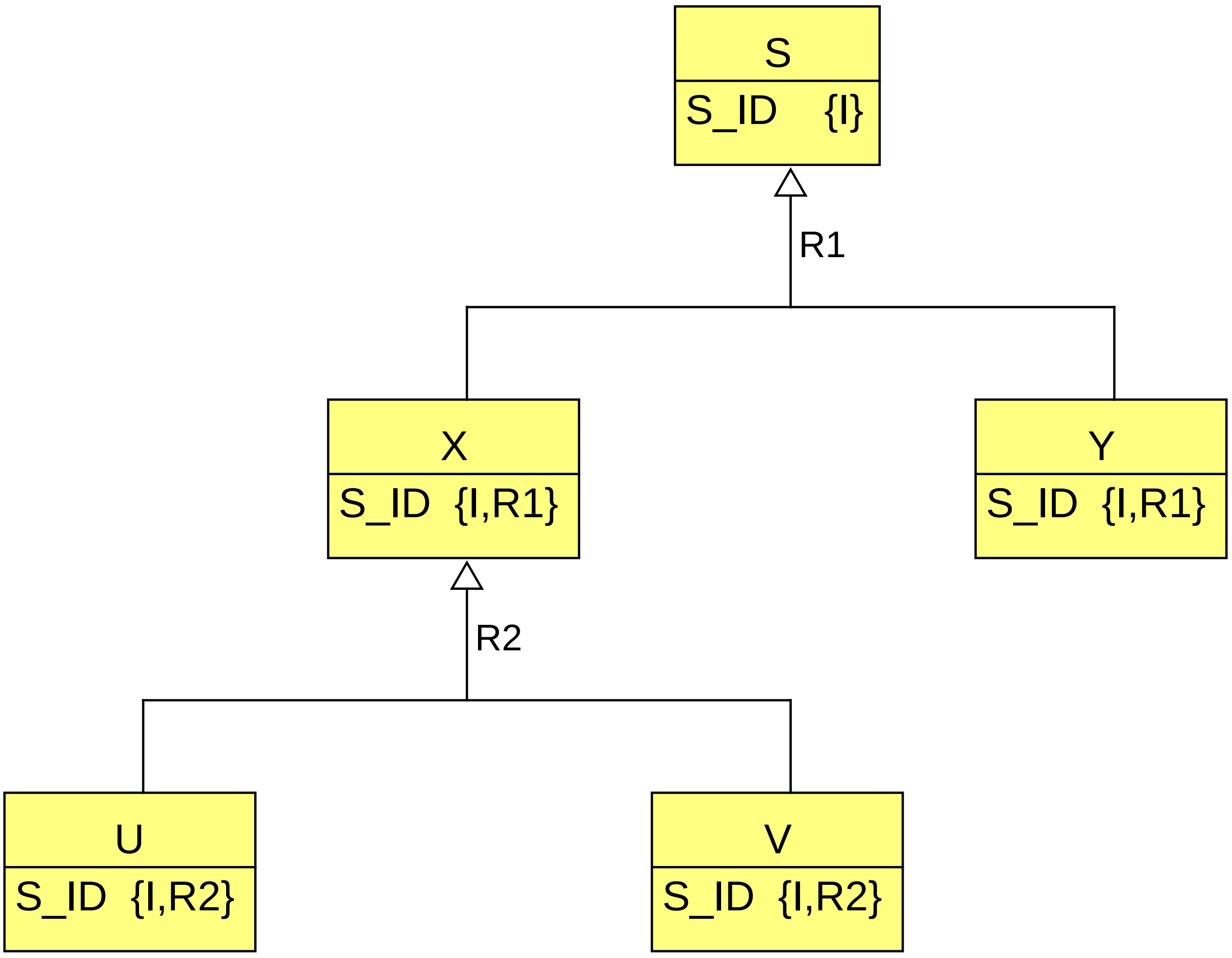


Figure 13: Repeated Specialization

In this figure, **X** is both a subclass of **R1** and a superclass of **R2**. Note that for every instance of **U** or **V** there are related instances of **X** and **S**.

A *multiple generalization* is the case where a class serves as the subclass for two independent generalization relationships. The following figure shows an example of a multiple generalization.

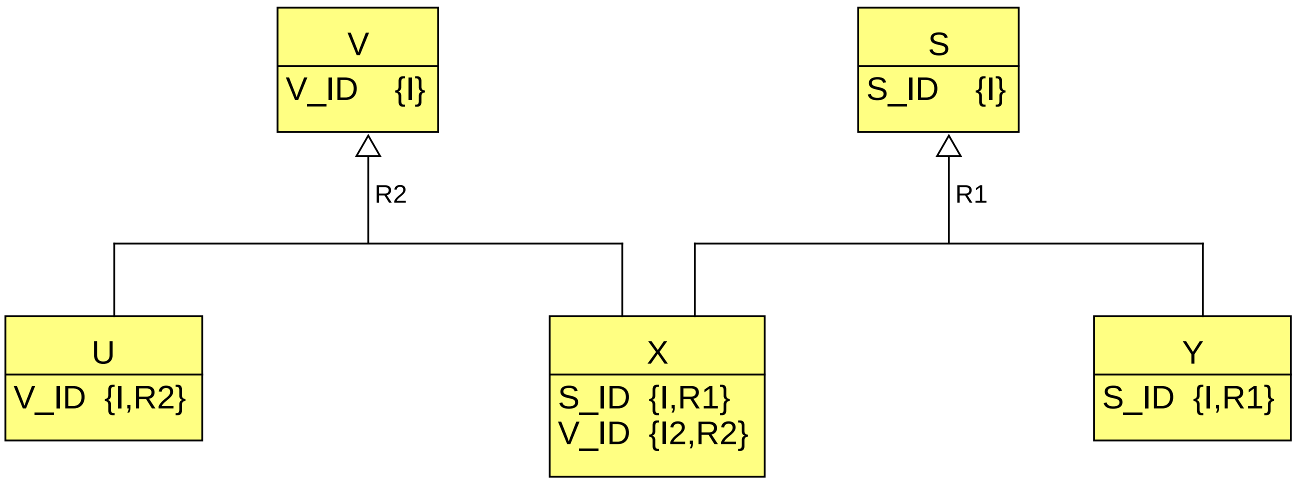


Figure 14: Multiple Generalization

In this figure, **X** is a subclass for both the **R1** and **R2** generalizations. Note that the so called *diamond* pattern, where a subclass in a multiple generalization has a common superclass ancestor, is not allowed since it would violate the singularity of the generalization between a subclass and superclass. So in the example figure, it is not possible for **V** and **S** to be subclasses of another generalization since a superclass instance must have exactly one related subclass instance and instances of **V** and **S** related to **X** must exist to satisfy **R1** and **R2**.

A *compound generalization* is the case where a class serves as the superclass for two independent generalizations. The following figure shows an example of a compound generalization.

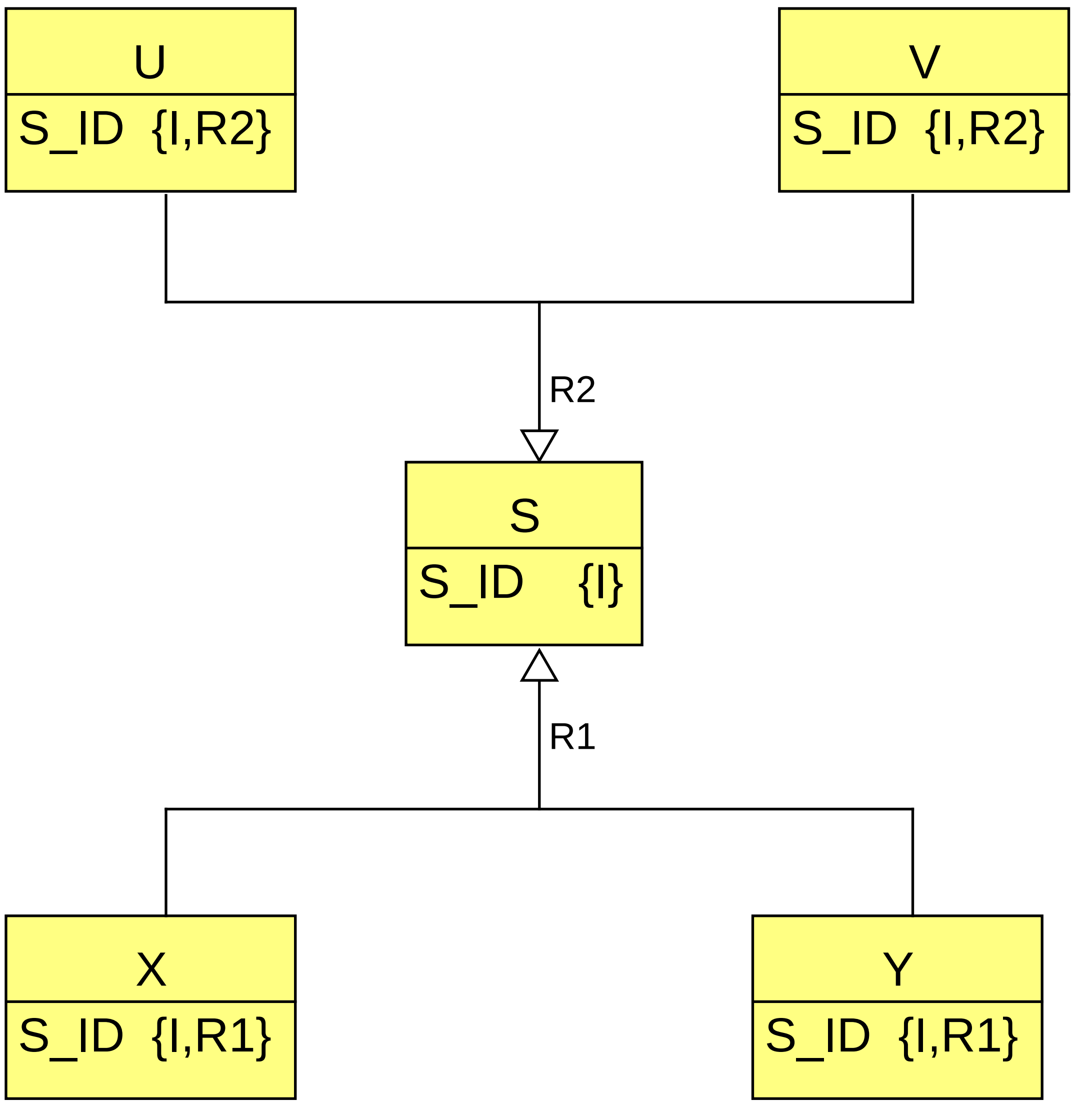


Figure 15: Compound Generalization

In this figure, **S** is the superclass for both **R1** and **R2**. The implication is that for each instance of **S**, there exists an instance of **U** or **V** *and* and instance of **X** or **Y**.

xUML Operations on Generalizations

Creating Generalization Instances

Given the one-to-one unconditional nature of the relationship between a superclass instance and a subclass instance in a generalization, when instances of the participating classes are created the entire hierarchy must be instanciated in a single transaction on the class model. Since subclass instances refer to their related superclass instances, the creation must happen from the *top down*. Just as in creating an instance of an association, the create operation for the subclass instance must be supplied with an instance reference for the related superclass instance. This implies that the superclass instance must be created first and hence the top down order that creating a generalization requires.

Deleting Generalization Instances

Similarly, deleting generalization instances requires deleting both the superclass and subclass instances in the same transaction on the data model. The order of deletion is arbitrary but navigating from subclass instance to superclass instance is more convenient (since one does not have to include logic about the subclass of the instance is actually related to a given superclass instance). Hence we tend to view generalization deletion as happening from the *bottom up*. Some care must be taken to navigate the generalization relationship to obtain the superclass instance before the subclass instance is deleted since deleting the subclass instance also destroys the generalization instance itself (as does deleting any instance that has referential attributes).

Reclassifying Generalization Instances

We also propose that action languages support a *reclassify* operation. Conceptually, reclassification is the deleting of a subclass instance followed by creating a new subclass instance that is related to the same superclass instance to which the deleted instance was related. Although there is no new fundamental operation involved in reclassification, we justify introducing it as a separate operation based on:

- It is a semantically significant and frequent action for generalization instances. When a generalization is used to model a *modal* lifecycle where subclasses have state models and events are made polymorphic, reclassification has a direct correspondence to a mode change. This is an effective way to model different behaviors for the same events that behavior varies over time given the changes in mode of the superclass instance as reflected by the reclassification of its subclass instances.
- When translating to some target environments, the translation mechanisms can make a significant optimization if they are aware that a reclassification is being undertaken. In these circumstances, the conceptual delete / create sequence can often be bypassed. Without a separate operation, one is reduced to determining reclassification as a combination of a delete followed by a creation. Deducing that an arbitrary action language sequence is actually a reclassification is difficult to accomplish precisely by traversing the syntax tree of an activity.

We also note that any `reclassify` operation must accept as arguments the values of any descriptive or referential attributes (excluding any reference to the superclass instance) that are present in the newly created subclass instance. This follows from the requirement that all attributes have valid values when an instance is created.

Summary

In this paper we have proposed a set of operations on relationship instances in xUML. Those operations are based on the following principles:

- All class attributes must have a valid value selected from the type of the attribute at all times, specifically those values must be supplied as part of the instance creation operation. There is no **NULL** value and no three-valued logic implied by the concept of **NULL**.
- Referential attributes values are supplied indirectly by using an instance reference to a class instance. The instance reference, combined with the model meta-data, provide a translation mechanism with the ability to determine the correct referential attribute values.
- Deleting an instance of a class which has referential attributes deletes the relationship instance to which referential attributes apply.
- A `reference` operation may be applied to associations to update the instance to which the association refers to be a different instance.
- A `reclassify` operation may be applied to generalizations to change the subclass instance to which a superclass instance is currently related.