ActiveCRL Simplified

# Introduction

This document explores in minute detail the thinking behind the CRL representations. It assumes that the reader has some familiarity with the CRL concepts (which can be found [here](https://github.com/pbrown12303/activeCRL/blob/master/activeCRL/docs/ActiveCRL.docx)). The purpose of the discussion is to explore the pros and cons of various representational alternatives including the nomenclature being used, ultimately arriving at the simplest possible structure for ActiveCRL.

Philosophically, the objective of this exercise is to create a minimalist model that provides the ability to represent anything that can be written down – particularly in a digital computer. It is believed that the present implementation is not minimalist.

# Concept Representation Language (CRL) Core

## Element

An Element is intended to represent a concept. The concept is identified with the conceptID[[1]](#footnote-1) attribute. This value is an immutable universal identifier for the concept and may never be altered.[[2]](#footnote-2)

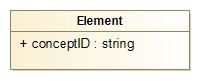


Figure 2‑1: Element AbstractiFigure 2‑2on

Each concept can be a part of another concept - this is the “black diamond” UML Composite shown in Figure 2‑3. The concept to which the part concept belongs is terms the owningConcept.

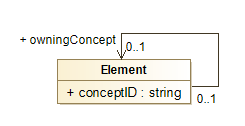


Figure 2‑3: Ownership Abstraction

A concrete data structure for representing this relationship is shown in Figure 2‑4. Note that from the data structure it is clear that the Element can only be owned by one concept. This value is allowed to be null: a concept does not have to be part of another concept.

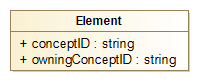


Figure 2‑4: Element Data Structure

## Reference

Sometimes the concept being represented is not a new concept at all, but rather a reference to an existing concept (Figure 2‑5). The Reference is a refinement (this concept will be discussed shortly) of Element with an attribute referencedConceptID[[3]](#footnote-3) that identifies the concept being referenced.

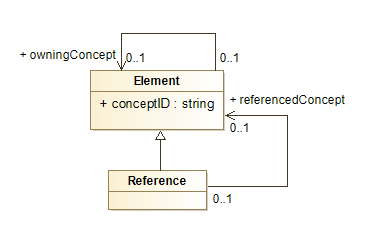


Figure 2‑5: Reference Abstraction

The data structure for representing a reference is shown in

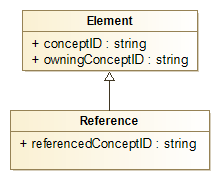


Figure 2‑6: Reference Implementation (partial)

References, as described above, indicate the Element (or its refinement) that is being referenced. But sometimes it is a specific attribute of that element and not the Element itself needs to be referenced. To support that we add a the referencedAttributeName to the Reference (Figure 2‑7). The AttributeName enumeration lists the allowed values. If the reference is to an Element (or refinement of Element), the AttributeName is NoAttribute. The remaining seven entries are the only non-Element values allowed in ActiveCRL and all are literal strings.

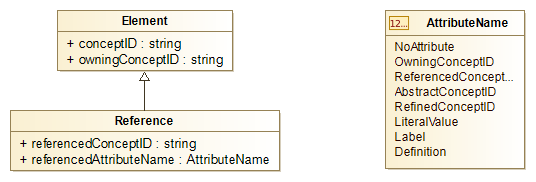


Figure 2‑7: Reference Implementation (complete)

## Literal (Value Holders)

The concepts represented by Elements are largely abstractions, identified only by the conceptID. While theoretically anything could be identified in such a manner, from a practical perspective sometimes the concept we want to represent is a symbol or string of symbols. Such a string could, indeed, be identified with a conceptID, but that still leaves us with the problem of representing the structure of that string and the set of symbols it comprises.

Modeling the structure of strings would require modeling symbols as concepts. This, of course, has been done many times: ASCII, EBCDIC, UTF-8, UTF-16, and UTF-32 are some common examples in which the “code” serves as the identifier and the relationship between the code and the symbol is given in graphical form. Fonts even add another dimension to this, providing an assortment of graphical variations for representing each character. Some fonts, such as Wingdings, provide graphical symbols that are unrelated to the normal character code meanings simply as a means of introducing new symbols.

While one might want to model these symbol and strings comprised of these symbols in ActiveCRL, we choose not to make strings and their modeling part of the core ActiveCRL model. Instead, we introduce the concept of a Literal value holder, which we shall simply call a Literal (Figure 2‑7, Figure 2‑8). The Literal is a refinement of Element[[4]](#footnote-4) and has an attribute literalValue to hold a string of literals. The actual encoding to be used is an implementation design decision.

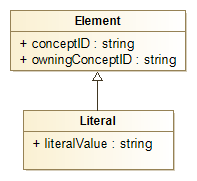


Figure 2‑9: Literal Implementation

## Refinement

Refinement is a binary directed association between concepts in which one of the concepts is identified as an abstract concept and another is identified as a refined variation of the abstract concept. This subsumes the UML notions of Generalization and Instantiation. The abstract representation of refinement is shown in Figure 2‑9, and the data structure refinement is shown in Figure 2‑10.

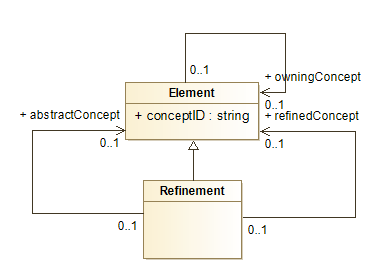


Figure 2‑10: Refinement Abstraction

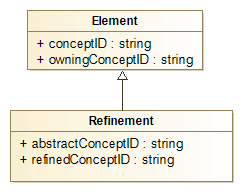


Figure 2‑11: Refinement Implementation

## Full CRL Core

The full CRL core model is shown in Figure 2‑11 and Figure 2‑12. [[5]](#footnote-5)

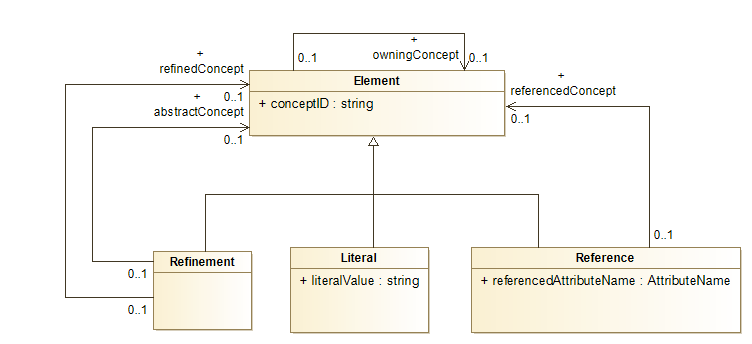


Figure 2‑12: CRL Core Abstraction

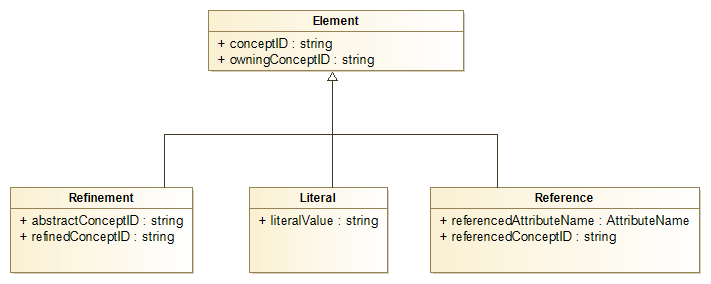


Figure 2‑13: CRL Core Implementation

# CRL Convenience Features

While identifiers may provide an unambiguous way to identify a concept, they are not expressive in human terms. In this section we describe three additional Element attributes: URI, label, definition.

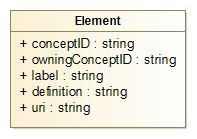


Figure 3‑1: Element with Convenience Attributes

A label provides a human-readable term for the concept. This should not be confused with the richer notion of a name: concepts can have many names (perhaps in different languages). In CRL, the idea of Name would be modeled as an explicit concept and the relationship between the Name concept and the Element having that name would be modeled explicitly as well.

The definition provides a human-readable description of the concept.

A URI provides a means of associating a separate, fixed, and meaningful identifier with a concept.

All core concepts are assigned URLs as follows:

Domain: https://activeCrl.com/core/CoreDomain  
Element: <https://activeCrl.com/core/Element>  
Literal: [https://activeCrl.com /core/Literal](https://activeCrl/core/Literal)  
Reference: [https://activeCrl.com /core/Reference](https://activeCrl/core/Reference)  
Refinement: [https://activeCrl.com /core/Refinement](https://activeCrl/core/Refinement)

When a concept is created, its conceptID (a UUID) is normally randomly generated. As a convenience in the implementation, an Element can be constructed with its conceptID derived from a URI so that both the URI and the UUID remain constant. Using this capability, for example, the core domain can be reconstructed programmatically. The benefit is that any for any other concepts that reference the core concepts, the reference would remain valid since the conceptID would remain the same.

# ActiveCRL

## ActiveCRL Data Structures

ActiveCRL adds the ability to dynamically modify CRL data structures and for Elements (and their refinements) to not only represent functions that respond to those changes but also to automatically execute implementations of those functions. Several attributes are added to support this (Figure 4‑1). The isCore attribute indicates that the concept is one of the core concepts of CRL and may not be modified. The user-settable readOnly attribute. After an element is created, this attribute may be set, after which it cannot be modified. The version attribute that is automatically incremented any time a change is made to an element or its descendants (ownedConcepts, recursively). Finally, there is the observers attribute, which will be explained shortly.

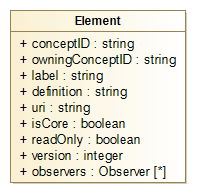


Figure 4‑1: ActiveCRL Element Implementation

When a CRL Element (or refined concept) is changed, it automatically sends change notifications to each reference that points to the Element. However, in implementation, there are sometimes classes that are not part of the CRL data structure but, nevertheless, need to be notified when changes occur. Generally these are classes that implement a user interface. If these classes implement the Observer interface, they may register (or deregister) themselves with specific Elements in order to be notified when they change. Each Element maintains a list of registered observers.

The Reference data structure is also enhanced with an additional attribute: the referencedConceptVersion (Figure 4‑2). This value is set automatically when the referencedConceptID is set. It is used when receiving notifications of change to determine whether the referenced concept has changed. After processing the notification, the value is set to the current version of the referenced concept.

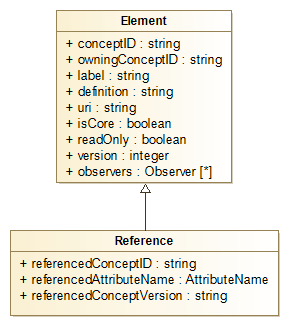


Figure 4‑2: ActiveCRL Reference Implementation

Similar extensions are added to the ActiveCRL Refinement (Figure 4‑3

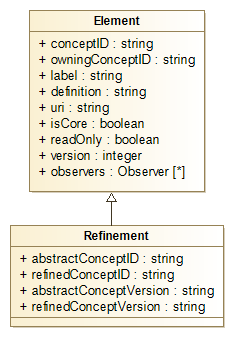


Figure 4‑3: ActiveCRL Refinement Data Structure

## Functions

Any Element with a URI can optionally have one or more functions associated with it. When a change occurs to an Element that is a refinement (directly or indirectly) of the Element associated with the functions, the functions associated with the ancestor are executed one at a time. There are two arguments to this function: the Element that changed and the ChangeNotification that delineates the nature of the changes.

Figure 4‑4 shows an example of an addition function being represented by an Element with a child establishing a URI for the function. An instance of the addition function would be represented by a refinement of the function, as shown in the diagram. Any change to the instance (for example, a change in one of its arguments) would cause the addition function to execute.

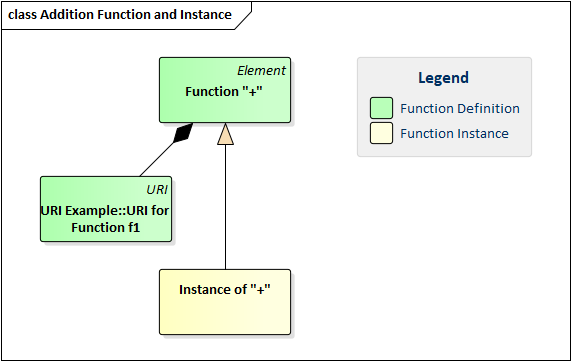


Figure 4‑4: Function Example: Addition Function and Instance

Most functions require arguments. While not strictly necessary, it is good practice to model the CRL structure expected by the function as part of the function representation (Figure 4‑5). The function has three children, arg1, arg2, and result, each represented by a Reference. Furthermore, each of these references points to a Literal, indicating that the expected value is a literal. This is an implicit constraint: refinements of these children are expected to reference literals or refinements of literals.[[6]](#footnote-6) This provides a full model of the concepts expected by the function.

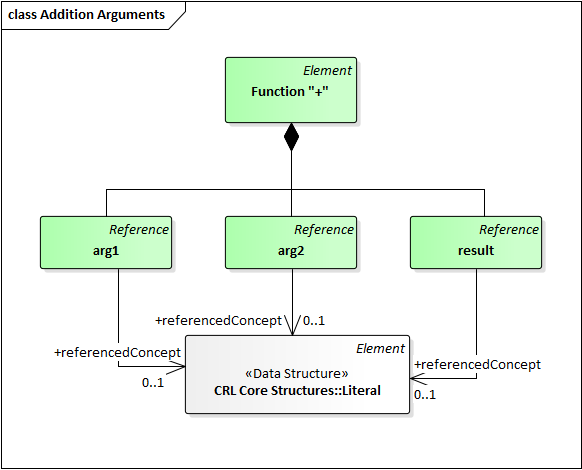


Figure 4‑5: Model (Prototype) of Function "+"

The resulting function model serves as a prototype for making an instance of that function: simply clone the structure and make each element of the clone a refinement of the corresponding element of the prototype. CRL provides a built-in function, CloneAsRefinement, that provides this capability. Applying this function to the addition prototype would yield the structure of Figure 4‑6. Note that none of the three references point to anything immediately after cloning. When arg1, arg2, and result point to appropriate literals, then the function will add the arg1 and arg2 values and place the result as the literalValue of the result. Note that the function must be designed to be well-behaved if portions of the data structure are missing.

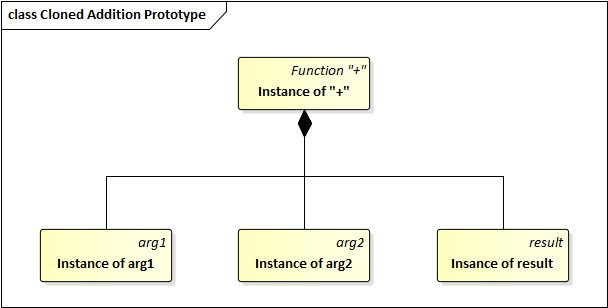


Figure 4‑6: Cloned Addition Prototype

The functions execute asynchronously after the lock on the changed Element has been released (see Locking below). The intent is that if there are multiple functions executing that they may be executed in parallel.

Of necessity, all of the function implementations are language specific (the reference implementation is in Go). All of the arguments to the functions are CRL structures. The CRL reference implementation provides all of the functions necessary to create and modify CRL structures along with CRL representations of each of those functions. Thus a program for creating and manipulate a CRL structure can itself be written as a CRL structure.

## Notification

To make CRL active, elements must be aware of changes that occur in related elements. This awareness has two components: the ChangeNotification and ChangePropagation.

### ChangeNotification

The ChangeNotification is the mechanism for communicating changes. The types of changes are shown in Figure 4‑7.

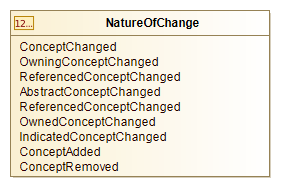


Figure 4‑7: Change Types

#### Direct Concept Changes

The first five identify the types of change that can be made to a concept.

* ConceptChanged is used to report any change to a concept that is not a change to a relationship with (a pointer to) another concept: pointer changes are reported separately.
* OwningConceptChanged is used to report a change to an Element’s pointer to its owner.
* ReferencedConceptChanged is used to report a change to a Reference’s referenced concept pointer.
* AbstractConceptChanged is used to report a change to a Refinement’s abstract concept pointer.
* RefinedConceptChanged is used to report a change to a Refinement’s refined concept pointer.

#### Forwarded Changes

The next two are forwarded notifications of change:

* OwnedConceptChanged is used to notify an Element that one of its owned concepts has changed. The actual change to the owned concept is conveyed as the underlying change
* IndicatedConceptChanged is used to notify an Element, Reference, or Refinement that one of the objects to which it has a pointer has changed. The actual change to the indicated concept is conveyed as the underlying change.

#### UniverseOfDiscourse Changes

The final types of change reflect the addition or removal of the concept from a UniverseOfDiscourse:

* ConceptAdded is used to report the addition of a concept to a UniverseOfDiscourse.
* ConceptRemoved is used to report the removal of a concept from a UniverseOfDiscourse.

Changes are communicated via the ChangeNotification data structure (Figure 4‑8). The natureOfChange indicates the type of change. The reportingElement is the element sending this particular notification.

For changes to a concept (ConceptChanged, OwningConceptChanged, ReferencedConceptChanged, AbstractConceptChanged, and RefinedConceptChanged), the beforeElementState is a snapshot of the concept prior to the change. For changes to a concept, the afterConceptState is a snapshot of the concept after the change.

For forwarded notifications (OwnedConceptChanged and IndicatedConceptChanged), the beforeConceptSpace and afterConceptSpace are null. The underlyingChange is the actual report of the change to the concept. If the type is one of the forwarding changes, the underlyingChange points to the notification that generated this one.

For UniverseOfDiscourse changes (ConceptAdded and ConceptRemoved), the beforeConceptState and afterConceptState are null.

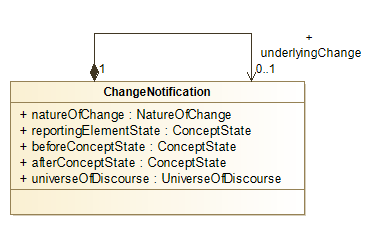


Figure 4‑8: Change Notification Data Structure

### Responding to Changes

There are two mechanisms in ActiveCRL for receiving change notifications, one intended for concepts within the UniverseOfDiscourse (UofD) so that they can respond to change notifications and possibly make further changes within the UofD; and the other to notify objects outside the UofD (components of the execution environment surrounding the UofD) so that they can respond to changes.

#### Notifications Within the *UniverseOfDiscourse*

Concepts within the UofD can have functions associated with them (see Section 4.2). Notifying concepts of change is accomplished via the UniverseOfDiscourse.callAssociatedFunctions(<listener>, <notification>, <transaction>) function. This function determines whether there are any functions associated with the listener (there can be more than one) and invokes each of the associated functions. This mechanism is the “Active” in ActiveCRL.

#### Notifications Outside the Domains

Objects in the working environment surrounding the domains may need to be notified of changes to concepts in the UofD. For example, the concept tree displayed in the CRLEditor needs to be informed about changes to the concepts being displayed in the tree. Environment objects that are interested in a concept can call Element.Register(<listener>) on the concept. In turn, when changes occur, the concept calls the <listener>.Update() method with the associated notification as an argument.

Environment objects that are interested in all changes to all concepts in the UofD can call Register() on the UofD itself. Listeners will be notified of all changes to the concepts in the UofD.

### Propagation of ChangeNotifications

When the state of a concept changes, the <concept>.propagateChange(<changeNotification>) function is called. The behavior of this function depends upon the nature of the change.

For state changes other than membership in the UniverseOfDiscourse, the following actions are taken:

* UniverseOfDiscourse.callAssociatedFunctions(<concept>,<stateChangeNotification>, <Transaction>) is called to invoke any functions associated with the concept whose state has changed.
* <concept>.NotifyListeners(<stateChangeNotification >, <Transaction>) is called to inform all elements with pointers to this concept that the concept has changed. This function creates an indicatedConceptChangedNotification with the stateChangeNotification as its underlying change. In this function, for each listener, two functions are called with this change:
  + UniverseOfDiscourse.callAssociatedFunctions(<listener>,< indicatedConceptChangedNotification >, <Transaction>) is called to invoke any functions associated with the concept.
  + <listener>.NotifyOwner(<indicatedConceptChangedNotification >) is called to inform the owner of the listener that the listener has received this notification.
  + <listener>.NotifyAll(<indicatedConceptChangedNotification >) is called to inform all direct observers of the concept.
* <concept>.NotifyOwner(<stateChangeNotification >, <Transaction>) is called to inform the owner of the change. In this function, if the concept has an owner, an ownedConceptChangedNotification is created with the concept as the reporting element and the stateChangeNotification as its underlying change. Two functions are called with this change (If the stateChangeNotification was an OwningConceptChanged, these functions are called for both the old owner and the new owner):
  + UniverseOfDiscourse.callAssociatedFunctions(<owner>,< ownedConceptChangedNotification >, <Transaction>) is called to invoke any functions associated with the concept.
  + <owner>.NotifyAll(<ownedConceptChangedNotification >) is called to inform all direct observers of the concept
* <concept>.NotifyAll(<stateChangeNotification >) is called to inform all direct observers of the concept
* <UofD>.NotifyAll(<stateChangeNotification >) is called to inform all direct observers of the Universe of Discourse of the change to the concept.

For changes in UniverseOfDiscourse membership (i.e. ConceptAdded and ConceptRemoved), only the UofD is notified:

* <UofD>.NotifyAll(<directChangeNotification>) is called to inform all direct observers of the Universe of Discourse of the change to the concept.

Note that there is no mechanism for membership change notifications to be sent to other concepts in the UofD: UofD membership is not part of the concept space.

# Constraints

Refinement requires that when a Reference is refined, the referenced element of the refinement be a refinement of the referenced element of the abstract Reference.

# Built-In Functions

### Element Functions

#### GetConceptId

### CloneAsRefinement

This function has two references: the PrototypeToBeCloned and the NewClone. The function takes the Element indicated by the PrototypeToBeCloned and replicates its structure recursively as defined by ownedConcepts.

* The derived concepts of Name, URI, and Definition are not cloned
* Each cloned element is made a refinement of its corresponding prototype element
* If a Reference’s referencedConcept points to an element that is, itself, cloned, then the cloned Reference’s referencedConcept points to the clone of the originally referenced element; otherwise the cloned Reference points to nil. The same holds true for the abstractConcept and refinedConcept of cloned Refinements

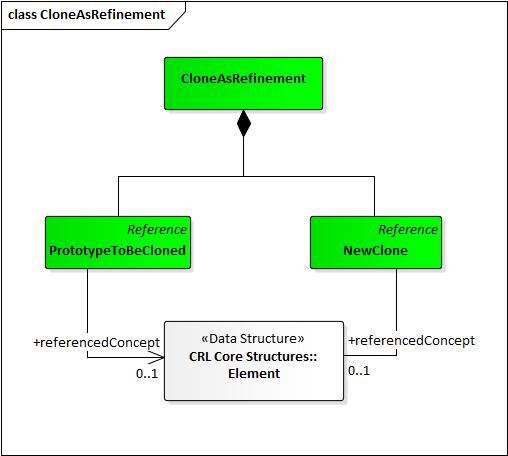


Figure 6‑1: CloneAsRefinement

# Mapping Domain

## CrlOneToOneMap

The CrlOneToOneMap is used to define a mapping between any two types of CRL Elements or between any two element attributes. This mapping does not support mappings between Elements and attributes. The map definition has a SourceReference and a TargetReference.

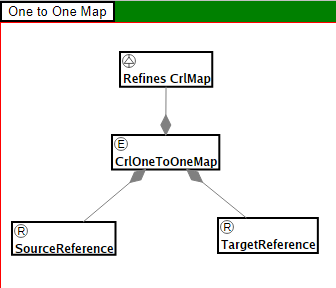


Figure 7‑1: CRLOneToOneMap

To define a map, one must first have a source and target data structure defined. These data structures represent the schema of the source and target data structures.

### Element Mapping

For each Element in the source data structure that you wish to map, a refinement of the CrlOneToOneMap is created with its SourceReference pointing to the Element of the source data structure and the TargetReference pointing to the Element of the target data structure. We will refer to this as the abstract map.

To execute the map, create a refinement of the abstract map and point the SourceReference to a refinement of the Element referenced in the source data structure. It is an error if the Element indicated by the SourceReference instance is not a refinement of the Element indicated by the abstract map’s SourceReference.

If these conditions are met, the map instance will create a refinement of the Element referenced by the abstract map’s TargetReference and make it the target of the map instance’s TargetReference.

The owner of the newly created Element depends upon the ownership of the abstract map.

1. If the owner of the abstract map is not another map, the owner of the map instance becomes the owner of the newly created Element.
2. If the owner of abstract map is another map, then the owner of the new Element will depend upon the Element indicated by the map instance’s owner’s TargetReference. If the parent map’s TargetReference is a refinement of the abstract map’s TargetReferencethe mapping will automatically set the owner of the

### Attribute Mapping

For each attribute of the source data structure that you wish to map, the SourceReference points to the source attribute and the TargetReference points to the target attribute.

1. This is the id attribute of the Element in the current implementation. [↑](#footnote-ref-1)
2. In the current implementation, the conceptID is a UUID. [↑](#footnote-ref-2)
3. In the present implementation the referencedConceptID value is held by another object called a Pointer (specifically, a ReferencedElementPointer). This pointer has its own identifier. It is conjectured that this concept of Pointer is not necessary. [↑](#footnote-ref-3)
4. In the present implementation Literal is not a refinement of Element but rather a member of the Value branch of the type hierarchy that is parallel to the Element branch. [↑](#footnote-ref-4)
5. [↑](#footnote-ref-5)
6. A richer model would refine Literal further to represent a Number and the references would then point to Number. [↑](#footnote-ref-6)