A Pedantic Walk Through ActiveCRL

# 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.

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)

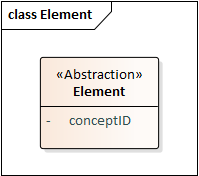


Figure 1: Element Abstraction

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

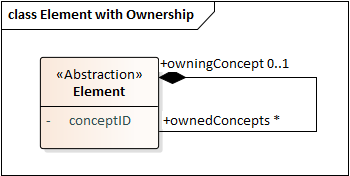


Figure 2: Ownership Abstraction

A concrete data structure for representing this relationship is shown in Figure 3. 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.

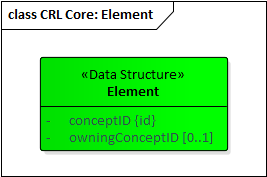


Figure 3: Element Data Structure

## Reference

Sometimes the concept being represented is not a new concept at all, but rather a reference to an existing concept (Figure 4). 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.

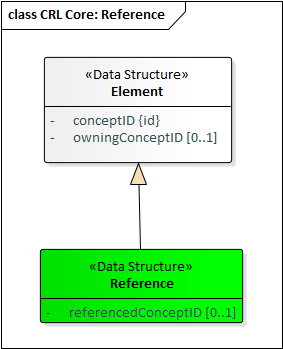


Figure 4: Reference Data Structure

In the present implementation there are refinements of the Reference concept differing in the kind of thing they reference: Element, BaseElement, Literal, Pointer, etc. It is conjectured that these refinements are not necessary, particularly because the BaseElement and the Value branch of the type hierarchy do not exist in this simplified model.

## 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 the string itself 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 4). 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.

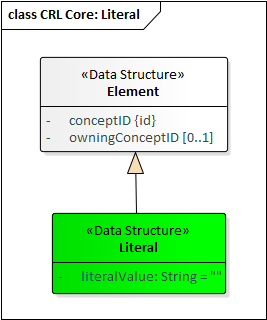


Figure 5: Literal Data Structure

## 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 representation of refinement is shown in Figure 6.

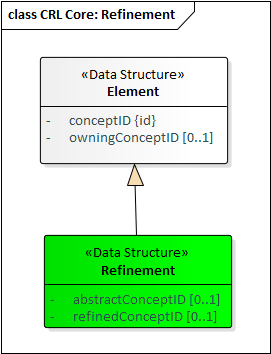


Figure 6: Refinement Data Structure

## Full CRL Core

The full CRL core model is shown in Figure 7. Note that in the current implementation Element is a refinement of a BaseElement (which holds the conceptID and owningConcept). There is another refinement of BaseElement, the Value, whose refinements are the family of Pointers and the Literal. It is conjectured that this branch of the type hierarchy is not necessary.

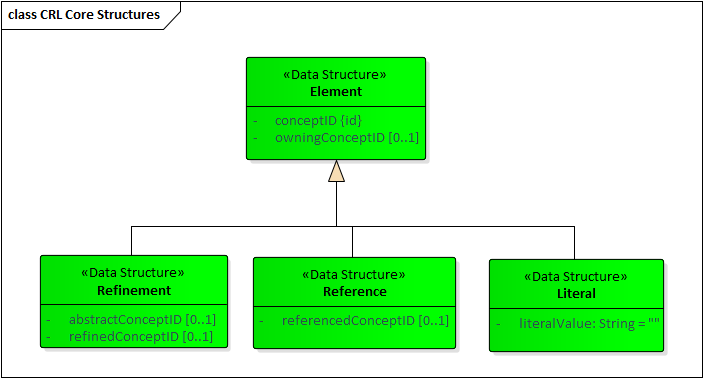


Figure 7: Full CRL Core Model

# 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 how to associate a label, URL, and definition with a concept.

## URI

The URI concept is a refinement of Literal with the literal value being the URI used to identify this concept in CRL: <https://activeCrl.com/core/URI> ().

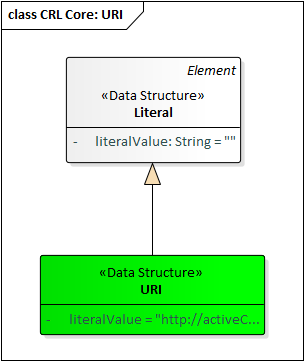


Figure 8: URL Structure

Figure 9 shows an example of how the URI concept is used. To give the concept X a URI, create a Literal that is a refinement of the URI concept and make it a child of the concept X.

For convenience, we add a derived attribute to Element named /uri (the slash indicates that it is a derived value). The meaning of this is, “Find the child Literal that is a refinement of the URL concept and return its literalValue; If none is found, return the empty string.”

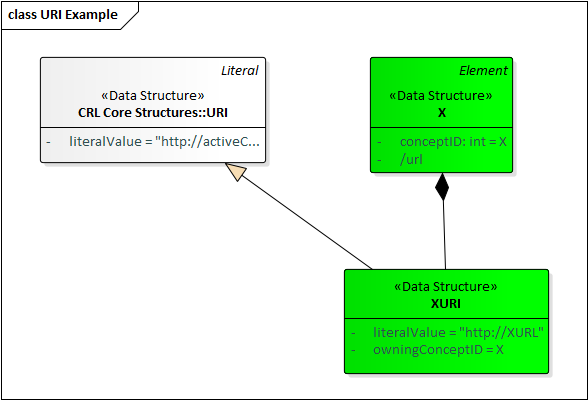


Figure 9: URL Example

All core concepts are assigned URLs as follows:

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)

## Label and Definition

Again, for convenience, the concepts of Label and Definition are created as refinements of Literal. Their URLs are:

Label: <https://activeCrl.com/core/Label>  
Definition: https://activeCrl.com/core/Definition

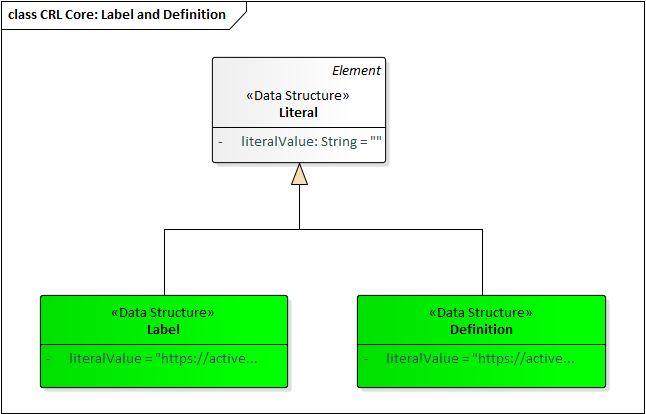


Figure 10: Label and Definition

## Element with Derived Attributes

Figure 11 shows the Element with the derived attributes. With these derived attributes a label, definition, and URI can be added to any concept representation.

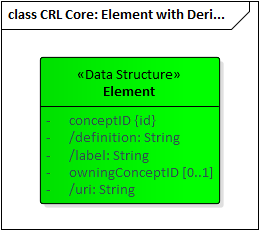


Figure 11: Element with Derived Attributes

# ActiveCRL

## 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 then the functions are queued for execution. There are two arguments to this function: the Element that changed and the list of ChangeNotifications (there may be more than one) that delineate the nature of the changes.

Figure 12 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.

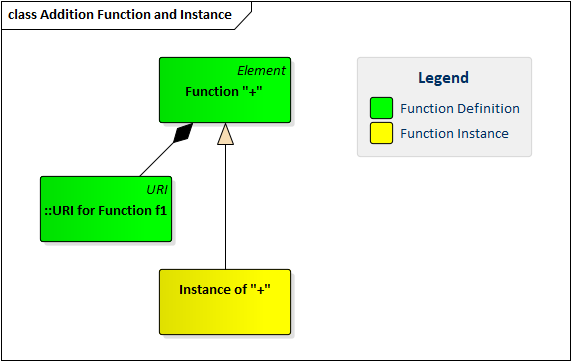


Figure 12: Function Example: Addition Function and Instance

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 13). 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.[[5]](#footnote-5) This provides a full model of the concepts expected by the function.

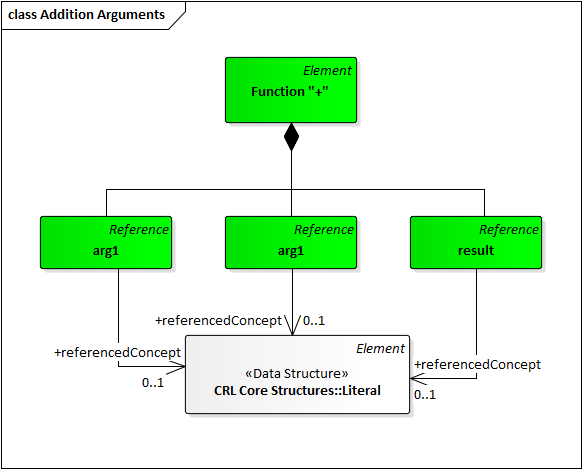


Figure 13: Model (Prototype) of Function "+"

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 14. 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.

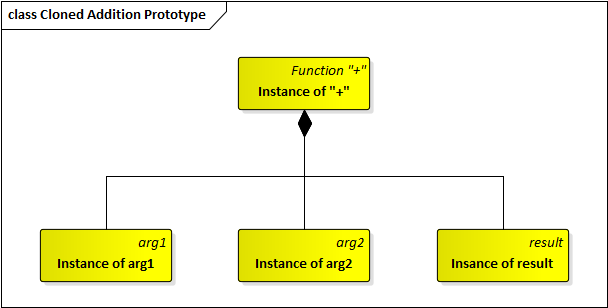


Figure 14: Cloned Addition Prototype

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. The ChangeNotification is the mechanism of awareness. There are three possible types of change notification as shown in Figure 21.

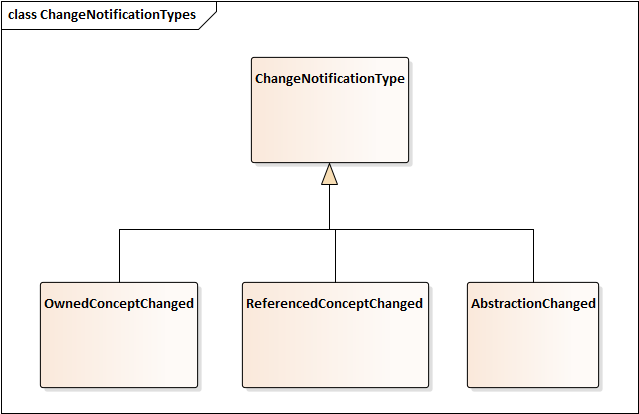


Figure 15: Change Notification Types

OwnedConceptChanged reports changes within the ownership hierarchy. These notifications pass upwards through the ownership hierarchy so that every element is aware of any change that occurs to its children and their descendants. This is the notification that updates the version number of an Element.

ReferencedElementChanged reports changes that occur in an element being referenced by either a Reference or a Refinement. These notifications report changes in the ownership hierarchy rooted at the element being referenced. These notifications pass upwards through the ownership hierarchy so that every element is aware that a change has occurred in an element referenced by some member of the hierarchy. This notification is not passed on via references.

AbstractionChanged reports changes that occur in the refinement hierarchy. Refined elements are notified of changes to their abstractions.

Changes are communicated via the ChangeNotification data structure (Figure 22). The type indicates the type of change. The currentReportingElement is the element sending this particular notification. The originalReportingElement, changedElementNew, changedElementOld, collectively, indicate the nature of the change.

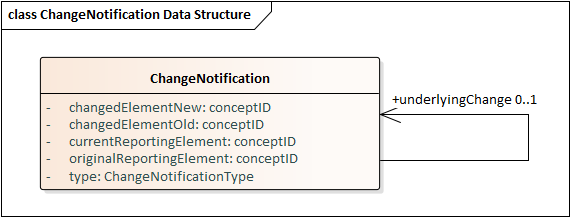


Figure 16: Change Notification Data Structure

There are essentially two fundamental changes that can occur: a change within an element (e.g. a Literal value changes), or a change in the identity of an element being referenced.

If

element that isnature of the change. If there is a change in which element is being referenced, the changedElementOld and changedElementNew provide the identifiers of the old and new elements, respectively. If an element has just changed (typically a Literal), the values of changedElementNew and ChangedElementOld

### Direct changes that can occur:

#### Element’s OwningConceptID changed:

* Propagates as an OwnedConceptAdded/ OwnedConceptRemoved to the new and old owners, respectively;
* Does not propagate to references

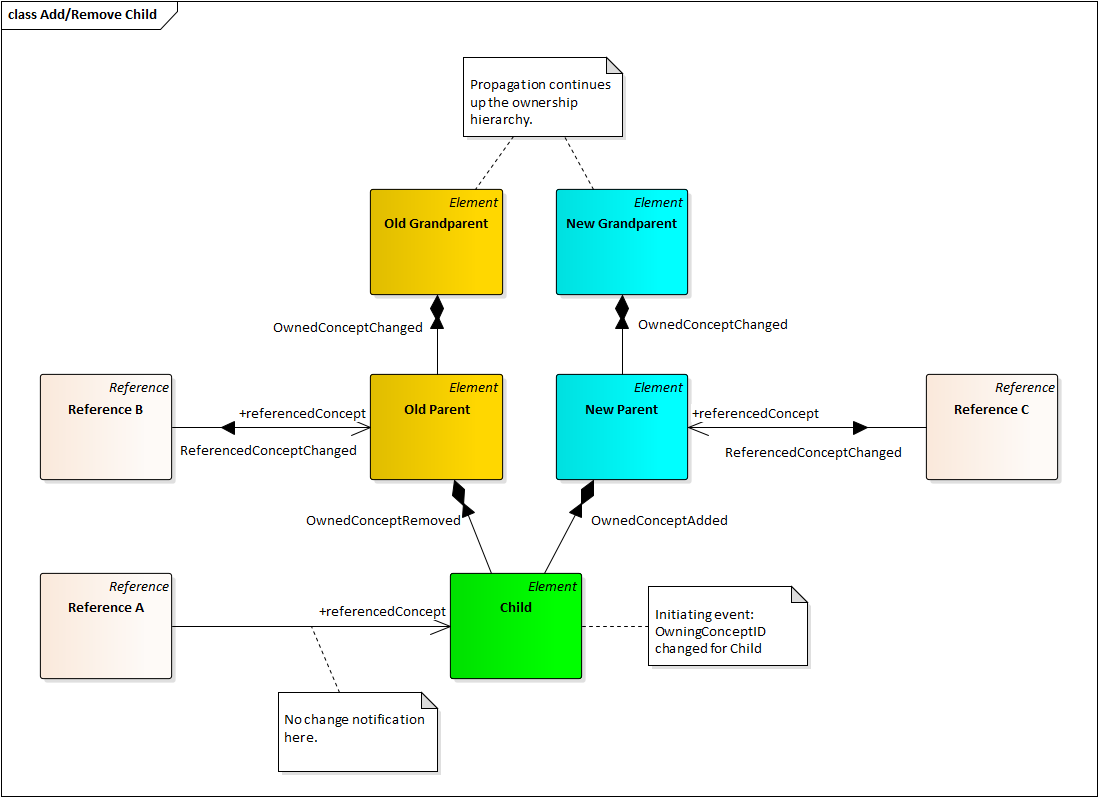


Figure 17: ChangeNotifications for Element OwningCOncpetID Changed

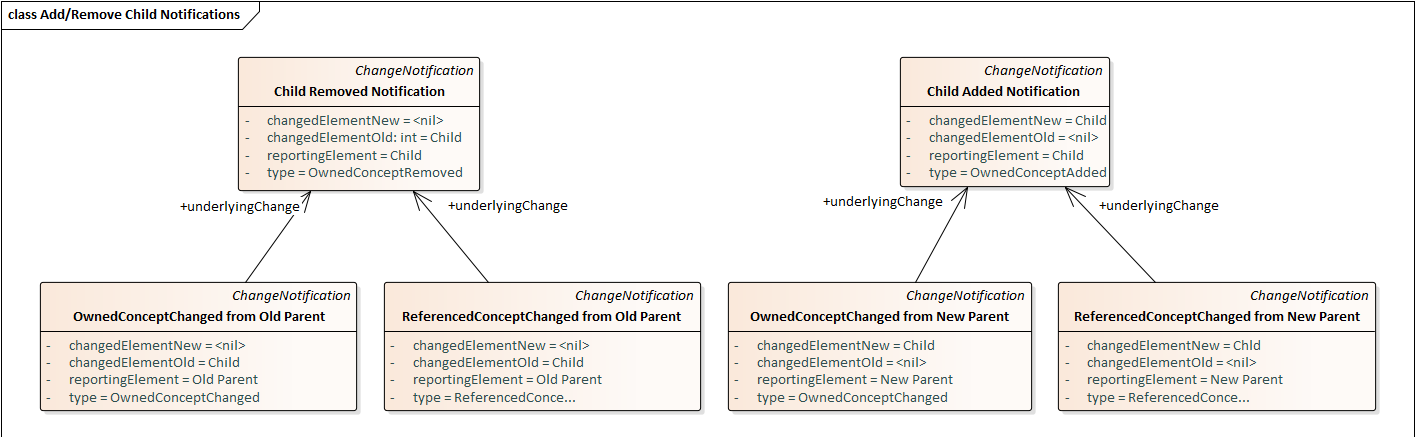


Figure 18: ChangeNotification Details forElement OwningConceptID Changed

#### Literal’s Literal Value Changed:

* Propagates as an OwnedConceptChanged to the owner;
* Propagates as a ReferencedConceptChanged to any references.

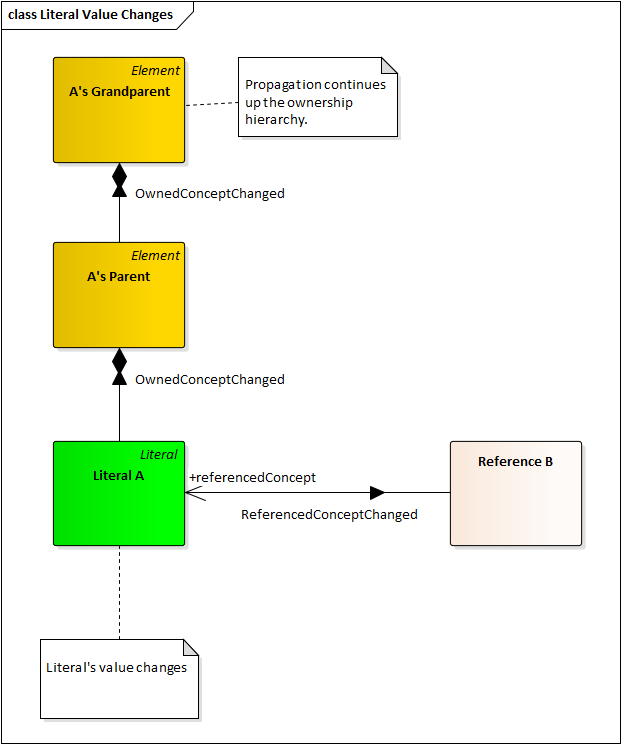


Figure 19: Literal Value Changes

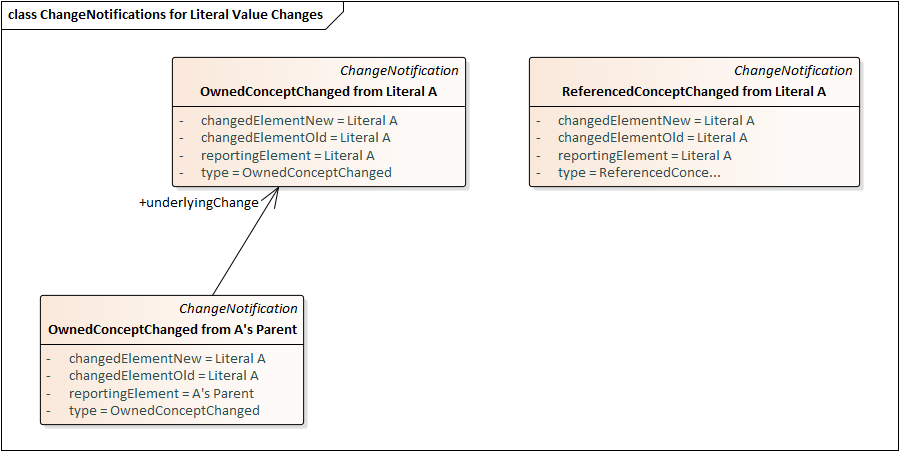


Figure 20: ChangeNotification Details for Literal Value Change

#### Reference’s ReferencedConceptID changed:

* Propagates as a ReferencedConceptChanged to the owner;
* Propagates as a ReferencedConceptChanged to any references

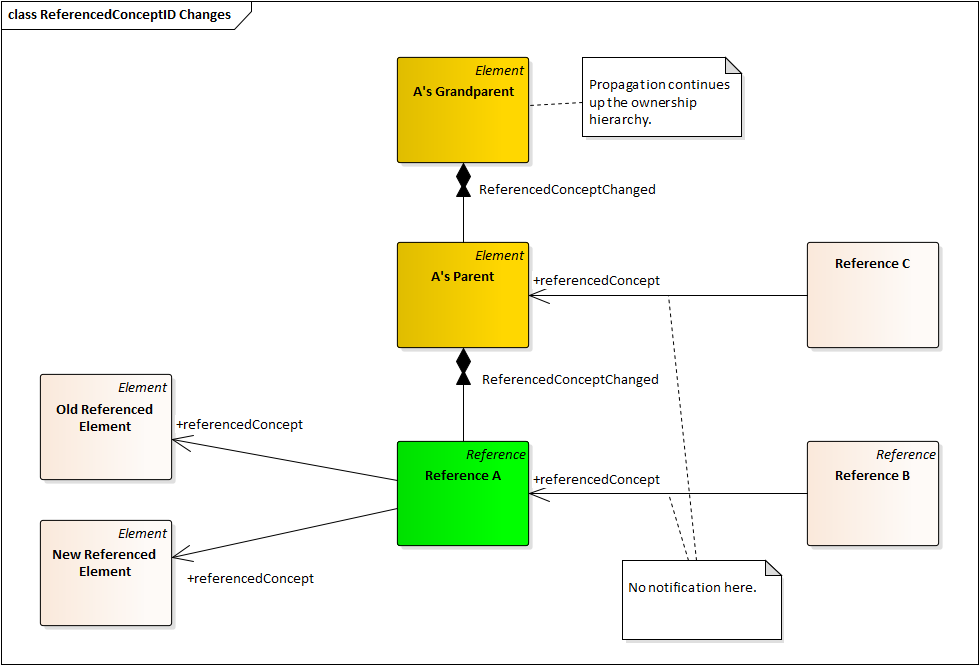


Figure 21: ReferencedConceptID Changes

#### Refinement’s AbstractConceptID changed:

* Propagates as a ReferencedConceptChanged to the owner;
* Does not propagate any references
* Propagates as an AbstractionChanged to refined concept

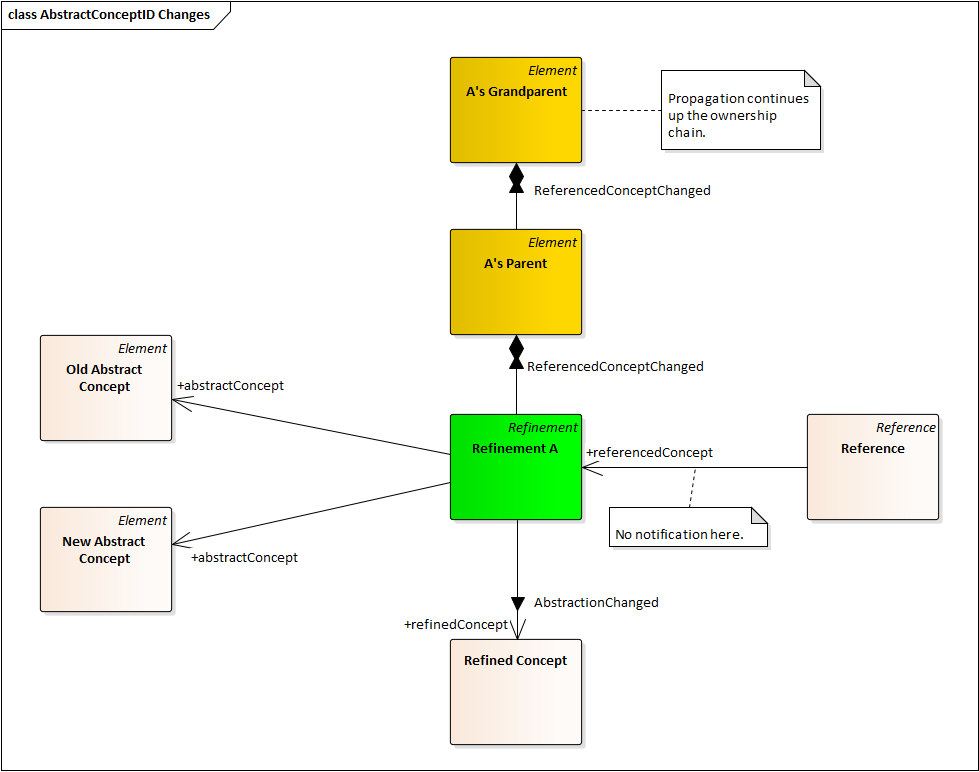


Figure 22: Refinement's AbstractConceptID changes

#### Refinement’s RefinedConceptID changed:

* Propagates as a ReferencedConceptChanged to the owner;
* Propagates as a ReferencedConceptChanged to any references

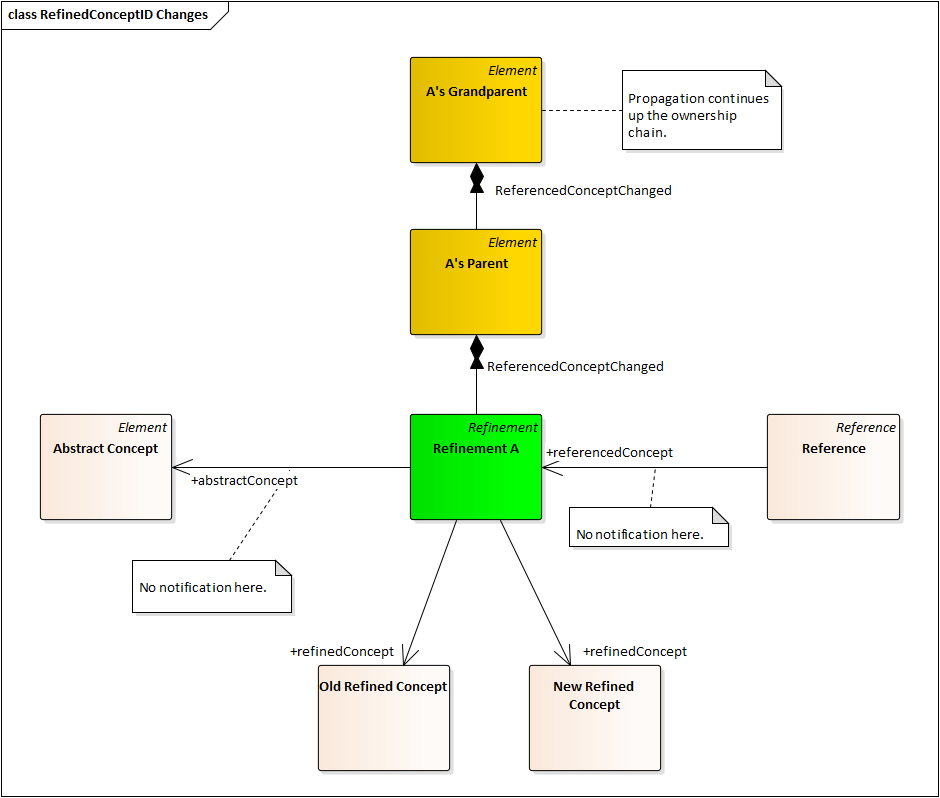


Figure 23: Refinement's RefinedConceptID Changes

### Propagation of Received Notifications

#### Reference or Refinement Receives ReferencedConceptChanged

* Propagates as ReferencedConceptChanged to owner
* Does not propagate to any references

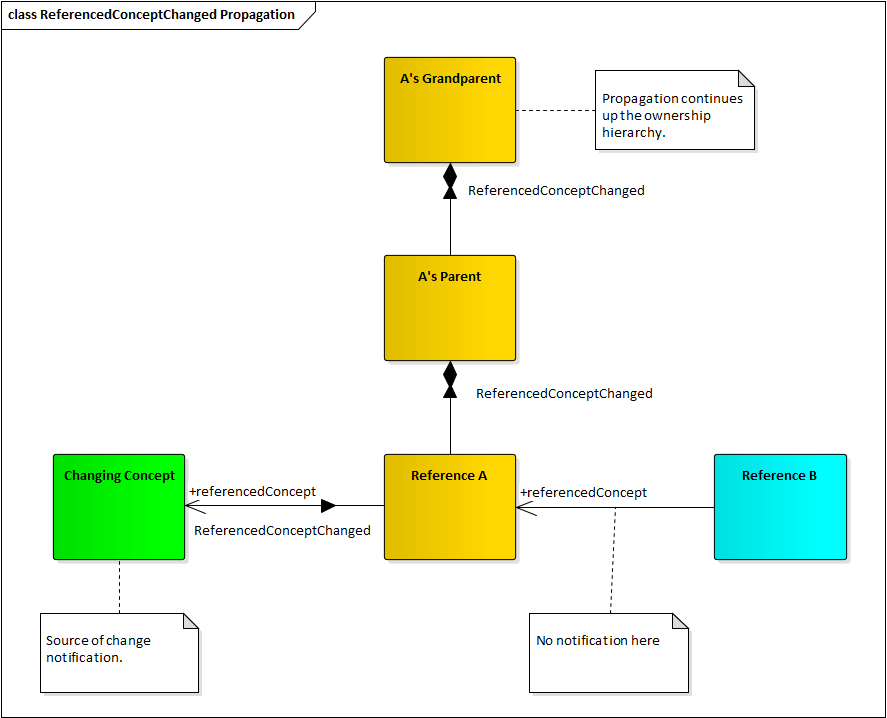


Figure 24: Reference Receives ReferencedConceptChanged

# 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

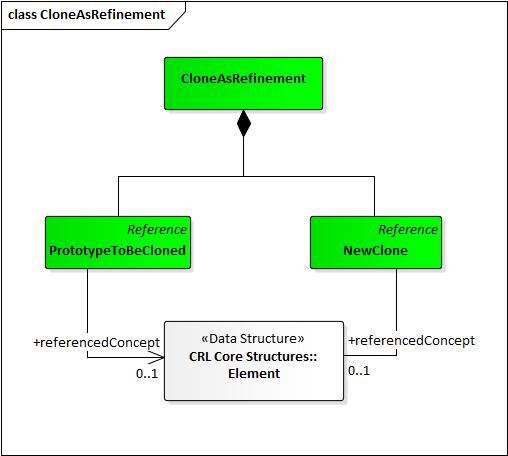


Figure 25: CloneAsRefinement

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. A richer model would refine Literal further to represent a Number and the references would then point to Number. [↑](#footnote-ref-5)