Simplifying 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, 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)

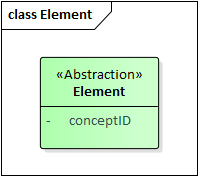


Figure ‑: Element AbstractiFigure ‑on

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.

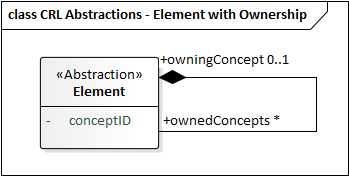


Figure ‑: Ownership Abstraction

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.

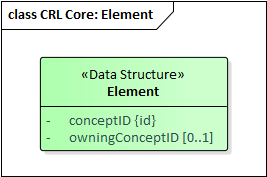


Figure ‑: Element Data Structure

## 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.

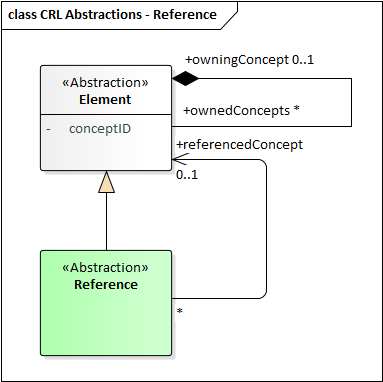


Figure ‑: Reference Abstraction

The data structure for representing a reference is shown in

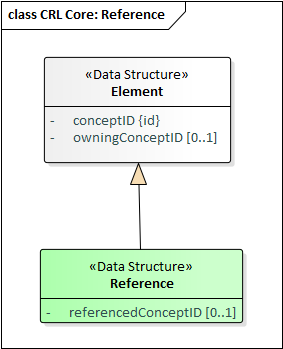


Figure ‑: Reference Data Structure

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 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.

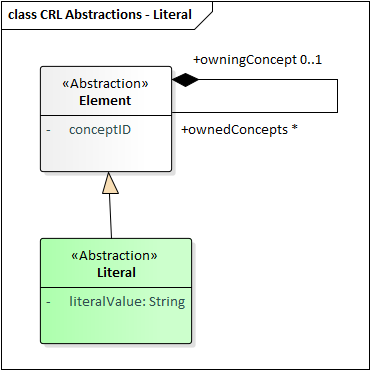


Figure ‑: Literal Abstraction

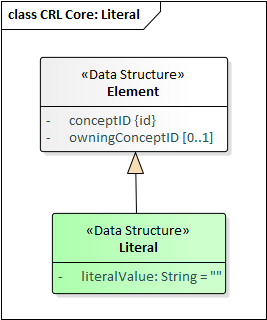


Figure ‑: Literal Data Structure

## 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 abstract representation of refinement is shown in Figure 2‑9, and the data structure refinement is shown in Figure 2‑10.

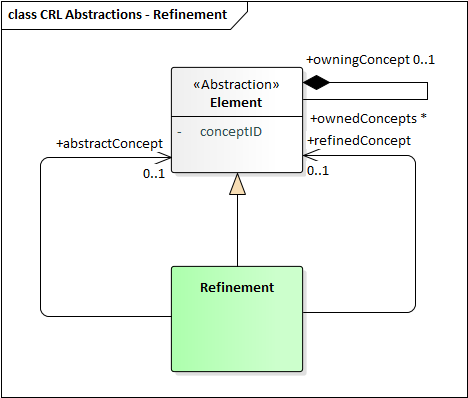


Figure ‑: Refinement Abstraction

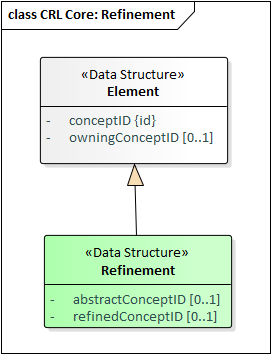


Figure ‑: Refinement Data Structure

## Full CRL Core

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

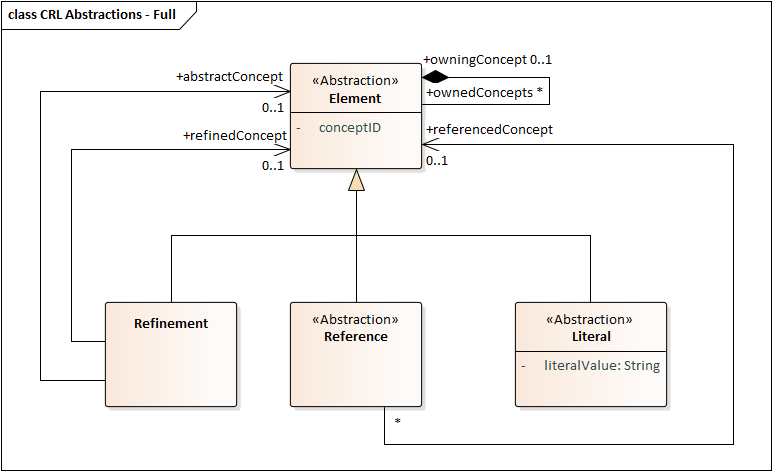


Figure ‑: Full Core Model Abstraction

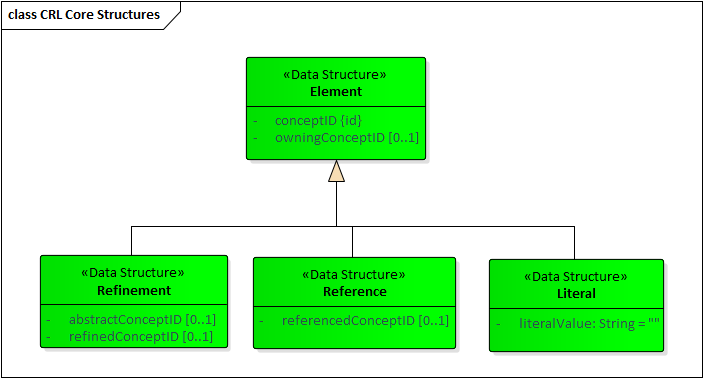


Figure ‑: Full CRL Core Model Data Structures

# 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> ().

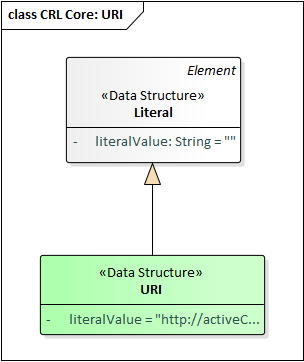


Figure ‑: URL Structure

Figure 3‑2 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.”

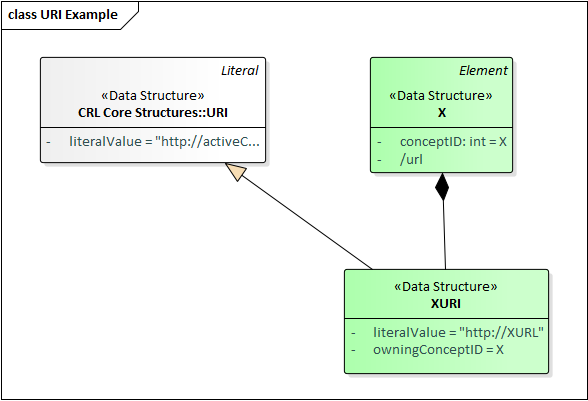


Figure ‑: URL Example

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

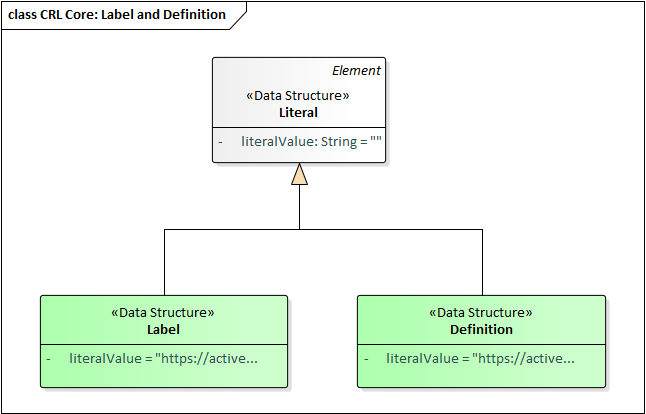


Figure ‑: Label and Definition

## Element with Derived Attributes

Figure 3‑4 shows the Element with the derived attributes. With these derived attributes a label, definition, and URI can be added to any concept representation.

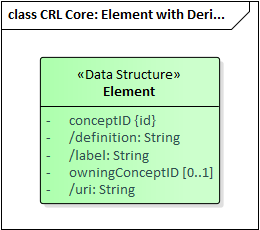


Figure ‑: Element with Derived Attributes

# ActiveCRL

## Data Structures

ActiveCRL adds the ability to dynamically modify data structures and for Elements to represent functions that respond to those changes. Several attributes are added to support this. One is the readOnly attribute. Another is the version attribute that is incremented any time a change is made to an element or its descendants (Figure 4‑1).

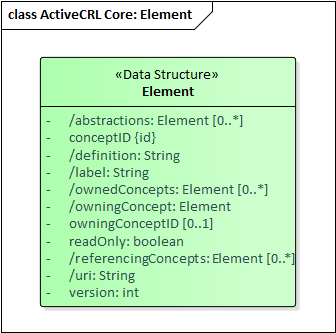


Figure ‑: ActiveCRL Element Data Structure

In addition to the version, there are a number of derived attributes that are added for computational convenience. The abstractions attribute contains a list of the Elements representing the abstractions (transitive closure) of this concept. The ownedConcepts attribute contains a list of the Elements representing the immediate children of this concept. The owningConcept attribute contains the Element that represents the owningConceptID. The referencingConcepts contains a list of the Elements that reference this concept.

The Reference data structure is also enhanced with two attributes. The referencedConcept is a cached pointer to the concept being referenced. The referencedConceptVersion records the version number of the referenced concept the last time the Reference was notified of a change to it. This allows the Reference to check the next time it receives a notification about the referenced element to see whether it has actually changed. If the version number has not changed, the notification does not propagate.

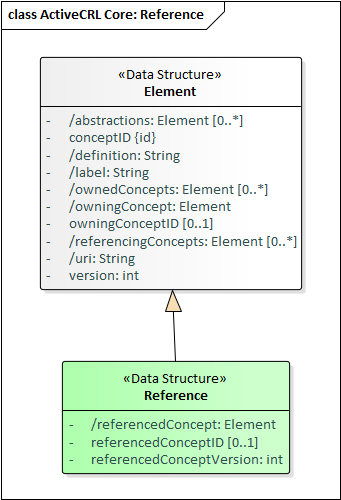


Figure ‑: ActiveCRL Reference Data Structure

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

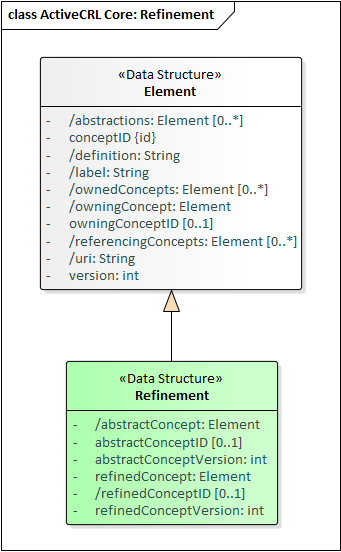


Figure ‑: ActiveCRL Refinement Data Structure

## 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 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 4‑2 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.

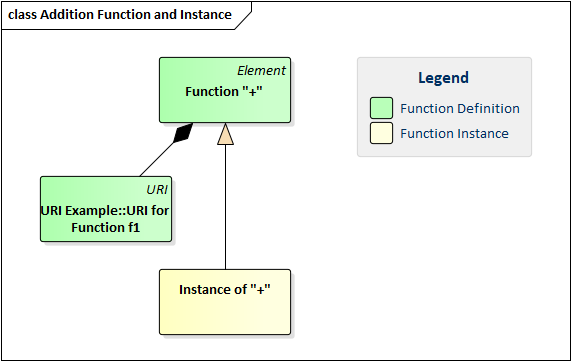


Figure ‑: Function Example: Addition Function and Instance

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 4‑3). 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.

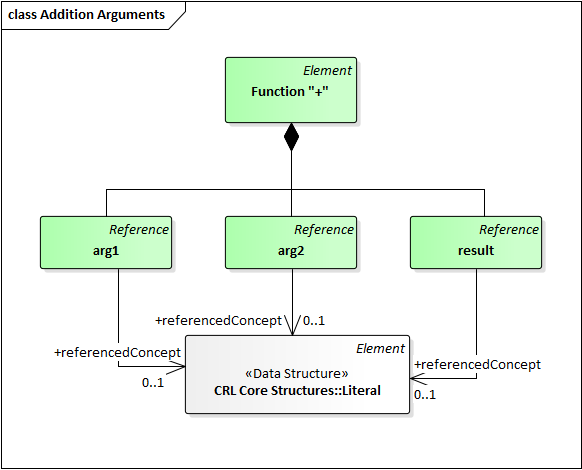


Figure ‑: Model (Prototype) of Function "+"

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 4‑4. 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.

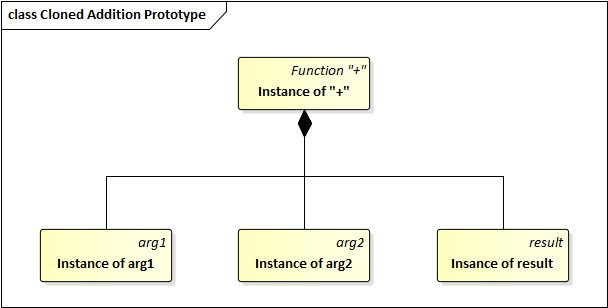


Figure ‑: Cloned Addition Prototype

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. It reports changes to impacted elements. These changes fall into two broad categories: ElementChange and ForwardingChange (Figure 4‑5).

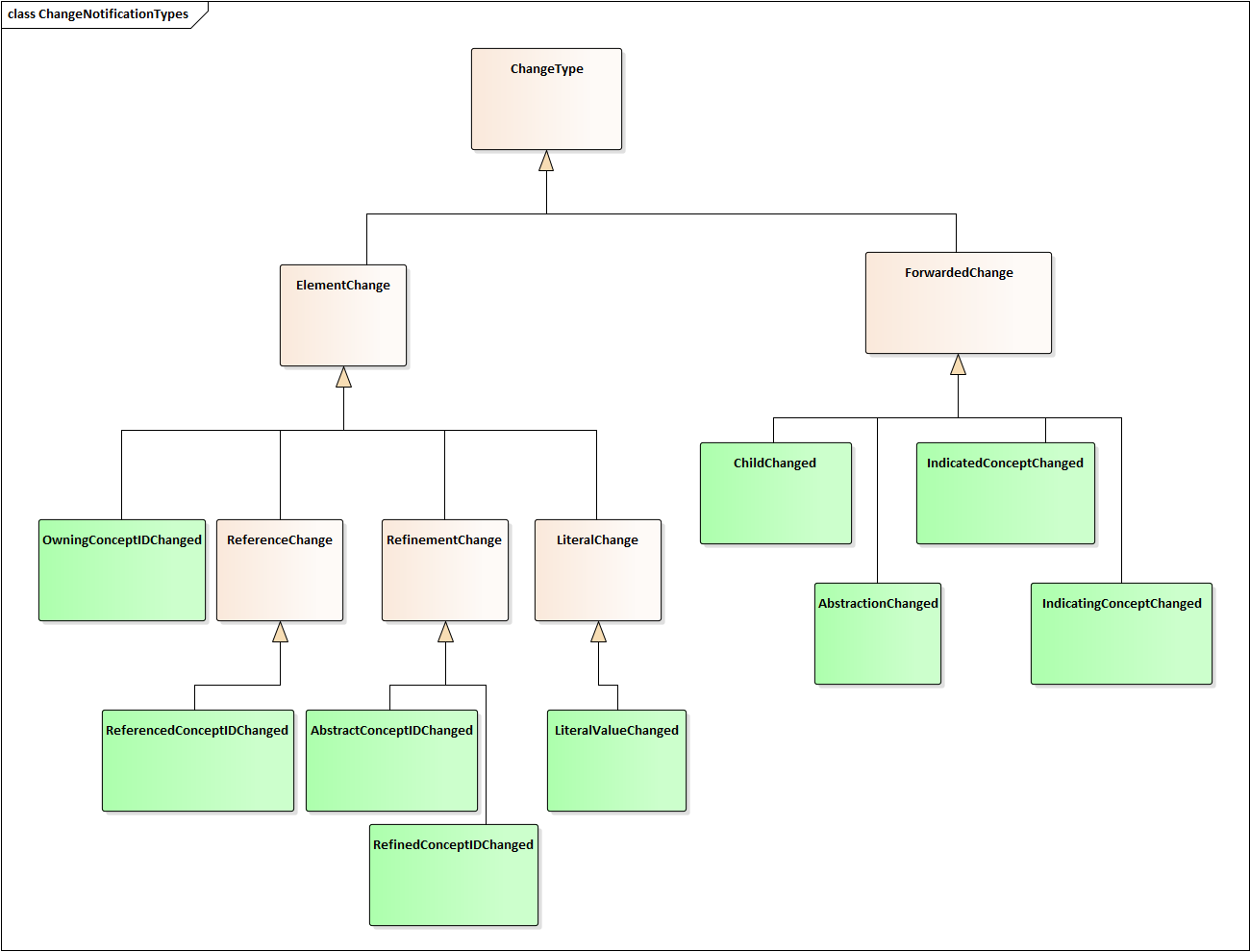


Figure ‑: Change Types

An ElementChange is a change directly to an Element. OwningConceptIDChanged is used to report that the value of the OwningConceptID has changed. Refinements of Element can report additional types of changes. A Refrerence uses ReferencedConceptIDChanged to report that the value of its ReferencedConceptID has changed. A Refinement uses AbstractConceptIDChanged and RefinedConceptIDChanged to report changes in these values. A Literal uses LiteralValueChanged to report a change in the literal value.

A ForwardingChange is used to pass on the notification of changes to related Elements. A ChildChanged is used to indicate that a change has occurred to an OwnedConcept.

An AbstractionChanged is used to indicate that a change has occurred in the abstraction hierarchy above an Element.

An IndicatedConceptChanged is used to indicate that a change has occurred to an indicated Element: either the ReferencedConcept of a Reference or the AbstractConcept or RefinedConcept of a Refinement.

An IndicatingConceptChanged is used to inform an Element that there has been a change in a pointer to the Element, either the ReferencedConceptID of a Reference or the AbstractConceptID or RefinedConceptID of a Refinement. This notification is necessary to maintain the Element’s list of ReferencingConcepts.

### ChangeNotification Data Structure

Changes are communicated via the ChangeNotification data structure (Figure 4‑6). The type indicates the type of change. The reportingElement is the element sending this particular notification. The newValue and oldValue give the specifics of the change. The last attribute is the underlyingChange pointer. If a change is being propagated to additional elements, it is not forwarded. Instead, a new ChangeNotification is created.

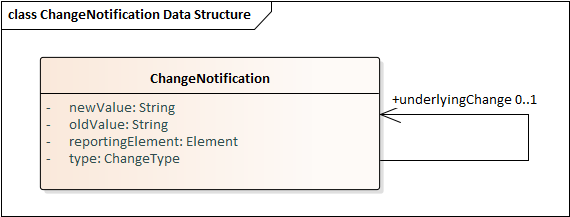


Figure ‑: Change Notification Data Structure

The following sections provide examples of the notifications and their propagation.

### Direct Changes to Elements

#### Element’s OwningConceptID changed:

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

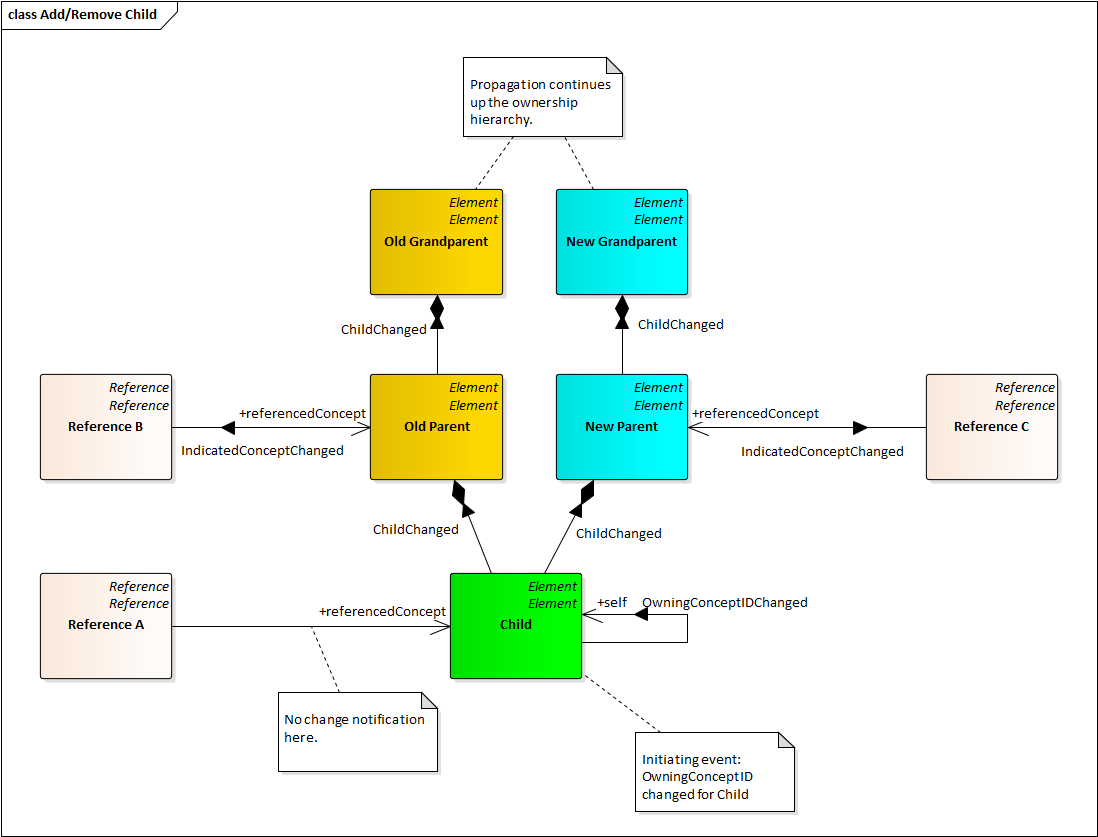


Figure ‑: ChangeNotifications for Element OwningConcpetID Changed

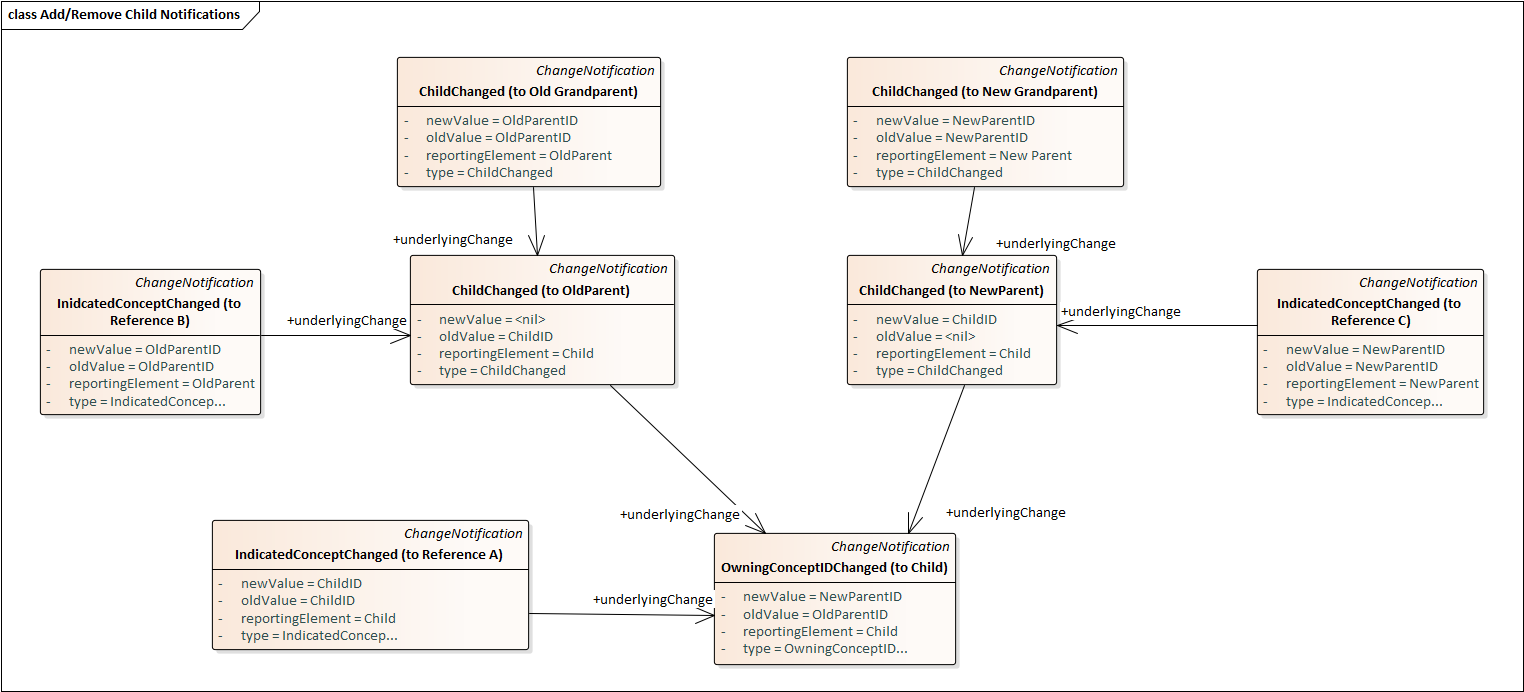


Figure ‑: ChangeNotification Details for OwningConceptID Changed

#### Literal’s Literal Value Changed:

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

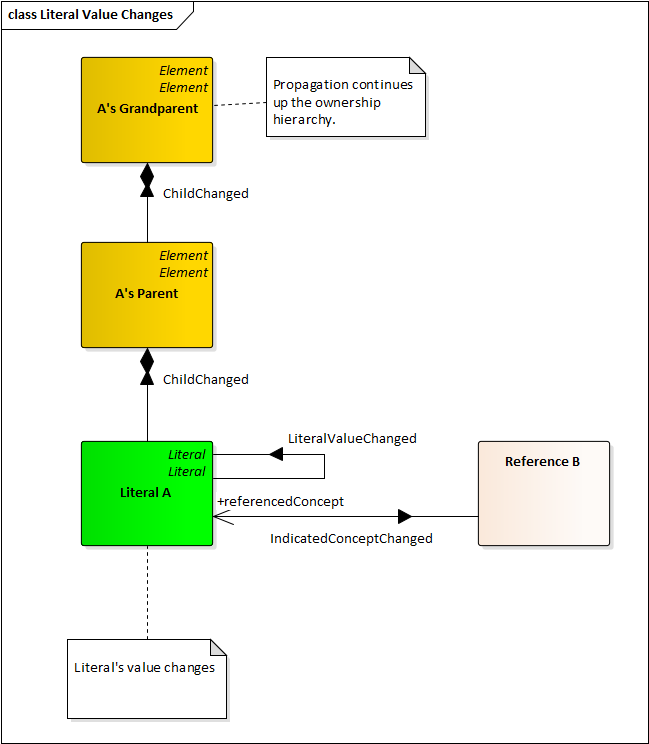


Figure ‑: Literal Value Changes

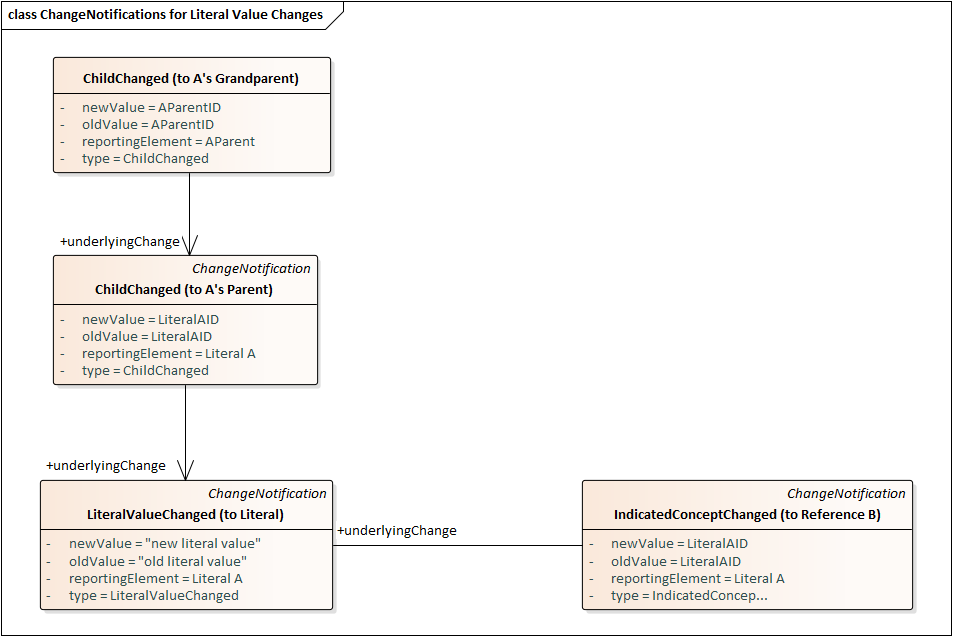


Figure ‑: ChangeNotification Details for Literal Value Change

#### Reference’s ReferencedConceptID changed:

* Propagates as an IndicatedConceptChanged to the owner;
* Does not propagate to references.

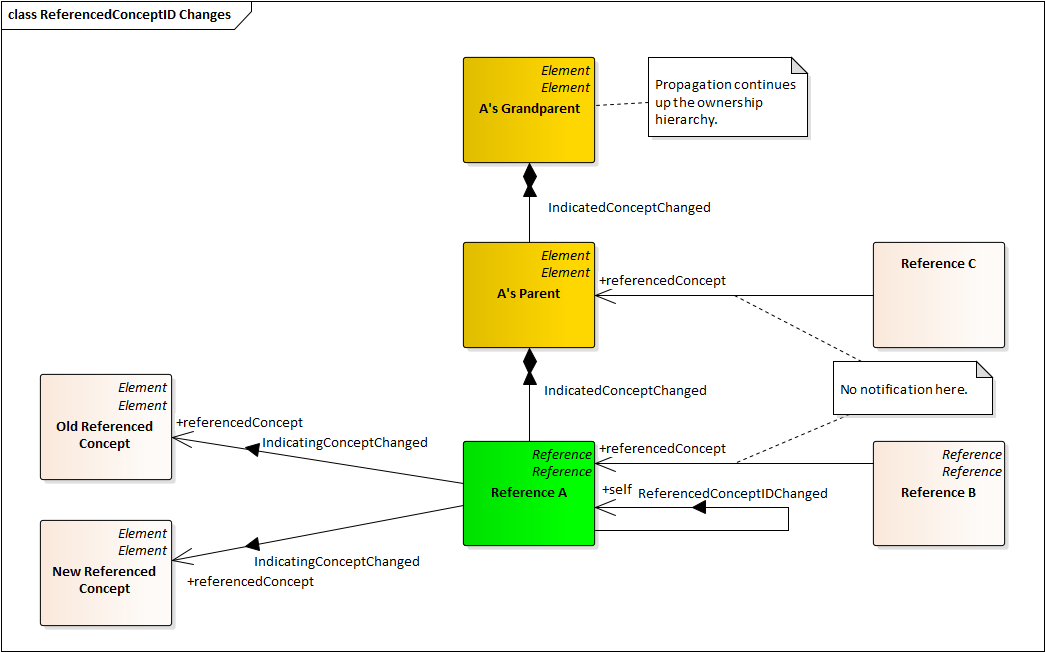


Figure ‑: ReferencedConceptIDChanged

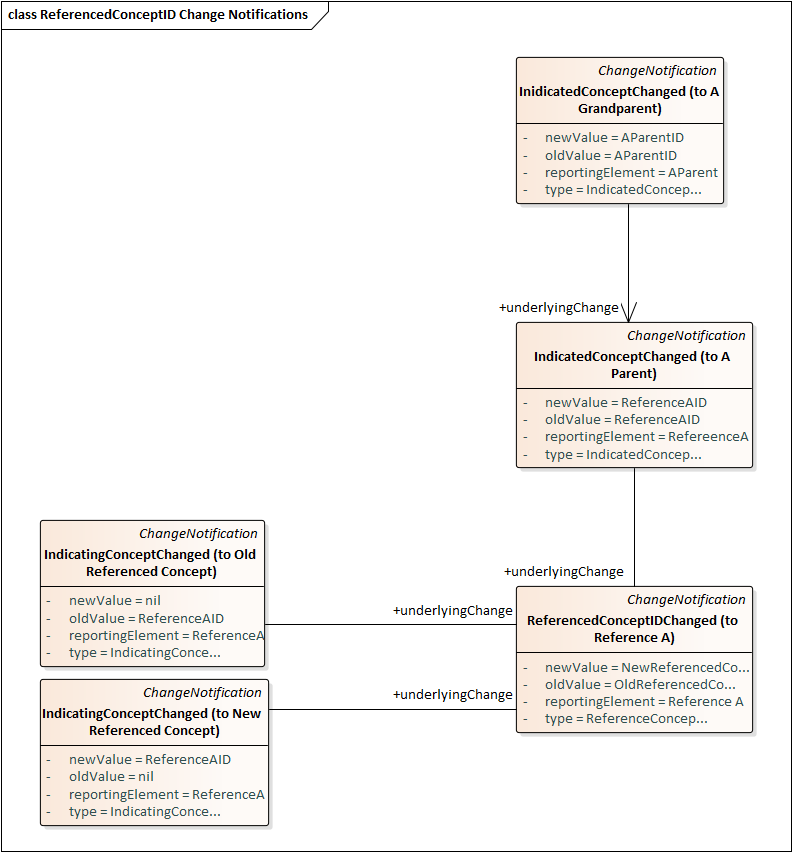


Figure ‑: ReferencedConceptIDChanged Notifications

#### Refinement’s AbstractConceptID changed:

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

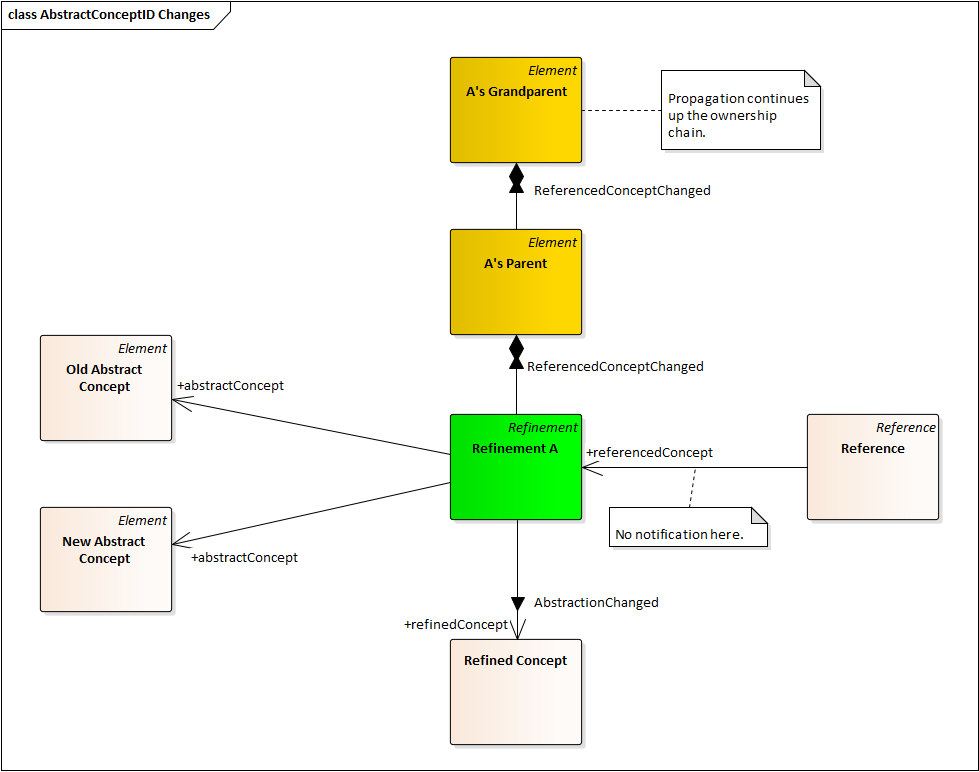


Figure ‑: Refinement's AbstractConceptID changes

#### Refinement’s RefinedConceptID changed:

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

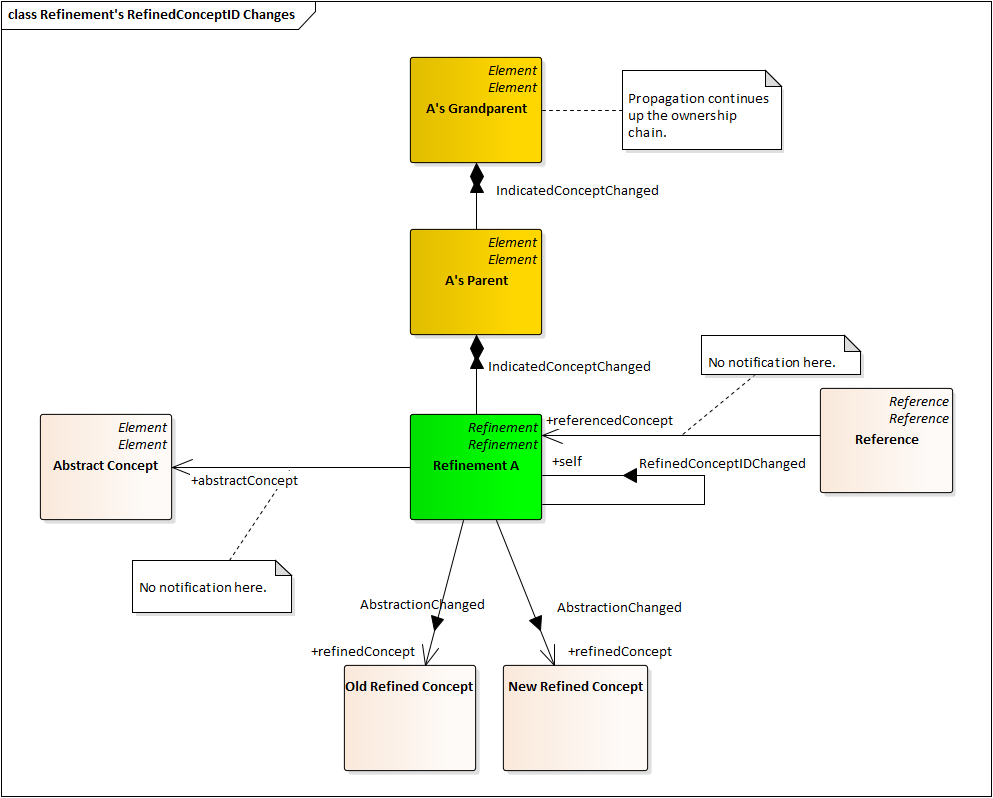


Figure ‑: Refinement's RefinedConceptID Changes

### Propagation of Received Notifications

#### Reference Receives IndicatedConceptChanged

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

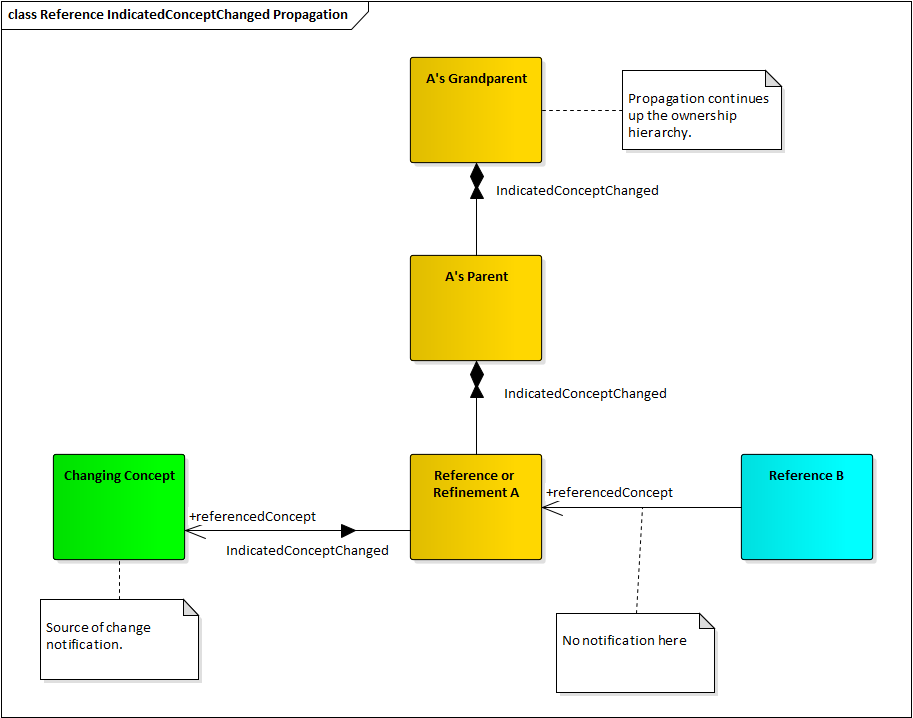
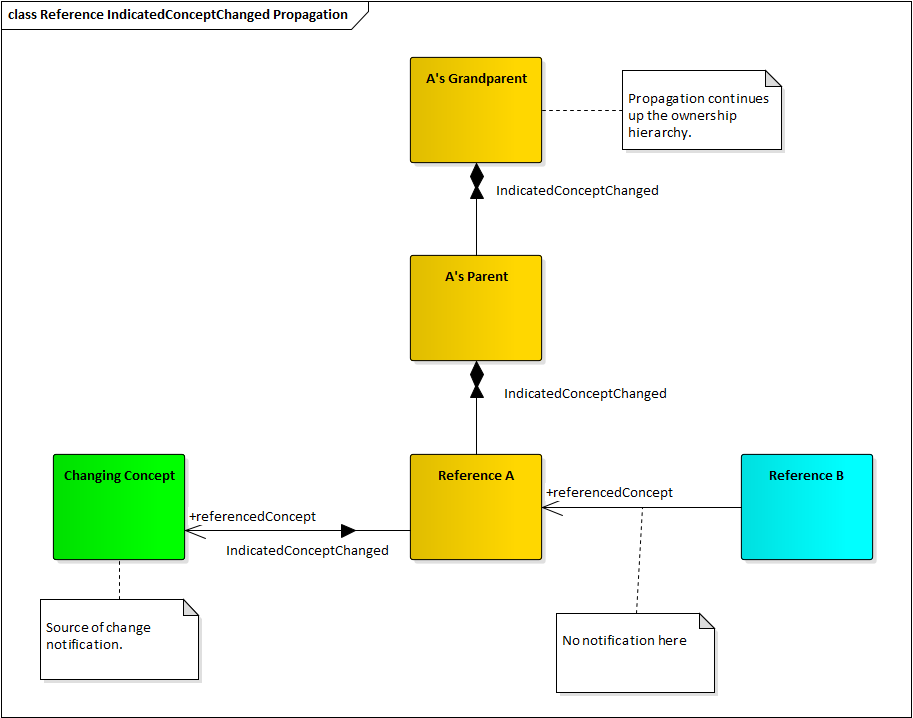
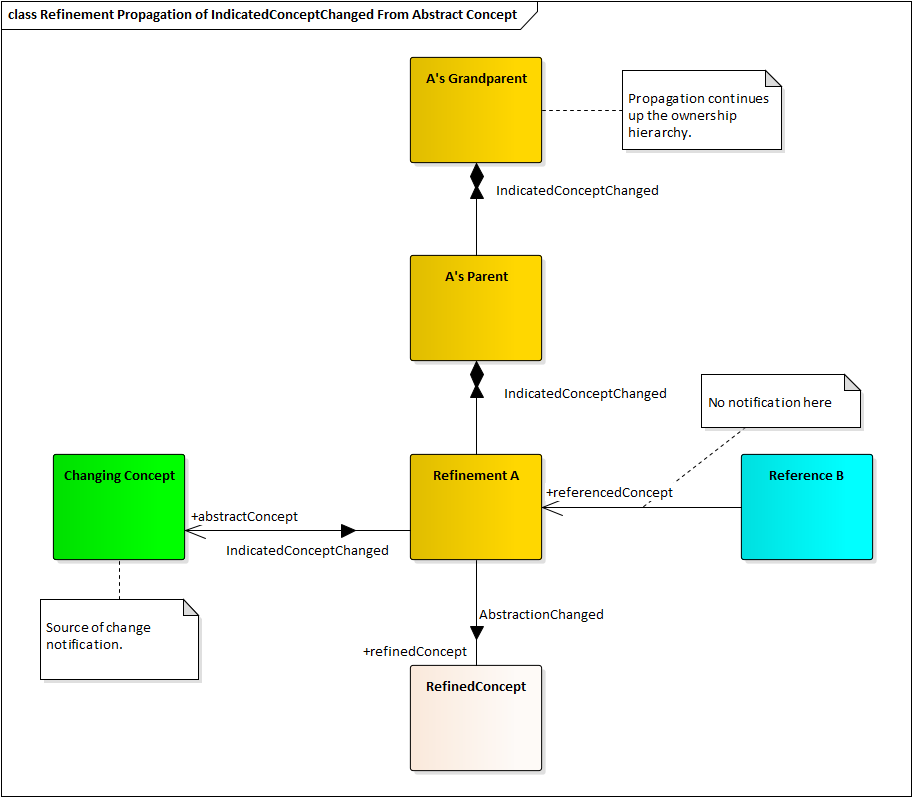


Figure ‑: Reference Receives IndicatedConceptChanged

#### Refinement Receives IndicatedConceptChanged From AbstractConcept



#### Refinement Receives IndicatedConceptChanged from RefinedConcept

This notification is ignored by the Refinement

# 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

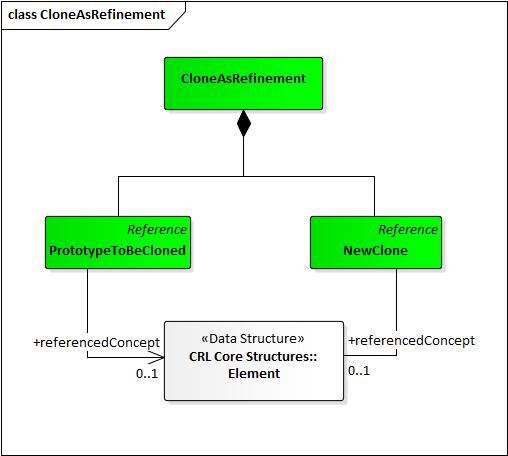


Figure ‑: CloneAsRefinement

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. 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. [↑](#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)