ActiveCRL

Overview and Architecture

Paul C. Brown  
  
June 25, 2017

Contents

[1 Introduction 9](#_Toc492908027)

[1.1 Background 9](#_Toc492908028)

[1.2 Analysis 10](#_Toc492908029)

[2 The Core CRL Representational Entities 10](#_Toc492908030)

[2.1 Reference 11](#_Toc492908031)

[2.2 Refinement 11](#_Toc492908032)

[3 Grounding the Model 11](#_Toc492908033)

[3.1 BaseElement Attributes 13](#_Toc492908034)

[3.2 ElementPointer and LiteralPointer Roles 14](#_Toc492908035)

[3.3 Versioning 14](#_Toc492908036)

[4 Core Representations 15](#_Toc492908037)

[4.1 BaseElement 15](#_Toc492908038)

[4.2 BaseElementPointer 15](#_Toc492908039)

[4.3 BaseElementReference 15](#_Toc492908040)

[4.4 Element 15](#_Toc492908041)

[4.5 ElementPointer 15](#_Toc492908042)

[4.6 ElementPointerPointer 15](#_Toc492908043)

[4.7 ElementPointerReference 15](#_Toc492908044)

[4.8 ElementPointerRole 15](#_Toc492908045)

[4.9 ElementReference 15](#_Toc492908046)

[4.10 Literal 15](#_Toc492908047)

[4.11 LiteralPointer 15](#_Toc492908048)

[4.12 LiteralPointerPointer 15](#_Toc492908049)

[4.13 LiteralPointerReference 15](#_Toc492908050)

[4.14 LIteralPointerRole 15](#_Toc492908051)

[4.15 LiteralReference 15](#_Toc492908052)

[4.16 Refinement 15](#_Toc492908053)

[5 Functions 15](#_Toc492908054)

[5.1 Making It Executable 15](#_Toc492908055)

[5.2 BaseElement Functions 16](#_Toc492908056)

[5.2.1 Delete 16](#_Toc492908057)

[5.2.2 GetId 17](#_Toc492908058)

[5.2.3 GetName 17](#_Toc492908059)

[5.2.4 GetOwningElement 18](#_Toc492908060)

[5.2.5 GetUri 18](#_Toc492908061)

[5.2.6 GetVersion 19](#_Toc492908062)

[5.2.7 SetOwningElement 19](#_Toc492908063)

[5.2.8 SetUri 20](#_Toc492908064)

[5.3 BaseElementPointer Functions 20](#_Toc492908065)

[5.3.1 CreateBaseElementPointer 21](#_Toc492908066)

[5.3.2 GetBaseElement 21](#_Toc492908067)

[5.3.3 GetBaseElementId 21](#_Toc492908068)

[5.3.4 GetBaseElementVersion 22](#_Toc492908069)

[5.3.5 SetBaseElement 22](#_Toc492908070)

[5.4 BaseElementReference Functions 23](#_Toc492908071)

[5.4.1 CreateBaseElementReference 23](#_Toc492908072)

[5.4.2 GetBaseElementPointer 24](#_Toc492908073)

[5.4.3 GetReferencedBaseElement 24](#_Toc492908074)

[5.4.4 SetReferencedBaseElement 25](#_Toc492908075)

[5.5 Element Functions 26](#_Toc492908076)

[5.5.1 CreateElement 26](#_Toc492908077)

[5.5.2 GetDefinition 26](#_Toc492908078)

[5.5.3 GetDefinitionLiteral 27](#_Toc492908079)

[5.5.4 GetDefinitionLiteralPointer 27](#_Toc492908080)

[5.5.5 GetNameLiteral 28](#_Toc492908081)

[5.5.6 GetNameLiteralPointer 28](#_Toc492908082)

[5.5.7 GetUriLiteral 29](#_Toc492908083)

[5.5.8 GetUriLiteralPointer 29](#_Toc492908084)

[5.5.9 SetDefinition 30](#_Toc492908085)

[5.5.10 SetName 30](#_Toc492908086)

[5.6 ElementPointer Functions 31](#_Toc492908087)

[5.6.1 CreateAbstractElementPointer 31](#_Toc492908088)

[5.6.2 CreateOwningElementPointer 32](#_Toc492908089)

[5.6.3 CreateReferencedElementPointer 32](#_Toc492908090)

[5.6.4 CreateRefinedElementPointer 33](#_Toc492908091)

[5.6.5 GetElement 33](#_Toc492908092)

[5.6.6 GetElementId 34](#_Toc492908093)

[5.6.7 GetElementPointerRole 34](#_Toc492908094)

[5.6.8 GetElementVersion 35](#_Toc492908095)

[5.6.9 SetElement 35](#_Toc492908096)

[5.7 ElementPointerPointer Functions 36](#_Toc492908097)

[5.7.1 CreateElementPointerPointer 36](#_Toc492908098)

[5.7.2 GetElementPointer 37](#_Toc492908099)

[5.7.3 GetElementPointerId 37](#_Toc492908100)

[5.7.4 GetElementPointerVersion 38](#_Toc492908101)

[5.7.5 SetElementPointer 38](#_Toc492908102)

[5.8 ElementPointerReference Functions 39](#_Toc492908103)

[5.8.1 CreateElementPointerReference 39](#_Toc492908104)

[5.8.2 GetElementPointerPointer 39](#_Toc492908105)

[5.8.3 GetReferencedElementPointer 40](#_Toc492908106)

[5.8.4 SetReferencedElementPointer 40](#_Toc492908107)

[5.9 ElementReference Functions 41](#_Toc492908108)

[5.9.1 CreateElementReference 41](#_Toc492908109)

[5.9.2 GetElementPointer 42](#_Toc492908110)

[5.9.3 GetReferencedElement 42](#_Toc492908111)

[5.9.4 SetReferencedElement 42](#_Toc492908112)

[5.10 Literal Functions 43](#_Toc492908113)

[5.10.1 CreateLiteral 43](#_Toc492908114)

[5.10.2 GetLiteralValue 44](#_Toc492908115)

[5.10.3 SetLiteralValue 44](#_Toc492908116)

[5.11 LiteralPointer Functions 44](#_Toc492908117)

[5.11.1 CreateNameLiteralPointer 44](#_Toc492908118)

[5.11.2 CreateDefinitionLiteralPointer 45](#_Toc492908119)

[5.11.3 CreateUriLiteralPointer 45](#_Toc492908120)

[5.11.4 CreateValueLiteralPointer 45](#_Toc492908121)

[5.11.5 GetLiteral 46](#_Toc492908122)

[5.11.6 GetLiteralId 46](#_Toc492908123)

[5.11.7 GetLiteralPointerRole 47](#_Toc492908124)

[5.11.8 GetLiteralVersion 47](#_Toc492908125)

[5.11.9 SetLiteral 48](#_Toc492908126)

[5.12 LiteralPointerPointer Functions 49](#_Toc492908127)

[5.12.1 CreateLiteralPointerPointer 49](#_Toc492908128)

[5.12.2 GetLiteralPointer 49](#_Toc492908129)

[5.12.3 GetLiteralPointerId 49](#_Toc492908130)

[5.12.4 GetLiteralPointerVersion 49](#_Toc492908131)

[5.12.5 SetLiteralPointer 49](#_Toc492908132)

[5.13 LiteralPointerReference 49](#_Toc492908133)

[5.13.1 CreateLiteralPointerReference 49](#_Toc492908134)

[5.13.2 GetReferencedLiteralPointer 49](#_Toc492908135)

[5.13.3 GetLiteralPointerPointer 49](#_Toc492908136)

[5.13.4 SetReferencedLiteralPointer 49](#_Toc492908137)

[5.14 LiteralReference 49](#_Toc492908138)

[5.14.1 CreateLiteralReference 49](#_Toc492908139)

[5.14.2 GetReferencedLiteral 49](#_Toc492908140)

[5.14.3 GetLiteralPointer 49](#_Toc492908141)

[5.14.4 SetReferencedLiteral 49](#_Toc492908142)

[5.15 Refinement Functions 49](#_Toc492908143)

[5.15.1 CreateRefinement 49](#_Toc492908144)

[5.15.2 GetAbstractElement 49](#_Toc492908145)

[5.15.3 GetAbstractElementPointer 49](#_Toc492908146)

[5.15.4 GetRefinedElement 49](#_Toc492908147)

[5.15.5 GetRefinedElementPointer 49](#_Toc492908148)

[5.15.6 SetAbstractElement 49](#_Toc492908149)

[5.15.7 SetRefinedElement 49](#_Toc492908150)

[6 Go Implementation 49](#_Toc492908151)

[6.1 Core Interfaces 50](#_Toc492908152)

[6.2 Core Implementation 51](#_Toc492908153)

[1 Introduction 9](#_Toc492908154)

[Figure 1‑1: SIMF Concept 9](#_Toc492908155)

[2 The Core CRL Representational Entities 10](#_Toc492908156)

[Figure 2‑1: Core CRL Representational Entities 10](#_Toc492908157)

[Figure 2‑2: Representing an Association 11](#_Toc492908158)

[3 Grounding the Model 11](#_Toc492908159)

[Figure 3‑1: Partially Grounded Model 12](#_Toc492908160)

[Figure 3‑2: Fully Grounded Model 13](#_Toc492908161)

[4 Core Representations 15](#_Toc492908162)

[5 Functions 15](#_Toc492908163)

[Figure 5‑1: Execution Concepts 16](#_Toc492908164)

[Figure 5‑2: BaseElement Functions 16](#_Toc492908165)

[Figure 5‑3: Delete Operation Structure 17](#_Toc492908166)

[Figure 5‑4: GetId Function Structure 17](#_Toc492908167)

[Figure 5‑5: GetName Function Structure 18](#_Toc492908168)

[Figure 5‑6: GetOwningElement Function Structure 18](#_Toc492908169)

[Figure 5‑7: GetUri Function Structure 19](#_Toc492908170)

[Figure 5‑8: GetVersion Function Structure 19](#_Toc492908171)

[Figure 5‑9: SetOwningElement Function Structure 20](#_Toc492908172)

[Figure 5‑10: SetUri Function Structure 20](#_Toc492908173)

[Figure 5‑11: BaseElementPointer Functions 20](#_Toc492908174)

[Figure 5‑12: CreateBaseElementPointer Function Structure 21](#_Toc492908175)

[Figure 5‑13: GetBaseElement Function Structure 21](#_Toc492908176)

[Figure 5‑14:GetBaseElementId Function Structure 22](#_Toc492908177)

[Figure 5‑15: GetBaseElementVersion Function Structure 22](#_Toc492908178)

[Figure 5‑16: SetBaseElement Function Structure 23](#_Toc492908179)

[Figure 5‑17: BaseElementReference Functions 23](#_Toc492908180)

[Figure 5‑18: CreateBaseElementReference Function Structure 24](#_Toc492908181)

[Figure 5‑19: GetBaseElementPointer Function Structure 24](#_Toc492908182)

[Figure 5‑20: GetReferencedBaseElement Function Structure 25](#_Toc492908183)

[Figure 5‑21: SetReferencedBaseElement Function Structure 25](#_Toc492908184)

[Figure 5‑22: Element Functions 26](#_Toc492908185)

[Figure 5‑23: CreateElement Function Structure 26](#_Toc492908186)

[Figure 5‑24: GetDefinition Function Structure 27](#_Toc492908187)

[Figure 5‑25: GetDefinitionLiteral Function Structure 27](#_Toc492908188)

[Figure 5‑26: GetDefinitionLiteralPointer Function Structure 28](#_Toc492908189)

[Figure 5‑27: GetNameLiteral Function Structure 28](#_Toc492908190)

[Figure 5‑28: GetNameLiteralPointer Function Structure 29](#_Toc492908191)

[Figure 5‑29: GetUriLiteral Function Structure 29](#_Toc492908192)

[Figure 5‑30: SetDefinition Function Structure 30](#_Toc492908193)

[Figure 5‑31: SetName Function Structure 31](#_Toc492908194)

[Figure 5‑32: ElementPointer Functions 31](#_Toc492908195)

[Figure 5‑33: CreateAbstractElementPointer Function Structure 32](#_Toc492908196)

[Figure 5‑34: CreateOwningElementPointer Function Structure 32](#_Toc492908197)

[Figure 5‑35: CreateReferencedElementPointer Function Structure 33](#_Toc492908198)

[Figure 5‑36: CreateRefinedElementPointer Function Structure 33](#_Toc492908199)

[Figure 5‑37: GetElement Function Structure 34](#_Toc492908200)

[Figure 5‑38: GetElementId Function Structure 34](#_Toc492908201)

[Figure 5‑39: GetElementPointerRole Function Structure 35](#_Toc492908202)

[Figure 5‑40: GetElementVersion Function Structure 35](#_Toc492908203)

[Figure 5‑41: SetElement Function Structure 36](#_Toc492908204)

[Figure 5‑42: ElementPointerPointer Functions 36](#_Toc492908205)

[Figure 5‑43: CreateElementPointerPointer Function Structure 36](#_Toc492908206)

[Figure 5‑44:GetElementPointer Function Structure 37](#_Toc492908207)

[Figure 5‑45: GetElementPointerId Function Structure 37](#_Toc492908208)

[Figure 5‑46: GetElementPointerVersion Function Structure 38](#_Toc492908209)

[Figure 5‑47: ElementPointerReference Functions 39](#_Toc492908210)

[Figure 5‑48: CreateElementPointerReference Function Structure 39](#_Toc492908211)

[Figure 5‑49: GetElementPointerPointer Function Structure 40](#_Toc492908212)

[Figure 5‑50: GetElementPointer Function Structure 40](#_Toc492908213)

[Figure 5‑51: SetReferencedElementPointer Function Structure 41](#_Toc492908214)

[Figure 5‑52: ElementReference Functions 41](#_Toc492908215)

[Figure 5‑53: CreateElementReference Function Structure 41](#_Toc492908216)

[Figure 5‑54: GetElementPointer Function Structure 42](#_Toc492908217)

[Figure 5‑55: SetReferencedElement Function Structure 43](#_Toc492908218)

[Figure 5‑56: Literal Functions 43](#_Toc492908219)

[Figure 5‑57: CreateLiteral Function Structure 43](#_Toc492908220)

[Figure 5‑58: GetLiteralValue Operatsion Structure 44](#_Toc492908221)

[Figure 5‑59: SetLiteralValue Function Structure 44](#_Toc492908222)

[Figure 5‑60: CreateNameLiteralPointer Function Structure 45](#_Toc492908223)

[Figure 5‑61: CreateDefinitionLiteralPointer Function Structure 45](#_Toc492908224)

[Figure 5‑62: CreateUriLiteralPointer Function Structure 45](#_Toc492908225)

[Figure 5‑63: CreateValueLiteralPointer Function Structure 46](#_Toc492908226)

[Figure 5‑64: GetLiteral Function Structure 46](#_Toc492908227)

[Figure 5‑65: GetLiteralId Function Structure 47](#_Toc492908228)

[Figure 5‑66: GetLiteralPointerRole Function Structure 47](#_Toc492908229)

[Figure 5‑67: GetLiteralVersion Function Structure 48](#_Toc492908230)

[Figure 5‑68: SetLiteral Function Structure 48](#_Toc492908231)

[6 Go Implementation 49](#_Toc492908232)

[Figure 6‑1: CRL Core Interfaces 50](#_Toc492908233)

[Figure 6‑2: Core Implementation 51](#_Toc492908234)

# Introduction

This document describes the activeCRL, a minimalist executable concept representation language. ActiveCRL is very much a work in progress and this document is intended not only to inform as to the intent and realization of activeCRL but also to solicit feedback and participation in the furthering of this development.

## Background

The evolution of computer science has brought with it a number of conceptual distinctions: programs vs. data, types vs. instances, entities vs. relationships, and human languages vs. computer languages, to name but a few.

As useful as these distinctions are, there are situations in which they seem to get in the way rather than help. While I have contemplated such situations for many years, it wasn’t until I began assembling a proposal in response to the OMG Semantic Information Modeling for Federation (SIMF) RFP that I felt I had to take a run at solving the problem.

The idea of SIMF was that a semantic model could be used as a bridge between different syntactic representations (Figure 1‑1). Through a series of mappings, one could transform a physical representation into a logical representation and then into a semantic representation, from which point the semantic representation could be transformed into a different logical representation and finally into a different physical representation.



Figure 1‑1: SIMF Concept

The problem, as I saw it, was that each of these types of representations – physical, logical, and semantic – typically uses a different kind of model – a different language – to describe it. Physical representations could be described by XML schema, JSON schema, SQL Schema, BNF schema, or just plain text documents. Logical representations might be done in Entity-Relationship diagrams or UML – possibly augmented by a custom UML Profile. Semantic models might be expressed in RDF/OWL, yet another variation of UML, or some form of logic.

Exacerbating this problem even further is that each of the transformations requires yet another language in which it can be expressed, and that language is itself dependent on the languages of both the source and target structures. What I saw was a Tower-of-Babel problem: so many languages would be involved that even if the result were conceptually and logically sound it would be impractical to learn and understand.

## Analysis

The problem, as I saw it, was not in the diversity of concepts: It was more that each of the languages, in addition to expressing its relevant concepts, had its own representation language for those concepts. Furthermore, when one considered mapping one of these languages to another, what representation language should be used? Note that such mapping may require relating some types in one language to some enumeration values in another: types in one language corresponded to instances in another.

While the diversity of concepts in the different languages seemed necessary, was the diversity in representation languages also necessary? Could we somehow separate the concepts from the language in which those concepts are represented? Could we create a concept representation language (CRL) for all concepts?

The starting point for creating CRL was to identify some distinctions that we did not want to “bake” into the representations. There should be no distinction between the representations of:

* Different levels of abstraction, e.g. types vs. instances[[1]](#footnote-1)
* Entities vs. relationships[[2]](#footnote-2)

# The Core CRL Representational Entities

The search for a minimalist set of representational entities seemed to converge on the model shown in Figure 2‑1. An Element is the representation of a concept, identified by an identifier. Elements can “own” other elements, in which case the ownedElement is a part of the owningElement.



Figure 2‑1: Core CRL Representational Entities

There are two special cases (refinements) of Element that also seem essential: References and Refinements.

## Reference

Sometimes what you want to represent is not a new concept but rather a pointer to an existing concept. That is the purpose of a Reference. The existing concept is referenced by pointing to the Element that represents the concept. A Reference is a refinement of an Element, i.e. a Reference is also an Element. The intended usage is that a Reference represents a role that the referencedElement plays with respect to the Reference’s owningElement.

Figure 2‑2 shows how a traditional association can be represented. An Element represents the association. The Element owns two References, each indicating a specific role that a particular concept plays with respect to the association.



Figure 2‑2: Representing an Association[[3]](#footnote-3)

## Refinement

The other special case is that of Refinement. A Refinement points to two Elements, indicating that one (the refinedElement) is a refinement of the other (the abstractElement). Refinement can be used to represent both generalization and the type-instance relationship. A Refinement is also an Element.

It should be immediately apparent that a refinement could also be represented the same way as the association was in Figure 2‑2. While this is true from a representational perspective, it would hide the special role that Refinement plays in the representational scheme: the concept of Refinement is needed to define the relationship between Element and Reference and the relationship between Element and Refinement. Thus Refinement is a first-class representational entity.

# Grounding the Model

As simple as the CRL model is at this point, it has two practical problems when it comes to creating an actual implementation of the model:

1. The model is recursive: Refinement is used to define Refinement
2. The model is not grounded: there is no representation of values such as literals or the pointers shown in the model (owningElement, abstractElement, refinedElement, referencedElement).

Figure 3‑1 shows a refined version of the model that begins to resolve these problems. A new base class, BaseElement, has been added as a common ancestor for both Element and Value. The identifier attribute is moved from Element to BaseElement and the ownedElements end of the Ownership relation is moved to BaseElement and is re-labeled ownedBaseElements.



Figure 3‑1: Partially Grounded Model

The model provides for two types of values: Literals and Pointers. But the introduction of these values introduces the need for references and pointers to values as well as to elements so that they can be mapped. The resulting fully-grounded model is shown in Figure 3‑2.



Figure 3‑2: Fully Grounded Model

## BaseElement Attributes

Each base element is uniquely identified by a system-generated identifier. Since the identifier is not meaningful to a person looking at the model, two attributes are added to aid in human readability: name and definition. Note that these attributes are not intended to be, in the formal sense, part of the model. In particular it is expected that the potentially complex relationship between names and concepts be explicitly modeled: the name and definition attributes are provided simply as a human aid to identify the concept being represented.

Versioning is built into the model and is automatically maintained by the system. Changes to the BaseElement or, in the case of Elements, any of the ownedBaseElements (recursively) will result in the version number being incremented. This value may be read, but never set.

An optional uri attributes is provided so that any concept may be given a unique human-generated (and presumably human readable) identifier for a concept.

## ElementPointer and LiteralPointer Roles

To avoid infinite recursion, the model-defined attributes of Elements and their subclasses are implemented as values that are part of the ownedBaseElements set, either ElementPointers or LiteralPointers. These pointers have a role enumeration that identifies the intended role for the pointer with respect to its owningElement. Note that attributes defined in this manner are shown as being derived with a leading “/” before the attribute name.

## Versioning

Every BaseElement maintains a version automatically. The version is an integer and is incremented any time there is a change to the BaseElement or, in the case of Elements, any ownedBaseElements (recursively).

Pointers of all types, in addition to the actual pointer to the object, maintain the identifier and version number of the referenced object. This is so that if the indicated object is not immediately available, when the object is finally loaded its current version can be compared to the maintained version to see whether there have been any changes.

# Core Representations

## BaseElement

## BaseElementPointer

## BaseElementReference

## Element

## ElementPointer

## ElementPointerPointer

## ElementPointerReference

## ElementPointerRole

## ElementReference

## Literal

## LiteralPointer

## LiteralPointerPointer

## LiteralPointerReference

## LIteralPointerRole

## LiteralReference

## Refinement

# Functions

All ActiveCRL functions are expected to be idempotent: they will be invoked numerous times as the function structure is assembled.

## Making It Executable

Any Element can represent a function and may be made executable by creating a CRL Execution Function (Figure 5‑1) and associating it with the URI of another Element (which represents the function itself) and then making the element representing the function an abstraction of the element representing an instance of the function.



Figure 5‑1: Execution Concepts

Whenever a change occurs to an element, its abstractions are searched to see whether any of them have associated functions. If so, the execution of each associated function is initiated in a separate thread.

The signature of the function is fixed. The first argument is the Element that changed. The second argument is the ChangeNotification that triggered the execution. Note that the change notification may be the result of an underlying notification and that this history is conveyed as part of the change notification.

Operations typically require sub-structure – owned references that indicate the objects involved in the function. These may be objects from which information is extracted, objects that are modified, or objects that are actually created by the function.

## BaseElement Functions



Figure 5‑2: BaseElement Functions

### Delete

The Delete function removes an object from the Universe of Discourse and nils all CRL pointers to the object – including the one in the DeletedElementRef.



Figure 5‑3: Delete Operation Structure

### GetId

The GetId function creates a literal (if it does not already exist) and assigns it as the CreatedLiteralRef’s referencedLiteral. It sets the literal value to be the id of the SourceBaseElementRef’s referencedBaseElement’s Id.



Figure 5‑4: GetId Function Structure

### GetName

The GetName function creates a literal (if it does not already exist) and assigns it as the referencedLiteral of the CreatedLiteralRef. It sets the literal value to be the source base element’s name.



Figure 5‑5: GetName Function Structure

### GetOwningElement

The GetOwningElement function locates the owningElement of the base element indicated by the SourceBaseElementRef and sets the OwingElementRef’s elementPointer to indicate this object. If there is no owner, the elementPointer’s value is nil.



Figure 5‑6: GetOwningElement Function Structure

### GetUri

The GetUri function creates a literal (if it does not already exist) and sets it as the value of the CreatedLiteralRef’s referencedLiteral. It sets the literal value to be the source base element’s URI.



Figure 5‑7: GetUri Function Structure

### GetVersion

The GetVersion function creates a target literal (if it does not already exist) and assigns it as the referencedLiteral of the CreatedLiteralRef. It sets the literal value to be the source base element’s version.



Figure 5‑8: GetVersion Function Structure

### SetOwningElement

SetOwningElement sets the modified base element’s owner to be the element to which the OwningElementRef points.



Figure 5‑9: SetOwningElement Function Structure

### SetUri

SetUri sets the modified base element’s URI to be the value indicated by the SourceUriRef.



Figure 5‑10: SetUri Function Structure

## BaseElementPointer Functions



Figure 5‑11: BaseElementPointer Functions

### CreateBaseElementPointer

CreateBaseElementPointer creates a BaseElementPointer and points the CreatedBaseElementPointerRef to it. Note that this function does not establish the owningElement of the created base element pointer.



Figure 5‑12: CreateBaseElementPointer Function Structure

### GetBaseElement

GetBaseElement locates the base element to which the SourceBaseElementPointer points and sets the pointer of the IndicatedBaseElementRef to this base element. Note that the SourceBaseElementPointerRef is not strongly typed: the function checks to see whether the indicated BaseElement is of the correct type. If it is not, or it is nil, the IndicatedBaseElementRef’s referencedBaseElement is set to nil.

****

Figure 5‑13: GetBaseElement Function Structure

### GetBaseElementId

GetBaseElementId retrieves the id from the SourceBaseElementPointerRef’s indicated base element pointer. It creates a literal (if it does not already exist) and sets its literalValue to the string value of the id. Note that the SourceBaseElementPointerRef is not strongly typed. If it points to the wrong type or is nil, the literalValue is set to the empty string.



Figure 5‑14:GetBaseElementId Function Structure

### GetBaseElementVersion

GetBaseElementVersion retrieves the version from the SourceBaseElementPointerRef’s indicated base element pointer. It creates a literal (if it does not already exist) and sets its literalValue to the string value of the version. Note that the SourceBaseElementPointerRef is not strongly typed. If it points to the wrong type or is nil, the literalValue is set to the empty string.



Figure 5‑15: GetBaseElementVersion Function Structure

### SetBaseElement

SetBaseElement operates on the base element pointer indicated by the ModifiedBaseElementPointerRef. If the BaseElementRef points to a base element, it makes that base element the indicated base element of the base element pointer, which is otherwise set to nil. Note that the ModifiedBaseElementPointerRef is not strongly typed: if it does not point to a base element pointer, nothing happens (a log entry is generated).



Figure 5‑16: SetBaseElement Function Structure

## BaseElementReference Functions



Figure 5‑17: BaseElementReference Functions

### CreateBaseElementReference

CreateBaseElementReference creates a BaseElementReference and sets it as the referenced element of the CreatedBaseElementReferenceRef. Note that this function does not establish an owner for the created BaseElementReference.



Figure 5‑18: CreateBaseElementReference Function Structure

### GetBaseElementPointer

GetBaseElementPointer locates the referenced element of the SourceBaseElementReferenceRef, which should be a BaseElementReference. If it is found and is of the correct type, the referenced base element of the IndicatedBaseElementPointerRef is set to its child BaseElementPointer. If there is no BaseElementReference or it is of the wrong type or it does not have a child BaseElementPointer, the referenced base element of the IndicatedBaseElementPointerRef is set to nil.



Figure 5‑19: GetBaseElementPointer Function Structure

### GetReferencedBaseElement

GetReferencedBaseElement locates the referenced element of the SourceBaseElementReferenceRef, which should be a BaseElementReference. If it is found and is of the correct type, the referenced base element of the IndicatedBaseElementRef is set to its referenced BaseElement. If there is no BaseElementReference or it is of the wrong type or it does not have a referenced BaseElement, the referenced base element of the TargetBaseElement is set to nil.



Figure 5‑20: GetReferencedBaseElement Function Structure

### SetReferencedBaseElement

SetReferencedBaseElement locates the modified base element of the ModifiedBaseElementReferenceRef and the indicated base element of the SourceBaseElementRef. If the modified base element is of the correct type (i.e. is a BaseElementReference), it sets its referencedBaseElement to the base element indicated by the SourceBaseElementRef. If the modified base element is not a BaseElementReference a log entry is generated.



Figure 5‑21: SetReferencedBaseElement Function Structure

## Element Functions



Figure 5‑22: Element Functions

### CreateElement

CreateElement creates an Element and sets it as the referenced element of the CreatedElement. Note that this function does not establish an owner for the created Element.



Figure 5‑23: CreateElement Function Structure

### GetDefinition

GetDefinition creates a literal and assigns it as the referencedLiteral of the CreatedLiteralRef. It sets the literal value to the name of the element indicated by the SourceElementRef.



Figure 5‑24: GetDefinition Function Structure

### GetDefinitionLiteral

GetDefinitionLiteral locates the element indicated by the SourceElementRef, finds its definition literal and sets it as the referencedLiteral of the IndicatedLiteralRef. If there is no definition literal, the referencedLiteral is set to nil.



Figure 5‑25: GetDefinitionLiteral Function Structure

### GetDefinitionLiteralPointer

GetDefinitionLiteralPointer locates the element indicated by the SourceElementRef and gets its definition literal pointer (if one exists). It sets the referencedLiteralPointer of the IndicatedLiteralPointerRef to this value.



Figure 5‑26: GetDefinitionLiteralPointer Function Structure

### GetNameLiteral

GetNameLiteral finds the element indicated by the SourceElementRef and gets its name literal. It sets the name literal as the referencedLiteral of the IndicatedLiteralRef.



Figure 5‑27: GetNameLiteral Function Structure

### GetNameLiteralPointer

GetNameLiteralPointer locates the element referenced by the SourceElementRef and retrieves its name literal pointer. This is set as the referencedLiteralPointer of the IndicatedLiteralPointerRef.



Figure 5‑28: GetNameLiteralPointer Function Structure

### GetUriLiteral

GetUriLiteral locates the element indicated by the SourceElementRef and locates its Uri literal. It then sets this as the referencedLiteral of the IndicatedLiteralRef.



Figure 5‑29: GetUriLiteral Function Structure

### GetUriLiteralPointer

GetUriLiteralPointer locates the element indicated by the SourceElementRef and locates its Uri literal pointer (if any). It then sets this as the referencedLiteralPointer of the IndicatedLiteralPointerRef.



### SetDefinition

SetDefinition locates the literal indicated by the SourceLiteralRef and assigns its value as the definition of the element indicated by the ModifiedElementRef.



Figure 5‑30: SetDefinition Function Structure

### SetName



Figure 5‑31: SetName Function Structure

## ElementPointer Functions

There are four create functions for element pointers, one for each of the roles that an element pointer can play. Once established, this role cannot be changed, only queried.



Figure 5‑32: ElementPointer Functions

### CreateAbstractElementPointer

CreateAbstractElementPointer creates an ElementPointer with its role set to ABSTRACT\_ELEMENT. It is set as the referencedBaseElement of the CreatedElementPointerRef. If this pointer already exists, no action is taken.



Figure 5‑33: CreateAbstractElementPointer Function Structure

### CreateOwningElementPointer

CreateOwningElementPointer creates an ElementPointer with its role set to OWNING\_ELEMENT. It is set as the referencedBaseElement of the CreatedElementPointerRef. If this pointer already exists, no action is taken.



Figure 5‑34: CreateOwningElementPointer Function Structure

### CreateReferencedElementPointer



Figure 5‑35: CreateReferencedElementPointer Function Structure

### CreateRefinedElementPointer

CreateRefinedElementPointer creates an ElementPointer with its role set to REFINED\_ELEMENT. It is set as the referencedBaseElement of the CreatedElementPointerRef. If this pointer already exists, no action is taken.



Figure 5‑36: CreateRefinedElementPointer Function Structure

### GetElement

GetElement locates the element to which the SourceElementPointer points and sets the pointer of the IndicatedElementRef to this element.



Figure 5‑37: GetElement Function Structure

### GetElementId

GetElementId retrieves the id from the SourceElementPointerRef’s indicated element pointer. It creates a literal (if it does not already exist) and sets its literalValue to the string value of the id.



Figure 5‑38: GetElementId Function Structure

### GetElementPointerRole

GetElementPointerRole retrieves the pointerRole from the SourceElementPointerRef’s indicated element pointer. It creates a literal (if it does not already exist) and sets its literalValue to the string value of the pointerRole. Possible values are:

* ABSTRACT\_ELEMENT
* REFINED\_ELEMENT
* OWNING\_ELEMENT
* REFERENCED\_ELEMENT



Figure 5‑39: GetElementPointerRole Function Structure

### GetElementVersion

GetElementVersioin retrieves the version from the SourceElementPointerRef’s indicated element pointer. It creates a literal (if it does not already exist) and sets its literalValue to the string value of the version.



Figure 5‑40: GetElementVersion Function Structure

### SetElement

SetElement gets the element indicated by the ElementRef, locates the element pointer indicated by the ModifiedElementPointerRef, and sets the element pointer’s element to the indicated element.



Figure 5‑41: SetElement Function Structure

## ElementPointerPointer Functions



Figure 5‑42: ElementPointerPointer Functions

### CreateElementPointerPointer

CreateElementPointerPointer creates an ElementPointerPointer (if one does not already exist) and sets it as the referencedElementPointerPointer of the CreatedElementPointerPointerRef.



Figure 5‑43: CreateElementPointerPointer Function Structure

### GetElementPointer

GetElementPointer locates the element pointer pointer that is the referencedBaseElement of the SourceElementPointerPointerRef, gets its element pointer and sets that as the referencedElementPointer of the IndicatedElementPointerRef. If the element pointer pointer is nil or of the wrong type, the referencedElementPointer is set to nil.



Figure 5‑44:GetElementPointer Function Structure

### GetElementPointerId

GetElementPointerId creates a Literal (if one does not already exist) and assigns it as the referencedLiteral of the CreatedLiteralRef. It then gets the referencedBaseElement of the SourceElementPointerPointerRef. If it is not nil and is of the correct type, the elementPointerId is retrieved and set as the literal value of the created literal.



Figure 5‑45: GetElementPointerId Function Structure

### GetElementPointerVersion

GetElementPointerVersion creates a Literal (if one does not already exist) and assigns it as the referencedLiteral of the CreatedLiteralRef. It then gets the referencedBaseElement of the SourceElementPointerPointerRef. If it is not nil and is of the correct type, the elementPointerVersion is retrieved and set as the literal value of the created literal.



Figure 5‑46: GetElementPointerVersion Function Structure

### SetElementPointer

SetElementPointer retrieves the element pointer referenced by the ElementPointerRef. It locates the element pointer pointer referenced by the ModifiedElementPointerPointerRef. If it exists and is of type ElementPointerPointer its elementPointer is set to the retrieved element pointer.



## ElementPointerReference Functions



Figure 5‑47: ElementPointerReference Functions

### CreateElementPointerReference

CreateElementPointerReference creates an ElementPointerReference (if one does not already exist) and assigns it as the referencedElementPointerReference of the CreatedElementPointerReferenceRef.



Figure 5‑48: CreateElementPointerReference Function Structure

### GetElementPointerPointer

GetElementPointer locates the element pointer reference indicated by the SourceElementPointerReferenceRef, finds its referencedElementPointer, and sets it as the referencedBaseElement of the IndicatedElementPointerPointerRef. Note that the SourceElementPointerReferenceRef is not strongly typed: if it is of the wrong type a log entry is generated and the IndicatedElementPointerPointerRef’s referencedBaseElementPointer is set to nil. 

Figure 5‑49: GetElementPointerPointer Function Structure

### GetReferencedElementPointer

GetReferencedElementPointer locates the element pointer reference indicated by the SourceElementPointerReferenceRef, finds its referencedElementPointer, and sets it as the referencedElementPointer of the IndicatedElementPointerRef. Note that the SourceElementPointerReferenceRef is not strongly typed: if it is of the wrong type a log entry is generated and the IndicatedElementPointerRef’s referencedElementPointer is set to nil.



Figure 5‑50: GetElementPointer Function Structure

### SetReferencedElementPointer

SetReferencedElementPointer locates the referencedElementPointer of the SourceElementPointerRef and sets it as the referencedElement of the ModifiedElementPointerReferenceRef.



Figure 5‑51: SetReferencedElementPointer Function Structure

## ElementReference Functions



Figure 5‑52: ElementReference Functions

### CreateElementReference

CreateElementReference creates an ElementReference (if one does not already exist) and sets it as the referencedElement of the CreatedElementReferenceRef.



Figure 5‑53: CreateElementReference Function Structure

### GetElementPointer

GetElementPointer gets the indicated source element reference and gets its element pointer. The element pointer is then set as the referencedElementPointer of the IndicatedElementPointerRef.



Figure 5‑54: GetElementPointer Function Structure

### GetReferencedElement

GetReferencedElement gets the indicated source element reference and gets its referencedElement. The referencedElement of the IndicatedElementRef is then set to this value.



### SetReferencedElement

SetReferencedElement locates the referencedElement of the ModifiedElementReferenceRef, verifies that it is an ElementReference, and sets its referencedElement to be the source element.



Figure 5‑55: SetReferencedElement Function Structure

## Literal Functions



Figure 5‑56: Literal Functions

### CreateLiteral

CreateLiteral creates a Literal (if one does not already exist) and assigns it as the referencedLiteral of the CreatedLiteralRef.



Figure 5‑57: CreateLiteral Function Structure

### GetLiteralValue

GetLiteralValue gets the literal value from the referencedLiteral of the SourceLiteralRef, creates a Literal (if one does not already exist), and assigns the retrieved literal value as the new literal’s literal value.



Figure 5‑58: GetLiteralValue Operatsion Structure

### SetLiteralValue

SetLiteralValue takes the literal value from the SourceLiteral and assigns it as the literal value of the ModifiedLiteral.



Figure 5‑59: SetLiteralValue Function Structure

## LiteralPointer Functions



Figure 5‑60: LiteralPointer Functions

### CreateDefinitionLiteralPointer

CreateDefinitionLiteralPointer creates a LiteralPointer (if one does not already exist) with a literalPointerRole equal DEFINITION and sets it as the referencedLiteralPointer of the CreatedLiteralPointerRef.



Figure 5‑61: CreateDefinitionLiteralPointer Function Structure

### CreateNameLiteralPointer

CreateNameLiteralPointer creates a LiteralPointer (if one does not already exist) with a literalPointerRole equal NAME and sets it as the referencedLiteralPointer of the CreatedLiteralPointerRef.



Figure 5‑62: CreateNameLiteralPointer Function Structure

### CreateUriLiteralPointer

CreateUriLiteralPointer creates a LiteralPointer (if one does not already exist) with a literalPointerRole equal URI and sets it as the referencedLiteralPointer of the CreatedLiteralPointerRef.



Figure 5‑63: CreateUriLiteralPointer Function Structure

### CreateValueLiteralPointer

CreateValueLiteralPointer creates a LiteralPointer (if one does not already exist) with a literalPointerRole equal VALUE and sets it as the referencedLiteralPointer of the CreatedLiteralPointerRef.



Figure 5‑64: CreateValueLiteralPointer Function Structure

### GetLiteral

GetLiteral locates the LiteralPointer indicated by the SourceLiteralPointerRef. It finds the Literal to which it points and assigns it as the referencedLiteral of the IndicatedLiteralRef.



Figure 5‑65: GetLiteral Function Structure

### GetLiteralId

GetLiteralId locates the LiteralPointer indicated by the SourceLiteralPointerRef and gets the id of the literal to which it refers. It finds the Literal referenced by the CreatedLiteralRef, creating it if it does not exist, and sets its literalValue to the id.



Figure 5‑66: GetLiteralId Function Structure

### GetLiteralPointerRole

GetLiteralPointerRole locates the LiteralPointer indicated by the SourceLiteralPointerRef and gets its literalPointerRole. It finds the Literal referenced by the CreatedLiteralRef, creating it if it does not exist, and sets its literalValue to the role. Possible values are:

* DEFINITION
* NAME
* URI
* VALUE



Figure 5‑67: GetLiteralPointerRole Function Structure

### GetLiteralVersion

GetLiteralVersion creates a literal (if one does not already exist) and assigns it as the referencedLiteral of the CreatedLiteralRef. It locates the LiteralPointer indicated by the SourceLiteralRef. It then gets the literal version from the literal pointer and assigns it as the literalValue of the created literal.



Figure 5‑68: GetLiteralVersion Function Structure

### SetLiteral



Figure 5‑69: SetLiteral Function Structure

## LiteralPointerPointer Functions



Figure 5‑70: LiteralPointerPointer Functions

### CreateLiteralPointerPointer

CreateLiteralPointerPointer creates a LiteralPointerPointer (if one does not already exist) and assigns it as the referencedLiteralPointerPointer of the CreatedLiteralPointerPointerRef.



Figure 5‑71: CreateLiteralPointerPointer Function Structure

### GetLiteralPointer

GetLiteralPointer locates the LiteralPointerPointer indicated by the SourceLiteralPointerPointerRef and verifies that it is of the correct type. It then locates the LiteralPointer to which it references and assigns it as the referencedLiteralPointer of the IndicatedLiteralPointerRef.



Figure 5‑72: GetLiteralPointer Function Structure

### GetLiteralPointerId

GetLiteralPointerId locates the LiteralPointerPointer indicated by the SourceLiteralPointerPointerRef and verifies that it is of the correct type. It creates a Literal (if one does not already exist) and assigns it as the referencedLiteral of the CreatedLiteralRef. It retrieves the id of the LiteralPointer indicated by the LiteralPointerPointer and assigns it as the literalValue of the Literal.



Figure 5‑73: GetLiteralPointerId Function Structure

### GetLiteralPointerVersion

GetLiteralPointerVersion locates the LiteralPointerPointer indicated by the SourceLiteralPointerPointerRef and verifies that it is of the correct type. It creates a Literal (if one does not already exist) and assigns it as the referencedLiteral of the CreatedLiteralRef. It retrieves the version of the LiteralPointer indicated by the LiteralPointerPointer and assigns it as the literalValue of the Literal.



Figure 5‑74: GetLiteralPointerVersion Function Structure

### SetLiteralPointer

SetLiteralPointer locates the LiteralPointerPointer indicated by the ModifiedLiteralPointerPointerRef and verifies that it is of the correct type. It locates the LiteralPointer indicated by the LiteralPointerRef and assigns it as the literalPointer of the LiteralPointerPointer.



Figure 5‑75: SetLiteralPointer Function Structure

## LiteralPointerReference



Figure 5‑76: LiteralPointerReference Functions

### CreateLiteralPointerReference

CreateLiteralPointerReference creates a LiteralPointerReference (if one does not already exist) and assigns it as the referencedLiteralPointerReference of the CreatedLiteralPointerReferenceRef.



Figure 5‑77: CreateLiteralPointerReference Function Structure

### GetLiteralPointerPointer

GetLiteralPointerPointer locates the LiteralPointerReference indicated by the sourceLiteralPointerReferenceRef and verifies that it is of the correct type. It then retrieves its LiteralPointerPointer and assigns it as the referencedBaseElement of the IndicatedLiteralPointerPointerRef.



Figure 5‑78: GetLiteralPointerPointer Function Structure

### GetReferencedLiteralPointer

GetReferencedLiteralPointer locates the LiteralPointerReference indicated by the SourceLiteralPointerReferenceRef and verifies that it is of the correct type. It then locates the LiteralPointer to which it points and assigns this as the referencedLiteralPointer of the IndicatedLiteralPointerRef.



### SetReferencedLiteralPointer

SetReferencedLiteralPointer locates the LiteralPointerReference indicated by the ModifiedLiteralPointerReferenenceRef and verifies that it is of the correct type. It locates the LiteralPointer indicated by the SourceLiteralPointer and assigns it as the referencedLiteralPointer of the LiteralPointerReference.



Figure 5‑79: SetReferencedLiteralPointer Function Structure

## LiteralReference



Figure 5‑80: LiteralReference Functions

### CreateLiteralReference

CreateLiteralReference creates a LiteralReference (if one does not already exist) and assigns it as the referencedElement of the CreatedLiteralReferenceRef.



Figure 5‑81: CreateLiteralReference Function Structure

### GetLiteralPointer

GetLiteralPointer locates the LiteralReference indicated by the SourceLiteralReferenceRef. If it is of the correct type, its owned LiteralPointer is located and set as the referencedLiteralPointer of the IndicatedLiteralPointerRef.



Figure 5‑82: GetLiteralPointer Function Structure

### GetReferencedLiteral

GetReferencedLiteral locates the LiteralReference indicated by the SourceLiteralReferenceRef and verifies that it is of the correct type. If so, it gets its referencedLiteral and assigns it as the referencedLiteral of the IndicatedLiteralRef.



Figure 5‑83: GetReferencedLiteral Function Structure

### SetReferencedLiteral

SetReferenceLiteral locates the LiteralReference indicated by the ModifiedLiteralReferenceRef and verifies that it is of the correct type. If so, it locates the Literal indicated by the SourceLiteralReference and assigns it as the referencedLiteral of the LiteralReference.



Figure 5‑84: SetReferencedLiteral Function Structure

## Refinement Functions

### CreateRefinement

CreateRefinement creates a Refinement (if one does not already exist) and assigns it as the referencedElement of the CreatedRefinementRef.



Figure 5‑85: CreateRefinement Function Structure

### GetAbstractElement

GetAbstractElement locates the Refinement indicated by the SourceRefinementRef and checks that it is a Refinement. If it is, it gets its abstractElement and assigns it as the referencedElement of the IndicatedElementRef.



Figure 5‑86: GetAbstractElement Function Structure

### GetAbstractElementPointer

GetAbstractElementPointer locates the Refinement indicated by the SourceRefinementRef and checks that it is a Refinement. If it is, it gets its abstractElementPointer and assigns it as the referencedElementPointer of the IndicatedElementPointerRef.



Figure 5‑87: GetAbstractElementPointer Function Structure

### GetRefinedElement

GetRefinedElement locates the Refinement indicated by the SourceRefinementRef and validates that it is a Refinement. If it is, it gets the refinedElement of the Refinement and assigns it as the referencedElement of the IndicatedElementRef.



### GetRefinedElementPointer

GetRefinedElementPointer locates the Refinement indicated by the SourceRefinementRef and verifies that it is a Refinement. If it is, it gets the refinedElementPointer and assigns it as the referencedElement of the IndicatedElementPointerRef.



Figure 5‑88: GetRefinedElementPointer Function Structure

### SetAbstractElement

SetAbstractElement locates the Refinement indicated by the ModifiedRefinementRef and verifies that it is a Refinement. It locates the Element indicated by the SourceElementRef and sets it as the abstractElement of the Refinement.



Figure 5‑89: SetAbstractElement Function Structure

### SetRefinedElement

SetRefinedElement locates the Refinement indicated by the ModifiedRefinementRef and verifies that it is a Refinement. It locates the Element indicated by the SourceElementRef and assigns it as the refinedElement of the Refinement.



Figure 5‑90: SetRefinedElement FunctionStructure

# Go Implementation

A working implementation of CRL has been developed in the Go programming language. The current version can be found at <https://github.com/pbrown12303/activeCRL>. [[4]](#footnote-4)

For those not familiar with the language it is important to understand the relationship between name capitalization and visibility in Go: types and functions that are capitalized are public, i.e. visible outside the package in which they are defined. Types and functions whose names are not capitalized are visible only within their defining package.

## Core Interfaces



Figure 6‑1: CRL Core Interfaces

## Core Implementation



Figure 6‑2: Core Implementation

1. The MOF specification (<http://www.omg.org/spec/MOF/2.5.1/PDF>) section 7.3 contains a discussion of the number of meta-layers there should be in the specification (which is built upon the UML specification <http://www.omg.org/spec/UML/2.5/PDF>). Regardless of the answer to the question, it should be possible to map to and from languages with different numbers of meta-layers, so the representations of meta-layers should not be fundamental to the representation language. [↑](#footnote-ref-1)
2. The concept of Customer may be an entity in one language and a relationship (e.g. between a Person and a Company) in another. A mapping between such languages would require mapping an entity in one language to a relationship in the other. To avoid having many different types of relationship representations, relationships should be able to relate relationships as well as entities. In other words, every relationship must also be a first-class entity [↑](#footnote-ref-2)
3. Note that the UML instance notation used here actually violates one of the CRL guiding principles: there should be no difference in the representation of a type and an instance. [↑](#footnote-ref-3)
4. An earlier implementation in Java was abandoned when experimentation with making the model executable began, for two reasons. The threading model was deemed to heavyweight for what was anticipated to be a fine-grained highly-parallel execution model. Furthermore, the tight coupling between the built-in change notification mechanism and a transactional semantics best suited for editing scenarios made the implementation of a highly-parallel multi-threaded notification/update mechanism problematic. [↑](#footnote-ref-4)