ActiveCRL

Overview and Architecture

Paul C. Brown  
  
June 25, 2017

Contents

[1 Introduction 4](#_Toc487618947)

[1.1 Background 4](#_Toc487618948)

[1.2 Analysis 5](#_Toc487618949)

[2 The Core CRL Representational Entities 5](#_Toc487618950)

[2.1 Reference 6](#_Toc487618951)

[2.2 Refinement 6](#_Toc487618952)

[3 Grounding the Model 6](#_Toc487618953)

[3.1 BaseElement Attributes 8](#_Toc487618954)

[3.2 ElementPointer and LiteralPointer Roles 9](#_Toc487618955)

[3.3 Versioning 9](#_Toc487618956)

[3.4 Making It Executable 9](#_Toc487618957)

[4 Go Implementation 10](#_Toc487618958)

[4.1 Core Interfaces 11](#_Toc487618959)

[4.2 Core Implementation 12](#_Toc487618960)

[1 Introduction 4](#_Toc487619041)

[Figure 2‑1: SIMF Concept 4](#_Toc487619042)

[2 The Core CRL Representational Entities 5](#_Toc487619043)

[Figure 3‑1: Core CRL Representational Entities 5](#_Toc487619044)

[Figure 3‑2: Representing an Association 6](#_Toc487619045)

[3 Grounding the Model 6](#_Toc487619046)

[Figure 4‑1: Partially Grounded Model 7](#_Toc487619047)

[Figure 4‑2: Fully Grounded Model 8](#_Toc487619048)

[Figure 4‑3: Execution Concepts 9](#_Toc487619049)

[4 Go Implementation 10](#_Toc487619050)

[Figure 5‑1: CRL Core Interfaces 11](#_Toc487619051)

[Figure 5‑2: Core Implementation 12](#_Toc487619052)

# 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

# Operations

## Making It Executable

Any Element can represent an operation and may be made executable by creating a CRL Execution Function (Figure 3‑3) 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 3‑3: 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.

## BaseElement Operations

* GetIdentifier
* GetVersion
* GetName
* GetUri
* Delete
* SetOwningElement
* SetUri

## BaseElementPointer Operations

* CreateBaseElementPointer
* GetBaseElement
* SetBaseElement

## BaseElementReference Operations

* CreateBaseElementReference
* GetReferencedBaseElement
* GetBaseElementPointer
* SetReferencedBaseElement

## Element Operations

* CreateElement
* GetDefinition
* GetDefinitionLiteral
* GetDefinitionLiteralPointer
* GetNameLiteral
* GetNameLiteralPointer
* GetUriLiteral
* GetUriLiteralPointer
* SetName
* SetDefinition
* SetName
* SetUri

## ElementPointer Operations

* CreateElementPointer
* GetElement
* GetElementRole
* SetElement
* SetElementRole

## ElementPointerPointer Operations

* CreateElementPointerPointer
* GetElementPointer
* SetElementPointer

## ElementPointerReference Operations

* CreateElementPointerReference
* GetElementPointer
* GetElementPointerPointer
* SetElementPointer

## ElementReference Operations

* CreateElementReference
* GetReferencedElement
* GetElementPointer
* SetReferencedElement

## Literal Operations

* CreateLiteral
* GetLiteralValue
* SetLiteralValue

## LiteralPointer Operations

* CreateLiteralPointer
* GetLiteral
* GetLiteralRole
* SetLiteral
* SetLiteralRole

## LiteralPointerPointer Operations

* CreateLiteralPointerPointer
* GetLiteralPointer
* SetLiteralPointer

## LiteralPointerReference

* CreateLiteralPointerReference
* GetReferencedLiteralPointer
* GetLiteralPointerPointer
* SetReferencedLiteralPointer

## LiteralReference

* CreateLiteralReference
* GetReferencedLiteral
* GetLiteralPointer
* SetReferencedLiteral

## Refinement Operations

* CreateRefinement
* GetAbstractElement
* GetAbstractElementPointer
* GetRefinedElement
* GetRefinedElementPointer
* SetAbstractElement
* SetRefinedElement

# 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 4‑1: CRL Core Interfaces

## Core Implementation



Figure 4‑2: 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)