Data Analytics and Visualization Environment for xAPI and the Total Learning Architecture: DAVE Learning Analytics Algorithms

Yet Analytics

October 15, 2019

Introduction

This report introduces the updated definition of learning analytics algorithms in terms of **Operations**, **Primitives** and **Algorithms** and presents an updated definition for each of the previously defined algorithms. The previous definitions will be included for reference. In a more general sense, this report establishes a set of style guidelines for the reporting of algorithms and associated visualization templates.

This document will be updated to include additional Operations, Primitives and Algorithms as they are defined by the Author of this report or members of the Open Source Community. Updates may also address refinement of existing definitions and this document should be understood to be an example of algorithm presentation and not the final state of any defined algorithm.

The structure of this documents is as follows:

1. An Introduction to Z notation and its usage in this document
2. A formal specification for xAPI written in Z
3. An Introduction to Terminology of Operations, Primitives and Algorithms
4. What is an Operation
5. What is a Primitive
6. What is an Algorithm
7. Foundational Operations
8. Example Primitives
9. An algorithm definition including
   (a) Init
   (b) Relevant?
   (c) Accept?
   (d) Step
   (e) Result
10. Previous Algorithm definitions where each consists of
    (a) an introduction for the algorithm
    (b) the structure of the ideal input data
    (c) how to retrieve input data from an LRS
    (d) the statement parameters which the algorithm will utilize

(e) notices regarding data collected during the 2018 pilot test of the TLA
(f) a summary of the algorithm
(g) the formal specification of the algorithm
(h) pseudocode representation of the algorithm
(i) JSONSchema for the output of the algorithm
(j) a description of the associated visualization
(k) a prototype of the visualization
(l) a collection of suggestions describing how the algorithm could be adapted to improve the quality of the visualization prototype

1 Z Notation Introduction

The following subsections provide a high level overview of select properties of Z Notation based on "The Z Notation: A Reference Manual" by J. M. Spivey. A copy of this reference manual can be found at dave/docs/z/Z-notation reference manual.pdf. In many cases, definitions will be pulled directly from the reference manual and when this occurs, the relevant page number(s) will be included. For a proper introduction with tutorial examples, see chapter 1, "Tutorial Introduction" from pages 1 to 23. For the $LaTeX$ symbols used to write Z, see the reference document found at dave/docs/z/zed-csp-documentation.pdf.

1.1 Decorations

The following decorations are used through this document and are taken directly from the reference manual. For a complete summary of the Syntax of Z, see chapter 6, Syntax Summary, startingon page 142.

$'$	[indicates final state of an operation]
?	[indicates input to an operation]
!	[indicates output of an operation]
Δ	[indicates the schema results in a change to the state space]
Ξ	[indicates the schema does not result in a change to the state space]
$\gg$	[indicates output of the left schema is input to the right schema]

1.2 Types

Objects have a type which characterizes them and distinguish them from other kinds of objects.

- Basic types are sets of objects which have no internal structure of interest meaning the concrete definition of the members is not relevant, only their shared type.
- Free types are used to describe (potentially nested and/or recursive) sets of objects. In the most simple case, a free type can be an enumeration of constants.

Within the xAPI Formal Specification, both of these types are used to describe the Inverse Functional Identifier property.

- Introduction of the basic types $MBOX$, $MBOX\_SHA1SUM$, $OPENID$ and $ACCOUNT$ allows the specification to talk about these constraints within the xAPI specification without defining their exact structure
- The free type IFI is defined as one of the above basic types meaning an object of type IFI is of type $MBOX$ or $MBOX\_SHA1SUM$ or $OPENID$ or $ACCOUNT$.

Types can be composed together to form composite types and thus complex objects.

$$[MBOX, MBOX\_SHA1SUM, OPENID, ACCOUNT]$$
$$IFI ::= MBOX \mid MBOX\_SHA1SUM \mid OPENID \mid ACCOUNT$$

Within the xAPI Formal Specification, IFI is used within the definition of an *agent* as presented in the schema $Agent$.

$$\begin{array}{|l}
\textit{Agent} \\
\hline
agent : AGENT \\
objectType : OBJECTTYPE \\
name : \mathbb{F}_1 \#1 \\
ifi : IFI \\
\hline
objectType = Agent \\
agent = \{ifi\} \cup \mathbb{P}\{name, objectType\} \\
\hline
\end{array}$$

See section 2.2, pages 28 to 34, and chapter 3, pages 42 to 85, for more information about Schemas and the Z Language.

1.3 Sets

A collection of elements that all share a type. A set is characterized solely by which objects are members and which are not. Both the order and repetition of objects are ignored. Sets are written in one of two ways:

- listing their elements
- by a property which is characteristic of the elements of the set.

such that the following law from page 55 holds for some object y

$$y \in \{x_1, ..., x_n\} \iff y = x_1 \vee ... \vee y = x_n$$

1.4 Ordered Pairs

Two objects (x, y) where x is paired with y. An n-tuple is the pairing of n objects together such that equality between two n-tuple pairs is given by the law from page 55

$$(x_1, ..., x_n) = (y_1, ..., y_n) \iff x_1 = y_1 \wedge ... \wedge x_n = y_n$$

When ordered pairs are used with respect to application (as seen on page 60)

$$fx \Rightarrow f(x) \iff (x, y) \in f$$

which states that $f(x)$ is defined if and only if there is a unique value y which result from fx Additionally, application associates to the left

$$fxy \Rightarrow (fx)y \Rightarrow (f(x), y)$$

meaning $f(x)$ results in a function which is then applied to y.

1.5 Sequences

A collection of elements where their ordering matters such that

$$\langle a_1, ..., a_n \rangle \Rightarrow \{1 \mapsto a_1, ..., n \mapsto a_n\}$$

as seen on page 115. Additionally, iseq is used to describe a sequence whose members are distinct.

1.6 Bags

A collection of elements where the number of times an element appears in the collection is meaningful.

$$[\![a_1, ..., a_n]\!] \Rightarrow \{a_1 \mapsto k_1, ..., k_n \mapsto k_n\}$$

As described on page 124, each element a_i appears k_i times in the list $a_1, ..., a_n$ such that the number of occurances of a_i within bag A is returned by

$$count\, A\, a_i \;\; \equiv A \,\#\, a_i$$

1.7 Maps

This document introduces a named subcategory of sets, *map* of the free type KV, which are akin to sequences and bags. To enumerate the members of a *map*, $\langle\!\langle ... \rangle\!\rangle$ is used but should not be confused with $d_i \langle\!\langle E_i[T] \rangle\!\rangle$ within a Free Type definition. The distinction between the two usages is context dependent but in general, if $\langle\!\langle ... \rangle\!\rangle$ is used outside of a constructor declaration within a Free Type definition, it should be assumed to represent a *map*.

$$KV ::= base \mid associate \langle\!\langle KV \times X \times Y \rangle\!\rangle$$

where

base	[is a constant which is the empty $KV \Rightarrow \langle\!\langle\rangle\!\rangle$]
associate	[is a constructor and is infered to be an injection]

The full enumeration of all properties, constraints and functions specific to a *map* with type KV will be defined elsewhere but *associate* can be understood to (in the most basic case) operate as follows.

$$associate(base, x_i, y_i) = \langle\!\langle (x_i, y_i) \rangle\!\rangle \Rightarrow \langle\!\langle x_i \mapsto y_i \rangle\!\rangle$$

The enumeration of a *map* was chosen to be $\langle\!\langle ... \rangle\!\rangle$ as a *map* is a collection of injections such that if M is the result of $associate(base, x_i, y_i)$ from above then

$$atKey(M, x_i) = y_i \iff x_i \mapsto y_i \wedge (x_i, y_i) \in M$$

1.8 Select Operations and Symbols

The follow are defined in Chpater 4 (The Mathematical Tool-kit) within the reference manual and are used extensively throughout this document. In many cases, the functions listed here will serve as Operations in the context of Primitives and Algorithms.

1.8.1 Functions

$\pfun$	[relate each x ∈ X to at most one y ∈ Y, page 105]
$\rightarrow$	[relate each x ∈ X to exactly one y ∈ Y, page 105]
$\pinj$	[map different elements of x to different y, page 105]
$\rightarrowtail$	[$\pinj$ that are also $\rightarrow$, page 105]
$\psurj$	[$X \pfun Y$ where whole of Y is the range, page 105]
$\twoheadrightarrow$	[$X \pfun Y$ whole of X as domain and whole of Y as range, page 105]
$\bij$	[map x ∈ X one-to-one with y ∈ Y, page 105]

$$
\begin{array}{l}
X \pfun Y == \{\, f : X \leftrightarrow Y \mid (\forall x : X; y1, y2 : Y \bullet \\
\qquad\qquad\qquad\qquad (x \mapsto y_1 \in f \;\wedge\; (x \mapsto y_2) \in f \Rightarrow y_1 = y_2))\} \\
X \rightarrow Y == \{\, f : X \pfun Y \mid \mathrm{dom}\, f = X \} \\
X \pinj Y == \{\, f : X \pfun Y \mid (\forall x_1, x_2 : \mathrm{dom}\, f \bullet f(x_1) = f(x_2) \Rightarrow x_1 = x_2)\} \\
X \rightarrowtail Y == (X \pinj Y) \cap (X \rightarrow Y) \\
X \psurj Y == \{\, f : X \pinj Y \mid \mathrm{ran}\, f = Y \} \\
X \twoheadrightarrow Y == (X \psurj Y) \cap (X \rightarrow Y) \\
X \bij Y == (X \twoheadrightarrow Y) \cap (X \rightarrowtail Y)
\end{array}
$$

1.8.2 Ordered Pairs, Maplet and Composition of Relations

first	[returns the first element of an ordered pair, page 93]
second	[returns the second element of an ordered pair, page 93]
$\mapsto$	[maplet is a graphic way of expressing an ordered pair, page 95]
dom	[set of all x ∈ X related to atleast one y ∈ Y by R, page 96]
ran	[set of all y ∈ Y related to atleast one x ∈ X by R, page 96]
$\fatsemi$	[The composition of two relationships, page 97]
$\circ$	[The backward composition of two relationships, page 97]

$$
\begin{array}{l}
[X, Y] \\
\hline
first : X \times Y \rightarrow X \\
second : X \times Y \rightarrow Y \\
\hline
\forall x : X; y : Y \bullet \\
\qquad first(x, y) = x \wedge \\
\qquad second(x, y) = y
\end{array}
$$

$$
\begin{array}{|l}
\hline [X, Y] \\
\_ \mapsto \_ : X \times Y \to X \times Y \\
\hline \forall x : X; y : Y \bullet \\
x \mapsto y = (x, y) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline [X, Y] \\
\mathrm{dom} : (X \leftrightarrow Y) \to \mathbb{P}\, X \\
\mathrm{ran} : (X \leftrightarrow Y) \to \mathbb{P}\, Y \\
\hline \forall R : X \leftrightarrow Y \bullet \\
\quad \mathrm{dom}\, R = \{\, x : X;\ y : Y \mid x \underline{R} y \bullet x \} \land \\
\quad \mathrm{ran}\, R = \{\, x : X;\ y : Y \mid x \underline{R} y \bullet y \} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline [X, Y, Z] \\
\_ \,⨾\, \_ : (X \leftrightarrow Y) \times (Y \leftrightarrow Z) \to (X \leftrightarrow Z) \\
\_ \circ \_ : (Y \leftrightarrow X) \times (X \leftrightarrow Y) \to (X \leftrightarrow X) \\
\hline \forall\, Q : X \leftrightarrow Y; R : Y \leftrightarrow Z \bullet \\
\quad Q \,⨾\, R = R \circ Q = \{\, x : X; y : Y; z : Z \mid \\
\qquad\qquad x\, \underline{Q}\, y \ \land\ y\, \underline{R}\, z \bullet x \mapsto z \} \\
\hline
\end{array}
$$

1.8.3 Numeric

succ	[the next natural number, page 109]
..	[set of integers within a range, page 109]
#	[number of members of a set, page 111]
min	[smallest number in a set of numbers, page 113]
max	[largest number in a set of numbers, page 113]

$$
\begin{array}{|l}
succ : \mathbb{N} \to \mathbb{N} \\
\_ \mathinner{\ldotp\ldotp} \_ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{P}\, \mathbb{Z} \\
\hline \forall n : \mathbb{N} \bullet succ(n) = n + 1 \\
forall a, b : \mathbb{Z} \bullet \\
\qquad a \mathinner{\ldotp\ldotp} b = \{\, k : \mathbb{Z} \mid a \leq k \leq b \}
\end{array}
$$

$$
\begin{array}{|l}
\hline [X] \\
\# : \mathbb{F}\, X \to \mathbb{N} \\
\hline \forall S : \mathbb{F}\, X \bullet \\
\quad \#S = (\mu\, n : \mathbb{N} \mid (\exists f : 1 \mathinner{\ldotp\ldotp} n \rightarrowtail X \bullet \mathrm{ran}\, f = S)) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\min : \mathbb{P}_1 \mathbb{Z} \rightarrow\!\!\!\!\!+ \mathbb{Z} \\
\max : \mathbb{P}_1 \mathbb{Z} \rightarrow\!\!\!\!\!+ \mathbb{Z} \\
\hline
\min = \{ S : \mathbb{P}_1 \mathbb{Z}; m : \mathbb{Z} \mid \\
\qquad\qquad m \in S \land (\forall n : S \bullet m \leq n) \bullet S \mapsto m \} \\
\max = \{ S : \mathbb{P}_1 \mathbb{Z}; m : \mathbb{Z} \mid \\
\qquad\qquad m \in S \land (\forall n : S \bullet m \geq n) \bullet S \mapsto m \}
\end{array}
$$

1.8.4 Sequences

$\frown$	[concatenation of two sequences, page 116]
rev	[reverse a sequence, page 116]
head	[first element of a sequence, page 117]
last	[last element of a sequence, page 117]
tail	[all elements of a sequence except for the first, page 117]
front	[all elements of a sequence except for the last, page 117]
$\upharpoonleft$	[sub seq based on provided indices, order maintained, page 118]
$\upharpoonright$	[sub seq based on provided condition, order maintained, page 118]
squash	[compacts a fn of positive integers into a sequence, page 118]
$\frown/$	[flatten seq of seqs into single seq, page 121]
disjoint	[pairs of sets in family have empty intersection, page 122]
partition	[union of all pairs of sets = the family set, page 122]

$$
\begin{array}{|l}
[X] \\
\hline
\_ \frown \_ : \operatorname{seq} X \times \operatorname{seq} X \rightarrow \operatorname{seq} X \\
rev : \operatorname{seq} X \rightarrow \operatorname{seq} X \\
\hline
\forall s, t : \operatorname{seq} X \bullet \\
\qquad s \frown t = s \cup \{ n : \operatorname{dom} t \bullet n + \#s \mapsto t(n) \} \\
\forall s : \operatorname{seq} X \bullet \\
\qquad rev s = (\lambda\, n : \operatorname{dom} s \bullet s(\#s - n + 1)) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
[X] \\
\hline
head, last : \operatorname{seq}_1 X \rightarrow X \\
tail, front : \operatorname{seq}_1 X \rightarrow \operatorname{seq} X \\
\hline
\forall s : \operatorname{seq}_1 X \bullet \\
\qquad head\, s = s(1) \land \\
\qquad last\, s = s(\#s) \land \\
\qquad tail\, s = (\lambda\, n : 1 \mathinner{..} \#s - 1 \bullet s(n + 1)) \land \\
\qquad front\, s = (1 \mathinner{..} \#s - 1) \lhd s \\
\hline
\end{array}
$$

$$
\begin{array}{l}
[X] \\
\hline
\_ \upharpoonleft \_ : \mathbb{P}\,\mathbb{N}_1 \times \mathrm{seq}\,X \rightarrow \mathrm{seq}\,X \\
\_ \upharpoonright \_ : \mathrm{seq}\,X \times \mathbb{P}\,X \rightarrow \mathrm{seq}\,X \\
squash : (\mathbb{N}_1 \mathrel{+\!\!\!\rightarrow} X) \rightarrow \mathrm{seq}\,X \\
\hline
\forall U : \mathbb{P}\,\mathbb{N}_1; s : \mathrm{seq}\,X \bullet \\
\qquad U \upharpoonleft s = squash(U \lhd s) \\
\forall s : \mathrm{seq}\,X; V : \mathbb{P}\,X \bullet \\
\qquad s \upharpoonright V = squash(s \rhd V) \\
\forall f : \mathbb{N}_1 \mathrel{+\!\!\!\rightarrow} X \bullet \\
\qquad squash f = f \circ (\mu\, p : 1 \mathbin{..} \# f \rightarrowtail\!\!\!\!\rightarrow \mathrm{dom}\, f \mid p \circ succ \circ p^{\sim} \subseteq (\_ < \_))
\end{array}
$$

$$
\begin{array}{l}
[X] \\
\hline
{\frown}/ : \mathrm{seq}(\mathrm{seq}\,X) \rightarrow \mathrm{seq}\,X \\
\hline
{\frown}/\langle\rangle = \langle\rangle \\
\forall s : \mathrm{seq}\,X \bullet {\frown}/\langle s \rangle = s \\
\forall q, r : \mathrm{seq}(\mathrm{seq}\,X) \bullet \\
\qquad {\frown}/(q \frown r) = ({\frown}/\,q) \frown ({\frown}/\,r)
\end{array}
$$

$$
\begin{array}{l}
[I, X] \\
\hline
\mathsf{disjoint}\ \_ : \mathbb{P}(I \mathrel{+\!\!\!\rightarrow} \mathbb{P}\,X) \\
\_\ \mathsf{partition}\ \_ : (I \mathrel{+\!\!\!\rightarrow} \mathbb{P}\,X) \leftrightarrow \mathbb{P}\,X \\
\hline
\forall S : I \mathrel{+\!\!\!\rightarrow} \mathbb{P}\,X; T : \mathbb{P}\,X \bullet \\
\qquad (\mathsf{disjoint}\ S \iff \\
\qquad\qquad (\forall i, j : \mathrm{dom}\,S \mid i \neq j \bullet S(i) \cap S(j) = \emptyset)) \land \\
\qquad (S\ \mathsf{partition}\ T \iff \\
\qquad\qquad \mathsf{disjoint}\ S \land \bigcup\{\, i : \mathrm{dom}\,S \bullet S(i)\} = T)
\end{array}
$$

1.8.5 Bags

count, #	[the number of times something appears in a bag, page 124]
$\otimes$	[scaling across a bag, page 124]
$\uplus$	[union of two bags, sum of occurances, page 126]
$\cup\!\!\!\!-$	[bag difference, subtract occurances or zero if negative, page 126]
items	[converstion from seq to bag, page 127]

$$
\begin{array}{l}
[X] \\
\hline
count : \mathrm{bag}\, X \rightarrowtail\!\!\!\!\rightarrow (X \rightarrow \mathbb{N}) \\
\_ \# \_ : \mathrm{bag}\, X \times X \rightarrow \mathbb{N} \\
\_ \otimes \_ : \mathbb{N} \times \mathrm{bag}\, X \rightarrow \mathrm{bag}\, X \\
\hline
\forall B : \mathrm{bag}\, X \bullet \\
\qquad countB = (\lambda x : X \bullet 0) \oplus B \\
\forall x : X; B : \mathrm{bag}\, x \bullet \\
\qquad B \# x = count\, B\, x \\
\forall n : \mathbb{N}; B : \mathrm{bag}\, X; x : X \bullet \\
\qquad (n \otimes B) \# x = n * (B \# x)
\end{array}
$$

$$
\begin{array}{l}
[X] \\
\hline
\_ \uplus \_, \_ \cup\!\!\!\!- \_ : \mathrm{bag}\, X \times \mathrm{bag}\, X \rightarrow \mathrm{bag}\, X \\
\hline
\forall B, C : \mathrm{bag}\, X; x : X \bullet \\
\qquad (B \uplus C) \# x = B \# x + C \# x \land \\
\qquad (B \cup\!\!\!\!- C) \# x = \max\{B \# x - C \# x, 0\}
\end{array}
$$

$$
\begin{array}{l}
[X] \\
\hline
items : \mathrm{seq}\, X \rightarrow \mathrm{bag}\, X \\
\hline
\forall s : \mathrm{seq}\, X; x : X \bullet \\
\qquad (items\, s) \# x = \#\{ i : \mathrm{dom}\, s \mid s(i) = x\}
\end{array}
$$

2 xAPI Formal Specification

The current formal specification only defines xAPI statements abstractly within the context of Z. A concrete definition for xAPI statements is outside the scope of this document.

2.1 Basic and Free Types

$[MBOX, MBOX\_SHA1SUM, OPENID, ACCOUNT]$

- Basic Types for the abstract representation of the different forms of Inverse Functional Identifiers found in xAPI

$[CHOICES, SCALE, SOURCE, TARGET, STEPS]$

- Basic Types for the abstract representation of the different forms of Interaction Components found in xAPI

$IFI ::= MBOX \mid MBOX\_SHA1SUM \mid OPENID \mid ACCOUNT$

- Free Type unique to Agents and Groups, The concrete definition of the listed Basic Types is outside the scope of this specification

$OBJECTTYPE ::= Agent \mid Group \mid SubStatement \mid StatementRef \mid Activity$

- A type which can be present in all activities as defined by the xAPI specification

$INTERACTIONTYPE ::= true{-}false \mid choice \mid fill{-}in \mid long{-}fill{-}in \mid matching \mid performance \mid sequencing \mid likert \mid numeric \mid other$

- A type which represents the possible interactionTypes as defined within the xAPI specification

$INTERACTIONCOMPONENT ::= CHOICES \mid SCALE \mid SOURCE \mid TARGET \mid STEPS$

- A type which represents the possible interaction components as defined within the xAPI specification
- the concrete definition of the listed Basic Types is outside the scope of this specification

$CONTEXTTYPES ::= parent \mid grouping \mid category \mid other$

- A type which represents the possible context types as defined within the xAPI specification

$[STATEMENT]$

- Basic type for an xAPI data point

$[AGENT, GROUP]$

- Basic types for Agents and collections of Agents

2.2 Id Schema

$$\begin{array}{|l} \textit{Id} \\ \hline id : \mathbb{F}_1\ \#1 \\ \hline \end{array}$$

- the schema Id introduces the component id which is a non-empty, finite set of 1 value

2.3 Schemas for Agents, Groups and Actors

$$\begin{array}{|l} \textit{Agent} \\ \hline agent : AGENT \\ objectType : OBJECTTYPE \\ name : \mathbb{F}_1\ \#1 \\ ifi : IFI \\ \hline objectType = Agent \\ agent = \{ifi\} \cup \mathbb{P}\{name, objectType\} \\ \hline \end{array}$$

- The schema $Agent$ introduces the component $agent$ which is a set consisting of an ifi and optionally an $objectType$ and/or $name$

$$\begin{array}{|l} \textit{Member} \\ \hline Agent \\ member : \mathbb{F}_1 \\ \hline member = \{a : AGENT \mid \forall a_n : a_i..a_j \bullet i \leq n \leq j \bullet a = agent\} \\ \hline \end{array}$$

- The schema $Member$ introduces the component $member$ which is a set of objects a, where for every a within $a_0..a_n$, a is an $agent$

$$\begin{array}{|l} \textit{Group} \\ \hline Member \\ group : GROUP \\ objectType : OBJECTTYPE \\ ifi : IFI \\ name : \mathbb{F}_1\ \#1 \\ \hline objectType = Group \\ group = \{objectType, name, member\} \vee \{objectType, member\} \vee \\ \qquad\qquad \{objectType, ifi\} \cup \mathbb{P}\{name, member\} \\ \hline \end{array}$$

- The schema $Group$ introduces the component $group$ which is of type $GROUP$ and is a set of either $objectType$ and $member$ with optionaly $name$ or $objectType$ and ifi with optionally $name$ and/or $member$

$$
\begin{array}{|l}
\hline
\textit{Actor} \\
\textit{Agent} \\
\textit{Group} \\
actor : AGENT \lor GROUP \\
\hline
actor = agent \lor group \\
\hline
\end{array}
$$

- The schema *Actor* introduces the component *actor* which is either an *agent* or *group*

2.4 Verb Schema

$$
\begin{array}{|l}
\hline
Verb \\
Id \\
display, verb : \mathbb{F}_1 \\
\hline
verb = \{id, display\} \lor \{id\} \\
\hline
\end{array}
$$

- The schema $Verb$ introduces the component *verb* which is a set that consists of either *id* and the non-empty, finite set *display* or just *id*

2.5 Object Schema

$$
\begin{array}{|l}
\hline
Extensions \\
extensions, extensionVal : \mathbb{F}_1 \\
extensionId : \mathbb{F}_1 \#1 \\
\hline
extensions = \{e : (extensionId, extensionVal)\ |\forall e_n : e_i..e_j \bullet i \leq n \leq j \bullet \\
\qquad (extensionId_i, extensionVal_i) \lor (extensionId_i, extensionVal_j) \land \\
\qquad (extensionId_j, extensionVal_i) \lor (extensionId_j, extensionVal_j) \land \\
\qquad extensionId_i \neq extensionId_j\} \\
\hline
\end{array}
$$

- The schema *Extensions* introduces the component *extensions* which is a non-empty, finite set that consists of ordered pairs of $extensionId$ and $extensionVal$. Different $extensionId$s can have the same $extensionVal$ but there can not be two identical $extensionId$ values
- $extensionId$ is a non-empty, finite set with one value
- $extensionVal$ is a non-empty, finite set

$$
\begin{array}{|l}
\hline
InteractionActivity \\
interactionType : INTERACTIONTYPE \\
correctResponsePattern : \mathrm{seq}_1 \\
interactionComponent : INTERACTIONCOMPONENT \\
\hline
interactionActivity = \{interactionType, correctReponsePattern, interactionComponent\} \lor \\
\qquad \{interactionType, correctResponsePattern\} \\
\hline
\end{array}
$$

- The schema $InteractionActivity$ introduces the component $interactionActivity$ which is a set of either $interactionType$ and $correctResponsePattern$ or $interactionType$ and $correctResponsePattern$ and $interactionComponent$

$$
\begin{array}{|l}
\hline
\textit{Definition} \\
InteractionActivity \\
Extensions \\
definition, name, description : \mathbb{F}_1 \\
type, moreInfo : \mathbb{F}_1 \#1 \\
\hline
definition = \mathbb{P}_1\{name, description, type, moreInfo, extensions, interactionActivity\} \\
\hline
\end{array}
$$

- The schema $Definition$ introduces the component $definition$ which is the non-empty, finite power set of $name$, $description$, $type$, $moreInfo$ and $extensions$

$$
\begin{array}{|l}
\hline
\textit{Object} \\
Id \\
Definition \\
Agent \\
Group \\
Statement \\
objectTypeA, objectTypeS, objectTypeSub, objectType : OBJECTTYPE \\
substatement : STATEMENT \\
object : \mathbb{F}_1 \\
\hline
substatement = statement \\
objectTypeA = Activity \\
objectTypeS = StatementRef \\
objectTypeSub = SubStatement \\
objectType = objectTypeA \lor objectTypeS \\
object = \{id\} \lor \{id, objectType\} \lor \{id, objectTypeA, definition\} \\
\qquad \lor \{id, definition\} \lor \{agent\} \lor \{group\} \lor \{objectTypeSub, substatement\} \\
\qquad \lor \{id, objectTypeA\} \\
\hline
\end{array}
$$

- The schema $Object$ introduces the component $object$ which is a non-empty, finite set of either id, id and $objectType$, id and $objectTypeA$, id and $objectTypeA$ and $definition$, $agent$, $group$, or $substatement$
- The schema $Statement$ and the corresponding component $statement$ will be defined later on in this specification

2.6 Result Schema

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Score}} \\
\hline
score : \mathbb{F}_1 \\
scaled, min, max, raw : \mathbb{Z} \\
\hline
scaled = \{n : \mathbb{Z} \mid -1.0 \leq n \leq 1.0\} \\
min = n < max \\
max = n > min \\
raw = \{n : \mathbb{Z} \mid min \leq n \leq max\} \\
score = \mathbb{P}_1\{scaled, raw, min, max\} \\
\hline
\end{array}
$$

- The schema *Score* introduces the component *score* which is the non-empty powerset of *min*, *max*, *raw* and *scaled*

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Result}} \\
\hline
Score \\
Extensions \\
success, completion, response, duration : \mathbb{F}_1 \#1 \\
result : \mathbb{F}_1 \\
\hline
success = \{true\} \vee \{false\} \\
completion = \{true\} \vee \{false\} \\
result = \mathbb{P}_1\{score, success, completion, response, duration, extensions\} \\
\hline
\end{array}
$$

- The schema *Result* introduces the component *result* which is the non-empty power set of *score*, *success*, *completion*, *response*, *duration* and *extensions*

2.7 Context Schema

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Instructor}} \\
\hline
Agent \\
Group \\
instructor : AGENT \vee GROUP \\
\hline
instructor = agent \vee group \\
\hline
\end{array}
$$

- The schema *Instructor* introduces the component *instructor* which can be ether an *agent* or a *group*

$$
\begin{array}{|l}
\hline \textit{Team} \\
\textit{Group} \\
team : GROUP \\
\hline
team = group \\
\hline
\end{array}
$$

- The schema *Team* introduces the component *team* which is a *group*

$$
\begin{array}{|l}
\hline \textit{Context} \\
Instructor \\
Team \\
Object \\
Extensions \\
registration, revision, platform, language : \mathbb{F}_1 \# 1 \\
parentT, groupingT, categoryT, otherT : CONTEXTTYPES \\
contextActivities, statement : \mathbb{F}_1 \\
\hline
statement = object \setminus (id, objectType, agent, group, definition) \\
parentT = parent \\
groupingT = grouping \\
categoryT = category \\
otherT = other \\
contextActivity = \{ca : object \setminus (agent, group, objectType, objectTypeSub, substatement)\} \\
contextActivityParent = (parentT, contextActivity) \\
contextActivityCategory = (categoryT, contextActivity) \\
contextActivityGrouping = (groupingT, contextActivity) \\
contextActivityOther = (otherT, contextActivity) \\
contextActivities = \mathbb{P}_1\{contextActivityParent, contextActivityCategory, \\
\qquad contextActivityGrouping, contextActivityOther\} \\
context = \mathbb{P}_1\{registration, instructor, team, contextActivities, revision, \\
\qquad platform, language, statement, extensions\} \\
\hline
\end{array}
$$

- The schema *Context* introduces the component *context* which is the non-empty powerset of *registration*, *instructor*, *team*, *contextActivities*, *revision*, *platform*, *language*, *statement* and *extensions*
- The notation $object \setminus agent$ represents the component *object* except for its subcomponent *agent*

2.8 Timestamp and Stored Schema

$$
\begin{array}{|l}
\hline \textit{Timestamp} \\
timestamp : \mathbb{F}_1 \# 1 \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline \textit{Stored} \\
stored : \mathbb{F}_1 \# 1 \\
\hline
\end{array}
$$

- The schema $Timestamp$ and $stored$ introduce the components $timestamp$ and $stored$ respectively. Each are non-empty, finite sets containing one value

2.9 Attachements Schema

$$
\begin{array}{|l}
\hline
\textit{Attachments} \\
display, description, attachment, attachments : \mathbb{F}_1 \\
usageType, sha2, fileUrl, contexntType : \mathbb{F}_1 \#1 \\
length : \mathbb{N} \\
\hline
attachment = \{usageType, display, contentType, length, sha2\} \cup \mathbb{P}\{description, fileUrl\} \\
attachments = \{a : attachment\} \\
\hline
\end{array}
$$

- The schema $Attachements$ introduces the componenet $attachements$ which is a non-empty, finite set of the component $attachement$
- The component $attachment$ is a non-empty, finite set of the componenets $usageType$, $display$, $contentType$, $length$, $sha2$ with optionally $description$ and/or $fileUrl$

2.10 Statement and Statements Schema

$$
\begin{array}{|l}
\hline
\textit{Statement} \\
Id \\
Actor \\
Verb \\
Object \\
Result \\
Context \\
Timestamp \\
Stored \\
Attachements \\
statement : STATEMENT \\
\hline
statement = \{actor, verb, object, stored\} \cup \\
\qquad\qquad \mathbb{P}\{\text{id}, result, context, timestamp, attachments\} \\
\hline
\end{array}
$$

- The schema $Statement$ introduces the component $statement$ which consists of the components $actor$, $verb$, $object$ and $stored$ and the optional components id, $result$, $context$, $timestamp$, and/or $attachments$
- The schema $Statement$ allows for subcomponent of $statement$ to refrenced via the . (selection) operator

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;Statements\;}} \\
Statement \\
IsoToUnix \\
statements : \mathbb{F}_1 \\
\hline
statements = \{s : statement \mid \forall s_n : s_i..s_j \bullet i \leq n \leq j \\
\qquad\qquad\qquad \bullet\ convert(s_i.timestamp) \leq convert(s_j.timestamp)\} \\
\hline
\end{array}
$$

- The schema *Statements* introduces the component *statements* which is a non-empty, finite set of the component *statement* which are in chronological order.

3 Operations, Primitives and Algorithms

The following sections introduce, define and explain Operations, Primitives and Algorithms generally using the Terminology presented below. Operations are the building blocks of Primitives whereas Primitives are the building blocks of Algorithms. The definitions which follow are flexible enough to support implementation across programing languages but have been inspired by the core concepts found within Lisp and Z. The focus of these sections is to define the properties of and interactions between Operations, Primitives and Algorithms in a general way which doesn't place unnecessary bounds on their range of possible functionality with respect to processing xAPI data.

3.1 Terminology

Within this document, (s) indicates one or more. When talking about some $x \in X$ at some index within a range $i \mathinner{..} n \mathinner{..} j$, the notation $i_X \vee n_X, \vee j_X$ may be used in cases where it is a more concise version of an equivalant expression.

3.1.1 Scalar

When working with xAPI data, Statements are written using JavaScript Object Notation (JSON). This data model supports a few fundamental types as described by JSON Schema. In order to speak about a singular valid JSON value (string, number, boolean, null) generically, the term Scalar is used. To talk about a scalar within a Z Schema, the following free and basic types are introduced.

$[STRING, NULL]$
$Boolean ::= true \mid false$
$Scalar ::= Boolean \mid STRING \mid NULL \mid \mathbb{Z}$

Arrays and Objects are also valid JSON values but will be referenced using the terms Collection and Map $\vee$ KV respectively.

3.1.2 Collection

a sequence $\langle ... \rangle$ of items c such that each $c : \mathbb{N} \times V \Rightarrow (\mathbb{N}, V) \Rightarrow \mathbb{N} \mapsto V$

$C : Collection$

$C = \langle c_i .. c_n .. c_j \rangle \Rightarrow \{ i \mapsto c_i, n \mapsto c_n, j \mapsto c_j \} \bullet i \leq n \leq j \Rightarrow i \prec n \prec j \iff i \neq n \neq j$

And the following free type is introducted for collections

$Collection ::= emptyColl \mid append \langle\langle Collection \times Scalar \vee Collection \vee KV \times \mathbb{N} \rangle\rangle$

$emptyColl$ [the empty Collection $\langle \rangle$]

$append$ [is a constructor and is infered to be an injection]

KV [a free type introduced bellow]

$append(emptyColl, c?, 0) = \langle c_0 \rangle \Rightarrow \{ 0 \mapsto c? \}$ [$append$ adds $c?$ to $\langle \rangle$ at $\mathbb{N}$]

3.1.3 Key

An identifier k paired with some value v to create an ordered pair (k, v). k can take on any valid JSON value (Scalar, Collection, KV) except for the Scalar null. The following free type is introduced for keys.

$$K ::= (Scalar \setminus NULL) \,|\, Collection \,|\, KV$$

3.1.4 Value

A value v is paired with an identifier k to create an ordered pair (k, v). v can be any valid JSON value (Scalar, Collection, KV) The following free type is introduced for values.

$$V ::= Scalar \,|\, Collection \,|\, KV$$

3.1.5 Map

Within the Z Notation Introduction section, Maps are introduced using the free type KV.

$$KV ::= base \,|\, associate \langle\!\langle\, KV \times X \times Y \,\rangle\!\rangle$$

This definition is more accurately

$$KV ::= base \,|\, associate \langle\!\langle\, KV \times K \times V \,\rangle\!\rangle$$

which indicates the usage of Key k and Value v within *associate*. Using this updated definition,

$$associate(base, k, v) = \langle\!\langle (k, v) \rangle\!\rangle$$

such that a Map is a Collection of ordered pairs (k_n, v_n) and thus a Collection of mappings

$$(k_n, v_n) \Rightarrow k_n \mapsto v_n$$

but Maps are special cases of Collections as k_n is the unique identifier of v_n within a Map but the opposite is not true. In fact, keys are their own identifiers

$$\mathrm{id}\, v_n = k_n$$
$$\mathrm{id}\, k_n \neq v_n$$
$$\mathrm{id}\, k_n = k_n$$

Given a Map $M = \langle\!\langle (k_i, v_i) \mathinner{..} (k_n, v_n) \mathinner{..} (k_j, v_j) \rangle\!\rangle$ the following demonstrates the uniqueness of Keys but the same is not true for all v within M

$$i_k \neq n_k \neq j_k$$
$$i_v = n_v \vee i_v \neq n_v \;\; i_v = j_v \vee i_v \neq j_v \;\; j_v = n_v \vee j_v \neq n_v$$

which can all be stated formally as

$$
\begin{array}{l}
[K, V] \\
\hline
Map : K \times V \rightarrowtail\!\!\!\!\rightarrow KV \\
\hline
Map = \langle\!\langle (k_i, v_i) \mathrel{..} (k_n, v_n) \mathrel{..} (k_j, v_j) \rangle\!\rangle \bullet \\
\quad \operatorname{dom} Map = \{ k_i \mathrel{..} k_n \mathrel{..} k_j \} \\
\quad \operatorname{ran} Map = \{ v_i \mathrel{..} v_n \mathrel{..} v_j \} \\
\quad first(k_i, v_i) \neq first(k_n, v_n) \neq first(k_j, v_j) \land \\
\quad i_v = n_v \lor i_v \neq n_v \; i_v = j_v \lor i_v \neq j_v \; j_v = n_v \lor j_v \neq n_v \land \\
\quad \mathrm{id}\ v_i = k_i \land \mathrm{id}\ v_n = k_n \land \mathrm{id}\ v_j = k_j \land \\
\quad \mathrm{id}\ k_i = k_i \land \mathrm{id}\ k_n = k_n \land \mathrm{id}\ k_j = k_j
\end{array}
$$

Given that v can be a Map M, or a Collection C, Arbitrary nesting is allowed within Maps but the properties of a Map hold at any depth.

$$M = \langle\!\langle (k_i, v_i) \mathrel{..} (k_n, \langle\!\langle (k_{ni}, v_{ni}) \rangle\!\rangle) \mathrel{..} (k_j, \langle v_{ji} \mathrel{..} \langle\!\langle (k_{jn}, v_{jn}) \rangle\!\rangle \mathrel{..} \langle v_{jji} \mathrel{..} v_{jjn} \mathrel{..} v_{jjj} \rangle \rangle) \rangle\!\rangle$$

such that $\langle\!\langle (k_{ni}, v_{ni}) \rangle\!\rangle$ and $\langle\!\langle (k_{nj}, v_{nj}) \rangle\!\rangle$ are both Maps and adhere to the constraints enumerated above.

3.1.6 Statement

Immutable Map conforming to the xAPI Specification as described in the xAPI Formal Definition section of this document. The imutability of a Statement s is demonstrated by the following which indicates that s was not altered when passed to *associate*.

$$
\begin{array}{l}
s!, s? : STATEMENT \\
k? : K \\
v? : V \\
\hline
s! = associate(s?, k?, v?) = s? \Rightarrow (k?, v?) \notin s! \Rightarrow s! = s?
\end{array}
$$

Additionally, given the schema *Statements* the following is true for all *Statement*(s)

$$
\begin{array}{l}
Statements \\
Keys : STRING \\
S : Collection \\
\hline
Keys = \{ id, actor, verb, object, result, context, attachments, timestamp, stored \} \\
\operatorname{dom} statement = K \lhd Keys \\
S = \langle statement_i \mathrel{..} statement_n \mathrel{..} statement_j \rangle \bullet \\
\quad atKey(statement_i, id) \neq atKey(statement_n, id) \neq atKey(statement_j, id) \Rightarrow \\
\quad id_i \neq id_n \neq id_j \iff statement_i \neq statement_n \neq statement_j
\end{array}
$$

Which confirms the constraints found in the schema *Statement* and adds an additional constraint to *Statements* such that every unique *Statement* in a *Collection* of *Statements* has a unique *id*.

3.1.7 Algorithm State

Mutable Map *state* without any domain restriction such that

$$
\begin{array}{|l}
state?\,, state! : KV \\
k? : K \\
v? : V \\
\hline
associate(state?\,, k?\,, v?\,) = state! \bullet (k, v) \in state! \Rightarrow state? \neq state!
\end{array}
$$

3.1.8 Option

Mutable Map *opt* which is used to alter the result of an Algorithm. The effect of *opt* on an Algorithm will be discussed in the Algorithm Result section bellow.

4 Operation

An Operation is a function of arbitrary arguments and is defined using Z. For example, Operations pulled directly from "The Z Notation: A Reference Manual" include

- $first$
- $second$
- $succ$
- $\min$
- $\max$
- $count \equiv \#$
- $\frown$
- rev
- $head$
- $last$
- $tail$
- $front$
- $\upharpoonleft$
- $\upharpoonright$
- $\frown/$
- $\mathsf{disjoint}$
- $\mathsf{partition}$
- $\otimes$
- $\uplus$
- $\uminus$
- $items$

4.1 Domain

The arguments passed to an Operation can be any of the following but the definition of an Operation may limit the domain to a subset of the following

- Key(s)
- Value(s)
- Set(s)
- Collection(s)
- Bag(s)
- KV(s)
- Statement(s)
- Algorithm State

4.2 Range

The result of an Operation can be any of the following but the definition of an Operation may limit this range to a subset of the following

- Key(s)
- Value(s)
- Set(s)
- Collection(s)
- Bag(s)
- KV(s)
- Statement(s)
- Algorithm State

5 Primitive

Primitives break the processing of xAPI data down into discrete units that can be composed to create new analytical functions. Primitives allow users to address the methodology of answering research questions as a sequence of generic algorithmic steps which establish the necessary data transformations, aggregations and calculations required to reach the solution in an implementation agnostic way.

Within this document, they will be defined as a Collection of Operations and/or Primitives where the output is piped from member to member. In this section, o_n and p_n can be used as to describe Primitive members but for simplicity, only o_n will be used.

$$p_{\langle i\,..\,n\,..\,j\rangle} = o_i \gg o_n \gg o_j$$

Within any given Primitive p, variables local to p and any global variables may be passed as arguments to any member of p and there is no restriction on the ordering of arguments with respect to the piping. In the following, $q?$ is a global variable where as the rest are local.

$$\begin{array}{|l}
x?, y?, z?, i!, n!, j!, p! : Value \\
o_i : Value \nrightarrow Value \\
o_n : Value \times Value \nrightarrow Value \\
o_j, p : Value \times Value \times Value \nrightarrow Value \\
\hline
i! = o_i(x?) \\
n! = o_n(i!, y?) \\
j! = o_j(z?, n!, q?) \\
p! = j! \Rightarrow o_j(z?, o_n(o_i(x?), y?), q?)
\end{array}$$

In the rest of this document, the following notation is used to distinguish between the functionality of a Primitive and its composition. This notation should be used when defining Primitives.

$$\begin{array}{|l}
primitiveName\,_ : _ \nrightarrow _ \\
\hline
primitiveName = \langle primitiveName_i\,_\,..\,primitiveName_n\,_\,..\,primitiveName_j\,_\rangle
\end{array}$$

- The top line indicates the Primitive
  - should be written using postfix notation within other schemas
  - is atleast a partial function from some input to some output
- The bottom line is an enumeration of the composing Operations and/or Primitives and their order of execution

This means the definition of p from above can be updated as follows.

$$
\begin{array}{|l}
p_{\_} : Value \times Value \times Value \nrightarrow Value \\
\hline
p = \langle\, o_i,\, o_n,\, o_j \rangle \\
p(x?\,, y?\,, z?\,) = o_j(z?\,, o_n(o_i(x?\,), y?\,), q?\,)
\end{array}
$$

Additionally, this notation supports declaration of recursive iteration via the presence of $recur_{\_}$ within a Primitive chain

$$
\begin{array}{|l}
primitiveName_i = \langle\langle primitiveName_{ii\,\_}, primitiveName_{in\,\_}\rangle,\ recur_{\_}\rangle^{\#\_} \\
\hline
\langle\langle primitiveName_{ii\,\_}, primitiveName_{in\,\_}\rangle,\ recur_{\_}\rangle^{\#\_} \Rightarrow \\
\qquad (primtiveName_{ii} \gg primitiveName_{in})^{\#\_} \bullet \\
\qquad\quad \forall\, n : i \,..\, j \bullet j = \#_{\_} \wedge i \le n \le j \mid \exists_1 p_n : \_ \nrightarrow \_\_ \nrightarrow \_ \bullet \\
\qquad\qquad let \quad p_i == primtiveName_{ii} \gg primitiveName_{in} \Rightarrow \\
\qquad\qquad\qquad p_{i\,\_} = primitiveName_{in}(primitiveName_{ii\,\_}) \\
\qquad\qquad\quad p_n == p_i \gg primtiveName_{ii} \gg primitiveName_{in} \Rightarrow \\
\qquad\qquad\qquad p_{n\,\_} = primitiveName_{in}(primitiveName_{ii}(p_{i\,\_})) \\
\qquad\qquad\quad p_j == p_n \gg primtiveName_{ii} \gg primitiveName_{in} \Rightarrow \\
\qquad\qquad\qquad p_{j\,\_} = primitiveName_{in}(primitiveName_{ii}(p_{n\,\_})) \\
\qquad\quad p_j = (primtiveName_{ii} \gg primitiveName_{in})^{\#\_} \bullet j = 3 \Rightarrow \\
\qquad\qquad\quad (primtiveName_{ii} \gg primitiveName_{in}) \gg \\
\qquad\qquad\quad (primtiveName_{ii} \gg primitiveName_{in}) \gg \\
\qquad\qquad\quad (primtiveName_{ii} \gg primitiveName_{in}) \Rightarrow \\
\qquad\qquad\qquad primitiveName_{in}(\\
\qquad\qquad\qquad\quad primitiveName_{ii}(\\
\qquad\qquad\qquad\qquad primitiveName_{in}(\\
\qquad\qquad\qquad\qquad\quad primitiveName_{ii}(p_{i\,\_}))))
\end{array}
$$

Here, p_i was chosen to only be two primitives $primtiveName_{ii} \wedge primitiveName_{in}$ for simplicity sake. The Primitive chain can be of arbitrary length. The number of iterations is described using the count operation $\#_{\_}$. Above $j = 3$ was used to demonstrate the piping between iterations but j is not exclusively $= 3$. Given above, the term Primitive Chain can be defined as:

$$
\begin{array}{l}
(primtiveName_i \gg primitiveName_n \gg primitiveName_j)^{\#\_} \bullet \\
\qquad \#_{\_} = 0 \Rightarrow primtiveName_i \gg primitiveName_n \gg primitiveName_j
\end{array}
$$

where a Primitive chain iterated to the 0 is just the chain itself hence recursion is not a requirement of, but is supported within, the definition of Primitives.

5.1 Domain

Any of the following dependent upon the Operations which compose the Primitive

- Key(s)
- Value(s)

- Set(s)
- Collection(s)
- Bag(s)
- KV(s)
- Statement(s)
- Algorithm State

5.2 Range

Any of the following dependent upon the Domain and Functionality of the Primitive

- Key(s)
- Value(s)
- Set(s)
- Collection(s)
- Bag(s)
- KV(s)
- Statement(s)
- Algorithm State

6 Algorithm

Given a Collection of statement(s) $S_{\langle a..b..c\rangle}$ and potentially option(s) *opt* and potentially an existing Algorithm State *state* an Algorithm A executes as follows

1. call *init*
2. for each $stmt \in S_{\langle a..b..c\rangle}$
   (a) *relevant*?
   (b) *accept*?
   (c) *step*
3. return *result*

with each process within A is enumerated as

```
(init [state] body)
 - init state

(relevant? [state statement] body)
 - is the statement valid for use in algorithm?

(accept? [state statement] body)
 - can the algorithm consider the current statement?

(step [state statement] body)
 - processing per statement
 - can result in a modified state

(result [state] body)
 - return without option(s) provided
 - possibly sets default option(s)

(result [state opt] body)
 - return with consideration to option(s)
```

- *body* is a collection of Primitive(s) which establishes the processing of inputs $\rightarrow$ outputs
- *state* is a mutable Map of type KV and synonymous with Algorithm State
- *statement* is a single statement within the collection of statements passed as input data to the Algorithm A
- *opt* are additional arguments passed to the algorithm A which impact the return value of the algorithm and synonymous with Option

An Algorithm must be passed an Algorithm State and a Collection of Statement(s). Option is optional.

- Statement(s)
- Algorithm State
- Option(s)

An Algorithm will return an Algorithm State.

- Algorithm State

An Algorithm can be described via its components. A formal definition for an Algorithm is presented at the end of this section. The following subsections go into more detail about the components of an Algorithm.

$$Algorithm ::= Init \fatsemi Relevant? \fatsemi Accept? \fatsemi Step \fatsemi Result$$

6.1 Initialization

First process to run within an Algorithm which returns the Algorithm State for the current iteration.

$$\begin{array}{|l}
\hline
Init[KV] \\
state?, state! : KV \\
init_ : KV \twoheadrightarrow KV \\
\hline
init = \langle body \rangle \\
state! = init(state?) \bullet state! = state? \lor state! \neq state? \\
\hline
\end{array}$$

such that some *state*! does not need to be related to its arguments *state*? but *state*! could be derived from some seed *state*?. This functionality is dependent upon the composition of *body* within *init*.

6.1.1 Domain

- Algorithm State

6.1.2 Range

- Algorithm State

6.2 Relevant?

First process that each *stmt* passes through $\Rightarrow relevant? \prec accept? \prec step$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Relevant?\,[KV, STATEMENT] \qquad\qquad\qquad\qquad}} \\
state? : KV \\
stmt? : STATEMENT \\
relevant?\ \_ : KV \times STATEMENT \rightarrow Boolean \\
\hline
relevant? = \langle body \rangle \\
relevant?\,(state?\,, stmt?\,) = true \vee false \\
\hline
\end{array}
$$

resulting in an indication of whether the *stmt* is valid within algorithm A. The criteria which determines validity of *stmt* within A is defined by the *body* of *relevant?*

6.2.1 Domain

- Statement
- Algorithm State

6.2.2 Range

- Boolean

6.3 Accept?

Second process that each *stmt* passes through $\Rightarrow relevant? \prec accept? \prec step$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Accept?\,[KV, STATEMENT] \qquad\qquad\qquad\qquad}} \\
state? : KV \\
stmt? : STATEMENT \\
accept?\ \_ : KV \times STATEMENT \rightarrow Boolean \\
\hline
accept? = \langle body \rangle \\
accept?\,(state?\,, stmt?\,) = true \vee false \\
\hline
\end{array}
$$

resulting in an indication of whether the *stmt* can be sent to *step* given the current *state*. The criteria which determines usability of *stmt* given *state* is defined by the *body* of *accept?*

6.3.1 Domain

- Statement
- Algorithm State

6.3.2 Range

- Scalar

6.4 Step

An Algorithm Step consists of a sequential composition of Primitive(s) where the output of some function is passed as an argument to the next function both within and across Primitives in *body*.

$$body = p_i \gg p_n \gg p_j \Rightarrow o_{ii} \gg o_{in} \gg o_{ij} \gg o_{ni} \gg o_{nn} \gg o_{nj} \gg o_{ji} \gg o_{jn} \gg o_{jj}$$

The selection and ordering of Operation(s) and Primitive(s) into an Algorithmic Step determines how the Algorithm State changes during iteration through Statement(s) passed as input to the Algorithm.

$$\begin{array}{|l}
P = \langle p_i \mathrel{..} p_n \mathrel{..} p_j \rangle \bullet i \leq n \leq j \Rightarrow i \prec n \prec j \iff i \neq n \neq j \bullet p_i \gg p_n \gg p_j \\
P' = \langle p_{i'} \mathrel{..} p_{n'} \mathrel{..} p_{j'} \rangle \bullet i' \leq n' \leq j' \Rightarrow i' \prec n' \prec j' \iff i' \neq n' \neq j' \bullet p_{i'} \gg p_{n'} \gg p_{j'} \\
P'' = \langle p_x \mathrel{..} p_y \mathrel{..} p_z \rangle \bullet x \leq y \leq z \Rightarrow x \prec y \prec z \iff x \neq y \neq z \bullet p_x \gg p_y \gg p_z \\
\hline
P = P' \iff i \mapsto i' \land n \mapsto n' \land j \mapsto j' \\
P = P'' \iff (i \mapsto x \land n \mapsto y \land j \mapsto z) \land (p_i \equiv p_x \land p_n \equiv p_y \land p_j \equiv p_z)
\end{array}$$

step may or may not update the input Algorithm State given the current Statement from the Collection of Statement(s).

$$\begin{array}{|l}
S : Collection \\
stmt_a, stmt_b, stmt_c : STATEMENT \\
state?\,, step_a!\,, step_b!\,, step_c! : KV \\
step_ : KV \times STATEMENT \twoheadrightarrow KV \\
\hline
S = \langle stmt_a .. stmt_b .. stmt_c \rangle \bullet a \leq b \leq c \Rightarrow a \prec b \prec c \iff a \neq b \neq c \\
step_a! = step(state?\,, stmt_a) \bullet step_a! = state? \lor step_a! \neq state? \\
step_b! = step(step_a!\,, stmt_b) \bullet step_b! = step_a! \lor step_b! \neq step_a! \\
step_c! = step(step_b!\,, stmt_c) \bullet step_c! = step_b! \lor step_c! \neq step_b!
\end{array}$$

In general, this allows *step* to be defined as

$$\begin{array}{|l}
\hline
Step[KV, STATEMENT] \\
state?\,, state! : KV \\
stmt? : STATEMENT \\
step_ : KV \times STATEMENT \twoheadrightarrow KV \\
\hline
step = \langle body \rangle \\
state! = step\,(state?\,, stmt?\,) = state? \lor state! \neq state? \\
\hline
\end{array}$$

A change of $state? \rightarrow state! \bullet state! \neq state?$ can be predicted to occur given

- The definition of individual Operations which constitute a Primitive
- The ordering of Operations within a Primitive
- The Primitive(s) chosen for inclusion within the body of *step*
- The ordering of Primitive(s) within the body of *step*
- The key value pair(s) in both Algorithm State and the current Statement
- The ordering of Statement(s)

6.4.1 Domain

- Statement
- Algorithm State

6.4.2 Range

- Algorithm State

6.5 Result

Last process to run within an Algorithm which returns the Algorithm State *state* when all $s \in S$ have been processed by *step*

$$\begin{aligned}
&relevant? \prec accept? \prec step \prec result \prec relevant? \iff S \neq \emptyset \\
&relevant? \prec accept? \prec step \prec result \iff S = \emptyset
\end{aligned}$$

and does so without preventing subsequent calls of A

$$\begin{array}{|l}
\multicolumn{1}{l}{Result[KV, KV]} \\
result!, state?, opt? : KV \\
result_ : KV \times KV \twoheadrightarrow KV \\
\hline
result = \langle body \rangle \\
result! = result(state?, opt?) = state? \lor state! \neq state? \\
\hline
\end{array}$$

such that if at some future point j within the timeline $i \mathinner{..} n \mathinner{..} j$

$$\begin{aligned}
&S(t_n) = \emptyset && \text{[S is empty at } t_n\text{]} \\
&S(t_j) \neq \emptyset && \text{[S is not empty at } t_j\text{]} \\
&S(t_{n-i}) && \text{[stmts(s) added to } S \text{ between } t_i \text{ and } t_n\text{]} \\
&S(t_{j-n}) && \text{[stmts(s) added to } S \text{ between } t_n \text{ and } t_j\text{]} \\
&S(t_{j-i}) = S(t_{n-i}) \ \cup \ S(t_{j-n}) && \text{[stmts(s) added to } S \text{ between } t_i \text{ and } t_j\text{]}
\end{aligned}$$

Algorithm A can pick up from a previous $state_n$ without losing track of its own history.

$$\begin{array}{|l}
state_{n-i} = A(state_i,\ S(t_{n-i})) \\
state_{n-1} = A(state_{n-2},\ S(t_{n-1})) \\
state_n = A(state_{n-1},\ S(t_n)) \\
state_{j-n} = A(state_n,\ S(t_{j-n})) \\
state_j = A(state_i,\ S(t_{j-i})) \\
\hline
state_n = state_{n-1} \iff S(t_n) = \emptyset \land S(t_{n-1}) \neq \emptyset \\
state_j = state_{j-n} \iff state_{n-i} = state_n = state_{n-1}
\end{array}$$

Which makes A capable of taking in some $S_{\langle i..n..j..\infty \rangle}$ as not all $s \in S_{\langle i..\infty \rangle}$ have to be considered at once. In other words, the input data does not need to persist across the history of A, only the effect of s on $state$ must be persisted. Additionally, the effect of opt is determined by the $body$ within $result$ such that

$$
\begin{aligned}
&A(state_n,\ S(t_{j-n}),\ opt) \\
&\quad \equiv A(state_i\ S(t_{j-i})) \\
&\quad \equiv A(state_i,\ S(t_{j-i}),\ opt) \\
&\quad \equiv A(state_n,\ S(t_{j-n}))
\end{aligned}
$$

implying that the effect of opt doesn't prevent backwards compatibility of $state$.

6.5.1 Domain

- Algorithm State
- Option(s)

6.5.2 Range

- Algorithm State

6.6 Algorithm Formal Definition

In previous sections, $A\_$ was used to indicate calling an Algorithm. In the rest of this document, that notation will be replaced with $algorithm\_$. This new notation is defined using the definitions of Algorithm Components presented above. The previous definition of an Algorithm

$$Algorithm ::= Init \mathbin{⨾} Relevant? \mathbin{⨾} Accept? \mathbin{⨾} Step \mathbin{⨾} Result$$

can be refined using the Operation $recur$ and Primitive $algorithmIter$ (defined in following subsections) to illustrate how an Algorithm processes a Collection of Statement(s).

$$
\begin{array}{l}
\textit{Algorithm}[KV, Collection, KV] \\
\hline
AlgorithmIter,\ Recur,\ Init,\ Result \\
opt?, state?, state! : KV \\
S? : Collection \bullet \forall s? \in S? \mid s? : STATEMENT \\
algorithm\_ : KV \times Collection \times KV \twoheadrightarrow KV \\
\hline
algorithm = \langle init\_, \langle algorithmIter\_,\ recur\_\rangle^{\# S?},\ result\_\rangle \\
state! = algorithm(state?, S?, opt?) \bullet \\
\quad let\ \ init! == init(state?) \bullet \\
\quad \forall s_n \in S? \mid s_n : STATEMENT, n : \mathbb{N} \bullet i \leq n \leq j\ \bullet \\
\quad\quad \exists_1 state_n \mid state_n : KV \bullet \\
\quad\quad\quad let\ \ S?_n = tail(S?)^{n-i} \\
\quad\quad\quad\quad state_i = algorithmIter(init!, S?_n) \Rightarrow S?_n = S? \iff n = i \\
\quad\quad\quad\quad state_n = recur(state_i, S?_n, \_algorithmIter\_)^{j-1} \iff n \neq i \wedge n \neq j \\
\quad\quad\quad\quad state_j = recur(state_n, (\{j-1, j\} \upharpoonright S?), \_algorithmIter\_) \iff n = j \\
\quad\quad\quad\quad state_{j+1} = state_j \Rightarrow recur(state_j, (j \upharpoonright S?), \_algorithmIter\_) \iff n = j+1 \\
\quad = result(state_j, opt?) \\
\hline
\end{array}
$$

Within the schema above, the following notation is intended to show that *algorithm* is a Primitive ⇒ Collection of Primitives and/or Operations.

$$\langle\, init\_, \langle\, algorithmIter\_, \, recur\_\rangle^{\#\, S?}, \, result\_\rangle$$

Within that notation, the following notation is intended to represent the iteration through the Statement(s) via tail recursion.

$$\langle\, algorithmIter\_, \, recur\_\rangle^{\#\, S?}$$

which implies that each Statement is passed to $algorithmIter\_$ and the result is then passed on to the next iteration of the loop. The completion of this loop is the prerequisites of $result\_$

6.6.1 Recur

The following schema introduces the Operation *recur* which expects an accumulator (KV), a *Collection* of Value(s) (V) being iterated over and a function ($\_ \nrightarrow \_$) which will be called as the result of *recur*. This Operation has been writen to be as general purpose as possible and represents the ability to perform tail recursion. Given this intention, *recur* must only ever be the last Operation within a Primitive

$$\begin{array}{|l}
p_{i\,..\,j} : \mathrm{seq}_1 \bullet \forall o \in p \mid o : _ \nrightarrow _ \\
\hline
p_{i\,..\,j} = \langle \forall n : \mathbb{N} \mid i \leq n \leq j \wedge o_n \in p_{i\,..\,j} \bullet \\
\qquad\qquad \exists_1 o_n \bullet o_n \neq recur \vee o_n = recur \iff n = j \rangle \Rightarrow \\
\qquad\qquad\qquad front(p_{i\,..\,j}) \upharpoonright recur = \langle\rangle
\end{array}$$

and results in a call to the passed in function where the accumulator $ack?$ and the Collection (minus the first member) are passed as arguments to $fn?$. If this would result in the empty Collection ($\langle\rangle$) being passed to $fn?$, instead the accumulator $ack?$ is returned.

$$\begin{array}{|l}
\hline
Recur[KV, Collection, (_ \nrightarrow _)] \\
ack? : KV \\
S? : Collection \\
fn? : (_ \nrightarrow _) \\
recur_ : KV \times Collection \times (_ \nrightarrow _) \leftrightarrow (KV \times Collection \nrightarrow _) \\
\hline
recur(ack?, S?, fn?) = fn?(ack?, tail(S?)) \iff tail(S?) \neq \langle\rangle \\
recur(ack?, S?, fn?) = first(ack?, tail(S?)) \iff tail(S?) = \langle\rangle \\
\hline
\end{array}$$

In the context of Algorithms,

$$ack? = AlgorithmState$$
$$S? = CollectionofStatement(s)$$
$$fn? = algorithmIter$$

6.6.2 Algorithm Iter

The following schema introduce the Primitive *algorithmIter* which demonstrates the life cycle of a single statement as its passed through the components of an Algorithm.

$$
\begin{array}{|l}
\hline
AlgorithmIter[KV, Collection] \\
Relevant?\,,\ Accept?\,,\ Step \\
state?\,, state! : KV \\
S? : Collection \\
s? : STATEMENT \\
algorithmIter\,\_ : KV \times STATEMENT \twoheadrightarrow KV \\
\hline
algorithmIter = \langle\, relevant?\ \_\,,\ accept?\ \_\,,\ step\,\_ \rangle \\
s? = head(S?\,) \\
state! = algorithmIter(state?\,, s?\,) \bullet \\
\qquad let \quad relevant! == relevant?\,(state?\,, s?\,) \\
\qquad\qquad accept! == accept?\,(state?\,, s?\,) \\
\qquad\qquad step! == step(state?\,, s?\,) \\
\qquad\ = (state? \iff relevant! = false \lor accept! = false) \lor \\
\qquad\quad (step! \iff relevant! = true \land accept! = true) \\
\hline
\end{array}
$$

If a statement if both relevant and acceptable, *state*! will be the result of *step*. Otherwise, the passed in state is returned $\Rightarrow step! = state?$.

7 Foundational Operations

The Operations in this section use the Operations pulled from the Z Reference Manual (section 1,4) within their own definitions. They are defined as Operations opposed to Primitives because they represent core functionality needed in the context of processing xAPI data given the definition of an Algorithm above. As such, these Operations are added to the global dictionary of symbols usable within the definition of Operations and Primitives throughout the rest of this document.

7.1 Collections

Operations which expect a Collection $X = \langle x_i .. x_n .. x_j \rangle$

7.1.1 Array?

The operation *array*? will return a boolean which indicates if the passed in argument is a Collection

$$
\begin{array}{|l}
\hline
Array?[V] \\
coll? : V \\
bol! : Boolean \\
array?\ \_ : V \rightarrow Boolean \\
\hline
bol! = array?(coll?) \bullet bol! = true \iff coll? : Collection \Rightarrow V \setminus (Scalar, KV) \\
\hline
\end{array}
$$

where $V \setminus (Scalar, KV)$ is used to indicate that *coll*? is of type V

$$V ::= Scalar \mid Collection \mid KV$$

but in order for $bol! = true$, *coll*? must not be of type $Scalar \lor KV$ such that

$$
\begin{aligned}
& X = \langle x_0, x_1, x_2, x_3, x_4 \rangle \\
& \quad x_0 = 0 \\
& \quad x_1 = foo \\
& \quad x_2 = \langle baz,\ qux \rangle \\
& \quad x_3 = \langle\!\langle abc \mapsto 123,\ def \mapsto 456 \rangle\!\rangle \\
& \quad x_4 = \langle \langle\!\langle ghi \mapsto 789,\ jkl \mapsto 101112 \rangle\!\rangle,\ \langle\!\langle ghi \mapsto 131415,\ jkl \mapsto 161718 \rangle\!\rangle \rangle \\
& array?(X) = true && \text{[collection by definition]} \\
& array?(x_2) = true && \text{[collection of } 0 \mapsto baz,\ 1 \mapsto qux\text{]} \\
& array?(x_4) = true && \text{[collection of maps]} \\
& array?(x_0) = false && \text{[Scalar]} \\
& array?(x_1) = false && \text{[String]} \\
& array?(x_3) = false && \text{[Map]}
\end{aligned}
$$

7.1.2 Append

The operation *append* will return a Collection with a Value added at a specified numeric Index.

$$
\begin{array}{l}
\textit{Append}[\textit{Collection}, V, \mathbb{N}] \\
\hline
\textit{coll}?\,, \textit{coll}! : \textit{Collection} \\
v? : V \\
\textit{idx}? : \mathbb{N} \\
\textit{append}\_ : \textit{Collection} \times V \times \mathbb{N} \rightarrowtail \textit{Collection} \\
\hline
\#\,\textit{idx}? = 1 \\
\textit{coll}! = \textit{append}(\textit{coll}?\,, v?\,, \textit{idx}?\,) \bullet \\
\quad \textit{let}\ \textit{coll}' == \textit{front}(\{\, i : \mathbb{N} \mid i \in 0 \mathrel{..} \textit{idx}?\,\} \upharpoonright \textit{coll}?\,) \frown v? \\
\quad\quad \textit{coll}'' == \{\, j : \mathbb{N} \mid j \in \textit{idx}? \mathrel{..} \#\,\textit{coll}?\,\} \upharpoonright \textit{coll}? \\
\quad = \textit{coll}' \frown \textit{coll}'' \Rightarrow \\
\quad\quad (\textit{front}(\textit{coll}') \frown v? \frown \textit{coll}'') \wedge \\
\quad\quad (\, v? \mapsto \textit{idx}? \in \textit{coll}!\,) \wedge \\
\quad\quad (\,\#\,\textit{coll}! = \#\,\textit{coll}? + 1)
\end{array}
$$

append results in the composition of $coll'$ and $coll''$ such that

$$\textit{coll}! = \textit{coll}' \frown \textit{coll}'' \ \wedge\ \textit{idx}? \mapsto v? \in \textit{coll}!$$

- $coll'$ is the items in $coll?$ up to and including $idx?$ but the value at $idx?$ is replaced with $v?$ such that $\textit{idx}? \mapsto \textit{coll}?_{\textit{idx}?} \notin \textit{coll}'$
- $coll''$ is the items in $coll?$ from $idx?$ to $\#\,\textit{coll}? \Rightarrow \textit{coll}?_{\textit{idx}?} \in \textit{coll}''$

The following example illustrates these properties.

$$
\begin{array}{l}
X = \langle x_0, x_1, x_2 \rangle \\
\quad x_0 = 0 \\
\quad x_1 = \textit{foo} \\
\quad x_2 = \langle a, b, c \rangle \\
\quad v? = \textit{bar} \\
\textit{append}(X, v?\,, 0) = \langle \textit{bar}, 0, \textit{foo}, \langle a, b, c \rangle \rangle \\
\textit{append}(X, v?\,, 1) = \langle 0, \textit{bar}, \textit{foo}, \langle a, b, c \rangle \rangle \\
\textit{append}(X, v?\,, 2) = \langle 0, \textit{foo}, \textit{bar}, \langle a, b, c \rangle \rangle \\
\textit{append}(X, v?\,, 3) = \langle 0, \textit{foo}, \langle a, b, c \rangle, \textit{bar} \rangle \\
\textit{append}(X, v?\,, 4) = \textit{append}(X, v?\,, 3) \iff 3 \notin \operatorname{dom} X
\end{array}
$$

7.1.3 Remove

The inverse of the *append* Operations.

$$remove(coll, idx) =^{\sim} append(coll, idx)$$

The operation *remove* will return a Collection minus the Value removed from the specified Numeric Index

$$
\begin{array}{|l}
\multicolumn{1}{l}{Remove[Collection, \mathbb{N}]} \\
\hline
coll?, coll! : Collection \\
idx? : \mathbb{N} \\
remove\_ : Collection \times \mathbb{N} \twoheadrightarrow Collection \\
\hline
\# idx? = 1 \\
coll! = remove(coll?, idx?) \bullet \\
\quad let\ coll' == front(\{ i : \mathbb{N} \mid i \in 0 \mathrel{..} idx? \} \upharpoonright coll?) \\
\quad\quad coll'' == tail(\{ j : \mathbb{N} \mid j \in idx? \mathrel{..} \# coll? \} \upharpoonright coll?) \\
\quad = coll' \frown coll'' \Rightarrow \\
\quad\quad (coll?_{idx?} \notin coll') \land \\
\quad\quad (coll?_{idx?} \notin coll'') \land \\
\quad\quad (\# coll! = \# coll? - 1) \\
\hline
\end{array}
$$

such that

$$
\begin{aligned}
&X = \langle x_0, x_1, x_2 \rangle \\
&\quad x_0 = 0 \\
&\quad x_1 = foo \\
&\quad x_2 = baz \\
&remove(X, 0) = \langle foo, baz \rangle && \text{[0 was removed from } X\text{]} \\
&remove(X, 1) = \langle 0, baz \rangle && \text{[foo was removed from } X\text{]} \\
&remove(X, 2) = \langle 0, foo \rangle && \text{[baz was removed from } X\text{]} \\
&remove(X, 3) = \langle 0, foo, baz \rangle = X && \text{[nothing at 3, X unaltered]}
\end{aligned}
$$

7.1.4 At Index

The operation *atIndex* will return the Value at a specified numeric index within a Collection or an empty Collection if there is no value at the specified index.

$$
\begin{array}{|l}
\hline
\textit{AtIndex}[\textit{Collection}, \mathbb{N}] \\
\textit{idx}? : \mathbb{N} \\
\textit{coll}? : \textit{Collection} \\
\textit{atIndex}\_ : \textit{Collection} \times \mathbb{N} \twoheadrightarrow V \\
\hline
\# \textit{idx}? = 1 \\
\textit{coll}! = \textit{atIndex}(\textit{coll}?, \textit{idx}?) = (\textit{head}\,(\textit{idx}? \upharpoonright \textit{coll}?)) \iff \textit{idx}? \in \textit{coll}? \\
\textit{coll}! = \textit{atIndex}(\textit{coll}?, \textit{idx}?) = \langle\rangle \iff \textit{idx}? \notin \textit{coll}? \\
\hline
\end{array}
$$

Given the definition of the *Collection* and V free types

$$
\begin{array}{l}
\textit{Collection} ::== \textit{emptyColl} \mid \textit{append}\langle\!\langle \textit{Collection} \times \textit{Scalar} \vee \textit{Collection} \vee \textit{KV} \times \mathbb{N} \rangle\!\rangle \\
V ::= \textit{Scalar} \mid \textit{Collection} \mid \textit{KV}
\end{array}
$$

The collection member $\textit{coll}?_{\textit{idx}?} : V$ is implied from *append* accepting the argument of type $\textit{Scalar} \vee \textit{Collection} \vee \textit{KV} \equiv V$ which means each Collection member is of type V. Given that extraction $(\_ \upharpoonright \_)$ returns a Collection,

$$
\begin{array}{|l}
\operatorname{seq} X : \textit{Collection} \\
\hline
\_ \upharpoonright \_ : \mathbb{P}\,\mathbb{N}_1 \times \operatorname{seq} X \rightarrow \operatorname{seq} X
\end{array}
$$

in order for *atIndex* to return the collection member without altering its type, the first member of $\textit{atIdx}'$ must be returned, not $\textit{atIdx}'$ itself.

$$
\begin{array}{|l}
\textit{atIdx}' : \textit{Collection} \\
\textit{coll}!, \textit{coll}?_{\textit{idx}?} : V \\
\hline
\textit{atIdx}' = (\textit{idx}? \upharpoonright \textit{coll}?) \Rightarrow \langle \textit{coll}?_{\textit{idx}?} \rangle \\
\textit{coll}! = \textit{head}(\textit{atIdx}') = \textit{coll}?_{\textit{idx}?}
\end{array}
$$

The *head* call is made possible by restricting $\textit{idx}?$ to be a single numeric value.

$$
\begin{array}{l}
\textit{idx}?, \textit{idx}' : \mathbb{N} \\
\quad \# \textit{idx}? = 1 \bullet (\textit{idx}? \upharpoonright \textit{coll}?) = \langle \textit{coll}?_{\textit{idx}?} \rangle \bullet \\
\qquad (\textit{head}(\textit{idx}? \upharpoonright \textit{coll}?)) = \textit{coll}?_{\textit{idx}?} \qquad [\text{expected return given } \textit{idx}?] \\
\quad \# \textit{idx}' \geq 2 \bullet (\textit{idx}' \upharpoonright \textit{coll}?) = \langle \textit{coll}?_{\textit{idx}'_i} \mathrel{..} \textit{coll}?_{\textit{idx}'_j} \rangle \bullet \\
\qquad (\textit{head}(\textit{idx}' \upharpoonright \textit{coll}?)) = \textit{coll}?_{\textit{idx}'_i} \qquad [\text{unexpected return given } \textit{idx}']
\end{array}
$$

Additionally, if the provided $\textit{idx}? \notin \textit{coll}?$ then an empty Collection will be returned given that *head* must be passed a non-empty Collection.

$$
\begin{array}{|l}
\textit{head} : \operatorname{seq}_1 X \rightarrow X \\
\hline
\textit{idx}? \notin \textit{coll}? \Rightarrow (\textit{idx}? \upharpoonright \textit{coll}?) = \langle\rangle \neg \operatorname{seq}_1
\end{array}
$$

The properties of *atIndex* are illustrated in the following examples.

$$X = \langle x_0, x_1, x_2 \rangle$$

$$
\begin{aligned}
&\quad x_0 = 0 \\
&\quad x_1 = foo \\
&\quad x_2 = \langle a, b, c \rangle \\
&atIndex(X, 0) = 0 && [head\,(\langle\, x_0 \,\rangle)] \\
&atIndex(X, 1) = foo && [head\,(\langle\, x_1 \,\rangle)] \\
&atIndex(X, 2) = \langle a, b, c \rangle && [head\,(\langle\, x_2 \,\rangle)] \\
&atIndex(X, 3) = \langle\rangle && [3 \notin X \Rightarrow x_3 \notin X]
\end{aligned}
$$

7.1.5 Update

The operation *update* will return a Collection *coll*! which is the same as the input Collection *coll*? except for at index *idx*?. The existing member $coll?_{idx?}$ is replcaed by the provided Value *v*? at *idx*? in *coll*! such that

$$idx? \mapsto v? \in coll! \land idx? \mapsto coll?_{idx?} \notin coll!$$

which is equivalent to $remove \gg append$

$$update(coll?\,, v?\,, idx?\,) \equiv append(remove(coll?\,, idx?\,), v?\,, idx?\,)$$

The functionality of *update* is further explained in the following schema.

$$
\begin{array}{|l}
\hline
Update[Collection, V, \mathbb{N}] \\
idx? : \mathbb{N} \\
coll?\,, coll! : Collection \\
v? : V \\
update\_ : Collection \times V \times \mathbb{N} \rightarrowtail Collection \\
\hline
1 = \#\, idx? \\
coll! = update(coll?\,, v?\,, idx?\,) \bullet \\
\qquad let\ coll' == \{\, i : \mathbb{N} \mid i \in 0 \mathinner{..} idx? \,\} \upharpoonright coll? \\
\qquad\quad coll'' == head(coll') \frown v? \\
\qquad\quad coll''' == \{\, j : \mathbb{N} \mid j \in idx? + 1 \mathinner{..} \#\, coll? \,\} \upharpoonright coll? \\
\qquad = coll'' \frown coll'' \Rightarrow \\
\qquad\qquad (append(remove(coll', idx?\,), v?\,, idx?\,) \frown coll'') \land \\
\qquad\qquad (\, v? \mapsto idx? \in coll!\,) \land \\
\qquad\qquad (\, \#\, coll! = \#\, coll?\,) \land \\
\hline
\end{array}
$$

The value which previously existed at $idx? \in coll?$ is replaced with *v*? to result in *coll*!

- $coll'$ is the items in *coll*? up to and including *idx*?
- $coll''$ is the items in *coll*? except the item at *idx*? has been replaced with *v*?

- $coll'''$ is the items in $coll?$ from $idx? + 1$ to $\# coll? \Rightarrow coll?_{idx?} \notin coll''$

The following example illustrates these properties.

$$
\begin{aligned}
&X = \langle x_0, x_1, x_2 \rangle \\
&\quad x_0 = 0 \\
&\quad x_1 = foo \\
&\quad x_2 = \langle a, b, c \rangle \\
&\quad v? = bar \\
&update(X, v?, 0) = \langle bar, foo, \langle a, b, c \rangle \rangle \\
&update(X, v?, 1) = \langle 0, bar, \langle a, b, c \rangle \rangle \\
&update(X, v?, 2) = \langle 0, foo, bar \rangle \\
&update(X, v?, 3) = \langle 0, foo, \langle a, b, c \rangle, bar \rangle \\
&update(X, v?, 4) = append(X, v?, 3) = update(X, v?, 3) \iff 3 \notin \operatorname{dom} X
\end{aligned}
$$

7.2 Key Value Pairs

Operations which expect a Map $M = \langle\!\langle k_i v_{k_i} .. k_n v_{k_n} .. k_j v_{k_j} \rangle\!\rangle$

7.2.1 Map?

The operation $map?$ will return a boolean which indicates if the passed in argument is a KV

$$
\begin{array}{|l}
\hline
Map?[V] \\
m? : V \\
bol! : Boolean \\
map?\ \_ : V \rightarrow Boolean \\
\hline
bol! = map?(m?) \bullet bol! = true \iff m? : KV \Rightarrow V \setminus (Scalar, Collection) \\
\hline
\end{array}
$$

where $V \setminus (Scalar, Collection)$ is used to indicate that $m?$ is of type V

$$V ::= Scalar \mid Collection \mid KV$$

but in order for $bol! = true$, $m?$ must not be of type $Scalar \lor Collection$ such that

$$
\begin{aligned}
&X = \langle\!\langle x_0, x_1, x_2, x_3, x_4 \rangle\!\rangle \\
&\quad x_0 = 0 \\
&\quad x_1 = foo \\
&\quad x_2 = \langle baz,\ qux \rangle \\
&\quad x_3 = \langle\!\langle abc \mapsto 123,\ def \mapsto 456 \rangle\!\rangle
\end{aligned}
$$

$$
\begin{array}{ll}
\quad x_4 = \langle \langle\!\langle ghi \mapsto 789,\ jkl \mapsto 101112 \rangle\!\rangle,\ \langle\!\langle ghi \mapsto 131415,\ jkl \mapsto 161718 \rangle\!\rangle \rangle & \\
map?\,(X) = true & \text{[KV by definition]} \\
map?\,(x_3) = true & \text{[KV]} \\
map?\,(x_2) = false & \text{[Collection]} \\
map?\,(x_4) = false & \text{[Collection of maps]} \\
map?\,(x_0) = false & \text{[Scalar]} \\
map?\,(x_1) = false & \text{[String]}
\end{array}
$$

7.2.2 Associate

The operation *associate* establishes a relationship between $k?$ and $v?$ at the top level of $m!$.

$$
\begin{array}{|l}
\hline
Associate[KV, K, V] \\
m?\,, m!\,, m' : KV \\
k? : K \\
v? : V \\
associate\_ : KV \times K \times V \rightarrowtail KV \\
\hline
m! = associate(m?\,, k?\,, v?\,) \bullet \\
\qquad let\ m' == m? \lhd k? \Rightarrow \\
\qquad\qquad\qquad (\mathrm{dom}\ m' = \mathrm{dom}\,(\,m? \setminus k?\,)) \land \\
\qquad\qquad\qquad (\,m? \setminus m' = k? \iff k? \in m?\,) \land \\
\qquad\qquad\qquad (\,m? \setminus m' = \emptyset \iff k? \notin m? \Rightarrow m? = m') \\
\quad = \langle\!\langle k? \mapsto v?\,\rangle\!\rangle \cup m' \\
\hline
\end{array}
$$

This implies that any existing mapping at $k? \in m?$ will be overwritten by *associate* but an existing mapping is not a precondition.

$$
\begin{array}{|l}
(k?\,, m?_{k?}\,) \in m? \lor (k?\,, m?_{k?}\,) \notin m? \\
(k?\,, m?_{k?}\,) \notin m! \\
(k?\,, v?\,) \in m! \\
\hline
m! = associate(m?\,, k?\,, v?\,)
\end{array}
$$

associate does not alter any other mappings within $m?$ and this property is illustrated by the definition of local variable m'

$$
\begin{array}{|l}
m' : KV \mid m' = m? \lhd k? \Rightarrow m' \lhd (m? \setminus k?\,) \\
\hline
\mathrm{dom}\ m? = \{\, k_i : K \mid 0 \mathrel{..} \# m? \bullet k_i \in m? \land 0 \le i \le \# m?\,\} \\
\mathrm{dom}\ m' = \{\, k_i' : K \mid 0 \mathrel{..} \# m' \bullet k_i' \in m? \land k_i' \neq k? \land 0 \le i \le \# m'\} \\
\mathrm{dom}\ m' = \mathrm{dom}\ m? \iff k? \notin m? \Rightarrow \forall k_i \in m? \mid k_i \neq k? \\
\# m' = \# m? \iff k? \notin m? \\
\# m' = \# m? -1 \iff k? \in m?
\end{array}
$$

and its usage within the definition of *associate*.

$$m! = m? \cup \langle\!\langle\, k? \mapsto v? \,\rangle\!\rangle \Rightarrow k? \notin m?$$
$$m! = m' \cup \langle\!\langle\, k? \mapsto v? \,\rangle\!\rangle \Rightarrow m' \neq m? \wedge k? \in m?$$

The following examples demonstrate the intended functionality of *associate*.

$$M = \langle\!\langle k_0 v_{k_0}, k_1 v_{k_1} \rangle\!\rangle$$
$$k_0 = abc \ \wedge v_{k_0} = 123 \qquad [k_0 v_{k_0} = abc \mapsto 123]$$
$$k_1 = def \ \wedge v_{k_1} = xyz \mapsto 456 \qquad [k_1 v_{k_1} = def \mapsto xyz \mapsto 456]$$
$$associate(M, baz, foo) = \langle\!\langle abc \mapsto 123,\ def \mapsto xyz \mapsto 456,\ baz \mapsto foo \rangle\!\rangle$$
$$associate(M, abc, 321) = \langle\!\langle abc \mapsto 321,\ def \mapsto xyz \mapsto 456 \rangle\!\rangle$$

7.2.3 Dissociate

The operation *dissociate* will remove some $k \mapsto v$ from KV given $k \in KV$

$Dissociate[KV, K]$

$m?, m! : KV$
$k? : K$
$dissociate\_ : KV \times K \twoheadrightarrow KV$

$$m! = dissociate(m?, k?) \bullet m! = m? \lhd k? \Rightarrow$$
$$(\mathrm{dom}\ m! = \mathrm{dom}\,(m? \setminus k?)) \wedge$$
$$(m? \setminus m! = k? \iff k? \in m?) \wedge$$
$$(m? \setminus m! = \emptyset \iff k? \notin m? \Rightarrow m? = m!) \wedge$$
$$((k?, m?_{k?}) \notin m!)$$

such that every mapping in $m?$ is also in $m!$ except for $k? \mapsto m?_{k?}$.

$$M = \langle\!\langle k_0 v_{k_0}, k_1 v_{k_1} \rangle\!\rangle$$
$$k_0 = abc \ \wedge v_{k_0} = 123 \qquad [k_0 v_{k_0} = abc \mapsto 123]$$
$$k_1 = def \ \wedge v_{k_1} = xyz \mapsto 456 \qquad [k_1 v_{k_1} = def \mapsto xyz \mapsto 456]$$
$$dissociate(M, abc) = \langle\!\langle def \mapsto xyz \mapsto 456 \rangle\!\rangle$$
$$dissociate(M, def) = \langle\!\langle abc \mapsto 123 \rangle\!\rangle$$
$$dissociate(M, xyz) = M \qquad [xyz \notin M]$$

7.2.4 At Key

The operation *atKey* will return the Value v at some specified Key k.

$$
\begin{array}{|l}
\hline
\quad AtKey[KV, K] \\
m? : KV \\
v! : V \\
k? : K \\
atKey\_ : KV \times K \twoheadrightarrow V \\
\hline
v! = atKey(m?, k?) \bullet \\
\qquad\qquad let\ coll == ((\operatorname{seq} m?) \upharpoonright (k?, m?_{k?})) \Rightarrow \langle (k?, m?_{k?}) \rangle \iff k? \in \operatorname{dom} m? \\
\quad = (second(head(coll)) \iff k? \mapsto m?_{k?} \in coll) \vee \\
\qquad (\emptyset \iff k? \notin \operatorname{dom} m?) \\
\hline
\end{array}
$$

In the schema above, $coll$ is the result of filtering for $(k?, m?_{k?})$ within $\operatorname{seq} m?$. If the mapping was in the original $m?$, it will also be in the sequence of mappings. This means we can filter over the sequence to look for the mapping and if found, it is returned as $\langle (k?, m?_{k?}) \rangle$. To return the mapping itself, $head(coll)$ is used to extract the mapping such that the value mapped to $k?$ can be returned.

$$v! = atKey(m?, k?) = second(head(coll)) = m?_{k?} \bullet m?_{k?} : V \iff k? \in \operatorname{dom} m?$$

The follow examples demonstrate the properties of $atKey$

$$
\begin{array}{ll}
M = \langle\!\langle k_0 v_{k_0}, k_1 v_{k_1} \rangle\!\rangle & \\
\quad k_0 = abc \wedge v_{k_0} = 123 & [k_0 v_{k_0} = abc \mapsto 123] \\
\quad k_1 = def \wedge v_{k_1} = xyz \mapsto 456 & [k_1 v_{k_1} = def \mapsto xyz \mapsto 456] \\
atKey(M, abc) = 123 & \\
atKey(M, def) = xyz \mapsto 456 & \\
atKey(M, foo) = \emptyset &
\end{array}
$$

7.3 Utility

Operations which are usefull in many Statement processing contexts.

7.3.1 Map

The map operation takes in a function $fn?$, Collection $coll?$ and additional Arguments $args?$ (as necessary) and returns a modified Collection $coll!$ with members $fn!_n$. The ordering of $coll?$ is maintained within $coll!$

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{Map[(\_ \nrightarrow \_), Collection, V]} \\
fn? : (\_ \nrightarrow \_) \\
args? : V \\
coll?, coll! : Collection \\
map\_ : (\_ \nrightarrow \_) \times Collection \times V \twoheadrightarrow Collection \\
\hline
coll! = map(fn?, coll?, args?) \bullet \\
\qquad \langle \forall n : i \mathrel{..} j \in coll? \mid i \leq n \leq j \wedge j = \# coll? \bullet \\
\qquad\quad \exists_1 fn!_n : V \mid fn!_n = \\
\qquad\qquad (fn?(coll?_n, args?) \iff args? \neq \emptyset) \vee \\
\qquad\qquad (fn?(coll?_n) \iff args? = \emptyset)\rangle \Rightarrow fn!_i \frown fn!_n \frown fn!_j \\
\hline
\end{array}
$$

Above, $fn!_n$ is introduced to handle the case where $fn?$ only requires a single argument. Additional arguments may be necessary but if they are not ($args? = \emptyset$) then only $coll?_n$ is passed to $fn?$.

$$X = \langle 1, 2, 3 \rangle$$

$$map\,(succ, X) = \langle 2, 3, 4 \rangle \qquad \text{[increment each member of } X\text{]}$$

$$map\,(+, X, 2) = \langle 3, 4, 5 \rangle \qquad \text{[add 2 to each member of } X\text{]}$$

7.3.2 Iso To Unix Epoch

The $isoToUnix$ operation converts an ISO 8601 Timestamp (see the xAPI Specification) to the number of seconds that have elapsed since January 1, 1970

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{IsoToUnix} \\
Timestamp \\
seconds! : \mathbb{N} \\
isoToUnix\_ : \mathbb{F}_1 \rightarrow \mathbb{N} \\
\hline
seconds! = isoToUnix(timestamp) \\
\hline
\end{array}
$$

$$ts = 2015 - 11 - 18T12 : 17 : 00 + 00 : 00 \equiv 2015 - 11 - 18T12 : 17 : 00Z$$

$$isoToUnixEpoch(ts) = 1447849020 \qquad \text{[ISO 8601} \rightarrow \text{Epoch time]}$$

7.3.3 Timeunit To Number of Seconds

The operation $toSeconds$ will return the number of seconds corresponding to the input $Timeunit$

$$Timeunit ::= second \mid minute \mid hour \mid day \mid week \mid month \mid year$$

such that the following schema defines $toSeconds$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad ToSeconds[Timeunit] \quad}} \\
t? : Timeunit \\
toSeconds\_ : Timeunit \nrightarrow \mathbb{N} \\
\hline
toSeconds(t?) = 1 \iff t? = second \\
toSeconds(t?) = 60 \iff t? = minute \\
toSeconds(t?) = 3600 \iff t? = hour \\
toSeconds(t?) = 86400 \iff t? = day \\
toSeconds(t?) = 604800 \iff t? = week \\
toSeconds(t?) = 2629743 \iff t? = month \\
toSeconds(t?) = 31556926 \iff t? = year \\
\hline
\end{array}
$$

7.4 Rate Of

The Operation $rateOf$ calculates the number of times something occured within an interval of time given a unit of time.

$$rateOf(nOccurances, start, end, unit)$$

Where the output translates to: the rate of occurance per unit within interval

- $nOccurances$ is the number of times something happened and should be an Integer (called $nO?$ bellow)
- $start$ is an ISO 8601 timestamp which serves as the first timestamp within the interval
- end is an ISO 8601 timestamp which servers as the last timestamp within the interval
- $unit$ is a String Enum representing the unit of time

This can be seen in the definition of $rateOf$ bellow.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad RateOf[\mathbb{N}, TIMESTAMP, TIMESTAMP, TIMEUNIT] \quad}} \\
nO? : \mathbb{N} \\
rate! : \mathbb{Z} \\
start?, end? : TIMESTAMP \\
unit? : TIMEUNIT \\
rateOf\_ : \mathbb{N} \times TIMESTAMP \times TIMESTAMP \times TIMEUNIT \rightarrow \mathbb{Z} \\
\hline
rate! = rateOf(nO?, start?, end?, unit?) \bullet \\
\qquad let \quad interval == isoToUnix(end) - isoToUnix(start) \\
\qquad\qquad unitS == toSeconds(unit?) \\
\qquad\ = nO? \div (interval \div units) \\
\hline
\end{array}
$$

The only other functionality required by $rateOf$ is supplied via basic arithmetic

$$start = 2015 - 11 - 18T12 : 17 : 00Z$$
$$end = 2015 - 11 - 18T14 : 17 : 00Z$$
$$unit = second$$
$$nO? = 10$$
$$startN = isoToUnix(start) = 1447849020$$
$$endN = isoToUnix(end) = 1447856220$$
$$interval = endN - StartN = 7200$$
$$unitN = toSeconds(unit) = 60$$
$$0.001389 = rateOf(nO?\,, start, end, unit) \Rightarrow 10 \div (7200 \div 60)$$
$$5 = rateOf(nO?\,, start, end, hour) \Rightarrow 10 \div (7200 \div 3600)$$

8 Common Primitives

There will be many Primitives used within Algorithm definitions in DAVE but navigation into a nested *Collection* or *KV* is most likely to be used across nearly all Algorithm definitions. Because of this, the first common Primitive to be introduced is *walk*. In order to define *walk* using the Operation *recur*, the following helper Operations are introduced. These two helper Operations will be used to describe the navigation into and then back out of a nested Value based on the provided Collection of identifiers.

$$
\begin{array}{|l}
\multicolumn{1}{l}{GetNext[V, Collection]} \\
in?, next! : V \\
id? : Collection \\
getNext\_ : V \times Collection \twoheadrightarrow V \\
\hline
next! = getNext(in?, id?) \bullet \\
\qquad = (atIndex(in?, head(id?)) \iff (array?(in?) = true) \land (head(id?) \in \mathbb{N})) \lor \\
\qquad\quad (atKey(in?, head(id?)) \iff (array?(in?) = false) \land (map?(in?) = true))
\end{array}
$$

- Navigation down into either a *Collection* or *KV* based on the type of *in*?

$$
\begin{array}{|l}
\multicolumn{1}{l}{Merge[(V, V), Collection]} \\
parent?, child?, parent! : V \\
at? : Collection \\
merge\_ : (V \times V) \times Collection \rightarrowtail\!\!\!\!\rightarrow V \\
\hline
parent! = merge((parent?, child?), at?) \bullet \\
\qquad = (associate(parent?, head(at?), child?) \\
\qquad\qquad \iff map?(parent?) = true) \lor \\
\qquad\quad (update(parent?, child?, head(at?)) \\
\qquad\qquad \iff (array?(parent?) = true) \land (head(at?) \in \mathbb{N}))
\end{array}
$$

- Updating of *parent*? to include *child*? at location indicated by $head(at?)$

The helper Operations defined above are necessary for describing the traversal of a heterogeneous nested Value. Collection and KV have different Fundamental Operations for navigation and update. Their usage in *walk* is touched on within the following summary and expanded further within the formal definition.

1. navigate down into the provided value *in*? up until the second to last value $in?_{path?_{j-1}}$ as described by the provided *path*?

$$
\begin{array}{l}
in?_{path?_{j-1}} : V \\
\hline
path?_{j-1} \Rightarrow path? \lhd j \Rightarrow path? \lhd (\operatorname{dom} path? \setminus \{j\})
\end{array}
$$

2. extract any existing data mapped to $atIndex(path?, j)$ from the result of step 1

$$\frac{in?_{path?} : V}{path? \Rightarrow path?_{j-1} \cup (j, atIndex(path?, j))}$$

3. create the mapping $(atIndex(path?, j), in?_{path?})$ labeled here as $args?$

$$\frac{args? = (atIndex(path?, j), in?_{path?})}{\begin{array}{l} args? \in in?_{path?_{j-1}} \\ first(args?) = atIndex(path?, j) \end{array}}$$

4. pass $in?_{path?}$ to the provided function $fn?$ to produce some output $fn!$

$$fn! = fn?\,(second(args?)) = fn?\,(in?_{path?})$$

5. replace the previous mapping $args?$ within $in?_{path?_{j-1}}$ with $fn!$ at $atIndex(path?, j)$

$$\frac{\begin{array}{l} child_j = first(args?) \mapsto fn! \\ in!?_{path?_{j-1}} = merge((in?_{path_{j-1}}, fn!), first(args?)) \end{array}}{\begin{array}{l} child_j \in in!?_{path?_{j-1}} \\ child_j \notin in?_{path?_{j-1}} \iff child_j \neq args? \\ args? \in in?_{path?_{j-1}} \\ args? \notin in!?_{path?_{j-1}} \iff args? \neq child_j \end{array}}$$

6. retrace navigation back up from $in!?_{path?_{j-1}}$, updating the mapping at each $path?_n \in path?$ without touching any other mappings.

$$\frac{in!?_{path?_{j-1}} \lhd first(args?) = in?_{path?_{j-1}} \lhd first(args?) \iff args? \neq child_j}{\begin{array}{l} args? \neq child_j \Rightarrow second(args?) \neq second(child_j) \\ in!?_{path?_{j-1}} \lhd first(args?) \Rightarrow in!?_{path?_{j-1}} \lhd (\operatorname{dom} in!?_{path?_{j-1}} \setminus first(args?)) \end{array}}$$

7. return $out!$ after the final update is made to $in?$.

$$\frac{\begin{array}{l} child_i = atIndex(path?, i) \mapsto in!?_{path?_i} \\ in!?_{path?_i} = merge((in?_{path?_i}, in!?_{path?_{i+1}}), atIndex(path?, i+1)) \end{array}}{\begin{array}{l} out! = merge((in?, second(child_i)), first(child_i)) \bullet \\ \quad in? \lhd head(path?) = out! \lhd head(path?) \Rightarrow \\ \qquad \forall (a, b) \in path? \bullet b = atIndex(path?, a) \mid \exists a \bullet in?_a = out!_a \iff a \neq head(path?) \end{array}}$$

The summary of $walk$ given above is formalized within the schema $Walk$ bellow where $Walk$ dives deeper into the properties/constraints provided for each step. The variables names used in the summary are NOT used in all cases within $Walk$.

$$
\begin{array}{|l}
\hline
Walk[V, Collection, (\_ \nrightarrow \_)] \\
GetNext, Merge, Recur \\
in?\,, out!\,, fn! : V \\
path? : Collection \\
fn? : (\_ \nrightarrow \_) \\
walk\_ : V \times Collection \times (\_ \nrightarrow \_) \rightarrowtail V \\
\hline
walk = \langle \langle getNext\_, recur\_ \rangle^{\# path? - 1}, (\_ \nrightarrow \_), \langle merge\_, recur\_ \rangle^{\# path? - 1} \rangle \\
out! = walk(in?\,, path?\,, fn?\,) \bullet \\
\;\forall n : i \mathrel{..} j - 1 \bullet j = first(last(path?\,)) \Rightarrow \\
\qquad\qquad\qquad\qquad\qquad first(j, path?_j\,) \mid \exists\, down_n \bullet \\
\qquad let\;\; path?_n == tail(path?\,)^{n-i} \\
\qquad\quad down_i == getNext(in?\,, path?_n\,) \Rightarrow \\
\qquad\qquad\qquad\qquad atIndex(in?\,, head(path?\,)) \vee atKey(in?\,, head(path?\,)) \iff n = i \\
\qquad\quad down_n == recur(down_i, path?_n\,, getNext\_)^{j-1} \\
\qquad\quad down_{j-1} == getNext(down_n, path?_n\,) \iff n = j - 2 \\
\qquad\quad down_j == getNext(down_{j-1}, path?_n\,) \bullet \\
\qquad\qquad\qquad\qquad path?_n \equiv (path? \upharpoonright j) \Rightarrow \langle j \mapsto atIndex(path?\,, j) \rangle \iff n = j - 1 \\
\qquad fn! = fn?\,(down_j) \\
\;\forall z : p \mathrel{..} q \bullet ((p = j - 1) \wedge (q = i + 1)) \Rightarrow \\
\qquad\qquad\qquad\qquad\qquad ((z = p \iff n = j - 1) \wedge (z = q \iff n = i + 1)) \mid \exists\, up_n \bullet \\
\qquad let\;\; path?_{rev} == rev(path?\,) \\
\qquad\quad path?_z == tail(path?_{rev}\,)^{p-z+1} \\
\qquad\quad up_p == merge((down_{j-1}, fn!\,), path?_z\,) \Rightarrow \\
\qquad\qquad\qquad (path?_z \equiv tail(path?_{rev}\,)) \wedge \\
\qquad\qquad\qquad\; (associate(down_{j-1}, head(path?_z\,), fn!\,) \vee \\
\qquad\qquad\qquad\; update(down_{j-1}, fn!\,, head(path?_z\,))) \iff z = p \\
\qquad\quad up_z == recur((down_n, up_p), path?_z\,, merge\_)^p \iff p = n + 1 \wedge z = n \\
\qquad\quad up_q == merge((down_{i+1}, up_z), path?_z\,) \iff z = q + 1 \Rightarrow z = i + 2 \\
\qquad\quad up_i == merge((down_i, up_q), path?_z\,) \iff z = q \Rightarrow z = i + 1 \Rightarrow up_i = up_{q-1} \\
out! = merge((in?\,, up_i), path?_n\,) \equiv merge((in?\,, up_i), (path? \upharpoonright i)) \iff (n = i = q - 1) \\
\hline
\end{array}
$$

The following examples demonstrate the functionality of the Primitive *walk*

$$
\begin{array}{l}
X = \langle x_0, x_1, x_2 \rangle \wedge fn! = fn(val?\,, idx?\,) = ZZZ \\
\quad x_0 = true \\
\quad x_1 = \langle a, b, c \rangle \\
\quad x_2 = \langle\!\langle foo \mapsto \langle\!\langle bar \mapsto buz, x \mapsto y \rangle\!\rangle \rangle\!\rangle \\
walk(X, \langle 0 \rangle, array?\,\_) = \langle false, x_1, x_2 \rangle \qquad\qquad\qquad [true \neg Collection] \\
walk(X, \langle 2, foo, z \rangle, fn\_) = \langle x_0, x_1, \langle\!\langle foo \mapsto \langle\!\langle bar \mapsto buz, x \mapsto y, z \mapsto ZZZ \rangle\!\rangle \rangle\!\rangle \rangle \\
walk(X, \langle 2, foo, x \rangle, fn\_) = \langle x_0, x_1, \langle\!\langle foo \mapsto \langle\!\langle bar \mapsto buz, x \mapsto ZZZ \rangle\!\rangle \rangle\!\rangle \rangle \\
walk(X, \langle 2, qux \rangle, fn\_) = \langle x_0, x_1, (x_2 \cup qux \mapsto ZZZ) \rangle \\
walk(X, \langle 1 \rangle, map(succ\_, x_1, 1)) = \langle x_0, \langle b, c, d \rangle, x_2 \rangle \\
walk(X, \langle 1, 0 \rangle, succ\_) = \langle x_0, \langle b, b, c \rangle, x_2 \rangle
\end{array}
$$

8.1 Collections

Primitives which expect a Collection $X = \langle x_i..x_n..x_j \rangle$

8.1.1 At Depth

The Primitive *atDepth* will return the Value at a specified depth of indices within a passed in Collection. The following helper Operation *getFirstIndex* is introduced to establish navigation into a nested Collection given a Collection of Indices.

$$
\begin{array}{l}
\textit{GetFirstIndex}[\textit{Collection}, \textit{Collection}] \\
\hline
\textit{coll}?, \textit{idxs}? : \textit{Collection} \\
v! : V \\
\textit{getFirstIndex}\,\_ : \textit{Collection} \times \textit{Collection} \twoheadrightarrow V \\
\hline
v! = \textit{getFirstIndex}(\textit{coll}?, \textit{idxs}?) \bullet v! = \textit{atIndex}(\textit{coll}?, \textit{head}(\textit{idxs}?))
\end{array}
$$

This allows for the navigation into a nested Collection to be defined as $\langle \textit{getFirstIndex}\,\_, \textit{recur}\,\_ \rangle^{\#\,idxs?}$ which represents a step down into *coll*? for each member of *idxs*?. If there is not a value at some specified index or navigation can't continue despite what is being dictated by *idxs*?, the empty sequence $\langle\rangle$ will be returned

$$
\begin{array}{l}
\textit{AtDepth}[\textit{Collection}, \textit{Collection}] \\
\hline
\textit{Recur}, \textit{GetFirstIndex} \\
\textit{coll}?, \textit{idxs}? : \textit{Collection} \\
v! : V \\
\textit{atDepth}\,\_ : \textit{Collection} \times \textit{Collection} \twoheadrightarrow V \\
\hline
\textit{atDepth} = \langle \textit{getFirstIndex}\,\_, \textit{recur}\,\_ \rangle^{\#\,idxs?} \\
v! = \textit{atDepth}(\textit{coll}?, \textit{idxs}?) \bullet \\
\quad \forall n : i \mathrel{..} j \in \operatorname{dom} \textit{idxs}? \bullet j = \textit{first}(\textit{last}(\textit{idxs}?)) \Rightarrow \textit{first}(j, \textit{idxs}?_j) \mid \exists_1 v_n \bullet \\
\qquad \textit{let} \quad \textit{idxs}_n == \textit{tail}(\textit{idxs}?)^{n-i} \\
\qquad\qquad v_i == \textit{getFirstIndex}(\textit{coll}?, \textit{idxs}_n) \\
\qquad\qquad v_n == \textit{recur}(v_i, \textit{idxs}_n, \textit{getFirstIndex}\,\_)^j \\
\qquad\qquad v_j == \textit{atIndex}(v_n, \textit{last}(\textit{idxs}?)) \iff n = j - 1 \\
v! = v_j \bullet v_j : \operatorname{seq}_1 \iff (v_{n-1} = \operatorname{seq}_1 \wedge \\
\qquad\qquad\qquad\qquad \textit{head}(\textit{idxs}_n) \mapsto v_n \in v_{n-i}) \vee \\
\qquad v_j : \operatorname{seq} \iff v_{n-1} = \langle\rangle \Rightarrow \textit{atIndex}(v_{n-2}, \textit{head}(\textit{idxs}_{n-1})) = \langle\rangle
\end{array}
$$

The following examples demonstrate the properties of *atDepth* described above.

$$
\begin{aligned}
&X = \langle x_0, x_1, x_2 \rangle \\
&\quad x_0 = 0 \\
&\quad x_1 = foo \\
&\quad x_2 = \langle a, b, c \rangle \\
&\textit{atDepth}(X, \langle 1 \rangle) = foo
\end{aligned}
$$

$atDepth(X, \langle 1, 0\rangle) = f \Rightarrow foo = \langle f, o, o\rangle$
$atDepth(X, \langle 2, 0\rangle) = a$
$atDepth(X, \langle 2, 5\rangle) = \langle\rangle$

8.1.2 Append At

The Primitive *appendAt* uses the Primitive *atDepth* to navigated into a nested collection $coll?$ (called *coll* bellow). The Value $v?$ passed to *appendAt* will be appended to *coll* at $idxs?_j$. This results in a $coll!$ which is equivalent to $coll?$ except for at the value at the path $idxs?_i \mathrel{..} idxs?_j \in coll?$.

$$
\begin{array}{|l}
\text{\_\_ } AppendAt[Collection, Collection, V] \text{ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
AtDepth \\
coll?, coll!, idxs? : Collection \\
v? : V \\
appendAt\_ : Collection \times Collection \times V \rightarrowtail Collection \\
\hline
appendAt = \langle atDepth\_, append\_, \langle\, atDepth\_, remove\_, append\_\rangle^{\#\, idxs? - 1}\rangle \\
coll! = appendAt(coll?, idxs?, v?) \bullet \\
\quad\quad let \quad coll == append(atDepth(coll?, idxs? \lhd idxs?_j), v?, idxs?_j) \bullet \\
\quad\quad\quad\quad \forall n : i \mathrel{..} j - 1 \bullet j = first(last(idxs?)) \mid \exists c_n \bullet \\
\quad\quad\quad\quad\quad\quad let \quad c_i == atDepth(coll?, (idxs? \upharpoonright i)) \Rightarrow atIndex(coll?, idxs?_i) \\
\quad\quad\quad\quad\quad\quad\quad\quad c_n == atDepth(c_{n-1}, (idxs? \upharpoonright n)) \\
\quad\quad\quad\quad\quad\quad\quad\quad c_{j-1} == atDepth(c_n, (idxs? \upharpoonright j - 1)) \iff n = j - 2 \\
\quad\quad\quad\quad\quad\quad\quad\quad c_j == append(c_{j-1}, v?, (idxs? \upharpoonright j)) \Rightarrow c_j = coll = coll!_j \\
\quad\quad\quad\quad\quad\quad\quad\quad coll!_{j-1} == append(remove(c_{j-2}, idxs?_{j-1}), c_j, idxs?_{j-1}) \\
\quad\quad\quad\quad\quad\quad\quad\quad coll!_n == append(remove(c_{n-1}, idxs?_n), coll!_n, idxs?_n) \\
\quad\quad\quad\quad\quad\quad\quad\quad coll!_i == append(remove(c_i, idxs?_n), coll!_n, idxs?_n) \iff n = i + 1 \\
\quad\quad = append(remove(coll?, idxs?_i), coll!_i, idxs?_i) \\
\hline
\end{array}
$$

The relationship described above $coll? \lhd idxs_i = coll! \lhd idxs_i$ is described above as $\langle\, atDepth\_, remove\_, append\_\rangle^{\#\, idxs? - 1}$. The variables $coll!_{i \mathrel{..} j}$ were used to describe the sub Collections which have to have a single index updated given $idxs?$. Those subcollections are combined together to produce $coll!$ such that the only difference between $coll? \wedge coll!$ is found at path $idxs?$ The following examples demonstrate the properties of *appendAt* described above.

$X = \langle x_0, x_1, x_2\rangle$
$\quad x_0 = 0$
$\quad x_1 = foo$
$\quad x_2 = \langle a, b, c\rangle$
$appendAt(X, \langle 1, 3\rangle, z) = \langle x_0, fooz, x_2\rangle \Rightarrow foo = \langle f, o, o\rangle$
$appendAt(X, \langle 0\rangle, 5) = \langle\langle 0, 5\rangle, x_1, x_2\rangle$ [existing item gets 0 index]
$appendAt(X, \langle 0, 0\rangle, 5) = \langle\langle 5, 0\rangle, x_1, x_2\rangle$ [overwritting default behavior]
$appendAt(X, \langle 2, 0\rangle, d) = \langle x_0, x_1, \langle d, a, b, c\rangle\rangle$

$appendAt(X, \langle 2 \rangle, d) = \langle x_0, x_1, \langle a, b, c, d \rangle \rangle$

8.1.3 Update At

The primitive *updateAt* performs a replacement at some depth within a Collection without changing any other values in the source Collection.

$$
\begin{array}{l}
\underline{UpdateAt[Collection, V, Collection]} \\
AtDepth \\
coll? , coll! , indices? : Collection \\
v? : V \\
updateAt\_ : Collection \times V \times Collection \rightarrowtail Collection \\
\hline
updateAt = \langle atDepth\_ , update\_ , \langle atDepth\_ , update\_ \rangle^{\# indices? - 1} \rangle \\
coll! = updateAt(coll? , v? , indices?) \bullet \\
\quad let \quad coll == update(atDepth(coll? , indices? \lhd indices?_j), v? , indices?_j) \bullet \\
\qquad \forall n : i \mathrel{..} j - 1 \bullet j = first(last(indices?)) \mid \exists c_n \bullet \\
\qquad\quad let \quad c_i == atDepth(coll? , (indices? \upharpoonright i)) \Rightarrow atIndex(coll? , indices?_i) \\
\qquad\qquad c_n == atDepth(c_{n-1}, (indices? \upharpoonright n)) \\
\qquad\qquad c_{j-1} == atDepth(c_n, (indices? \upharpoonright j - 1)) \iff n = j - 2 \\
\qquad\qquad c_j == update(c_{j-1}, v? , (indices? \upharpoonright j)) \Rightarrow c_j = coll = coll!_j \\
\qquad\qquad coll!_{j-2} == update(c_{j-2}, c_j , indices?_{j-1}) \\
\qquad\qquad coll!_n == update(c_{n-1}, coll!_n , indices?_n) \\
\qquad\qquad coll!_i == update(c_i, coll!_n , indices?_n) \iff n = i + 1 \\
\quad = update(coll? , coll!_i , indices?_i)
\end{array}
$$

The following examples demonstrate the properties of *updateAt*

$$
\begin{array}{ll}
X = \langle x_0, x_1, x_2 \rangle & \\
\quad x_0 = 0 & \\
\quad x_1 = foo & \\
\quad x_2 = \langle a, b, c \rangle & \\
updateAt(X, \langle 1, 3 \rangle, z) = \langle x_0, fooz, x_2 \rangle \Rightarrow foo = \langle f, o, o \rangle & \\
updateAt(X, \langle 1, 0 \rangle, z) = \langle x_0, zoo, x_2 \rangle \Rightarrow zoo = \langle z, o, o \rangle & \\
updateAt(X, \langle 0 \rangle, 5) = \langle 5, x_1, x_2 \rangle & \\
updateAt(X, \langle 0, 0 \rangle, 5) = \langle \langle 5 \rangle, x_1, x_2 \rangle & \text{[5 in seq]} \\
updateAt(X, \langle 0, 1 \rangle, 5) = \langle \langle 0, 5 \rangle, x_1, x_2 \rangle & \text{[5 and 0 in seq]} \\
updateAt(X, \langle 2, 0 \rangle, d) = \langle x_0, x_1, \langle d, b, c \rangle \rangle & \\
updateAt(X, \langle 2 \rangle, d) = \langle x_0, x_1, d \rangle &
\end{array}
$$

Which indicates that if navigation would result in stepping into a non-nested Value, an empty sequence is created and then populated with the non-nested Value so navigation can continue.

$X! = updateAt(X, \langle 0, 0 \rangle, 5) \mid i \mapsto 0 \wedge j \mapsto 0 \wedge i \neq j \bullet$

$$
\begin{aligned}
&let \quad X_i == atIndex(X, i) = 0 \\
&\qquad X_{i+1} == atIndex(X_i, j) = \langle\rangle \\
&\qquad X_j == append(X_{i+1}, X_i, 0) = \langle 0 \rangle \qquad \text{[only item hence 0 index]} \\
&\qquad X'_j == update(X_j, 5, j) = \langle 5 \rangle \\
&= update(X, X'_j, i) \\
&= \langle \langle 5 \rangle, x_1, x_2 \rangle
\end{aligned}
$$

- The nesting which created $\langle 0 \rangle$ also specified that the first index of that sub Collection should have the value of 5 which is why $X! = \langle \langle 5 \rangle, x_1, x_2 \rangle$

$$
\begin{aligned}
&X! = updateAt(X, \langle 0, 1 \rangle, 5) \mid i \mapsto 0 \wedge j \mapsto 1 \wedge i \neq j \bullet \\
&\quad let \quad X_i == atIndex(X, i) = 0 \\
&\qquad X_{i+1} == atIndex(X_i, j) = \langle\rangle \\
&\qquad X_j == append(X_{i+1}, X_i, 0) = \langle 0 \rangle \qquad \text{[only item hence 0 index]} \\
&\qquad X'_j == update(X_j, 5, j) = \langle 0, 5 \rangle \\
&\quad = update(X, X'_j, i) \\
&\quad = \langle \langle 0, 5 \rangle, x_1, x_2 \rangle
\end{aligned}
$$

- The nesting which created $\langle 0 \rangle$ now indicates that the second index of that sub Collection should have the value of 5 which is why $X! = \langle \langle 0, 5 \rangle, x_1, x_2 \rangle$

$$
\begin{aligned}
&X! = updateAt(X, \langle 0 \rangle, 5) \mid i \mapsto 0 \bullet \\
&\qquad \# indices? = 1 \Rightarrow updateAt(X, \langle 0 \rangle, 5) \equiv update(X, i, 5) \bullet \\
&\quad = update(X, i, 5) = \langle 5, x_1, x_2 \rangle
\end{aligned}
$$

- Here there is no further nesting which can't be performed so the update happens as if the Operation *update* was used.

8.2 Key Value Pairs

Primitives which expect a Map $M = \langle\!\langle k_i v_{k_i} .. k_n v_{k_n} .. k_j v_{k_j} \rangle\!\rangle$

8.2.1 Associate At

The Primitive *associateAt* establishes a relationship between $k?_j$ and $v?$ at the nesting $k?_i\ ..k?_{j-1}$ within Map $m!$

$$
\begin{array}{|l}
k? = \langle k?_i\ ..k?_j \rangle \\
(k?_j, m?_{k?}) \in m? \vee (k?_j, m?_{k?}) \notin m? \\
(k?_j, m?_{k?}) \notin m! \iff m?_{k?} \neq v? \\
(k?_j, v?) \in m! \\
\hline
m! = associateAt(m?, k?, v?)
\end{array}
$$

This implies that any existing mapping at $k?_j \in m?$ will be overwritten by *associateAt* but an existing mapping is not a precondition. The following helper Operation *getFirstKey* is introduced to establish navigation into a nested Map given a Collection of Keys.

$$
\begin{array}{|l}
\hline
\textit{GetFirstKey}[KV, \textit{Collection}] \\
m? : KV \\
k? : \textit{Collection} \\
v! : V \\
\textit{getFirstKey}\_ : KV \times \textit{Collection} \twoheadrightarrow V \\
\hline
v! = \textit{getFirstKey}(m?, k?) \bullet v! = \textit{atKey}(m?, \textit{head}(k?)) \\
\hline
\end{array}
$$

This allows for the navigation into a nested Map to be defined as $\langle \textit{getFirstKey}\_, \textit{recur}\_\rangle^{\# k? - 1}$ which represents a step down for each $k \in (k? \setminus k?_j)$. Once at $k?_{j-1}$, the mapped value v_{j-1} has $(k?_j, v?)$ added to it. This update is localized within $m?$ and all other mappings within $m?$ are left alone.

$$
\begin{array}{|l}
\hline
\textit{AssociateAt}[KV, \textit{Collection}, V] \\
\textit{GetFirstKey}, \textit{Recur} \\
m?, m! : KV \\
k? : \textit{Collection} \bullet \forall k?_n \in k?_{\langle i \,..\, j\rangle} \mid k_n : K \\
v? : V \\
\textit{associateAt}\_ : KV \times \textit{Collection} \times V \rightarrowtail KV \\
\hline
\textit{associateAt} = \langle\langle \textit{getFirstKey}\_, \textit{recur}\_\rangle^{\# k? - 1}, \langle \textit{associate}\_\rangle^{\# k}\rangle \\
m! = \textit{associateAt}(m?, k?, v?) \bullet \\
\qquad \forall n : i \,..\, j - 1 \in \operatorname{dom} k? \bullet j = \textit{first}(\textit{last}(k?)) \Rightarrow \textit{first}(j, k?_j) \mid \exists_1 v_n \bullet \\
\qquad\qquad \textit{let} \quad c_n == \textit{tail}(k?)^{n-i} \\
\qquad\qquad\qquad v_i == \textit{getFirstKey}(m?, c_n) \Rightarrow \\
\qquad\qquad\qquad\qquad c_n = k? \iff n = i \bullet v_i = \textit{atKey}(m?, \textit{head}(k?)) \\
\qquad\qquad\qquad v_n = \textit{recur}(v_i, c_n, \textit{getFirstKey}\_)^{j-1} \\
\qquad\qquad\qquad v_{j-1} == \textit{getFirstKey}(v_n, (j - 1 \upharpoonright k?)) \iff n = j - 2 \\
\qquad\qquad\qquad v_j == \textit{associate}(v_{j-1}, \textit{last}(k?), v?) \Rightarrow \langle\!\langle k?_j \mapsto v? \rangle\!\rangle \cup v_{j-1} \lhd k?_j \\
\qquad = (v_j \cup v_{n-succ(1)} \lhd k?_{n-succ(0)})^{j-1} \bullet n \leq j - 1 \Rightarrow \\
\qquad\qquad v_{j-4} \cup (v_{j-3} \cup (v_j \cup v_{j-2} \lhd k?_{j-1}) \lhd k?_{j-2}) \lhd k?_{j-3} \bullet \\
m! = \textit{associate}(m?, k?_i, \textit{associate}(v_i, k?_n, \textit{associate}(v_n, k?_{j-1}, \textit{associate}(v_{j-1}, k?_j, v?)))) \\
\hline
\end{array}
$$

In the schema above, the localization of the change and the retention of the other mappings is indicated via

$$
\begin{array}{l}
(v_j \cup v_{n-succ(1)} \lhd k?_{n-succ(0)})^{j-1} \bullet n \leq j - 1 \Rightarrow \\
\qquad v_{j-4} \cup (v_{j-3} \cup (v_j \cup v_{j-2} \lhd k?_{j-1}) \lhd k?_{j-2}) \lhd k?_{j-3}
\end{array}
$$

and is the reason why $\langle \textit{associate}\_\rangle^{\# k}$ is included in the definition of *associateAt* and is equivalent to

$$
\textit{associate}(m?, k?_i, \textit{associate}(v_i, k?_n, \textit{associate}(v_n, k?_{j-1}, \textit{associate}(v_{j-1}, k?_j, v?))))
$$

which gracefully walks into and back out of a KV regardless of $k?_n \in m?_{k?_{n-1}}$.

$$
\begin{aligned}
&M = \langle\!\langle k_i \mapsto v_i,\ k_n \mapsto \langle\!\langle k_{ni} \mapsto v_{ni},\ k_{nj} \mapsto v_{nj} \rangle\!\rangle \rangle\!\rangle \\
&\quad associateAt(M, \langle k_n, k_{nn} \rangle, v?\,) = \langle\!\langle k_i \mapsto v_i,\ k_n \mapsto \langle\!\langle k_{ni} \mapsto v_{ni},\ k_{nj} \mapsto v_{nj},\ k_{nn} \mapsto v?\, \rangle\!\rangle \rangle\!\rangle \\
&\quad associateAt(M, \langle k_j, k_{ji} \rangle, v?\,) = \langle\!\langle k_i \mapsto v_i,\ k_n \mapsto \langle\!\langle k_{ni} \mapsto v_{ni},\ k_{nj} \mapsto v_{nj} \rangle\!\rangle, k_j \mapsto \langle\!\langle k_{ji} \mapsto v?\, \rangle\!\rangle \rangle\!\rangle \\
&\quad associateAt(M, \langle k_i \rangle, v?\,) = \langle\!\langle k_i \mapsto v?\,,\ k_n \mapsto \langle\!\langle k_{ni} \mapsto v_{ni},\ k_{nj} \mapsto v_{nj} \rangle\!\rangle \rangle\!\rangle \\
&\quad associateAt(M, \langle k_i, k_{ii} \rangle, v?\,) = \langle\!\langle k_i \mapsto \langle\!\langle k_{ii} \mapsto v?\, \rangle\!\rangle,\ k_n \mapsto \langle\!\langle k_{ni} \mapsto v_{ni},\ k_{nj} \mapsto v_{nj} \rangle\!\rangle \rangle\!\rangle
\end{aligned}
$$

The last example demonstrates what happens when the value at some key is not a Map.

$$
\begin{aligned}
&atKey(v_i, k_{ii}) = \emptyset \iff k_{ii} \notin \operatorname{dom} v_i \bullet \\
&\quad associateAt(M, \langle k_i, k_{ii} \rangle, v?\,) \Rightarrow \\
&\qquad associate(M, k_i, associate(\emptyset, k_{ii}, v?\,))
\end{aligned}
$$

8.3 Utility

Primitives which are usefull in many Statement processing contexts.

8.4 Accumulate

Performs an update at *path* within *state* using the supplied *item* or $k \ \wedge \ v$

$$accumulate(state, path, item) \rightarrow state'$$

$$accumulate(state, path, k, v) \rightarrow state'$$

8.4.1 Arguments

- *state* is an Algorithm State
- *path* is a Collection of Key(s) used to navigate into *state*
- *item* is a Scalar which should be reflected within $state'$ at *path*
- k is a Value used as the target
  - Index within some Collection
  - Key within some KV such that $k \mapsto v \in KV$
- v is a Value
  - Added to some Collection at index k
  - Mapped to k within some KV such that $k \mapsto v \in KV$

8.4.2 Relevant Operations

The primitive *accumulate* uses the operations

- array?
- object?
- append
- associate
- atKey
- count

8.4.3 Summary

accumulate will do one of the following things

- replace an existing non-array Scalar or KV

$$accumulate(state, path, item) \equiv associate(state, path, item)$$

- update an existing array Scalar or Collection

$$accumulate(state, path, item) \equiv associate(state, path, append(state_{path}, item, count(state_{path})))$$

- updates an existing Scalar object or KV

$$accumulate(state, path, k, v) \equiv associate(state, append(path, k, count(path)), v)$$

- updates an existing array Scalar or Collection

$$accumulate(state, path, k, v) \equiv associate(state, path, append(state_{path}, v, k))$$

- create a new Collection containing $state_{path}$ and v

$$accumulate(state, path, k, v) \equiv associate(state, path, append(append(<>, state_{path}, 0), v, k))$$

8.4.4 Usage of Operations

In order to update the argument *state* at *path* using *item* or $k \ \wedge \ v$ the first step is always retrieving the value at *path* using the operation $atKey$. This operation is used because by definition, *state* is a KV

$$state_{path} = atKey(state, path)$$

to determine its type

$$state_{path} = Object \ \vee \ KV \ \vee \ x \ \vee X$$

such that the following bullet points represent the beahvior of *accumulate* under various conditions

- $object?\,(state_{path}) = true$
  - and $item$ passed in as argument

$$updatedState = associate(state, path, item)$$

  - and $k \wedge v$ passed in as argument

$$index = count(path)$$

$$fullPath = append(path, k, index)$$

$$updatedState = associate(state, fullPath, v)$$

- $array?\,(state_{path}) = true$
  - and $item$ passed in as argument

$$index = count(state_{path})$$

$$updatedArray = append(state_{path}, item, index)$$

$$updatedState = associate(state, path, updatedArray)$$

  - and $k \ \wedge \ v$ passed in as argument

$$updatedArray = append(state_{path}, v, k)$$

$$updatedState = associate(state, path, updatedArray)$$

- $array?\,(state_{path}) = false \ \wedge \ object?\,(state_{path}) = false$
  - and $item$ passed in as argument

$$updatedState = associate(state, path, item)$$

  - and $k \ \wedge \ v$ passed in as argument

$$newArray = append(<>, state_{path}, 0)$$

$$updatedArray = append(newArray, v, k)$$

$$updatedState = associate(state, path, updatedArray)$$

Which shows that $accumulate$ has common steps across all conditions

$$state_{path} = atKey(state, path)$$

$$objectAtPath? = object?\,(state_{path})$$

$$arrayAtPath? = array?\,(state_{path})$$

but then the steps deviate based $item$ vs $k \ \wedge \ v$ such that the action of $accumulate$ when $item$ is passed in results in either

- an overwrite of $state_{path}$ via $associate(state, path, item)$
  - $objectAtPath? = true$
  - $objectAtPath? = false \ \wedge \ arrayAtpath? = false$
- an updated $state_{path}$ via $associate(state, path, append(state_{path}, item, count(state_{path})))$
  - $arrayAtPath? = true$

and the action of $accumulate$ when $k \ \wedge \ v$ is passed in results in either

- $objectAtPath? = true$
  - an update of $state_{path}$ to include $k \mapsto v$

$$associate(state, append(path, k, count(path)), v)$$

- $arrayAtPath? = true$
  - an update of $state_{path}$ to include v at index k

$$associate(state, path, append(state_{path}, v, k))$$

- $objectAtPath? = false \ \wedge \ arrayAtpath? = false$
  - creation of a new array which contains $state_{path}$ and v at index k

$$associate(state, path, append(append(<>, state_{path}, 0), v, k))$$

8.4.5 Example output

To demonstrate the functionality of $accumulate$, the following assumptions will be made

$$state =< a \mapsto< b \mapsto< 1, 2, 3 >, c \mapsto 4 > d \mapsto foo, e \mapsto< 4, 5, 6 >>$$

$$\Rightarrow$$

$$state_a =< b \mapsto< 1, 2, 3 >, c \mapsto 4 >$$

$$state_d = foo$$

$$state_e =< 4, 5, 6 >$$

such that

$$accumulate(state, < d >, baz) =< state_a, \ d \mapsto baz, \ state_e >$$

and

$$accumulate(state, < a >, baz) =< a \mapsto baz, \ state_d, \ state_e >$$

and

$$accumulate(state, < a, c >, baz) =< a \mapsto< b \mapsto< 1, 2, 3 >, c \mapsto baz >, \ state_d, \ state_e >$$

and

$$accumulate(state, < e >, 7) =< state_a,\ state_d,\ e \mapsto < 4, 5, 6, 7 >>$$

and

$$accumulate(state, < e >, < 7, 8, 9 >) =< state_a,\ state_d,\ e \mapsto < 4, 5, 6, < 7, 8, 9 >>>$$

and

$$accumulate(state, < a >, b, < 3, 2, 1 >) =< a \mapsto < b \mapsto < 3, 2, 1 >, c \mapsto 4 >,\ state_d,\ state_e >$$

and

$$accumulate(state, < a >, q, baz) =< a \mapsto < b \mapsto < 1, 2, 3 >, c \mapsto 4, q \mapsto baz >,\ state_d,\ state_e >$$

and

$$accumulate(state, < a, q >, r, baz) =< a \mapsto < b \mapsto < 1, 2, 3 >, c \mapsto 4, q \mapsto r \mapsto baz >,\ state_d,\ state_e >$$

and

$$accumulate(state, < e >, 1, 7) =< state_a,\ state_d,\ e \mapsto < 4, 7, 5, 6 >>$$

and

$$accumulate(state, < d >, 0, baz) =< state_a,\ < baz, foo >,\ state_e >$$

and

$$accumulate(state, < d >, 1, < baz, bar >) =< state_a,\ < foo, < baz, bar >>,\ state_e >$$

8.5 At JSONPath

Performs a lookup at *path* within *source* similar to atKey

$$atJsonPath(source, path)$$

such that the fundamental functionality of JSONPath is covered in this definition.

- A more complete definition will come at a future date if/as necessary

8.5.1 Arguments

- *source* is an object Scalar, KV, Statement or an Algorithm State
- *path* is a JSONPath string which adheres to the additional requirements, clarifications, and additions placed on JSONPath by the xAPI Profile Specification

8.5.2 Relevant Operations

The primitive $atJsonPath$ uses the operations

- atKey
- atIndex
- append
- count

8.5.3 Summary

$atJsonPath$ will return a v found within $source$ after converting

$$path \rightarrow < path_{i+1}..path_j >$$

such that if

$$path = \$.a.b$$

then

$$path \rightarrow < a, b >$$

so that

$$atJsonPath(< a \mapsto b \mapsto 123 >, \$.a.b) = 123$$

8.5.4 Usage of Operations

In order to convert

$$path \rightarrow < path_{i+2}..path_j >$$

an empty Collection $keyState$ is introduced

$$keyState = <>$$

so that the relevant k'(s) can be stored in $keyState$ during iteration over $path$

$$\forall n : i..j \ \bullet i = 0 \ \wedge j = count(path) - 1$$

and the number of stored keys can be tracked using $curKeyStateIndex$

$$curKeyStateIndex = count(keyState) - 1$$

such that the current $path_n$ can be retrieved

$$curKey = atIndex(path, n)$$

and $keepKey?$ can indicate the relevance of $path_n$

$$keepKey? = true \iff curKey \neq \$ \ \wedge curKey \neq .$$

such that during each iteration n, $keyState$ will be updated if necessary

$keyState = append(keyState,\ curKey,\ curKeyStateIndex) \iff keepKey? = true$

so at the end of the loop

$$keyState =< path_{i+2}..path_n..path_j >$$

which provides the Collection of Key(s) necessary for calling $atKey$

$$valueInSource = atKey(source, keyState)$$

such that

$$atJsonPath(source, path) \equiv atKey(source, keyState)$$

8.5.5 Example output

Given an example $source$

$$source =< a \mapsto< b \mapsto 123, c \mapsto 456 >, d \mapsto foo >$$

then

$$atJsonPath(source, \$.a) =< b \mapsto 123, c \mapsto 456 >$$

and

$$atJsonPath(source, \$.a.b) = 123$$

and

$$atJsonPath(source, \$.a.c) = 456$$

and

$$atJsonPath(source, \$.d) = foo$$

Updated Algorithm Definitions

The following are examples of the new way in which Algorithms were defined. These sections are either in draft form or are a work in progress.

9 Rate of Completions

As learners engage in activities supported by a learning ecosystem, they will build up a history of learning experiences. When the digital resources of that learning ecosystem adhere to a framework dedicated to supporting and understanding the learner, such as the Total Learning Architecture (TLA), it becomes possible to retell their learning story through data and data visualization. One important aspect of that story is the rate of completion of the various digital resources within the learning ecosystem.

9.1 Initialization

$init(state)$ sets up an empty KV within $state$ for the Algorithm to update at each $step$

$$init(state) = state_0$$

where

$$state_0 = associate(state, < state, completions >, <>) \iff atKey(state, < state, completions >) = nil$$

otherwise

$$state_0 = state$$

such that if

$$state =< a \mapsto b >$$

then

$$state_0 =< a \mapsto b, state \mapsto completions \mapsto <>>$$

9.2 Relevant?

$relevant?(state, stmt)$ determines if $stmt$ is valid for use within $step$ of $rateOfCompletions$ and does so by looking into various $k \to v$ within $stmt$. The following Primitives are used as the $body$ of $relevant?(state, stmt)$

- is the Object of the Statement an Activity?

$$activityType = atKey(stmt, < object, objectType >)$$

$$activity?(activityType) = true \iff activityType = Activity \lor activityType = nil$$

- is the Verb indicative of a completion event?

$$verbId = atKey(stmt, < verb, id >)$$

$$completionVerb?(verbId) = true$$

$$\iff$$

$$verbId = http://adlnet.gov/expapi/verbs/passed$$

$$\vee$$

$$verbId = https://w3id.org/xapi/dod-isd/verbs/answered$$

$$\vee$$

$$verbId = http://adlnet.gov/expapi/verbs/completed$$

- does the *stmt* indicate completion using Result?

$$result = atKey(stmt, < result, completion >)$$

$$resultCompletion = true \iff result = true$$

such that the body of *relevant*? contains

$$p_a(stmt) = activity?\,(atKey(stmt, < object, objectType >))$$

and

$$p_v(stmt) = completionVerb?\,(atKey(stmt, < verb, id >))$$

and

$$p_r(stmt) = resultCompletion(atKey(stmt, < result, completion >))$$

which are used to form higher level Primitives

$$p_{continue}(stmt) = stmt \iff p_a(stmt) = true$$

and

$$p_{completed?}(stmt) = stmt \iff p_v(stmt) = true \ \vee \ p_r(stmt) = true$$

which results in a final Primitive $p_{return?}$

$$p_{return?}(stmt) = object?\,(p_{completed?}(p_{continue}(stmt)))$$

which defines the *body* of *relevant*?

$$relevant?\,(stmt) = p_{return?}(stmt) \Rightarrow object?\,(p_{completed?}(p_{continue}(stmt)))$$

and can be summarized as

$$relevant?\,(state, stmt) = true$$

$$\iff$$

$$activity?\,(activitType) = true$$

$$\wedge$$

$$completionVerb?\,(verbId) = true \ \vee \ resultCompletion = true$$

9.3 Accept?

$rateOfCompletions$ does not require further boolean logic to determine if $stmt$ and $state$ can be passed to $step$

$$accept?\,(state, stmt) = object?\,(stmt)$$

which should always return true assuming valid xAPI Statements are passed to $rateOfCompletions$

9.4 Step

9.4.1 summary

$step(state, stmt)$ updates $state$ to include

$$\$.object.id \mapsto < domain, statementCount, name >$$

where

$$domain \mapsto < start, end >$$

$$statementCount \mapsto \mathbb{R}$$

$$name \mapsto < \$.object.definition.name >$$

at

$$< state, completions, \$.object.id >$$

9.4.2 processing

$step$ starts by extracting the relevant information from $stmt$

- $currentTime$

$$currentTime = atKey(stmt, timestamp)$$

- $name$

$$name_{stmt} = atKey(stmt, < object, definition, name >)$$

- $objectId$

$$objectId = atKey(stmt, < object, id >)$$

which allows for the previous $state$ to be resolved using $objectId$

- $domain$

$$domain_{state} = atKey(state, < state, completions, objectId, domain >)$$

$$start_{state} = first(domain_{state})$$

$$end_{state} = last(domain_{state})$$

- *statementCount*

  $$statementCount_{state} = atKey(state, < state, completions, objectId, statementCount >)$$

- *name*

  $$name_{state} = atKey(state, < state, completions, objectId, name >)$$

so that the previous state can be used along side the information parsed from *stmt*

- does $start_{state}$ need to be updated to $currentTime$?

  where
  $$inSeconds_{stmt} = isoToUnixEpoch(currentTime)$$
  $$inSeconds_{start} = isoToUnixEpoch(start_{state}) \iff start_{state} \neq nil$$
  such that
  $$start(state, stmt) = currentTime$$
  $$\iff$$
  $$start_{state} = nil$$
  $$\vee$$
  $$inSeconds_{stmt} \leq inSeconds_{start}$$
  otherwise
  $$start(state, stmt) = start_{state}$$

- does end_{state} need to be updated to $currentTime$?

  where
  $$inSeconds_{stmt} = isoToUnixEpoch(currentTime)$$
  $$inSeconds_{end} = isoToUnixEpoch(end_{state}) \iff end_{state} \neq nil$$
  such that
  $$end(state, stmt) = currentTime$$
  $$\iff$$
  $$end_{state} = nil$$
  $$\vee$$
  $$inSeconds_{stmt} \geq inSeconds_{end}$$
  otherwise
  $$end(state, stmt) = end_{state}$$

- what should *statementCount* be?

$$nStmts(state) = 1 \iff statementCount_{state} = 0 \ \vee \ nil$$

$$\vee$$

$$nStmts(state) = 1 + statementCount_{state} \iff statementCount_{state} \geq 1$$

- do we need to add a new *name*?

$$allNames(state, stmt) = append(name_{state}, name_{stmt}, count(name_{state}))$$

$$\iff$$

$$name_{stmt} \notin name_{state}$$

otherwise

$$allNames(state, stmt) = name_{state}$$

which allows for the following primitives to be defined

$$p_{start}(state, stmt) = start(state, stmt)$$

$$p_{end}(state, stmt) = end(state, stmt)$$

$$p_{stmtCount}(state, stmt) = nStmts(state)$$

$$p_{names}(state, stmt) = allNames(state, stmt)$$

and establish relevant paths into *state*

$$K_{domain} =< state, completions, objectId, domain >$$

$$K_{stmtCount} =< state, completions, objectId, statementCount >$$

$$K_{names} =< state, completions, objectId, name >$$

which are used within higher level primitives concerned with updating *state*

$$p_{updateStart}(state, stmt)$$

$$\equiv$$

$$associate(state, K_{domain}, append(remove(domain_{state}, 0), p_{start}(state, stmt), 0))$$

and

$$p_{updateEnd}(state, stmt)$$

$$\equiv$$

$$associate(state, K_{domain}, append(remove(domain_{state}, 1), p_{end}(state, stmt), 1))$$

and

$$p_{updatedCount}(state, stmt)$$

$$\equiv$$

$$associate(state, K_{stmtCount}, p_{stmtCount}(state, stmt))$$

and

$$p_{updatedNames}(state, stmt)$$
$$\equiv$$
$$associate(state, K_{names}, p_{names}(state, stmt))$$

such that *body* of *step* is defined as

$$step(state, stmt) = p_{updateNames}(p_{updateCount}(p_{updateEnd}(p_{updateStart}(state, stmt), stmt), stmt), stmt)$$

where

$$state' = p_{updateStart}(state, stmt)$$

and

$$state'' = p_{updateEnd}(state', stmt)$$

and

$$state''' = p_{updateCount}(state'', stmt)$$

such that

$$step(state, stmt) = p_{updateNames}(state''', stmt)$$

9.5 Result

The only *opts* used by $rateOfCompletions$ is $timeUnit$

$$timeUnit = second \ \vee minute \ \vee \ hour \ \vee day \ \vee month \ \vee year$$

and will default to *day* if not passed to $rateOfCompletions$

$$result(state) = result(state, < timeUnit \mapsto day >)$$

which is passed to $rateOf$ along with the arguments parsed from *state*

$$unit = atKey(opts, timeUnit)$$

$$allCompletions(state) = atKey(state, < state, completions >)$$

such that

$$\forall k_n : i..n..j \in allCompletions(state)$$

the following primitives are called each iteration

$$getCount(state, k_n) = atKey(allCompletions(state), < k_n, statementCount >)$$

$$getStart(state, k_n) = atKey(allCompletions(state), < k_n, domain, start >)$$
$$getEnd(state, k_n) = atKey(allCompletions(state), < k_n, domain, end >)$$
$$getName(state, k_n) = atKey(allCompletions(state), < k_n, name >)$$

which allows for

$$rate_n(state, k_n, unit) = rateOf(getCount(state, k_n), getStart(state, k_n), getEnd(state, k_n), unit)$$

such that

$$value_n(state, k_n, unit) =< x_n, y_n >$$

where

$$name_n(state, k_n) = first(getName(state, k_n))$$

$$x_n = x \mapsto name_n(state, k_n) \iff name_n(state, k_n) \neq nil$$

otherwise

$$x_n = x \mapsto k_n$$

and

$$y_n = y \mapsto rate_n(state, k_n, unit)$$

such that

$$value_n(state, k_n, unit) =< name_n(state, k_n),\ rate_n(state, k_n, unit) >$$

and

$$value(state, unit) = \forall k_n : i..n..j \in allCompletions(state) \exists!\, value_n(state, k_n, unit) =< x_n, y_n >$$

$$\Rightarrow$$

$$value(state, unit) =< value_i(state, k_i, unit)..value_n(state, k_n, unit)..value_j(state, k_j, unit) >$$

which allows the body of *result* to be defined using

$$unit = atKey(opts, timeUnit)$$

$$K_{store} =< state, completions, values, unit >$$

so that *result* returns an updated *state* with the rate of completions per *unit* located at K_{store}

$$result(state, opts) = associate(state, K_{store}, value(state, unit))$$

10 Timeline Of Learner Success

Intro text about the Algorithm

10.1 Initialization

What does $state_0$ look like?

10.2 Relevant?

What primitives are used to determine if a Statement is relevant

10.3 Accept?

What primitives are used to determine if a Statement is accepted

10.4 Step

What primitives are used to process a Statement to update *state*

10.5 Result

What *opts* are used if any + what does the *state* look like?

11 Which Assessment Questions are the Most Difficult

Intro text about the Algorithm

11.1 Initialization

What does $state_0$ look like?

11.2 Relevant?

What primitives are used to determine if a Statement is relevant

11.3 Accept?

What primitives are used to determine if a Statement is accepted

11.4 Step

What primitives are used to process a Statement to update *state*

11.5 Result

What *opts* are used if any + what does the *state* look like?

12 How Often are Recommendations Followed

Intro text about the Algorithm

12.1 Initialization

What does $state_0$ look like?

12.2 Relevant?

What primitives are used to determine if a Statement is relevant

12.3 Accept?

What primitives are used to determine if a Statement is accepted

12.4 Step

What primitives are used to process a Statement to update *state*

12.5 Result

What *opts* are used if any + what does the *state* look like?

Previous Algorithm Definitions

The following are examples of the previous way in which Algorithms were defined.

13 Rate of Completions

As learners engage in activities supported by a learning ecosystem, they will build up a history of learning experiences. When the digital resources of that learning ecosystem adhere to a framework dedicated to supporting and understanding the learner, such as the Total Learning Architecture (TLA), it becomes possible to retell their learning story through data and data visualization. One important aspect of that story is the rate of completion[1] of the various digital resources within the learning ecosystem.

13.1 Ideal Statements

In order to accurately portray the rates of completion, there are a few base requirements of the data produced by a Learning Record Provider (LRP). They are as follows:

- statements describing a learner completing an activity should[2] use the verb $http://adlnet.gov/expapi/verbs/completed$
- statements describing a learner completing an activity should report if the learner was successful or not via $\$.result.success$
- statement describing a learner completing a scored activity should report the learners score via $\$.result.score.raw$, $\$.result.score.min$ and $\$.result.score.max$
- activites must be uniquely and consistently identified across all statements
- The time at which a learner completed a learning activity must be recorded
  - The timestamp should contain an appropriate level of specificity.
  - ie. Year, Month, Day, Hour, Minute, Second, Timezone
- statements describing a learner completing an activity should report the amount of time taken to complete the activity via $\$.result.duration$

13.2 Input Data Retrieval

How to query an LRS via a GET request to the Statements Resource via curl. The following section contains the appropriate parameters with example values as well as the curl command necessary for making the request.[3][4][5]

[1] Completion can be defined by the presence of the verb completed or by the presence of $\$.result.completion$ set equal to true. In this algorithm, completion is defined by the presence of the verb completed regardless of $\$.result.completion$. This decision affects how statements are retrieved and filtered. In the case where completion is defined by $\$.result.completion$, the query to the LRS would not include the verb paramater and there would need to be a filtering process which looks for the presence of $\$.result.completion$ = true

[2] See footnote 4

[3] See footnote 1.

[4] See footnote 2.

[5] See footnote 3.

```
Verb = "verb=http://adlnet.gov/expapi/verbs/completed"

Since = "since=2018-07-20T12:08:47Z"

Until = "until=2018-07-21T12:08:47Z"

Base = "https://example.endpoint/statements?"

endpoint = Base + Verb + "&" + Since + "&" + Until

Auth = Hash generated from basic auth

S = curl -X GET -H "Authorization: Auth"
         -H "Content-Type: application/json"
         -H "X-Experience-API-Version: 1.0.3"
         Endpoint
```

13.3 Statement Parameters to Utilize

The statement parameter locations here are written in JSONPath. This notation is also compatable with the xAPI Z notation due to the defined hierarchy of components. Within the Z specifications, a variable name will be used instead of the $

- $\$.timestamp$
- $\$.object.id$

13.4 2018 Pilot TLA Statement Problems

The initial pilot test data supports the core requirements of this algorithm but completion statements only reports completion scores via $\$.result.scaled$ instead of $\$.result.score.raw$, $\$.result.score.min$ and $\$.result.score.max$.[6] Given that the offical 2018 pilot test is scheduled to take place on July 27th, 2018, this section may require updates pending future data review.

13.5 Summary

1. Query an LRS via a GET request to the statemetns endpoint using the paramters verb, since and until.

[6] The one potential issue with using scaled score is the calculation of scaled is not stricly defined by the xAPI specification but is instead up to the authors of the LRP. This results in the inability to reliably compare scaled scores across LRPs. if $\$.result.score.raw$, $\$.result.score.min$ and $\$.result.score.max$ are reported for all questions, it becomes possible to reliably compare scores across LRPs by generating a scaled score in a consistent way.

2. group statements by their $\$.object.id$

3. select time range unit for use within rate calculation. Will default to day.

4. determine the amount of time between the first and last instance of a $\$.object.id$ (in seconds) and divide it by the time unit. ie if the unit is minute, you would divide by 60.

5. calculate the rate by dividing the count of a group (2) by the number of time units covered by the statements (4) so that the rate is the number of completions per activity per time unit.

13.6 Formal Specification

13.6.1 Basic Types

$TIMEUNIT ::= \{second\}|\{minute\}|\{hour\}|\{day\}|\{week\}|\{month\}|\{year\}$

13.6.2 System State

$$
\begin{array}{|l}
\hline
RateOfCompletion \\
Statements \\
S_{completions} : \mathbb{F}_1 \\
S_{grouped}, S_{timeunit}, S_{processed} : \mathbb{F} \\
\hline
S_{completions} = statements \\
S_{grouped} = \{byId : \mathrm{seq}_1\ statement\} \\
S_{withRate} = \{byGroup : (\mathrm{seq}_1\ statement, \mathbb{N})\} \\
S_{processed} = \{rate : (id, \mathbb{N}, TIMEUNIT)\} \\
\hline
\end{array}
$$

- The set $S_{completions}$ is a non-empty, finite set and is the component *statements* which contains the results of the query to the LRS.

- The sets $S_{grouped}$, $S_{withRate}$ and $S_{processed}$ are all finite sets

- the set $S_{grouped}$ is a finite set of objects *byId* which are non-empty, finite sequences of the component *statement*

- the set $S_{withRate}$ is a finite set of objects *byGroup* which are ordered pairs of non-empty, finite sequences of the component *statement* and a natural number

- the set $S_{processed}$ is a finite set of objects *rate* where each contains the component *id*, a natural number and the type $TIMEUNIT$

13.6.3 Initial System State

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad InitRateOfCompletion \quad}} \\
RateOfCompletion \\
T : TIMEUNIT \\
\hline
S_{completions} \neq \emptyset \\
S_{grouped} = \emptyset \\
S_{withRate} = \emptyset \\
S_{processed} = \emptyset \\
T = \{day\} \\
\hline
\end{array}
$$

- The set $S_{completions}$ is a non-empty set which contains the results of the GET request(s) to the LRS
- The sets $S_{grouped}$, $S_{withRate}$ and $S_{processed}$ are all initially empty
- the variable T has the type $TIMEUNIT$ and the value $\{day\}$

13.6.4 Calculate Rate

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad IsoToUnix \quad}} \\
convert : \mathbb{F}_1 \rightarrow \mathbb{N}\#1 \\
c? : \mathbb{F}_1 \\
c! : \mathbb{N}\#1 \\
\hline
c! = convert(c?) \\
\hline
\end{array}
$$

- The schema $IsoToUnix$ introduces the function *convert* which takes in a finit set of one thing (a timestamp) and converts it to a single natural number.
- the purpose of this function is to convert an ISO 8601 timestamp to the Unix epoch. The concrete definition of the conversion is outside the scope of this document
  - The Unix epoch is the number of seconds that have elapsed since January 1, 1970 (midnight UTC/GMT), not counting leap seconds.

$$
\begin{array}{|l}
\hline
\textit{CalcRateByUnit} \\
\textit{Statement} \\
\textit{IsoToUnix} \\
\textit{CountPerGroup} \\
\textit{unit}? : \textit{TIMEUNIT} \\
s?\,,s! : \mathbb{F} \\
r : \mathbb{N} \\
\textit{rate} : (\mathbb{F}, \textit{TIMEUNIT}) \to \mathbb{F} \\
\hline
\textit{unit}? = \{\textit{second}\} \Rightarrow 1 \vee \{\textit{minute}\} \Rightarrow 60 \vee \{\textit{hour}\} \Rightarrow 3600 \vee \\
\qquad\qquad \{\textit{day}\} \Rightarrow 86400 \vee \{\textit{week}\} \Rightarrow 604800 \vee \\
\qquad\qquad \{\textit{month}\} \Rightarrow 2629743 \vee \{\textit{year}\} \Rightarrow 31556926 \\
s? = \{g : \textit{seq}_1\,\textit{statement}\} \\
s! = \textit{rate}(s?\,, \textit{unit}?\,) \\
s! = \{s : (g, r) \mid \forall g_n : g_i .. g_j \bullet i \leq n \leq j \bullet \exists\, s_n : (g_n, r_n) \bullet \\
\qquad r_n = \textit{count}(g_n) \div ((\textit{convert}(\textit{last}\, g_n.\textit{timestamp}) - \textit{convert}(\textit{head}\, g_n.\textit{timestamp})) \div \textit{unit}?\,)\} \\
\hline
\end{array}
$$

- The schema *CalcRateByUnit* introduces the function *rate* where the input $s?$ is a set of objects g which are each a non-empty, finite sequence of statements and the input *unit*? represents a unit of time.
- for every g_n within the range $g_i..g_j$, there exists an associated object s_n which is an orderd pair of (g_n, r_n) where r_n is equal to the number of items within g_n divided by the number of *unit*?s within the time range of $\textit{last}\, g_n.\textit{timestamp} - \textit{head}\, g_n.\textit{timestamp}$
- the output of the function *rate* is $s!$, the set of all s_n

13.6.5 Processes Results

$$
\begin{array}{|l}
\hline
\textit{AggergateCompletionStatements} \\
\Delta RateOfCompletion \\
GroupByActivityId \\
CalcRateByUnit \\
grouped, processed, withRate : \mathbb{F} \\
r : \mathbb{N} \\
T? : TIMEUNIT \\
\hline
T? = \{day\} \\
grouped = \emptyset \\
grouped' = group(S_{completions}) \\
S'_{grouped} = S_{grouped} \cup grouped' \\
withRate \subseteq S'_{grouped} \\
withRate' = rate(withRate, T?) \\
S'_{withRate} = withRate' \cup S_{withRate} \\
processed \subseteq S'_{withRate} \\
processed' = \{p : (id, r, T?) \mid \\
\qquad \textbf{let } \{processed_i..processed_j\} == \{b_i..b_j\} \bullet \\
\qquad \forall b_n : b_i..b_j \bullet i \leq n \leq j \bullet \exists p_n : (id_n, r_n, T?) \bullet \\
\qquad id_n = (head\,(first\,b_n)).object.id \land \\
\qquad r_n = (second\,b_n)\} \\
S'_{processed} = processed' \cup S_{processed} \\
\hline
\end{array}
$$

- The schema *AggergateCompletionStatements* outlines how to calculate the rate of completion per $\$.object.id$ per $second|minute|hour|day|week|month|year$

  1. $S'_{grouped}$ is the result of grouping the statements within $S_{completions}$ by their $\$.object.id$
  2. The groups from (1) are passed to the function *rate* with the variable $T?$ which controls the unit of time, ie per day vs per week
  3. the result of (2) is then processed to create a triplet of $\$.object.id$, rate, unit of time for all unique $\$.object.id$ within $S_{completions}$

13.6.6 Return

$$
\begin{array}{|l}
\hline
\textit{ReturnAggergateCompletionStatements} \\
\Xi RateOfCompletion \\
AggergateCompletionStatements \\
S_{processed}! : \mathbb{F} \\
\hline
S_{processed}! = S_{processed} \\
\hline
\end{array}
$$

- The return value $S_{processed}!$ is equal to $S_{processed}$ after the operation described by *AggergateCompletionStatements*

13.7 Pseudocode

Algorithm 1: Rate of Completions

```
Input: S_completed, timeUnit
Result: ratePerObjTu'
context = {};
ratePerObjTu = [];
while S_completion ≠ ∅ do
    foreach s ∈ S_completion do
        id ← s.object.id;
        ts ← convert(s.timestamp);
        if id ∉ context then
            do
            times = [ts];
            context' ← {id : times};
            S'_completion ← S_completion \ s;
            recur context', S'_completion;
        else
            do
            times' ← context.id ⌢ ts;
            context' ← {id : times'};
            S'_completion ← S_completion \ s;
            recur context', S'_completion;
        end
    end
end
foreach k ∈ context' do
    allTs ← context'.k;
    totalDuration ← max(allTs) − min(allTs);
    totalCount ← count(allTs);
    rate ← totalCount ÷ (totalDuration ÷ timeUnit);
    subVec = [k, rate, timeUnit];
    ratePerObjTu' ← ratePerObjTu ⌢ subVec;
    recur ratePerObjTu';
end
return ratePerObjTu'
```

- Values from Z schemas are used within this pseudocode
- the result of the algorithm is an array of arrays where each subarray contains a *statement.object.id*, the *rate* and the *timeUnit* used to calculate *rate*.

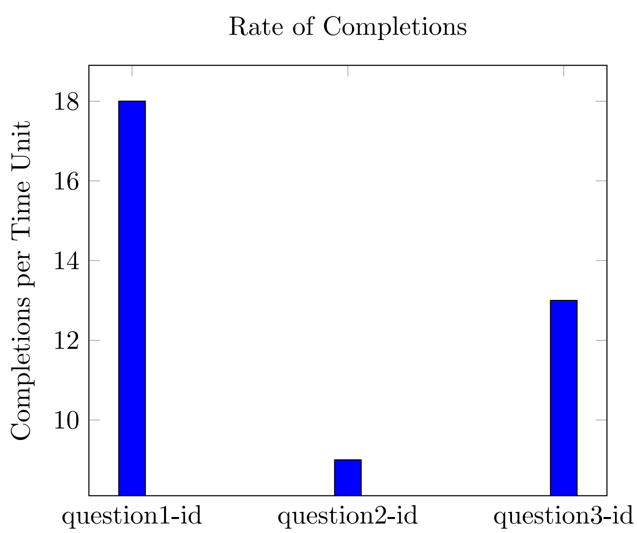
13.8 JSON Schema

```
{"type":"array",
   "items":{"type":"array",
      "items":[{"type":"string"}, {"type":"number"},
{"type":"string"}]}}
```

13.9 Visualization Description

The **Rate of Completions** visualization will be a bar chart where the domain consists of *statement.object.id* and the range is a number greater than 0 (the rate of completions for that *statement.object.id*). Every subarray within the array $ratePerObjTu$ will be a grouping within the bar chart. The pseudocode specifies an input paramter $timeUnit$ which controls the calculation of the rate (range of the visualization). $timeUnit$ could be per minute, per day, per week, etc.

13.10 Visualization prototype



13.11 Prototype Improvement Suggestions

Additional features may be implemented on top of this base specification but they would require adding aditional values to each subarray returned by the algorithm. These additional values can be retrieved via (1) performing metadata lookup within or independently of the algorithm (2) by utilizing additional xAPI statement paramters and/or (3) by performing additional computations. The following examples assume the metadata is contained within each statement available to the algorithm.

- use $statement.object.definition.name$ instead of $statement.object.id$ for x axis label
- populate a tooltip with the people who have completed the activity. This could also include the number of times they have completed it.
- populate a tooltip with the breakdown of which devices or platforms the activity was completed on. This would require the device type or platform to be reported within $statement.context.platform$
- populate a tooltip with the breakdown of percentage successful for all completions of the activty. This would require $statement.result.success$
- populate a tooltip with the breakdown of scores earned (if appliciable) for the completions. This would require $statement.result.score.raw$, $statement.result.score.min$ and $statement.result.score.max$
- populate a tooltip with the competency assocaited with the completed activities. The competency should be reported via $statement.context.contextActivities$
- populate a tooltip with the average duration spent to reach completions. This would require $statement.result.duration$ to be reported.

14 Timeline Of Learner Success

As learners engage in activities supported by a learning ecosystem, they will build up a history of learning experiences. When the digital resources of that learning ecosystem adhere to a framework dedicated to supporting and understanding the learner, such as the Total Learning Architecture (TLA), it becomes possible to retell their learning story through data and data visualization. One important aspect of that story is the' learners history of success.

14.1 Ideal Statements

In order to accurately portray a learner's timeline of success, there are a few base requirements of the data produced by a Learning Record Provider (LRP). They are as follows:

- the learner must be uniquely and consistently identified across all statements
- learning activities which evaluate a learner's understanding of material must report if the learner was successful or not
  - the grade earned by the learner must be reported
  - the minimum and maximum possible grade must be reported
- The learning activities must be uniquely and consistently identified across all statements
- The time at which a learner completed a learning activity must be recorded
  - The timestamp should contain an appropriate level of specificity.
  - ie. Year, Month, Day, Hour, Minute, Second, Timezone

14.2 Input Data Retrieval

How to query an LRS via a GET request to the Statements Resource via curl. The following section contains the appropriate parameters with example values as well as the curl command necessary for making the request.[7][8][9]

[7] S is the set of all statements parsed from the statements array within the HTTP response to the Curl request(s). It may be possible that multiple Curl requests are needed to retrieve all query results. If multiple requests are necessary, S is the result of concatenating the result of each request into a single set

[8] Querying an LRS will not be defined within the following Z specifications but the results of the query will be utilized

[9] If you want to query across the entire history of a LRS, omit Since and Until from the endpoint(s) and remove the associated & symbols.

```
Agent = "agent={"account":
  {"homePage": "https://example.homepage",
    "name": 123456}}"

Since = "since=2018-07-20T12:08:47Z"

Until = "until=2018-07-21T12:08:47Z"

Base = "https://example.endpoint/statements?"

endpoint = Base + Agent + "&" + Since + "&" + Until

Auth = Hash generated from basic auth

S = curl -X GET -H "Authorization: Auth"
-H "Content-Type: application/json"
-H "X-Experience-API-Version: 1.0.3"
Endpoint
```

14.3 Statement Parameters to Utilize

The statement parameter locations here are written in JSONPath. This notation is also compatable with the xAPI Z notation due to the defined hierarchy of components. Within the Z specifications, a variable name will be used instead of the $

- $\$.timestamp$
- $\$.result.success$
- $\$.result.score.raw$
- $\$.result.score.min$
- $\$.result.score.max$
- $\$.verb.id$

14.4 2018 Pilot TLA Statement Problems

The initial pilot test data supports this algorithm. This section may require updates pending future data review following iterations of the TLA testing.

14.5 Summary

1. Query an LRS via a GET request to the statements endpoint using the parameters agent, since and until

2. Filter the results to the set of statements where:
   - $.*verb.id* is one of:
     - http://adlnet.gov/expapi/verbs/passed
     - https://w3id.org/xapi/dod-isd/verbs/answered
     - http://adlnet.gov/expapi/verbs/completed
   - $.*result.success* is true
3. process the filtered data
   - extract $.*timestamp*
   - extract the score values from $.*result.score.raw*, $.*result.score.min* and $.*result.score.max* and convert them to the scale 0..100
   - create a pair of [$.*timestamp*, #]

14.6 Formal Specification

14.6.1 Basic Types

$COMPLETION ::==$
$\{http://adlnet.gov/expapi/verbs/passed\} \mid$
$\{https://w3id.org/xapi/dod-isd/verbs/answered\} \mid$
$\{http://adlnet.gov/expapi/verbs/completed\}$

$SUCCESS ::== \{true\}$

14.6.2 System State

$$
\begin{array}{|l}
\hline
\textit{TimelineLearnerSuccess} \\
Statements \\
S_{all} : \mathbb{F}_1 \\
S_{completion}, S_{success}, S_{processed} : \mathbb{F} \\
\hline
S_{all} = statements \\
S_{completion} \subseteq S_{all} \\
S_{success} \subseteq S_{completion} \\
S_{processed} = \{pair : (statement.timestamp, \mathbb{N})\} \\
\hline
\end{array}
$$

- The set S_{all} is a non-empty, finite set and is the component *statements*
- The sets $S_{completion}$ and $S_{success}$ are both finite sets
- the set $S_{completion}$ is a subset of S_{all} which may contain every value within S_{all}
- the set $S_{success}$ is a subset of $S_{completion}$ which may contain every value within $S_{completion}$
- the set $S_{processed}$ is a finite set of pairs where each contains a *statement.timestamp* and a natural number

14.6.3 Initial System State

$$\begin{array}{|l}
\hline
\textit{InitTimelineLearnerSuccess} \\
\textit{TimelineLearnerSuccess} \\
\hline
S_{all} \neq \emptyset \\
S_{completion} = \emptyset \\
S_{success} = \emptyset \\
S_{processed} = \emptyset \\
\hline
\end{array}$$

- The set S_{all} is a non-empty set
- The sets $S_{completion}, S_{success}$ and $S_{processed}$ are all initially empty

14.6.4 Filter for Completion

$$\begin{array}{|l}
\hline
\textit{Completion} \\
\textit{Statement} \\
\textit{completion} : \textit{STATEMENT} \mathrel{+\!\!\!\rightarrow} \mathbb{F} \\
s? : \textit{STATEMENT} \\
s! : \mathbb{F} \\
\hline
s? = \textit{statement} \\
s! = \textit{completion}(s?) \\
\textit{completion}(s?) = \textbf{if } s?.\textit{verb.id} : \textit{COMPLETION} \\
\qquad\qquad\qquad\qquad \textbf{then } s! = s? \\
\qquad\qquad\qquad\qquad \textbf{else } s! = \emptyset \\
\hline
\end{array}$$

- The schema *Completion* inroduces the function *completion* which takes in the variable $s?$ and returns the variable $s!$
- The variable $s?$ is the component *statement*
- $s!$ is equal to $s?$ if $\$.verb.id$ is of the type $COMPLETION$ otherwise $s!$ is an empty set

$$\begin{array}{|l}
\hline
\textit{FilterForCompletion} \\
\Delta \textit{TimelineLearnerSuccess} \\
\textit{Completion} \\
\textit{completions} : \mathbb{F} \\
\hline
\textit{completions} \subseteq S_{all} \\
\textit{completions}' = \{ s : \textit{STATEMENT} \mid \textit{completion}(s) \neq \emptyset \} \\
S'_{completion} = S_{completion} \cup \textit{completions}' \\
\hline
\end{array}$$

- the set *completions* is a subset of S_{all} which may contain every value within S_{all}

- The set $completions'$ is the set of all statements s where the result of $completion(s)$ is not an empty set
- the updated set $S'_{completion}$ is the union of the previous state of set $S_{completion}$ and the set $completions'$

14.6.5 Filter for Success

$$
\begin{array}{l}
\underline{\quad Success \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad} \\
Statement \\
success : STATEMENT \rightarrow \mathbb{F} \\
s? : STATEMENT \\
s! : \mathbb{F} \\
\hline
s? = statement \\
s! = success(s?) \\
success(s?) = \textbf{if } s?.result.success : SUCCESS \\
\qquad\qquad\qquad\qquad \textbf{then } s! = s? \\
\qquad\qquad\qquad\qquad \textbf{else } s! = \emptyset \\
\hline
\end{array}
$$

- the schema $Success$ introduces the function $success$ which takes in the variable $s?$ and returns the variable $s!$
- the variable $s?$ is the component $statement$
- $s!$ is equal to $s?$ if $\$.result.success$ is of the type $SUCCESS$ otherwise $s!$ is an empty set

$$
\begin{array}{l}
\underline{\quad FilterForSuccess \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad} \\
\Delta TimelineLearnerSuccess \\
Success \\
successes : \mathbb{F} \\
\hline
successes \subseteq S_{completion} \\
successes' = \{ s : STATEMENT \mid success(s) \neq \emptyset \} \\
S'_{success} = S_{success} \cup successes' \\
\hline
\end{array}
$$

- the set $successes$ is a subset of $S_{completion}$ which may contain every value within $S_{completion}$
- The set $successes'$ contains elements s of type $STATEMENT$ where $success(s)$ is not an empty set
- The updated set $S'_{success}$ is the union of the previous state of $S_{success}$ and $successes'$

14.6.6 Processes Results

$$
\begin{array}{|l}
\hline
\textit{Scale} \\
scaled! : \mathbb{N} \\
raw?, min?, max? : \mathbb{Z} \\
scale : \mathbb{Z} \rightarrow \mathbb{N} \\
\hline
scaled! = scale(raw?, min?, max?) \\
scale(raw?, min?, max?) = \\
\qquad\qquad\qquad (raw? * ((0.0 - 100.0)\ div\ (min? - max?))) + \\
\qquad\qquad\qquad (0.0 - (min? * ((0.0 - 100.0) div (min? - max?)))) \\
\hline
\end{array}
$$

- The schema *Scale* introduces the function *scale* which takes 3 arguments, *raw?*, *min?* and *max?*. The function converts *raw?* from the range $min? .. max?$ to 0.0..100.0

$$
\begin{array}{|l}
\hline
\textit{ProcessStatements} \\
\Delta TimelineLearnerSuccess \\
Scale \\
FilterStatements \\
processed : \mathbb{F} \\
\hline
processed \subseteq S_{success} \\
processed' = \{p : (\mathbb{F}_1 \#1, \mathbb{N}) \mid \\
\qquad\qquad \mathbf{let}\ \{processed_i .. processed_j\} == \{s_i .. s_j\} \bullet \\
\qquad\qquad i \leq n \leq j \bullet \forall s_n : s_i .. s_j \bullet \exists p_n : p_i .. p_j \bullet \\
\qquad\qquad first\ p_n = s_n.timestamp \wedge \\
\qquad\qquad second\ p_n = scale(s_n.result.score.raw, \\
\qquad\qquad\qquad\qquad s_n.result.score.min, \\
\qquad\qquad\qquad\qquad s_n.result.score.max)\} \\
S'_{processed} = S_{processed} \cup processed' \\
\hline
\end{array}
$$

- The operation *ProcessStatements* introduces the variable *processed* which is a subset of $S_{success}$ which may contain every value within $S_{success}$
- $S_{success}$ is the result of the operation *FilterStatements*
- The operation defines the variable $processsed'$ which is a set of objects p which are ordered pairs of (1) a finite set containing one value and (2) a single positive number.
- The first component of every object p, is the timestamp from the associated *statement* within *processed* ie. *s.timestamp*
- The second component of every object p is the result of the function *scale*. The score values contained within the associated *statement s* are the arguments passed to *scale*. ie $scale(s.result.score.raw, s.result.score.min, s.result.score.max)$
- The result of the operation *ProcessStatements* is to updated the set $S_{processed}$ with the values contained within $processed'$

14.6.7 Sequence of Operations

$FilterStatements \mathrel{\widehat=} FilterForCompletion \mathbin{\S} FilterForSuccess$

- The schema $FilterStatements$ is the sequential composition of operation schemas $FilterForCompletion$ and $FilterForSuccess$
- $FilterForCompletion$ happens before $FilterForSuccess$

$ProcessedStatements \mathrel{\widehat=} FilterStatements \mathbin{\S} ProcessStatements$

- The schema $ProcessedStatements$ is the sequential composition of operation schemas $FilterStatements$ and $ProcessStatements$
- $FilterStatements$ happens before $ProcessStatements$

14.6.8 Return

$$
\begin{array}{|l}
\hline
\textit{Return} \\
\Xi TimelineLearnerSuccess \\
ProcessedStatements \\
S_{processed}! : \mathbb{F} \\
\hline
S_{processed}! = S_{processed} \\
\hline
\end{array}
$$

- The returned variable $S_{processed}!$ is equal to the current state of variable $S_{processed}$ after the operations $FilterForCompletion$, $FilterForSuccess$ and $ProcessStatements$

14.7 Pseudocode

Algorithm 2: Timeline of Learner Success

```
Input: S_all
Result: coll′
coll = [];
while S_all ≠ ∅ do
    foreach s ∈ S_all do
        if s.verb.id = COMPLETION then
            do
            S′_completion ← s ∪ S_completion;
            S′_all ← S_all \ s ;
            recur S′_completion, S′_all;
        else
            do
            S′_all ← S_all \ s;
            recur S′_all;
        end
    end
end
while S′_completion ≠ ∅ do
    foreach sc ∈ S′_completion do
        if sc.result.success = SUCCESS then
            do
            S′_success ← sc ∪ S_success;
            S′_completion ← S_completion \ sc;
            recur S′_success, S′_completion;
        else
            do
            S′_completion ← S_completion \ sc;
            recur S′_completion;
        end
    end
end
foreach ss ∈ S′_success do
    raw? ← ss.result.score.raw;
    max? ← ss.result.score.max;
    min? ← ss.result.score.min;
    scaled ← scale(raw?, min?, max?);
    subVec ← [ss.timestamp, scaled];
    coll′ ← coll ⌢ subVec;
    recur coll′
end
return coll′
```

- The Z schemas are used within this pseudocode
- The return value coll is an array of arrays, each containing a *statement.timestamp* and a scaled score.

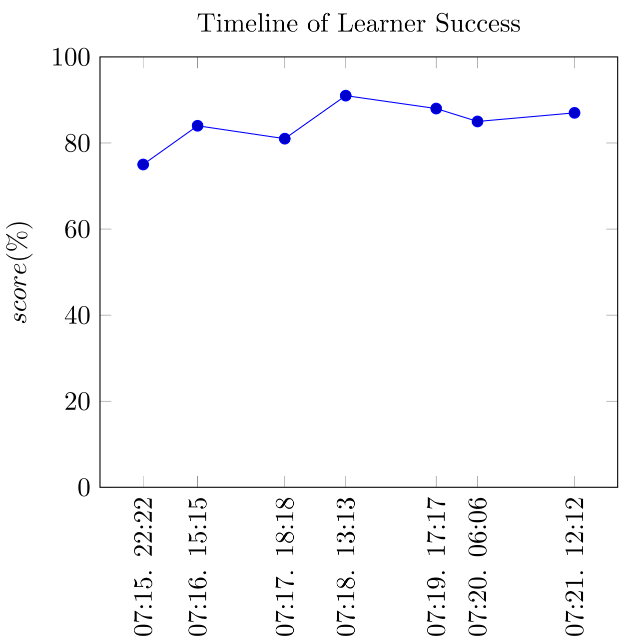
14.8 JSON Schema


```
{"type":"array",
    "items":{"type":"array",
        "items":[{"type":"string"}, {"type":"number"}]}}
```


14.9 Visualization Description

The **Timeline of Learner Success** visualization will be a line chart where the domain is time and the range is score on a scale of 0.0 to 100.0. Every subarray will be a point on the chart. The domain of the graph should be in chronological order.

14.10 Visualization prototype



14.11 Prototype Improvement Suggestions

Additional features may be implemented on top of this base specification but they would require adding aditional values to each subarray returned by the algorithm. These additional values can be retrieved via (1) performing metadata lookup within or independently of the algorithm (2) by utilizing additional xAPI statement paramters and/or (3) by performing additional computations. The following examples assume the metadata is contained within each statement available to the algorithm.

- A tooltip containing the name of an activity when hovering over a specific point on the chart
  - this would require utilizing $\$.object.definition.name$
- A tooltip containing the device on which the activity was experienced
  - this would require utilizing $\$.context.platform$
- A tooltip containing the instructor associated with a particular data point
  - this would require utilizing $\$.context.instructor$

15 Which Assessment Questions are the Most Difficult

As learners engage in activities supported by a learning ecosystem, they will experience learning content as well as assessment content. Assessments are designed to measure the effectiveness of learning content and help assess knowledge gained. It is possible that certain assessment questions do not accurately represent the concepts contained within learning content and this may be indicated by a majority of learners getting the question wrong. It is also possible that the question accurately represents the learning content but is very difficult. The following algorithm will identify these types of questions but will not be able to deduce why learners answer them incorrectly.

15.1 Ideal Statements

In order to accurately determine which assessment questions are the most dificult, there are a few requirements of the data produced by a LRP. They are as follows:

- statements describing a learner answering a question must report if the learner got the question correct or incorrect via $\$.result.success$
- if it is possible to get partial credit on a question, the amount of credit should be reported within the statement
  - the credit earned by the learner should be reported within $\$.result.score.raw$
  - the minimum and maximum possible credit amount should be reported within $\$.result.score.min$ and $\$.result.score.max$ respectively
- If it is possible to get partial credit on a question, it must still be reported if the learner reached the threshold of success via $\$.result.success$
- Statements describing a learner answering a question should contain activities of the type *cmi.interaction*
- activities must be uniquely and consistently identified across all statements
- Statements describing a learner answering a question should[10] use the verb $http://adlnet.gov/expapi/verbs/answered$

[10] it is possible to use another verb iri but if another is used, that will need to be accounted for in data retrieval

15.2 Input Data Retrieval

How to query an LRS via a GET request to the Statements Resource via curl. The following section contains the appropriate parameters with example values as well as the curl command necessary for making the request.[11][12][13]

```
Verb = "verb=http://adlnet.gov/expapi/verbs/answered"

Since = "since=2018-07-20T12:08:47Z"

Until = "until=2018-07-21T12:08:47Z"

Base = "https://example.endpoint/statements?"

endpoint = Base + Verb + "&" + Since + "&" + Until

Auth = Hash generated from basic auth

S = curl -X GET -H "Authorization: Auth"
-H "Content-Type: application/json"
-H "X-Experience-API-Version: 1.0.3"
Endpoint
```

15.3 Statement Parameters to Utilize

The statement parameter locations here are written in JSONPath. This notation is also compatable with the xAPI Z notation due to the defined hierarchy of components. Within the Z specifications, a variable name will be used instead of the $

- $\$.result.success$
- $\$.object.id$

15.4 2018 Pilot TLA Statement Problems

The initial pilot test data supports this algorithm. Given that the offical 2018 pilot test is scheduled to take place on July 27th, 2018, this section may require updates pending future data review.

15.5 Summary

1. Query an LRS via a GET request to the statements endpoint using the parameters verb, since and until

[11] See footnote 1.
[12] See footnote 2.
[13] See footnote 3.

2. Filter the results to the set of statements where:
   - $\$.result.success$ is false
3. process the filtered data
   - group by $\$.object.id$
   - determine the count of each group
   - create a collection of pairs = [$\$.object.id$, #]

15.6 Formal Specification

15.6.1 Basic Types

$INCORRECT ::= \{false\}$

15.6.2 System State

$$
\begin{array}{|l}
\hline
MostDifficultAssessmentQuestions \\
Statements \\
S_{all} : \mathbb{F}_1 \\
S_{incorrect}, S_{grouped}, S_{processed} : \mathbb{F} \\
\hline
S_{all} = statements \\
S_{incorrect} \subseteq S_{all} \\
S_{grouped} = \{groups : \mathrm{seq}_1\ statement\} \\
S_{processed} = \{pair : (id, \mathbb{N})\} \\
\hline
\end{array}
$$

- The set S_{all} is a non-empty, finite set and is the component *statements*
- The sets $S_{incorrect}$, $S_{grouped}$ and $S_{processed}$ are all finite sets
- the set $S_{incorrect}$ is a subset of S_{all} which may contain every value within S_{all}
- the set $S_{grouped}$ is a finite set of objects *groups* which are non-empty, finite sequences of the component *statement*
- the set $S_{processed}$ is a finite set of pairs where each contains the component *id* and a natural number

15.6.3 Initial System State

$$
\begin{array}{|l}
\hline
InitMostDifficultAssessmentQuestions \\
MostDifficultAssessmentQuestions \\
\hline
S_{all} \neq \emptyset \\
S_{incorrect} = \emptyset \\
S_{grouped} = \emptyset \\
S_{processed} = \emptyset \\
\hline
\end{array}
$$

- The set S_{all} is a non-empty set
- The sets $S_{incorrect}$, $S_{grouped}$ and $S_{processed}$ are all initially empty

15.6.4 Filter for Incorrect

$$
\begin{array}{|l}
\hline
\textit{Incorrect} \\
\textit{Statement} \\
incorrect : STATEMENT \nrightarrow \mathbb{F} \\
s? : STATEMENT \\
s! : \mathbb{F} \\
\hline
s? = statement \\
s! = incorrect(s?) \\
incorrect(s?) = \textbf{if } s?.result.success : INCORRECT \\
\qquad\qquad\qquad \textbf{then } s! = s? \\
\qquad\qquad\qquad \textbf{else } s! = \emptyset \\
\hline
\end{array}
$$

- the schema *Incorrect* introduces the function *incorrect* which takes in the variable $s?$ and returns the variable $s!$
- the variable $s?$ is the component *statement*
- $s!$ is equal to $s?$ if \$.*result.success* is of the type $INCORRECT$ otherwise $s!$ is an empty set

$$
\begin{array}{|l}
\hline
\textit{FilterForIncorrect} \\
\Delta MostDifficultAssessmentQuestions \\
\textit{Incorrect} \\
incorrects : \mathbb{F} \\
\hline
incorrects \subseteq S_{all} \\
incorrects' = \{s : STATEMENT \mid incorrect(s) \neq \emptyset\} \\
S'_{incorrect} = S_{incorrect} \cup incorrects' \\
\hline
\end{array}
$$

- the set *incorrects* is a subset of S_{all} which may contain every value within S_{all}
- The set $incorrects'$ contains elements s of type $STATEMENT$ where $incorrect(s)$ is not an empty set
- The updated set $S'_{incorrect}$ is the union of the previous state of $S_{incorrect}$ and $incorrects'$

15.6.5 Processes Results

$$
\begin{array}{|l}
\hline
\textit{GroupByActivityId} \\
\textit{Statements} \\
g? : \mathbb{F} \\
g! : \mathbb{F} \\
\textit{group} : \mathbb{F} \to \mathbb{F} \\
\hline
g? = \textit{statements} \Rightarrow \{g : \textit{statement}\} \\
g! = \textit{group}(g?) \\
g! = \{\textit{groups} : \mathrm{seq}_1\ \textit{statement} \mid \\
\quad \textbf{let}\ \mathrm{seq}_1\ \textit{statement}_i..\textit{statement}_j == \mathrm{seq}_1\ s_i..s_j \bullet \\
\quad \forall s_n : s_i..s_j \bullet i \leq n \leq j \bullet s_i.\textit{object}.\textit{id} = s_j.\textit{object}.\textit{id} = s_n.\textit{object}.\textit{id}\} \\
\hline
\end{array}
$$

- The schema *GroupByActivityId* introduces the function *group* which has the input of $g?$ and the output of $g!$
- The input variable $g?$ is the component *statements* which implies its a set of objects g which are each a *statement*
- the output variable $g!$ is a set of objects *groups* which are each a non-empty, finite sequence of *statement* where each member of the sequence $s_i..s_j$ has the same $\$.object.id$

$$
\begin{array}{|l}
\hline
\textit{CountPerGroup} \\
\textit{Statement} \\
c? : \mathrm{seq}_1\ \textit{statement} \\
c! : \mathbb{N} \\
\textit{count} : \mathrm{seq}_1\ \textit{statement} \to \mathbb{N} \\
\hline
c! = \textit{count}(c?) \\
c! \geq 1 \\
\textit{count}(c?) = \forall c_n? : \langle c?_i..c?_j \rangle \bullet i \leq n \leq j \wedge i = 0 \bullet \\
\qquad \exists_1\, c! : \mathbb{N} \bullet \textbf{if}\ n = i\ \textbf{then}\ c! = n + 1\ \textbf{else}\ c! = j + 1 \\
\hline
\end{array}
$$

- The schema *CountPerGroup* introduces the function *count* which has the input of $c?$ and the output of $c!$
- The input variable $c?$ is a non-empty, finite sequence in which each element is a *statement*
- The function *count* reads: for all elements $c?_n$ within the sequence $\langle c?_i..c?_j \rangle$, such that n is greater than or equal to i and less than or equal to j, i is equal to zero and there exits a number $c!$ which is equal to $n + 1$ (when $n = i \Rightarrow n = 0$) or equal to n

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\ AggregateQuestionStatements\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta MostDifficultAssessmentQuestions \\
FilterForIncorrect \\
GroupByActivityId \\
CountPerGroup \\
grouped, processed : \mathbb{F} \\
\hline
grouped = \emptyset \\
grouped' = group(S_{incorrect}) \\
S'_{grouped} = S_{grouped} \cup grouped' \\
processed \subseteq S'_{grouped} \\
processed' = \{p : (id, \mathbb{N}) \mid \\
\qquad\qquad\qquad \mathbf{let}\ \{\langle processed_i \rangle .. \langle processed_j \rangle\} == \{g_i .. g_j\} \bullet \\
\qquad\qquad\qquad i \leq n \leq j \bullet \forall g_n : g_i .. g_j \bullet \exists\, p_n : p_i .. p_j \bullet \\
\qquad\qquad\qquad first\, p_n = head\, g_n.object.id \wedge second\, p_n = count(g_n) \\
S'_{processed} = S_{processed} \cup processed' \\
\hline
\end{array}
$$

- The schema *AggregateQuestionStatements* introduces the variables *grouped* and *processed*
- *grouped* starts as an empty set but then becomes $grouped'$ which is the output of applying the function *group* to the set of statements $S_{incorrect}$ created by the opperation $FilterForIncorrect$
- $grouped'$ is a set of sequences. The elements of those sequences are statements which all have the same *statement.object.id*
- The set $S_{grouped}$ is updated to the set $S'_{grouped}$ which is the union of $S_{grouped}$ and $grouped'$
- the variable *processed* is a subset of $S'_{grouped}$ which can contain every value within $S'_{grouped}$
- the variable *processed* is updated to be the variable $processed'$ which is a set of objects p which are ordered pairs of the component *id* and a natural number. p is defined as:
  - for all sequences $g_i..g_j$ within the set *processed*, there exists an ordered pair p_n such that:
    - the first element of p_n is equal to the *object.id* of the first statement within the sequence g_n.
    - The second element of p_n is equal to the value returned when g_n is passed to the function *count*.
- The set $S'_{processed}$ is the union of the sets $S_{processed}$ and $processed'$

15.6.6 Sequence of Operations

$ProcessedQuestions \mathrel{\widehat=} FilterForIncorrect \mathbin{⨾} AggregateQuestionStatements$

- The schema $ProcessedQuestions$ is the sequential composition of operation schemas $FilterForIncorrect$ and $AggregateQuestionStatements$
- $FilterForIncorrect$ happens before $AggregateQuestionStatements$

15.6.7 Return

$$
\begin{array}{|l}
\textit{ReturnAggregate} \\
\hline
\Xi MostDifficultAssessmentQuestions \\
ProcessedQuestions \\
S_{processed}! : \mathbb{F} \\
\hline
S_{processed}! = S_{processed} \\
\hline
\end{array}
$$

- The returned variable $S_{processed}!$ is equal to the current state of variable $S_{processed}$ after the operations $FilterForIncorrect$ and $AggregateQuestionStatements$

15.7 Pseudocode

Algorithm 3: Most Difficult Assessment Questions

```
Input: S_all, displayN
Result: display''
context = {};
display = [];
while S_all ≠ ∅ do
    foreach s ∈ S_all do
        if s.result.success = INCORRECT then
            do
            S'_incorrect ← s ∪ S_incorrect;
            S'_all ← S_all \ s;
            recur S'_all, S'_incorrect;
        else
            do
            S'_all ← S_all \ s;
            recur S'_all
        end
    end
end
while S'_incorrect ≠ ∅ do
    foreach si ∈ S'_incorrect do
        id ← si.object.id;
        if id ∉ context then
            do
            count = 1;
            context' ← {id : count};
            S'_incorrect ← S_incorrect \ si;
            recur context', S'_incorrect;
        else
            do
            count' ← inc(context.id);
            context' ← {id : count'};
            S'_incorrect ← S_incorrect \ si;
            recur context', S'_incorrect;
        end
    end
end
foreach id ∈ context' do
    IdToCount ← [id, context.id];
    display' ← display ⌢ IdToCount;
    recur display'
end
return display'' ← take(sortBySubArray(display'), displayN)
```

- The Z schemas are used within this pseudocode
- The return value display is an array of length display-n, where each element of display is an array of [*statement.object.id*, #] where # representing the number of times *statement.object.id* appeared within $S'_{incorrect}$

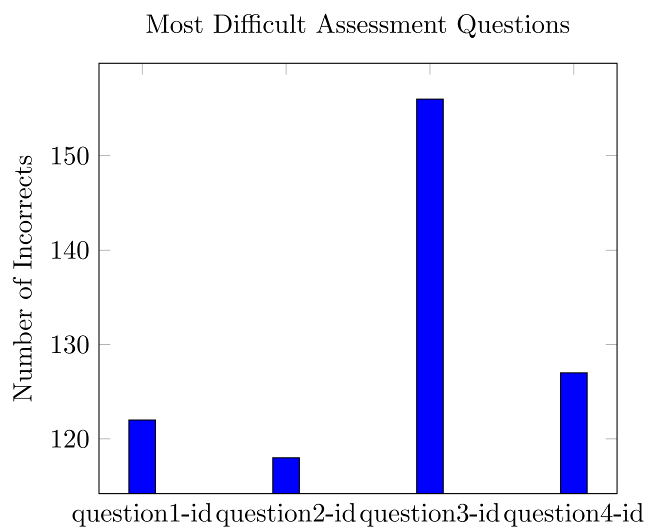
15.8 JSON Schema


```
{"type":"array",
  "items":{"type":"array",
    "items":[{"type":"string"}, {"type":"number"}]}}
```


15.9 Visualization Description

The **Most Difficult Assessment Questions** visualization will be a bar chart where the domain consists of *statement.object.id* and the range is a number greater than or equal to 1. Every subarray within the array display will be a grouping within the bar chart. The pseudocode specifies an input paramter display-n which controls the length of the array display and therefor the number of groups contained within the visualization.

15.10 Visualization prototype



15.11 Prototype Improvement Suggestions

Additional features may be implemented on top of this base specification but they would require adding aditional values to each subarray returned by the algorithm. These additional values can be retrieved via (1) performing metadata lookup within or independently of the algorithm (2) by utilizing additional xAPI statement paramters and/or (3) by performing additional computations. The following examples assume the metadata is contained within each statement available to the algorithm.

- Use the name of the activity for the x-axis label instead of its id.
  - $\$.object.definition.name$
  - grouping of statements should still happen by $\$.object.id$ to ensure an accurate count
- a tooltip containing contextual information about the question such as:
  - The question text
    * $\$.object.definition.description$
  - Interaction Type
    * $\$.object.definition$ which contains interaction properties
  - Answer choices
    * $\$.object.definition$ which contains interaction properties
  - Correct answer
    * $\$.object.definition$ which contains interaction properties
  - Most popular incorrect answer
    * This would require an extra step of processing and all statements would need to utilize interaction properties within $\$.object.definition$
  - average partial credit earned (if applicable)
    * $\$.result.score.scaled$
    * The one potential issue with using scaled score is the calculation of scaled is not stricly defined by the xAPI specification but is instead up to the authors of the LRP. This results in the inability to reliably compare scaled scores across LRPs.
    * if $\$.result.score.raw$, $\$.result.score.min$ and $\$.result.score.max$ are reported for all questions, it becomes possible to reliably compare scores across questions and LRPs.
  - average number of re-attempts
    * this would require additional steps of processing so that $\$.actor$ is considered as well
    * due to the problem of actor unification, ie the same person being identified differently across statements, this metric may not be accurate.
  - average time spent on the question
    * $\$.result.duration$
    * this would require additional steps of processing to extract the duration and average it.

- a tooltip containing contextual information about the course and/or assessment the question was within
  - the instructor for the course
    - $\$.context.instructor$
  - competency associated with the question and/or course
    - $\$.context.contextActivities$
  - metadata about the learning content associated with the question such as average time spent engaging with associated content before attempting the question.
  - this would require additional steps of processing to retrieve metadata about the content and its usage.
    - $\$.context.contextActivities$

16 How Often are Recommendations Followed

As learners engage in activities supported by a learning ecosystem, they will build up a history of learning experiences. When the digital resources of that learning ecosystem adhere to a framework dedicated to supporting and understanding the learner, such as the Total Learning Architecture (TLA), it becomes possible to retell their learning story through data and data visualization. One important aspect of that story is the recommendations provided to the learner and whether or not the learner follows those recommendations.

16.1 Ideal Statements

In order to accurately determine if a learner is following recommendations, there are a few requirements of the data produced by a LRP and the recommender itself. They are as follows:

- Every time the recommender makes a recommendation, a statement should be produced which uses the verb $https://w3id.org/xapi/dod{-}isd/verbs/recommended$[14] and has the recommended piece of content as the object.
  - the content should be uniquely and consistently identified across all statements.
- When a learner launches recommended content, the resulting launched statement should use the verb $http://adlnet.gov/expapi/verbs/launched$[15] and contain a refrence to the recommened content statement within $\$.context.statement$
  - Launching of content should use the above IRI regardless of why the content was launched
  - If it not possible to refrence the exact recommended content statement, the launch statement should have some indication that it was the result of a recommendation.[16]

[14] See footnote 4

[15] See footnote 4

[16] It is possible to determine if recommendations are followed (with some level of error) without this explicit linking of launched to recommended but this severly complicates the algorithm. In this case, in order to optmize for accuracy, the algorithm would need to consider the actor and their general activity within a session, the object of both launched and recommended statements generated within the session, the time lapse between recommendations and launches with a predefined lapse value which determines if a launch was close enough in time to a recommendation to be considered a result of the recommendation. An additonal constraint on the above case is the recommendation statements should contain a reference to to the person recieving the recommendation, otherwise determining the 1:1 relationships between recommendations and launches requires aditional complexity and will still not be 100% accurate due to the reliance on the time lapse value.

16.2 Input Data Retrieval

How to query an LRS via a GET request to the Statements Resource via curl.[17][18][19]

```
R = "verb=https://w3id.org/xapi/dod-isd/verbs/recommended"
L = "verb=http://adlnet.gov/expapi/verbs/launched"

Since = "since=2018-07-20T12:08:47Z"
Until = "until=2018-07-21T12:08:47Z"
Base = "https://example.endpoint/statements?"

endpoint1 = Base + R + "&" + Since + "&" + Until
endpoint2 = Base + L + "&" + Since + "&" + Until

Auth = Hash generated from basic auth

SR = curl -X GET -H "Authorization: Auth"
        -H "Content-Type: application/json"
        -H "X-Experience-API-Version: 1.0.3"
        endpoint1
SL = curl -X GET -H "Authorization: Auth"
        -H "Content-Type: application/json"
        -H "X-Experience-API-Version: 1.0.3"
        endpoint2

S = SR + SL
```

16.3 Statement Parameters to Utilize

The statement parameter locations here are written in JSONPath. This notation is also compatable with the xAPI Z notation due to the defined hierarchy of components. Within the Z specifications, a variable name will be used instead of the $

- $.*verb.id*
- $.*context.statement*

16.4 2018 Pilot TLA Statement Problems

At the time of writing this document, launched statements do not include a statement reference or any indication of a connection between recommendations and launches. The authors of this document do not have access to the LRS

[17] footnote 1 applies to both S1 and S2.
[18] See footnote 2.
[19] See footnote 3.

containing the recommended statements and thus can not draw any conclusions about any issues which may be present within those statements or any aspects of those statements which may correlate them to launch statements. The following algorithm is going to assume that the input set of statements follow the guidlines outlined in section 5.1 as the additional algorithmic considerations brought on by non ideal statements, as specified within footnote 16, result in an algoirthm which is not optimal for near real time visualizations.

16.5 Summary

1. Query an LRS via a GET request to the statements endpoint using the paramters verb, since and until to gather all statements with the verb $http://adlnet.gov/expapi/verbs/launched$.
2. Query an LRS via a GET request to the statements endpoint using the paramters verb, since and until to gather all statements with the verb $https://w3id.org/xapi/dod-isd/verbs/recommended$.[20]
3. Group all collections of statements by a $TIMEUNIT$
4. seperate the collection of grouped launched statements into a collection of those which were the result of a recommendation and those which were not.
5. Take the count of all groups of statements
   - Recommended statements per $TIMEUNIT$
   - Launches due to recommendations per $TIMEUNIT$
   - Launches not due to recommendations per $TIMEUNIT$
6. Calculate summary statistics for the overall time range and per $TIMEUNIT$
   - Divide launches due to recommendations by the total number of launches to determine the percentage of launches due to recommendations
   - Divide launches due to recommendations by the total number of recommendations to determine the percentage of recommendations which are followed.

[20] If since and until are specified, they should be the same in both requests.

16.6 Formal Specification

16.6.1 System State

$$
\begin{array}{l}
\textit{FollowedRecommendations} \\
\hline
\textit{Statements} \\
\textit{CountPerGroup} \\
S_{recommended}, S_{launched} : \mathbb{F}_1 \\
ordered_L, ordered_R, grouped_{launched}, grouped_{recommended}, \\
onlyRecommended, cPerGroup_{launched}, cPerGroup_{recommended}, \\
cPerGroup_{followed}, combined : \mathrm{seq} \\
t_{start}, N_{launched}, N_{recommended}, N_{followed}, P_{followed}, P_{dueto} : \mathbb{N} \\
tr_{start}, tr_{end} : \mathbb{F} \\
unit? : TIMEUNIT \\
\hline
S_{recommended} = statements \\
S_{launched} = statements \\
combined = \langle (tr_{start}, tr_{end}, N_{launched}, N_{recommended}, N_{followed}, P_{followed}, P_{dueto}) \rangle \\
count(grouped_{launched}) = count(grouped_{recommended}) \\
count(onlyRecommended) = count(grouped_{launched}) \Rightarrow \\
count(onlyRecommended) = count(grouped_{recommended}) \\
count(cPerGroup_{launched}) = count(cPerGroup_{followed}) = count(cPerGroup_{recommended}) \\
\hline
\end{array}
$$

- $S_{recommended}, S_{launched}$ are both non-empty, finite sets.
  - $S_{recommended}$ and $S_{launched}$ contain the results of querying an LRS for recommended and launched statements respectively.
- $ordered_L, ordered_R, grouped_{launched}, grouped_{recommended}, onlyRecommended, cPerGroup_{launched}, cPerGroup_{recommended}, cPerGroup_{followed}$ and $combined$ are all finite sequences.
  - $ordered_L$ and $ordered_R$ are the sequences of statements within $S_{launched}$ and $S_{recommended}$ respectively and sorted by timestamp.
  - $grouped_{launched}$ is the result of grouping the statements within $ordered_L$ by $unit?$.
  - $grouped_{recommended}$ is the result of grouping the statements within $ordered_R$ by $unit?$.
  - $onlyRecommended$ is the result of filtering the statements within the sequence $grouped_{launched}$ to only include statements where $statement.context.statement$ is present
  - $cPerGroup_{launched}, cPerGroup_{recommended}, cPerGroup_{followed}$ are all sequences of numbers which represent the count within each subsequence of $grouped_{launched}$, $grouped_{recommended}$ and $onlyRecommended$ respectively.

– *combined* is a sequence of ordered pairs where each pair consists of tr_{start}, tr_{end}, $N_{launched}$, $N_{recommended}$, $N_{followed}$, $P_{followed}$ and P_{dueto}

- $t_{start}, N_{launched}, N_{recommended}, N_{followed}, P_{followed}, P_{dueto}$ are all natural numbers
- tr_{start}, tr_{end} are both timestamps which represent the the start and end of the time range for each a group of statements.
- *unit*? is an input representing a time interval, ie day vs month vs hour.
- all sequences are the same length so that each subsequence represents the same time grouping. In other words, indexes are comparable across sequences.

16.6.2 Initial System State

$$
\begin{array}{|l}
\hline
\textit{InitFollowedRecommendations} \\
\textit{FollowedRecommendations} \\
\hline
S_{recommended} \neq \emptyset \\
S_{launched} \neq \emptyset \\
unit? = \{day\} \\
ordered_L = \langle\rangle \\
ordered_R = \langle\rangle \\
grouped_{launched} = \langle\rangle \\
grouped_{recommended} = \langle\rangle \\
onlyRecommended = \langle\rangle \\
cPerGroup_{launched} = \langle\rangle \\
cPerGropu_{recommended} = \langle\rangle \\
cPerGroup_{followed} = \langle\rangle \\
combined = \langle\rangle \\
t_{start} = 0 \\
N_{launched} = 0 \\
N_{recommended} = 0 \\
N_{followed} = 0 \\
P_{followed} = 0 \\
P_{dueto} = 0 \\
\hline
\end{array}
$$

- $S_{recommended}$ and $S_{launched}$ are initially not empty sets
- all sequences are initially empty
- all numbers are initially zero
- the default $TIMEUNIT$ is set to day

16.6.3 Group by Timestamp

$$
\begin{array}{|l}
\hline
\textit{SortByTimestamp} \\
Statement \\
IsoToUnix \\
orderByTimestamp : \mathbb{F}_1 \rightarrow seq_1 \\
o? : \mathbb{F}_1 \\
o! : seq_1 statement \\
\hline
o? = \{o : statement\} \\
o! = orderByTimestamp(o?) \\
o! = \langle o_i..o_j \rangle \bullet \forall o_n : o_i..o_j \bullet o_n : STATEMENT \wedge i \leq n \leq j \bullet \\
\qquad convert(o_i.timestamp) \leq convert(o_n.timestamp) \leq convert(o_j.timestamp) \\
\hline
\end{array}
$$

- The schema $SortByTimestamp$ introduces the function $orderByTimestamp$ which takes in a non-empty, finite set and returns a non-empty, finite sequence.
- $orderByTimestamp$ is a sequence of statements ordered from earliest to latest.

$$
\begin{array}{|l}
\hline
\textit{WithinRange} \\
withinRange : (\mathbb{N}, \mathbb{N}, \mathbb{N}, TIMEUNIT) \rightarrow \mathbb{F}_1 \#1 \\
in?, start?, state? : \mathbb{N} \\
unit? : TIMEUNIT \\
out! : \{TRUE\} \vee \{FALSE\} \\
\hline
unit? = \{second\} \Rightarrow 1 \vee \{minute\} \Rightarrow 60 \vee \{hour\} \Rightarrow 3600 \vee \\
\qquad \{day\} \Rightarrow 86400 \vee \{week\} \Rightarrow 604800 \vee \\
\qquad \{month\} \Rightarrow 2629743 \vee \{year\} \Rightarrow 31556926 \\
out! = withinRange(in?, start?, state?, unit?) \\
withinRange(in?, start?, state?, unit?) = \\
\qquad \textbf{if } in? \leq start? + ((state? + 1) * unit?) \\
\qquad\quad \textbf{then } out! = \{TRUE\} \\
\qquad\quad \textbf{else } out! = \{FALSE\} \\
\hline
\end{array}
$$

- The schema $WithinRange$ introduces the function $withinRange$ which takes in three numbers and a $TIMEUNIT$ and returns either $\{TRUE\}$ or $\{FALSE\}$
- $withinRange$ checks to see if $in?$ is less than or equal to a start time $start?$ plus the result of multiplying the numeric conversion for $unit?$ by the $state?$.
- $state?$ represents the current group, ie. day 1 vs day 2 vs day 3. The +1 is to account for array indexes starting at 0.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\ GroupByTimeUnit\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
Statement \\
IsoToUnix \\
WithinRange \\
groupByTimeUnit : (\mathrm{seq}_1, \mathbb{N}, TIMEUNIT) \rightarrow \mathrm{seq}_1 \\
g?\,, g! : \mathrm{seq}_1 \\
t_{start}? : \mathbb{N} \\
\hline
g? = \langle g?_i \,..g?_j \,\rangle \bullet \forall g?_n : g?_i \,..g?_j \bullet i \leq n \leq j \bullet g?_n = statement \land \\
\quad convert(g?_i \,.timestamp) \leq convert(g?_n \,.timestamp) \leq convert(g?_j \,.timestamp) \\
g! = groupByTimeUnit(g?\,, t_{start}?\, state?\,, unit?\,) \\
g! = \langle g : seq \mid \forall g?_n : g?_i \,..g?_j \bullet \exists_1 \langle g_r \rangle : \langle g_q \rangle..\langle g_s \rangle \bullet q \leq r \leq s \land r = state? \bullet \\
\qquad\qquad \textbf{if } withinRange(convert(g?_n \,.timestamp), t_{start}, r, unit?\,) = \{TRUE\} \\
\qquad\qquad\quad \textbf{then } g? \upharpoonright g?_n \land g?_n \text{ in } \langle g_r \rangle \Rightarrow \langle g_r \rangle = \langle g_{ri}..g_{rn}..g_{rj} \rangle \bullet ri \leq rn \leq rj \bullet g_{rn} = g?_n \\
\qquad\qquad\quad \textbf{else if } \forall g_n? : g_i?\,..g_j? \bullet withinRange(g?_n\,, t_{start}, r, unit?\,) = \{FALSE\} \\
\qquad\qquad\qquad \textbf{then } groupByTimeUnit((g? \upharpoonright \langle g_r \rangle), t_{start}?\,(r+1), unit?\,) \bullet \langle g_r \rangle = \langle\rangle \lor \neq \langle\rangle\rangle \\
\hline
\end{array}
$$

- The schema *GroupByTimeUnit* intorudces the function *groupByTimeUnit* which takes as arguments a non-empty, finite sequence, a natural number and a $TIMEUNIT$ and outputs a non-empty, finite sequence of sequences.

- For every statement within the input sequence, *groupByTimeUnit* checks to see if the timestamp of that statement is within the range of t_{start} and $unit?$. If it is, that statement is removed from the input sequence $g?$ and added to the current subsequence $\langle g_r \rangle$. If none of the remaining statements within the input sequence are within the range of t_{start} and $unit?$, then the variable $state?$ is incremented, the current subsequence $\langle g_r \rangle$ is either a collection of matched statements or is an empty sequence and the search for remaining subsequences $\langle g_{r+state?} \rangle$ continues.

- because the input sequence $g?$ is orderd chronologically, this implies that once a statement does not fit into a range, the rest of the statements remaining in the input sequence will not fit into that range and $state?$ must be incremented to generate a new subsequence $\langle g_{r+state?} \rangle$ so that the remaining statements can be grouped.

16.6.4 Processes Results

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\ OrderStatements\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta FollowedRecommendations \\
SortByTimestamp \\
\hline
ordered'_L = orderByTimestamp(S_{launched}) \\
ordered'_R = orderByTimestamp(S_{recommended}) \\
t'_{start} = convert((head\, ordered'_L).timestamp) \\
\hline
\end{array}
$$

- The schema *OrderStatements* updates the system state defined by the schema *FollowedRecommendations*.
- $ordered'_L$ is the result of ordering the statements contained within the set $S_{launched}$ chronologically.
- $ordered'_R$ is the result of ordering the statements contained within the set $S_{recommended}$ chronologically.
- t'_{start} is the timestamp from the first statement within $ordered'_L$ converted to unix time.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\ GroupByTime\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta FollowedRecommendations \\
GroupByTimeUnit \\
\hline
grouped'_{launched} = groupByTimeUnit(ordered'_L, t'_{start}, 0, unit?) \\
grouped'_{recommended} = groupByTimeUnit(ordered'_R, t'_{start}, 0, unit?) \\
\hline
\end{array}
$$

- The schema *GroupByTime* updates the state defined by the schema *FollowedRecommendations*.
- $grouped'_{launched}$ is the result of passing $ordered'_L$, t'_{start}, 0 and $unit?$ to the function *groupByTimeUnit*.
- $grouped'_{recommended}$ is the result of passing $ordered'_R$, t'_{start}, 0 and $unit?$ to the function *groupByTimeUnit*.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\ OnlyRecommendedLaunches\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta FollowedRecommendations \\
\hline
onlyRecommended' = \langle o : seq \mid \mathbf{let}\ grouped'_{launched} == gl \Rightarrow \\
\quad \langle\langle gl_i\rangle..\langle gl_j\rangle\rangle \Rightarrow \langle\langle gl_{ii}..gl_{ij}\rangle..\langle gl_{ji}..gl_{jj}\rangle\rangle \bullet \\
\quad \forall\langle gl_n\rangle : \langle gl_i\rangle..\langle gl_j\rangle \bullet \exists_1\langle o_n\rangle : \langle o_i\rangle..\langle o_j\rangle \Rightarrow \langle\langle o_{ii}..o_{ij}\rangle..\langle o_{ji}..o_{jj}\rangle\rangle \bullet \\
\quad ((\forall o_{in} : o_{ii}..o_{ij} \bullet o_{in}.context.statement \neq \emptyset \wedge o_{in}\ \mathsf{in}\ gl_i)\wedge \\
\quad (\forall o_{jn} : o_{ji}..o_{jj} \bullet o_{jn}.context.statement \neq \emptyset \wedge o_{jn}\ \mathsf{in}\ gl_j))\vee \\
\quad \langle o_n\rangle = \langle\rangle\rangle \\
\hline
\end{array}
$$

- The schema *OnlyRecommendedLaunches* updates the state defined by the schema $FollowedRecommendations$.
- $onlyRecommended'$ is the sequence of objects o where o is a sequence consisting of statements (or no statements) from the corresponding sequences within $grouped'_{launched}$ where *statement.context.statement* exists.
- $onlyRecommended'$ maintains the same number and ordering of time groups (subsequences) as $grouped'_{launched}$ and $grouped'_{recommended}$.

$$
\begin{array}{|l}
\hline
\textit{GetCounts} \\
\Delta FollowedRecommendations \\
CountPerGroup \\
\hline
cPerGroup'_{launched} = \langle c : \mathbb{N} \mid \textbf{let } grouped'_{launched} == gl \Rightarrow \langle \langle gl_i \rangle .. \langle gl_j \rangle \rangle \bullet \\
\qquad \forall \langle gl_n \rangle : \langle gl_i \rangle .. \langle gl_j \rangle \bullet \exists_1 c_n : \mathbb{N} \bullet \\
\qquad \textbf{if } gl_n = \langle \rangle \\
\qquad\quad \textbf{then } c_n = 0 \\
\qquad\quad \textbf{else } c_n = count(\langle gl_n \rangle) \rangle \\
cPerGroup'_{recommended} = \langle c : \mathbb{N} \mid \textbf{let } grouped'_{recommended} == gr \Rightarrow \langle \langle gr_i \rangle .. \langle gr_j \rangle \rangle \bullet \\
\qquad \forall \langle gr_n \rangle : \langle gr_i \rangle .. \langle gr_j \rangle \bullet \exists_1 c_n : \mathbb{N} \bullet \\
\qquad \textbf{if } gr_n = \langle \rangle \\
\qquad\quad \textbf{then } c_n = 0 \\
\qquad\quad \textbf{else } c_n = count(\langle gr_n \rangle) \rangle \\
cPerGroup'_{followed} = \langle c : \mathbb{N} \mid \textbf{let } onlyRecommended' == or \Rightarrow \langle \langle or_i \rangle .. \langle or_j \rangle \rangle \bullet \\
\qquad \forall \langle or_n \rangle : \langle or_i \rangle .. \langle or_j \rangle \bullet \exists_1 c_n : \mathbb{N} \bullet \\
\qquad \textbf{if } or_n = \langle \rangle \\
\qquad\quad \textbf{then } c_n = 0 \\
\qquad\quad \textbf{else } c_n = count(\langle or_n \rangle) \rangle \\
\hline
\end{array}
$$

- The schema *GetCounts* updates the state defined by the schema $FollowedRecommednations$.
- $cPerGroup'_{launched}$ is a sequence of numbers c where each c is either 0 or the result of passing the current subsequence of $grouped'_{launched}$ (gl_n) to the function *count*.
- $cPerGroup'_{recommended}$ is a sequence of numbers c where each c is either 0 or the result of passing the current subsequence of $grouped'_{recommended}$ (gr_n) to the function *count*.
- $cPerGroup'_{followed}$ is a sequence of numbers c where each c is either 0 or the result of passing the current subsequence of $onlyRecommended'$ (or_n) to the function *count*.

$$
\begin{array}{l}
\underline{\quad CombineSequences \quad}\\
\Delta FollowedRecommendations\\
\hline
combined' = \langle c : (tr'_{start}, tr'_{end}, N'_{launched}, N'_{recommended}, N'_{followed}, P'_{followed}, P'_{dueto}) \mid \\
\quad \textbf{let } grouped'_{launched} == gl \Rightarrow \langle \langle gl_i \rangle .. \langle gl_n \rangle .. \langle gl_j \rangle \rangle \\
\quad\quad cPerGroup'_{launched} == cl \Rightarrow \langle cl_i .. cl_n .. cl_j \rangle \\
\quad\quad cPerGroup'_{recommended} == cr \Rightarrow \langle cr_i .. cr_n .. cr_j \rangle \\
\quad\quad cPerGroup'_{followed} == cf \Rightarrow \langle cf_i .. cf_n .. cf_j \rangle \\
\quad \bullet \forall \langle gl_n \rangle : \langle gl_i \rangle .. \langle gl_j \rangle \bullet i \leq n \leq j \bullet \\
\quad \exists_1 c_n : (tr_{startn}, tr_{endn}, N_{launchedn}, N_{recommendedn}, N_{followedn}, P_{followedn}, P_{dueton}) \bullet \\
\quad tr_{startn} = (head\, gl_n).timestamp \\
\quad tr_{endn} = (last\, gl_n).timestamp \\
\quad N_{launchedn} = cl_n \\
\quad N_{recommendedn} = cr_n \\
\quad N_{followedn} = cf_n \\
\quad P_{followedn} = cf_n \div cr_n \\
\quad P_{dueton} = cf_n \div cl_n \rangle
\end{array}
$$

- The schema $CombineSequences$ changes the state defined by the schema $FollowedRecommendations$.
- $combined'$ is a sequence of objects c where each c is an ordered pair of $tr'_{start}, tr'_{end}, N'_{launched}, N'_{recommended}, N'_{followed}, P'_{followed}, P'_{dueto}$.
- for each c_n:
  - $tr'_{start} \leadsto tr_{startn}$ which is equal to the timestamp for the first statement within gl_n
  - $tr'_{end} \leadsto tr_{endn}$ which is equal to the timestamp for the last statement within gl_n.
  - $N'_{launched} \leadsto N_{launchedn}$ which is equal to the current count of launched statements within the nth time grouping aka cl_n.
  - $N'_{recommended} \leadsto N_{recommendedn}$ which is equal to the current count of recommended statements within the nth time grouping aka cr_n.
  - $N'_{followed} \leadsto N_{followedn}$ which is equal to the current count of recommended statements within the nth time grouping aka cf_n.
  - $P'_{followed} \leadsto P_{followedn}$ which is equal to the result of dividing cf_n by cr_n.
  - $P'_{dueto} \leadsto P_{dueton}$ which is equal to the result of dividing cf_n by cl_n.

16.6.5 Sequence of Operations

$ProcessFollowedRecommendations \mathrel{\widehat=}$
$OrderStatements \mathbin{⨾} GroupByTime \mathbin{⨾} OnlyRecommendedLaunches \mathbin{⨾}$
$GetCounts \mathbin{⨾} CombineSequences$

- The schema *ProcessFollowedRecommendations* defines the order of operations for the steps within the *FollowedRecommendations* algorithm.

16.6.6 Return

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ReturnFollowedRecommendations}} \\
\Xi FollowedRecommendations \\
ProcessFollowedRecommendations \\
combined! : \mathrm{seq} \\
\hline
combined! = combined' \\
\hline
\end{array}
$$

- The schema *ReturnFollowedRecommendations* describes the return value of the system defined by the schema *FollowedRecommendations*

- The return value *combined*! is the variable *combined*′ defined wtihin the schema *CombineSequences*

16.7 Pseudocode

Algorithm 4: Followed Recommendations

Input: $S_{recommended}$, $S_{launched}$ $timeUnit$
Result: $combined'$
$ordered'_L \leftarrow orderByTimestamp(S_{launched})$;
$ordered'_R \leftarrow orderByTimestamp(S_{recommended})$;
$t'_{start} \leftarrow convert((head\, ordered'_L).timestamp)$;
$grouped'_{launched} \leftarrow groupByTimeUnit(ordered'_L, t'_{start}, 0, timeUnit)$;
$grouped'_{recommended} \leftarrow$
$groupByTimeUnit(ordered'_R, t'_{start}, 0, timeUnit)$;
$grouped_{followed} \leftarrow []$;
foreach G in $grouped'_{launched}$ **do**
  $curGrouping \leftarrow []$;
  foreach G_n in G **do**
    if $G_n.context.statement \neq nil$ **then**
      do
      $curGrouping' \leftarrow curGrouping \frown G_n$**;**
      recur curGrouping'
    else
      recur curGrouping'
    end
  end
  $grouped'_{followed} \leftarrow grouped_{followed} \frown curGrouping'$**;**
  recur $grouped'_{followed}$
end
$C_{launched} \leftarrow$ **map count() grouped**$'_{\mathbf{launched}}$;
$C_{recommended} \leftarrow$ **map count() grouped**$'_{\mathbf{recommended}}$;
$C_{followed} \leftarrow$ **map count() grouped**$'_{\mathbf{followed}}$;
$combined \leftarrow []$;
for $i \leftarrow 0$ **to** $count(c_{launched})$ **by** 1 **do**
  $tr_{starti} \leftarrow (first(nth(grouped'_{launched}, i))).timestamp$;
  $tr_{endi} \leftarrow (last(nth(grouped'_{launched}, i))).timestamp$;
  $N_{Li} \leftarrow nth(C_{launched}, i)$;
  $N_{Ri} \leftarrow nth(C_{recommended}, i)$;
  $N_{Fi} \leftarrow nth(C_{followed}, i)$;
  $P_{Fi} \leftarrow N_{fi} \div N_{Ri}$;
  $P_{duetoi} \leftarrow N_{fi} \div N_{Li}$;
  $subVec_i \leftarrow [tr_{starti}, tr_{endi}, N_{Li}, N_{Ri}, N_{Fi}, P_{Fi}, P_{duetoi}]$;
  $combined' \leftarrow combined \frown subVec_i$
end
return $combined'$

- **map count()** $grouped'_{...}$ means apply the function **count()** to every sequence within the sequence $grouped_{...}$ and put all results into a single array.

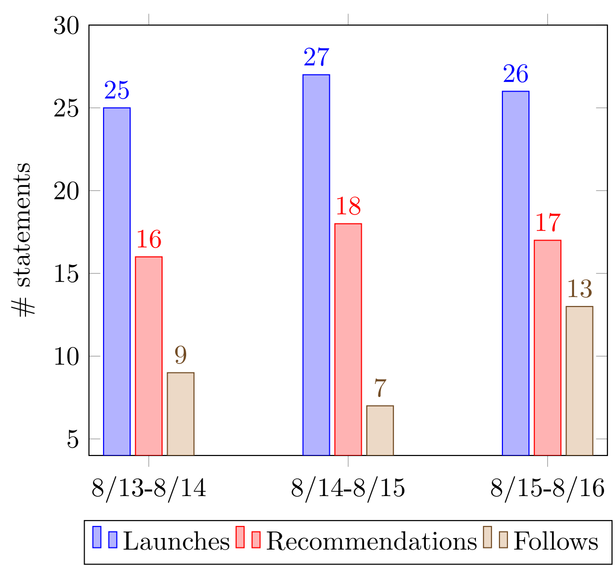
16.8 JSON Schema

```
{"type":"array",
   "items":{"type":"array",
      "items":[{"type":"string"}, {"type":"string"},
               {"type":"number"}, {"type":"number"},
               {"type":"number"}, {"type":"number"},
               {"type":"number"}]}}
```

16.9 Visualization Description

The **Followed Recommendations** visualization can be a bar chart where the domain is time ranges and the range is a number representing the total count of statements recorded. For each time range, there will be three groups: 1) the number of launched statements 2) the number of recommended statements 3) the number of launches which are due to recommendations. Above each grouping or on hover, summary statistics can be desplayed which describe the percentage of launches due to recommendations and the percentage of recommendations which were followed.

16.10 Visualization prototype



- The percentages described in section 5.9 are not displayed here.

16.11 Prototype Improvement Suggestions

Additional features may be implemented on top of this base specification but they would require adding aditional values to each subarray returned by the algorithm. These additional values can be retrieved via (1) performing metadata lookup within or independently of the algorithm (2) by utilizing additional xAPI statement paramters and/or (3) by performing additional computations. The following examples assume the metadata is contained within each statement available to the algorithm.

- populate a tooltip with the most popular launched, recommended and followed activity.
- populate a tooltip with the number of actors associated with the launches and follows.
- populate a tooltip with the actor who most often and the actor who lease often follows recommendations.

Appendex A: Visualization Exemplars

Appendex A includes a typology of data visualizations which may be supported within DAVE workbooks. These visualizations can either be one to one or one to many in regards to the algorithms defined within this document. Future iterations of this document will increasingly include these typologies within the domain-question template exemplars.

Line Charts

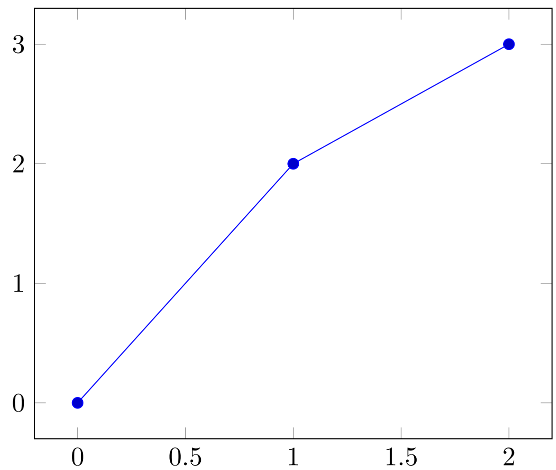


Figure 1: Line Chart

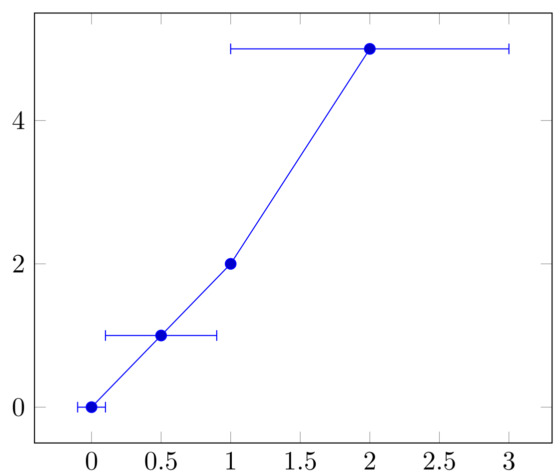


Figure 2: Line Chart with Error

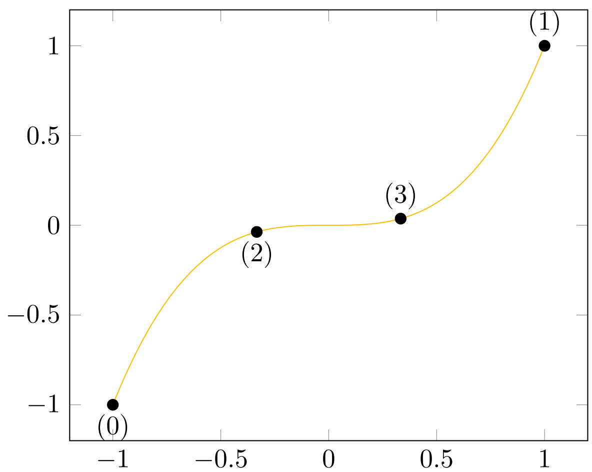


Figure 3: Spline Chart

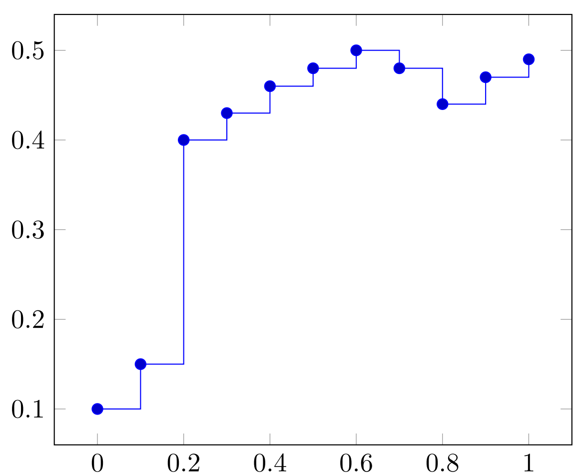


Figure 4: Quiver Chart

Grouping Charts

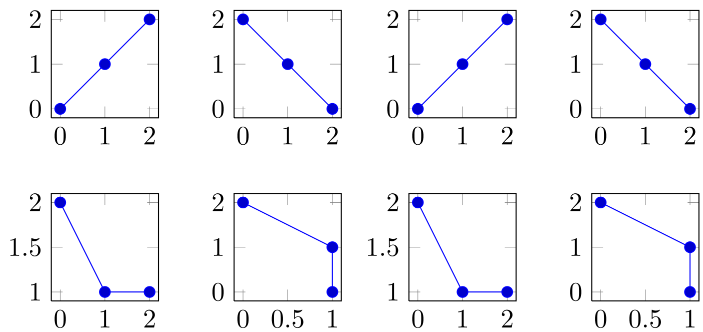


Figure 5: Grouped Line Charts

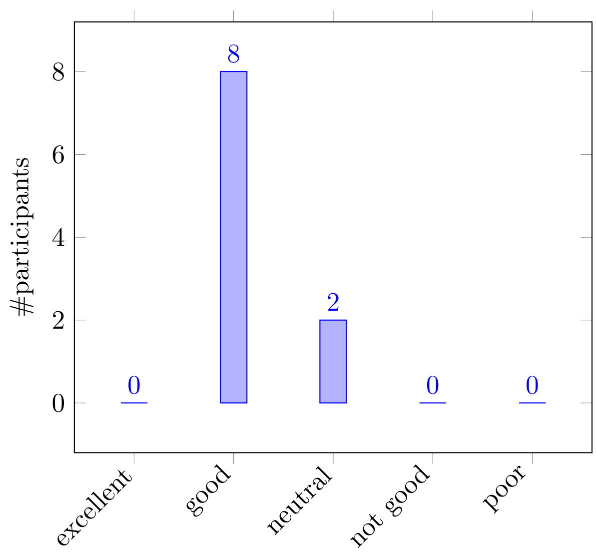


Figure 6: Histogram

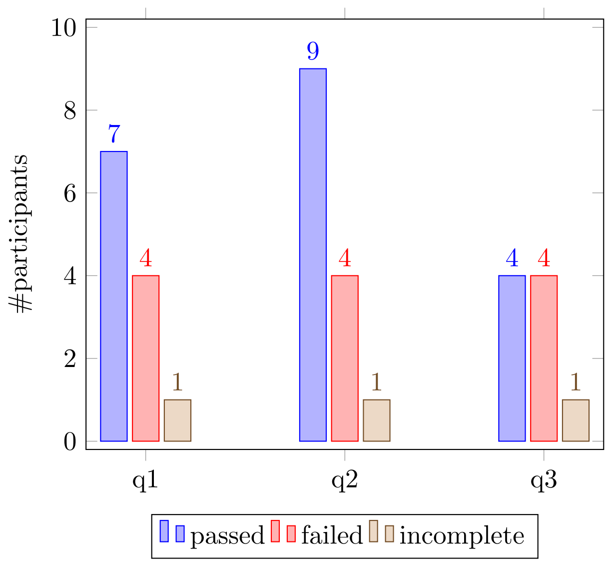


Figure 7: Bar Chart

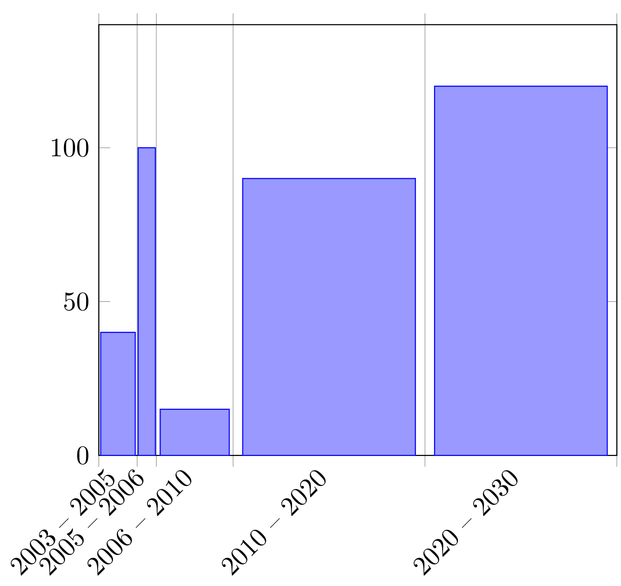


Figure 8: Bar Chart Grouped by Time Range

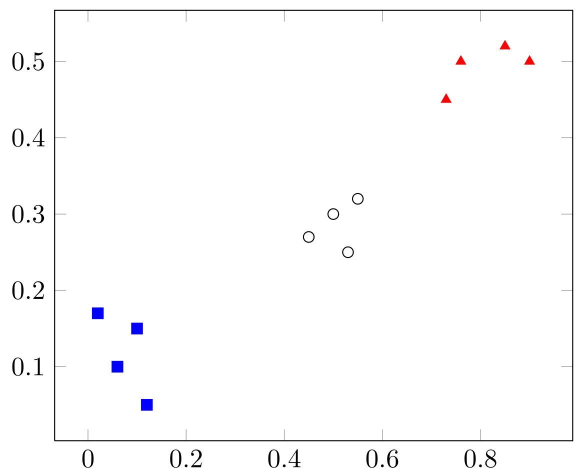


Figure 9: Scatter Plot

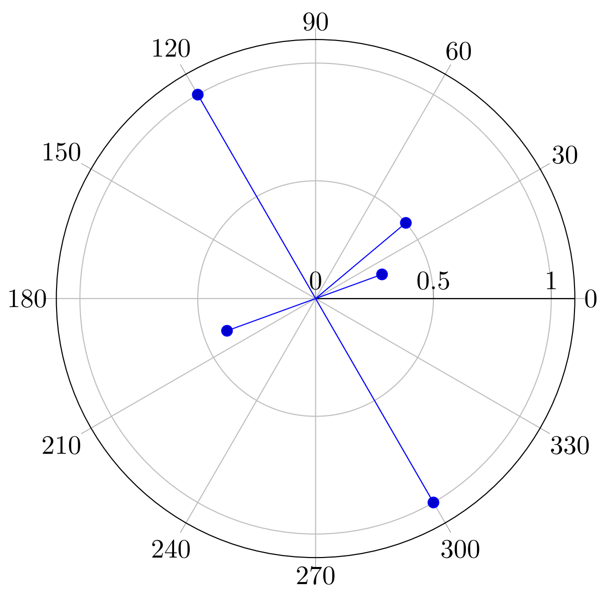


Figure 10: Polar Chart

Specialized Charts

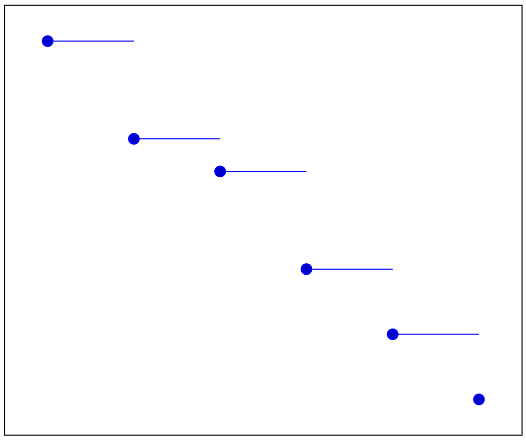


Figure 11: Gantt Chart

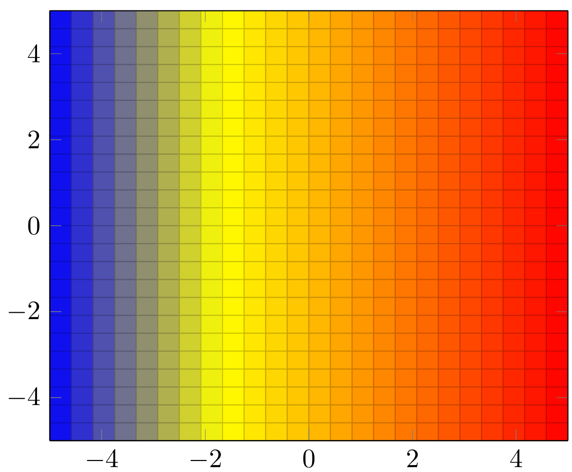


Figure 12: Heat Map

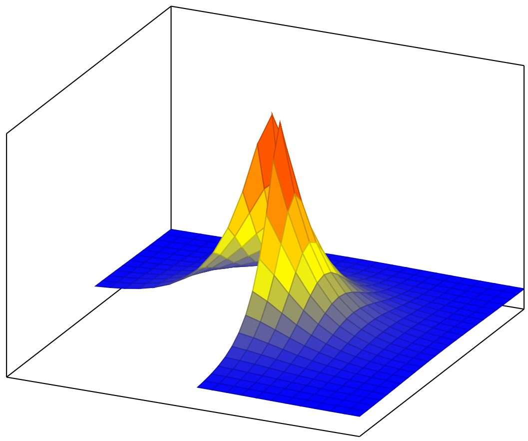


Figure 13: 3D Plot

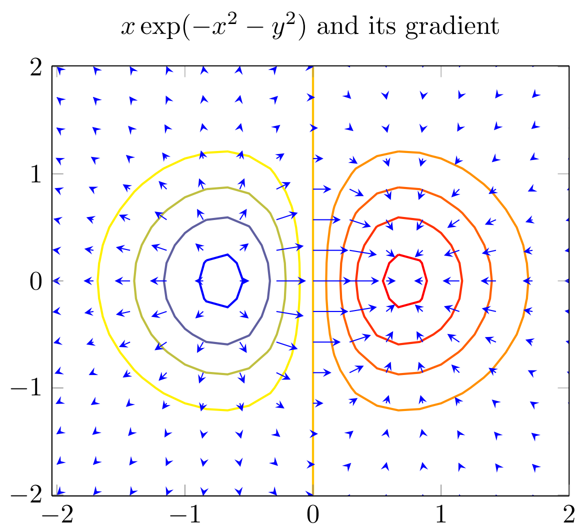


Figure 14: Gradient Plot