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

Yet Analytics

October 1, 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*.

Agent
$agent : AGENT$
$objectType : OBJECTTYPE$
$name : \mathbb{F}_1 \, \#1$
$ifi : IFI$

$objectType = Agent$
$agent = \{ifi\} \cup \mathbb{P}\{name, objectType\}$

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

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

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

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

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.

4

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.

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

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

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

$$X \nrightarrow Y == \{\, f : X \leftrightarrow Y \mid (\forall x : X; y1, y2 : Y \bullet$$
$$(x \mapsto y_1 \in f \ \wedge \ (x \mapsto y_2) \in f \Rightarrow y_1 = y_2))\}$$
$$X \rightarrow Y == \{\, f : X \nrightarrow Y \mid \mathrm{dom}\, f = X \}$$
$$X \rightarrowtail\!\!\!\rightarrow Y == \{\, f : X \nrightarrow Y \mid (\forall x_1, x_2 : \mathrm{dom}\, f \bullet f(x_1) = f(x_2) \Rightarrow x_1 = x_2)\}$$
$$X \rightarrowtail Y == (X \rightarrowtail\!\!\!\rightarrow Y) \cap (X \rightarrow Y)$$
$$X \twoheadrightarrow Y == \{\, f : X \rightarrowtail\!\!\!\rightarrow Y \mid \mathrm{ran}\, f = Y \}$$
$$X \twoheadrightarrow Y == (X \twoheadrightarrow Y) \cap (X \rightarrow Y)$$
$$X \rightarrowtail\!\!\!\twoheadrightarrow Y == (X \twoheadrightarrow Y) \cap (X \rightarrowtail Y)$$

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 $\in$ X related to atleast one y $\in$ Y by R, page 96]
ran	[set of all y $\in$ Y related to atleast one x $\in$ X by R, page 96]
$\fatsemi$	[The composition of two relationships, page 97]
$\circ$	[The backward composition of two relationships, page 97]

$\begin{array}{l} =\!\!=[X,Y]=\!\!=\!\!=\!\!=\!\!=\!\!=\!\!=\!\!=\!\!=\!\!= \\ \quad first : X \times Y \rightarrow X \\ \quad second : X \times Y \rightarrow Y \\ \rule{4cm}{0.4pt} \\ \quad \forall x : X; y : Y \bullet \\ \qquad first(x, y) = x \ \wedge \\ \qquad second(x, y) = y \\ \rule{0pt}{0.4pt} \end{array}$

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

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

$$\begin{array}{l}
[X,Y,Z] \\
\hline
\_\,{}^{\circ}_{9}\,\_ : (X \leftrightarrow Y) \times (Y \leftrightarrow Z) \to (X \leftrightarrow Z) \\
\_\,\circ\,\_ : (Y \leftrightarrow X) \times (X \leftrightarrow Y) \to (X \leftrightarrow X) \\
\hline
\forall Q : X \leftrightarrow Y; R : Y \leftrightarrow Z \bullet \\
\quad Q\,{}^{\circ}_{9}\,R = R \circ Q = \{\,x : X; y : Y; z : Z \mid \\
\qquad\qquad\qquad x\,\underline{Q}\,y\ \wedge\ y\,\underline{R}\,z \bullet x \mapsto z\,\}
\end{array}$$

1.8.3 Numeric

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

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

7

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

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]

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

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

$$\begin{array}{|l}
\underline{}[X]\rule{0pt}{0pt}\\
\hline
\_\upharpoonleft\_ : \mathbb{P}\,\mathbb{N}_1 \times \operatorname{seq} X \to \operatorname{seq} X \\
\_\restriction\_ : \operatorname{seq} X \times \mathbb{P}\, X \to \operatorname{seq} X \\
squash : (\mathbb{N}_1 \nrightarrow X) \to \operatorname{seq} X \\
\hline
\forall U : \mathbb{P}\,\mathbb{N}_1; s : \operatorname{seq} X \bullet \\
\quad U \upharpoonleft s = squash(U \lhd s) \\
\forall s : \operatorname{seq} X; V : \mathbb{P}\, X \bullet \\
\quad s \restriction V = squash(s \rhd V) \\
\forall f : \mathbb{N}_1 \nrightarrow X \bullet \\
\quad squash\, f = f \circ (\mu\, p : 1\mathrel{..}\#f \rightarrowtail \operatorname{dom} f \mid p \circ succ \circ p^{\sim} \subseteq (\_ < \_))
\end{array}$$

$$\begin{array}{|l}
\underline{}[X]\rule{0pt}{0pt}\\
\hline
^{\frown}\!/ : \operatorname{seq}(\operatorname{seq} X) \to \operatorname{seq} X \\
\hline
^{\frown}\!/\langle\rangle = \langle\rangle \\
\forall s : \operatorname{seq} X \bullet {}^{\frown}\!/\langle s\rangle = s \\
\forall q, r : \operatorname{seq}(\operatorname{seq} X) \bullet \\
\quad {}^{\frown}\!/(q \frown r) = ({}^{\frown}\!/\,q) \frown ({}^{\frown}\!/\,r)
\end{array}$$

$$\begin{array}{|l}
\underline{}[I, X]\rule{0pt}{0pt}\\
\hline
\mathsf{disjoint}\ \_ : \mathbb{P}(I \nrightarrow \mathbb{P}\, X) \\
\_\ \mathsf{partition}\ \_ : (I \nrightarrow \mathbb{P}\, X) \leftrightarrow \mathbb{P}\, X \\
\hline
\forall S : I \nrightarrow \mathbb{P}\, X; T : \mathbb{P}\, X \bullet \\
\quad (\mathsf{disjoint}\ S \iff \\
\qquad (\forall i, j : \operatorname{dom} S \mid i \neq j \bullet S(i) \cap S(j) = \emptyset)) \wedge \\
\quad (S\ \mathsf{partition}\ T \iff \\
\qquad \mathsf{disjoint}\ S \wedge \bigcup\{\, i : \operatorname{dom} S \bullet S(i)\,\} = T)
\end{array}$$

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]
$\underline{\cup}$	[bag difference, subtract occurances or zero if negative, page 126]
$items$	[converstion from seq to bag, page 127]

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

$$
\begin{array}{l}
\underline{\quad[X]} \\
\hline
\_\,\uplus\,\_ , \_\,\uplus\,\_ : \mathrm{bag}\, X \times \mathrm{bag}\, X \to \mathrm{bag}\, X \\
\hline
\forall B\,,\, C : \mathrm{bag}\, X; x : X \bullet \\
\quad (B \uplus C) \# x = B \# x + C \# x \,\wedge \\
\quad (B \uplus C) \# x = \max\{B \# x - C \# x, 0\} \\
\hline
\end{array}
$$

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

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

```
┌─ Id ──────────────────────────────
│ id : 𝔽₁ #1
```

- 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

```
┌─ Agent ──────────────────────────────
│ agent : AGENT
│ objectType : OBJECTTYPE
│ name : 𝔽₁ #1
│ ifi : IFI
├──────────────────────────────────────
│ objectType = Agent
│ agent = {ifi} ∪ ℙ{name, objectType}
```

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

```
┌─ Member ──────────────────────────────
│ Agent
│ member : 𝔽₁
├────────────────────────────────────────
│ member = {a : AGENT | ∀aₙ : aᵢ..aⱼ • i ≤ n ≤ j • a = agent}
```

- 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*

```
┌─ Group ──────────────────────────────
│ Member
│ group : GROUP
│ objectType : OBJECTTYPE
│ ifi : IFI
│ name : 𝔽₁ #1
├────────────────────────────────────────
│ objectType = Group
│ group = {objectType, name, member} ∨ {objectType, member}∨
│         {objectType, ifi} ∪ ℙ{name, member}
```

- 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*

$$
\begin{array}{|l}
\hline \_Actor_____ \\
Agent \\
Group \\
actor : AGENT \vee GROUP \\
\hline
actor = agent \vee group \\
\hline
\end{array}
$$

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

2.4 Verb Schema

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

- 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

$$
\begin{array}{|l}
\hline \_Extensions_____ \\
extensions, extensionVal : \mathbb{F}_1 \\
extensionId : \mathbb{F}_1 \,\#1 \\
\hline
extensions = \{e : (extensionId, extensionVal) \mid \forall e_n : e_i..e_j \bullet i \le n \le j \bullet \\
\qquad\qquad (extensionId_i, extensionVal_i) \vee (extensionId_i, extensionVal_j) \wedge \\
\qquad\qquad (extensionId_j, extensionVal_i) \vee (extensionId_j, extensionVal_j) \wedge \\
\qquad\qquad extensionId_i \ne extensionId_j\} \\
\hline
\end{array}
$$

- 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

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

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

$\quad$┌─ $Definition$ ─────────────────────────────
$\quad$│ $InteractionActivity$
$\quad$│ $Extensions$
$\quad$│ $definition, name, description : \mathbb{F}_1$
$\quad$│ $type, moreInfo : \mathbb{F}_1 \#1$
$\quad$├──────────────────────────────────────
$\quad$│ $definition = \mathbb{P}_1\{name, description, type, moreInfo, extensions, interactionActivity\}$
$\quad$└──────────────────────────────────────

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

$\quad$┌─ $Object$ ─────────────────────────────
$\quad$│ Id
$\quad$│ $Definition$
$\quad$│ $Agent$
$\quad$│ $Group$
$\quad$│ $Statement$
$\quad$│ $objectTypeA, objectTypeS, objectTypeSub, objectType : OBJECTTYPE$
$\quad$│ $substatement : STATEMENT$
$\quad$│ $object : \mathbb{F}_1$
$\quad$├──────────────────────────────────────
$\quad$│ $substatement = statement$
$\quad$│ $objectTypeA = Activity$
$\quad$│ $objectTypeS = StatementRef$
$\quad$│ $objectTypeSub = SubStatement$
$\quad$│ $objectType = objectTypeA \vee objectTypeS$
$\quad$│ $object = \{id\} \vee \{id, objectType\} \vee \{id, objectTypeA, definition\}$
$\quad$│ $\qquad \vee \{id, definition\} \vee \{agent\} \vee \{group\} \vee \{objectTypeSub, substatement\}$
$\quad$│ $\qquad \vee \{id, objectTypeA\}$
$\quad$└──────────────────────────────────────

- 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

```
┌─ Score ─────────────────────────────────────────
│ score : $\mathbb{F}_1$
│ scaled, min, max, raw : $\mathbb{Z}$
├─────────────────────────────────────────────────
│ scaled = $\{n : \mathbb{Z} \,|\, -1.0 \leq n \leq 1.0\}$
│ min = $n < max$
│ max = $n > min$
│ raw = $\{n : \mathbb{Z} \,|\, min \leq n \leq max\}$
│ score = $\mathbb{P}_1\{scaled, raw, min, max\}$
└─────────────────────────────────────────────────
```

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

```
┌─ Result ────────────────────────────────────────
│ Score
│ Extensions
│ success, completion, response, duration : $\mathbb{F}_1$ #1
│ result : $\mathbb{F}_1$
├─────────────────────────────────────────────────
│ success = $\{true\} \vee \{false\}$
│ completion = $\{true\} \vee \{false\}$
│ result = $\mathbb{P}_1\{score, success, completion, response, duration, extensions\}$
└─────────────────────────────────────────────────
```

- 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

```
┌─ Instructor ────────────────────────────────────
│ Agent
│ Group
│ instructor : $AGENT \vee GROUP$
├─────────────────────────────────────────────────
│ instructor = $agent \vee group$
└─────────────────────────────────────────────────
```

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

```
┌─ Team ──────────────────────────────────────────────
│ Group
│ team : GROUP
│─────────────────
│ team = group
└─────────────────────────────────────────────────────
```

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

```
┌─ Context ───────────────────────────────────────────
│ Instructor
│ Team
│ Object
│ Extensions
│ registration, revision, platform, language : 𝔽₁ #1
│ parentT, groupingT, categoryT, otherT : CONTEXTTYPES
│ contextActivities, statement : 𝔽₁
│──────────────────
│ statement = object \ (id, objectType, agent, group, definition)
│ parentT = parent
│ groupingT = grouping
│ categoryT = category
│ otherT = other
│ contextActivity = {ca : object \ (agent, group, objectType, objectTypeSub, substatement)}
│ contextActivityParent = (parentT, contextActivity)
│ contextActivityCategory = (categoryT, contextActivity)
│ contextActivityGrouping = (groupingT, contextActivity)
│ contextActivityOther = (otherT, contextActivity)
│ contextActivities = ℙ₁{contextActivityParent, contextActivityCategory,
│                       contextActivityGrouping, contextActivityOther}
│ context = ℙ₁{registration, instructor, team, contextActivities, revision,
│              platform, language, statement, extensions}
└─────────────────────────────────────────────────────
```

- 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* \ *agent* represents the component *object* except for its subcomponent *agent*

2.8 Timestamp and Stored Schema

```
┌─ Timestamp ─────────────────────────────────────────
│ timestamp : 𝔽₁ #1
└─────────────────────────────────────────────────────
```

```
┌─ Stored ────────────────────────────────────────────
│ stored : 𝔽₁ #1
└─────────────────────────────────────────────────────
```

- 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

```
┌─ Attachments ────────────────────────────────────────
│ display, description, attachment, attachments : \mathbb{F}_1
│ usageType, sha2, fileUrl, contextntType : \mathbb{F}_1 \#1
│ length : \mathbb{N}
├──────────
│ attachment = {usageType, display, contentType, length, sha2} \cup \mathbb{P}{description, fileUrl}
│ attachments = {a : attachment}
└────────────────────────────────────────────────
```

- 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

```
┌─ Statement ────────────────────────────────────────
│ Id
│ Actor
│ Verb
│ Object
│ Result
│ Context
│ Timestamp
│ Stored
│ Attachements
│ statement : STATEMENT
├──────────
│ statement = {actor, verb, object, stored} \cup
│                \mathbb{P}{id, result, context, timestamp, attachments}
└────────────────────────────────────────────────
```

- 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

$$
\begin{array}{|l}
\hline
\;\underline{Statements}\underline{\hspace{6cm}} \\
\; Statement \\
\; IsoToUnix \\
\; statements : \mathbb{F}_1 \\
\hline
\; statements = \{s : statement \mid \forall s_n : s_i..s_j \bullet i \leq n \leq j \\
\qquad\qquad\qquad \bullet\, convert(s_i.timestamp) \leq convert(s_j.timestamp)\} \\
\hline
\end{array}
$$

- 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 $i \ldots n \ldots j$, 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 respectively.

3.1.2 Collection

a sequence $\langle \ldots \rangle$ of items c such that

$$C = \langle c_i..c_n..c_j \rangle \Rightarrow \{ i \mapsto c_i, n \mapsto c_n, j \mapsto c_j \}$$

$$i \leq n \leq j \Rightarrow i \prec n \prec j \iff i \neq n \neq j$$

And the following free type is introduced for collections

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

$emptyColl$	[the empty Collection $\langle \rangle$]
$append$	[is a constructor and is infered to be an injection]
Map	[a free type introduced bellow]

$$append(emptyColl, c?, 0) = \langle c_0 \rangle \Rightarrow \{ 0 \mapsto c? \}$$

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, Map) but the Scalar null should be avoided. The following free type is introduced for keys.

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

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, Map) The following free type is introduced for values.

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

3.1.5 Map

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

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

This definition is more accurately

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

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

$$\operatorname{id} v_n = k_n$$
$$\operatorname{id} k_n \neq v_n$$
$$\operatorname{id} k_n = k_n$$

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

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

which can all be stated formally as

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

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.

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

such that $\langle\!\langle\!\langle (k_{ni}, v_{ni}) \rangle\!\rangle\!\rangle$ and $\langle\!\langle\!\langle (k_{nj}, v_{nj}) \rangle\!\rangle\!\rangle$ 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)

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

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

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

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

- $head$

- $last$

- $tail$

- $front$

- $\uparrow$

- $\upharpoonright$

- $\frown/$

- disjoint

- partition

- $\otimes$

- $\uplus$

- $\sqcup\!\!\!\sqcap$

- $items$

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)
- Map(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)
- Map(s)
- Statement(s)
- Algorithm State

5 Primitive

A collection of Operations where the output of an Operation o is passed as an argument to the next Operation.

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

Within any given Primitive p, variables local to p and any global variables may be passed as arguments to any o within 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.

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

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.

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)
- Map(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)

- Map(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

Such that the execution of A can be described as

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

6.1 Domain

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

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

6.2 Range

An Algorithm will return an Algorithm State.

- Algorithm State

6.3 Initialization

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

$$
\begin{array}{|l}
\hline Init[KV] \underline{\hspace{5cm}} \\
\hline state?, state!: KV \\
init: KV \rightarrow KV \\
\hline
state! = init(state?) \bullet state! = state? \lor state! \neq state? \\
\hline
\end{array}
$$

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 $body$ of an Algorithms $init$

6.3.1 Domain

- Algorithm State

6.3.2 Range

- Algorithm State

6.4 Relevant?

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

$$
\begin{array}{|l}
\hline Relevant?\,[KV, STATEMENT] \underline{\hspace{4cm}} \\
\hline state?: KV \\
stmt?: STATEMENT \\
relevant?: KV \times STATEMENT \rightarrow Boolean \\
\hline
relevant?\,(state?, stmt?) = true \lor false \\
\hline
\end{array}
$$

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?*

6.4.1 Domain

- Statement
- Algorithm State

6.4.2 Range

- Boolean

6.5 Accept?

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

$$
\begin{array}{|l}
\underline{\quad Accept?\,[KV, STATEMENT]\,} \rule{6cm}{0.4pt} \\
\quad state?: KV \\
\quad stmt?: STATEMENT \\
\quad accept?: KV \times STATEMENT \rightarrow Boolean \\
\hline
\quad accept?\,(state?\,, stmt?\,) = true \vee false \\
\end{array}
$$

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?*

6.5.1 Domain

- Statement
- Algorithm State

6.5.2 Range

- Scalar

6.6 Step

An Algorithm Step consists of a sequential composition of Primitive(s) and therefore sequential composition of all Operation(s) where the output of a Primitive is passed as an argument to the next Primitive.

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

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.

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

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

$$
\begin{aligned}
&S : Collection \\
&stmt_a, stmt_b, stmt_c : STATEMENT \\
&state?\,, step_a!\,, step_b!\,, step_c! : KV \\
&step : KV \times STATEMENT \rightarrow KV \\
\hline
&S = \langle stmt_a\,..\,stmt_b\,..\,stmt_c \rangle \bullet a \leq b \leq c \Rightarrow a \prec b \prec c \iff a \neq b \neq c \\
&step_a! = step(state?\,, stmt_a) \bullet step_a! = state? \vee step_a! \neq state? \\
&step_b! = step(step_a!\,, stmt_b) \bullet step_b! = step_a! \vee step_b! \neq step_a! \\
&step_c! = step(step_b!\,, stmt_c) \bullet step_c! = step_b! \vee step_c! \neq step_b!
\end{aligned}
$$

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

$$
\begin{aligned}
&\underline{Step[KV, STATEMENT]} \\
&state?\,, state! : KV \\
&stmt? : STATEMENT \\
&step : KV \times STATEMENT \rightarrow KV \\
\hline
&state! = step\,(state?\,, stmt?\,) = state? \vee \neq state?
\end{aligned}
$$

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) p chosen for inclusion within $P_{\langle i\,..\,n\,..\,j \rangle}$

- The ordering of $p \in P_{\langle i\,..\,n\,..\,j \rangle}$ which implies the ordering of $o \in p_{\langle i\,..\,n\,..\,j \rangle} \in P_{\langle ii\,..\,ij\,..\,ni\,..\,nj\,..\,ji\,..\,jj \rangle}$

- The key value pair(s) $(K, V) \in stmt$

- The ordering of Statement(s) $stmt \in S_{\langle a\,..\,b\,..\,c \rangle}$

6.6.1 Domain

- Statement

- Algorithm State

6.6.2   Range

- Algorithm State

6.7   Result

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

$$relevant? \prec accept? \prec step \prec result \prec relevant? \iff S \neq \emptyset$$
$$relevant? \prec accept? \prec step \prec result \iff S = \emptyset$$

and does so without preventing subsequent calls of A

$$\begin{array}{l} \underline{\;result[KV,KV]\;}\\[4pt] \quad result!, state?, opt? : KV \\ \quad result : KV \times KV \to KV \\ \hline\\[-8pt] \quad result! = result(state?, opt?) = state? \vee \neq state? \end{array}$$

such that if at some future point j within the timeline $i \ldots n \ldots j$

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

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

$$\begin{array}{l} state_{n-i} = A(state_i, \; S(t_{n-i})) \\ state_{n-1} = A(state_{n-2}, \; S(t_{n-1})) \\ state_n = A(state_{n-1}, \; S(t_n)) \\ state_{j-n} = A(state_n, \; S(t_{j-n})) \\ state_j = A(state_i, \; S(t_{j-i})) \\ \hline\\[-8pt] state_n = state_{n-1} \iff S(t_n) = \emptyset \wedge S(t_{n-1}) \neq \emptyset \\ state_j = state_{j-n} \iff state_{n-i} = state_n = state_{n-1} \end{array}$$

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

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

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

6.7.1 Domain

- Algorithm State

- Option(s)

6.7.2 Range

- Algorithm State

6.8 Algorithm Formal Definition

Given the definitions of Algorithm Components presented above, the previous definition of an Algorithm A

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

can be refined using the helper function $recur$

$$
\begin{array}{|l}
\underline{recur[KV, Collection]} \\
\quad state?, state!: KV \\
\quad S?, S!: Collection \\
\quad recur!: (KV, COllection) \\
\quad recur: KV \times Collection \rightarrow KV \times Collection \\
\hline
\quad recur! = recur(state?, S?) \Rightarrow (state!, S!) \; \bullet \\
\qquad state? = state! \wedge S! = tail(S?)
\end{array}
$$

to illustrate how an Algorithm processes a Collection of Statement(s)

$\boxed{\begin{array}{l}
Algorithm[KV, Collection] \\
\hline
Init[KV] \\
Relevant?\,[KV, STATEMENT] \\
Accept?\,[KV, STATEMENT] \\
Step[KV, STATEMENT] \\
result[KV, KV] \\
recur[KV, Collection] \\
opt?\,, state?\,, state_s!\,, state_S!\,: KV \\
S?\,: Collection \bullet \forall s? \in S?\, \mid s?\,: STATEMENT \\
algorithmIter : KV \times Collection \times KV \to KV \times Collection \\
algorithm : KV \times Collection \times KV \to KV \\
\hline
\end{array}}$

$$state_s! = algorithmIter(state?\,, S?\,, opt?\,) \bullet$$
$$let \quad s_{current} == head(S?\,)$$
$$init! == init(state?\,)$$
$$relevant! == relevant?\,(init!\,, s_{current})$$
$$passon_{relevant}? == recur(init!\,, S?\,) \iff relevant! = false$$
$$accept! == accept?\,(init!\,, s_{current}) \iff relevant! = true$$
$$passon_{accept}? == recur(init!\,, S?\,) \iff accept! = false$$
$$step! == step(init!\,, s_{current}) \iff accept! = true$$
$$passon_{step}? == recur(step!\,, S?\,) \iff tail(S?\,) \neq \emptyset$$
$$result! == result(step!\,, opt?\,) \iff tail(S?\,) = \emptyset$$
$$passon_{result}? == recur(result!\,, S?\,)$$
$$passon_s? == passon_{relevant}? \iff relevant! = false \;\vee$$
$$passon_{accept}? \iff accept! = false \;\vee$$
$$passon_{step}? \iff relevant! = true \wedge accept! = true \wedge tail(S?\,) \neq \emptyset$$
$$passon_{result}? \iff relevant! = true \wedge accept! = true \wedge tail(S?\,) = \emptyset$$
$$\Rightarrow \; = init!\, \mathbin{\raise0.3ex\hbox{$\scriptstyle 9$}} relevant! \;\vee$$
$$= init!\, \mathbin{\raise0.3ex\hbox{$\scriptstyle 9$}} relevant! \mathbin{\raise0.3ex\hbox{$\scriptstyle 9$}} accept! \;\vee$$
$$= init!\, \mathbin{\raise0.3ex\hbox{$\scriptstyle 9$}} relevant! \mathbin{\raise0.3ex\hbox{$\scriptstyle 9$}} accept! \mathbin{\raise0.3ex\hbox{$\scriptstyle 9$}} step! \;\vee$$
$$= init!\, \mathbin{\raise0.3ex\hbox{$\scriptstyle 9$}} relevant! \mathbin{\raise0.3ex\hbox{$\scriptstyle 9$}} accept! \mathbin{\raise0.3ex\hbox{$\scriptstyle 9$}} step! \mathbin{\raise0.3ex\hbox{$\scriptstyle 9$}} result!$$
$$= passon_s? \Rightarrow (first(passon_{relevant}? \vee passon_{accept}? \vee passon_{step} \vee passon_{result}), tail(S?\,))$$
$$state_S! = algorithm(state?\,, S?\,, opt?\,)$$
$$= first(algorithmIter(state?\,, S?\,, opt?\,)^{\#S?}) \bullet$$
$$\forall\, s? \in S?\, \exists_1\, passon_s : KV \times Collection \bullet$$
$$passon_s = algorithmIter(state?\,, S?\,, opt?\,) \bullet$$
$$let\, (state_s == first(passon_s)) \bullet$$
$$passon_{s'} = algorithmIter(state_s, tail(S?\,), opt?\,) \Rightarrow (state_{s'}, tail(tail(S?\,)))$$
$$passon_{s''} = algorithmIter(first(passon_{s'}), second(passon_{s'}), opt?\,)$$

The above introduces $algorithmIter$ to describe the processing per $s \in S$ which
results in $algorithm$ being an iteration of $algorithmIter$ over S. This means
there is a $passon_s$ for all $s \in S$ and the result of $algorithm(state, S, opt)$ is

$$first(algorithmIter(state, S, opt)^{\#S}) \Rightarrow first(passon_{result}) \iff second(passon_{result}) = \emptyset$$

7 Foundational Operations

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

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

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

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

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

$$
\begin{aligned}
X = &\langle x_0, x_1, x_2, x_3, x_4 \rangle \\
&x_0 = 0 \\
&x_1 = foo \\
&x_2 = \langle baz,\ qux \rangle \\
&x_3 = \langle\!\langle abc \mapsto 123,\ def \mapsto 456 \rangle\!\rangle \\
&x_4 = \langle \langle\!\langle ghi \mapsto 789,\ jkl \mapsto 101112 \rangle\!\rangle,\ \langle\!\langle ghi \mapsto 131415,\ jkl \mapsto 161718 \rangle\!\rangle \rangle
\end{aligned}
$$

$$
\begin{aligned}
array?\,(X) &= true & &\text{[collection by definition]} \\
array?\,(x_2) &= true & &\text{[collection of } 0 \mapsto baz,\ 1 \mapsto qux] \\
array?\,(x_4) &= true & &\text{[collection of maps]} \\
array?\,(x_0) &= false & &\text{[Scalar]} \\
array?\,(x_1) &= false & &\text{[String]} \\
array?\,(x_3) &= false & &\text{[Map]}
\end{aligned}
$$

7.1.2 Append

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

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

where $coll'$ is the items in $coll?$ up to idx but the value at $idx?$ is replaced with $v?$ and $coll''$ is the items in $coll?$ from idx to $\# coll?$ and is inclusive of $coll?_{idx?}$. The composition of the two Collections results in $coll!$ which contains $idx? \mapsto v?$ and all subsquent $idx \mapsto v \in coll?$ are now $idx + 1 \mapsto v_{idx}$. The following example illustrates these properties.

$$X = \langle x_0, x_1, x_2 \rangle$$
$$x_0 = 0$$
$$x_1 = foo$$
$$x_2 = \langle a, b, c \rangle$$
$$v? = bar$$
$$append(X, v?, 0) = \langle bar, 0, foo, \langle a, b, c \rangle \rangle$$
$$append(X, v?, 1) = \langle 0, bar, foo, \langle a, b, c \rangle \rangle$$
$$append(X, v?, 2) = \langle 0, foo, bar, \langle a, b, c \rangle \rangle$$
$$append(X, v?, 3) = \langle 0, foo, \langle a, b, c \rangle, bar \rangle$$

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

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

such that

$$X = \langle x_0, x_1, x_2 \rangle$$
$$x_0 = 0$$
$$x_1 = foo$$
$$x_2 = baz$$

$remove(X, 0) = \langle foo, baz \rangle$	[0 was removed from X]
$remove(X, 1) = \langle 0, baz \rangle$	[foo was removed from X]
$remove(X, 2) = \langle 0, foo \rangle$	[baz was removed from X]
$remove(X, 3) = \langle 0, foo, baz \rangle = X$	[nothing at 3, X unaltered]

7.1.4   At Index

The operation *atIndex* will return the Value at a specified index or depth of indices

$$
\begin{array}{|l}
\underline{\;atIndex[Collection, \mathbb{N} \vee Collection]\;} \\
idx? : \mathbb{N} \\
coll? : Collection \\
idxColl? : Collection \\
atIndex : Collection \times \mathbb{N} \to V \ \vee \ Collection \times Collection \to V \\
\hline
idxColl? = \langle \forall\, i \in idxColl? \bullet i : \mathbb{N} \rangle \\
atIndex(coll?, idx?) = (head(idx? \upharpoonright coll?)) \iff \#(idx? \upharpoonright coll?) = 1 \\
atIndex(coll?, idx?) = (idx? \upharpoonright coll?) \iff \#(idx? \upharpoonright coll?) \geq 2 \\
atIndex(coll?, idxColl?) = \forall i : 0 \mathinner{.\,.} j \in idxColl \bullet \\
\qquad\qquad\qquad\qquad coll! = atIndex((atIndex(atIndex(coll?, i)), i+1), j)
\end{array}
$$

$$X = \langle x_0, x_1, x_2 \rangle$$
$$x_0 = 0$$
$$x_1 = foo$$
$$x_2 = \langle a, b, c \rangle$$

$atIndex(X, 0) = 0$	
$atIndex(X, 1) = foo$	
$atIndex(X, \langle 1, 0 \rangle) = f$	[foo $\Rightarrow \langle f, o, o \rangle$]
$atIndex(X, \langle 1, 2 \rangle) = o$	[foo $\Rightarrow \langle f, o, o \rangle$]
$atIndex(X, 2) = \langle a, b, c \rangle$	
$atIndex(X, \langle 2, 1 \rangle) = b$	
$atIndex(X, 3) = \langle \rangle$	
$atIndex(X, \langle 2, 3 \rangle) = \langle \rangle$	

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

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

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

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

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

$$
\begin{aligned}
X &= \langle\!\langle x_0, x_1, x_2, x_3, x_4 \rangle\!\rangle \\
x_0 &= 0 \\
x_1 &= foo \\
x_2 &= \langle baz, \; qux \rangle \\
x_3 &= \langle\!\langle abc \mapsto 123, \; def \mapsto 456 \rangle\!\rangle \\
x_4 &= \langle \langle\!\langle ghi \mapsto 789, \; jkl \mapsto 101112 \rangle\!\rangle, \; \langle\!\langle ghi \mapsto 131415, \; jkl \mapsto 161718 \rangle\!\rangle \rangle \\
\end{aligned}
$$

$$
\begin{array}{ll}
map?\,(X) = true & \text{[KV by definition]} \\
map?\,(x_3) = true & \text{[KV]} \\
map?\,(x_2) = false & \text{[Collection]} \\
map?\,(x_4) = false & \text{[Collection of maps]} \\
map?\,(x_0) = false & \text{[Scalar]} \\
map?\,(x_1) = false & \text{[String]} \\
\end{array}
$$

7.2.2 Associate

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

$$\begin{array}{|l}
\underline{\;Associate[KV, K, V]\;\rule{3cm}{0pt}} \\
m?, m!, m' : KV \\
k? : K \\
v? : V \\
associate : KV \times K \times V \rightarrowtail KV \\
\rule{6cm}{0.4pt} \\
m! = associate(m?, k?, v?) \bullet \\
\quad let\; m' == m? \lhd k? \Rightarrow \\
\qquad\qquad (dom\; m' = dom\,(m? \setminus k?)) \wedge \\
\qquad\qquad (m? \setminus m' = k? \iff k? \in m?) \wedge \\
\qquad\qquad (m? \setminus m' = \emptyset \iff k? \notin m? \Rightarrow m? = m') \\
\quad = \langle\!\langle k? \mapsto v? \rangle\!\rangle \cup m'
\end{array}$$

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

$$\begin{array}{|l}
(k?, m?_{k?}) \in m? \vee (k?, m?_{k?}) \notin m? \\
(k?, m?_{k?}) \notin m! \\
(k?, v?) \in m! \\
\rule{5cm}{0.4pt} \\
m! = associate(m?, k?, v?)
\end{array}$$

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

$$\begin{array}{|l}
m' : KV \mid m' = m? \lhd k? \Rightarrow m' \lhd (m? \setminus k?) \\
\rule{7cm}{0.4pt} \\
dom\; m? = \{\, k_i : K \mid 0\mathinner{..}\#m? \bullet k_i \in m? \wedge 0 \leq i \leq \#m? \,\} \\
dom\; m' = \{\, k'_i : K \mid 0\mathinner{..}\#m' \bullet k'_i \in m? \wedge k'_i \neq k? \wedge 0 \leq i \leq \#m' \} \\
dom\; m' = dom\; m? \iff k? \notin m? \Rightarrow \forall k_i \in m? \mid k_i \neq k? \\
\#m' = \#m? \iff k? \notin m? \\
\#m' = \#m? - 1 \iff k? \in m?
\end{array}$$

and its usage within the definition of *associate*.

$$\begin{array}{|l}
m! = m? \cup \langle\!\langle k? \mapsto v? \rangle\!\rangle \Rightarrow k? \notin m? \\
m! = m' \cup \langle\!\langle k? \mapsto v? \rangle\!\rangle \Rightarrow m' \neq m? \wedge k? \in m?
\end{array}$$

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

$$\begin{array}{ll}
M = \langle\!\langle k_0 v_{k_0}, k_1 v_{k_1} \rangle\!\rangle & \\
\quad k_0 = abc \;\wedge v_{k_0} = 123 & [k_0 v_{k_0} = abc \mapsto 123] \\
\quad k_1 = def \;\wedge v_{k_1} = xyz \mapsto 456 & [k_1 v_{k_1} = def \mapsto xyz \mapsto 456] \\
associate(M, baz, foo) = \langle\!\langle abc \mapsto 123,\; def \mapsto xyz \mapsto 456,\; baz \mapsto foo \rangle\!\rangle & \\
associate(M, abc, 321) = \langle\!\langle abc \mapsto 321,\; def \mapsto xyz \mapsto 456 \rangle\!\rangle &
\end{array}$$

7.2.3 Dissociate

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

$$dissociate(KV, k) \rightarrow KV'$$

such that if KV is a collection of Key Value pair(s) $< k_0 v_{k_0}, k_1 v_{k_1} >$ where

$$k_0 = abc \ \wedge v_{k_0} = 123$$

$$\Rightarrow$$

$$k_0 v_{k_0} = abc \mapsto 123$$

and

$$k_1 = def \ \wedge v_{k_1} = xyz \mapsto 456$$

$$\Rightarrow$$

$$k_1 v_{k_1} = def \mapsto xyz \mapsto 456$$

such that

$$KV =< abc \mapsto 123, \ def \mapsto xyz \mapsto 456 >$$

Which means $disocciate(KV, k)$ results in

- $KV \neq KV' \iff k \in KV$ where $k \mapsto v_k$ is removed from KV

$$dissociate(KV, abc) =< def \mapsto xyz \mapsto 456 >= KV'$$

$$dissociate(KV, def) =< abc \mapsto 123 >= KV'$$

- $KV = KV' \iff k \notin KV$ where nothing is removed from KV

$$dissociate(KV, cba) =< abc \mapsto 123, \ def \mapsto xyz \mapsto 456 >= KV' = KV$$

7.2.4 At Key

The operation $atKey$ will return the Value v at some specified

- Top level Key k within KV

$$atKey(KV, k) \rightarrow v$$

- Nested location $K =< k_i..k_n..k_j >$ within KV

$$atKey(KV, K) \rightarrow v$$

such that if KV is a collection of Key Value pair(s) $< k_0 v_{k_0}, k_1 v_{k_1} >$ where

$$k_0 = abc \ \wedge v_{k_0} = 123$$

$$\Rightarrow$$

$$k_0 v_{k_0} = abc \mapsto 123$$

and

$$k_1 = def \ \wedge v_{k_1} = xyz \mapsto 456$$

$$\Rightarrow$$

$$k_1 v_{k_1} = def \mapsto xyz \mapsto 456$$

such that

$$KV =< abc \mapsto 123, \ def \mapsto xyz \mapsto 456 >$$

When k is a single Key

- $k \notin KV$, $atKey$ will return the representation of nothingness

$$atKey(KV, cba) = nil$$

- $k \in KV$, $atKey$ will return v_k

$$atKey(KV, k_0) \Rightarrow atKey(KV, abc) = 123$$

$$atKey(KV, k_1) \Rightarrow atKey(KV, def) = xyz \mapsto 456$$

When K is a Collection of Key(s)

- $K \notin KV$, $atKey$ will return the representation of nothingness

$$atKey(KV, < cba, 321 >) = nil$$

- $k_i \in KV \ \wedge k_j \notin KV$, $atKey$ will return the representation of nothingness

$$atKey(KV, < def, abc >) = nil$$

- $k_i \in KV \ \wedge k_j \in KV$, $atKey$ will return the nested value

$$atKey(KV, < def, xyz >) = 456$$

7.3 Utility

7.3.1 Map

The *map* operation takes in an Operation O or Primitive P, Collection *coll?* and additional Arguments *args?* and returns a modified Collection *coll!* where each member $coll!_i$ is the result of passing $coll?_i$ and *args?* to $P \vee O$. The ordering of *coll?* is maintained within *coll!*

$$
\begin{array}{|l}
\underline{map[O \vee P, Collection, V]} \\
operation? : O \\
primitive? : P \\
args? : V \\
coll?\,, coll! : Collection \\
map : (O \vee P \times Collection \times V \rightarrow Collection) \vee (O \vee P \times Collection \rightarrow Collection) \\
\hline
coll! = map(operation?\,, coll?\,) \bullet \\
\qquad\qquad \langle\, \forall i : 0\,..\,j \in coll? \mid \\
\qquad\qquad\quad operation?\,(coll?_0\,) \frown operation?\,(coll?_i\,) \frown operation?\,(coll?_j\,)\rangle \\
coll! = map(operation?\,, coll?\,, args?\,) \bullet \\
\qquad\qquad \langle\, \forall i : 0\,..\,j \in coll? \mid \\
\qquad\qquad\quad operation?\,(coll?_0\,, args?\,) \frown operation?\,(coll?_i\,, args?\,) \frown operation?\,(coll?_j\,, args?\,)\rangle \\
coll! = map(primitive?\,, coll?\,) \bullet \\
\qquad\qquad \langle\, \forall i : 0\,..\,j \in coll? \mid \\
\qquad\qquad\quad primitive?\,(coll?_0\,) \frown primitive?\,(coll?_i\,) \frown primitive?\,(coll?_j\,)\rangle \\
coll! = map(primitive?\,, coll?\,, args?\,) \bullet \\
\qquad\qquad \langle\, \forall i : 0\,..\,j \in coll? \mid \\
\qquad\qquad\quad primitive?\,(coll?_0\,, args?\,) \frown primitive?\,(coll?_i\,, args?\,) \frown primitive?\,(coll?_j\,, args?\,)\rangle \\
\end{array}
$$

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

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

7.3.2 Iso To Unix Epoch

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

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

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

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

7.3.3 Timeunit To Number of Seconds

The operation $timeunit \rightarrow seconds$ will return the number of seconds corresponding to the input $Timeunit$

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

such that the following schema defines $timeunit \rightarrow second$

$\_\_toSeconds[Timeunit]$ $_____$

$timeunit? : Timeunit$
$timeunit \rightarrow secdons : Timeunit \rightarrow \mathbb{N}$

$timeunit \rightarrow seconds(timeunit?) = 1 \iff timeunit? = second$
$timeunit \rightarrow seconds(timeunit?) = 60 \iff timeunit? = minute$
$timeunit \rightarrow seconds(timeunit?) = 3600 \iff timeunit? = hour$
$timeunit \rightarrow seconds(timeunit?) = 86400 \iff timeunit? = day$
$timeunit \rightarrow seconds(timeunit?) = 604800 \iff timeunit? = week$
$timeunit \rightarrow seconds(timeunit?) = 2629743 \iff timeunit? = month$
$timeunit \rightarrow seconds(timeunit?) = 31556926 \iff timeunit? = year$

---

8 Common Primitives

8.1 Accumulate

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

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

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

8.1.1 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 $k \mapsto v \in KV$

8.1.2 Relevant Operations

The primitive *accumulate* uses the operations

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

8.1.3 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

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

8.1.4 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

- $objectAtPath? = true$

45

- 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

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

- $objectAtPath? = false \land arrayAtpath? = false$

  - creation of a new array which contains $state_{path}$ and v at index k

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

8.1.5 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

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

8.2 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

8.2.1 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

8.2.2 Relevant Operations

The primitive *atJsonPath* uses the operations

- atKey

- atIndex

- append

- count

8.2.3 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

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

8.2.4 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

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

8.2.5 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
$$atJsonPath(source, \$.d) = foo$$

8.3 Rate Of

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

8.3.1 Arguments

- *nOccurances* is the number of times something happened and should be an Integer

- *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

8.3.2 Relevant Operations

- isoToUnixEpoch

- $timeunit \rightarrow seconds$

8.3.3 Summary

$rateOf$ determines the number of seconds within the interval $start \rightarrow end$

$$intervalSeconds = isoToUnixEpoch(end) - isoToUnixEpoch(start)$$

and resolves the numer of seconds corresponding to $unit$

$$unitSeconds = timeunit \rightarrow seconds(unit)$$

so that the interval can be converted from Seconds to $unit$

$$per = intervalSeconds/unitSeconds$$

and the rate can be calculated

$$rateOf(nOccurances, start, end, unit) \equiv nOccurances \div per$$

8.3.4 Usage of Operations

The Operations used within $rateOf$ convert from a String to a Integer

- $isoToUnixEpoch$ is used to convert

$$end \ \wedge start \rightarrow \mathbb{R}$$

- $timeunit \rightarrow seconds$ is used to convert

$$unit \rightarrow \mathbb{R}$$

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

8.3.5 Example output

Given an example $start$ and end

$$start = 2015 - 11 - 18T12 : 17 : 00Z$$

$$end = 2015 - 11 - 18T14 : 17 : 00Z$$

Then the Unix Epoch of each is

$$nStart = 1447849020$$

$$nEnd = 1447856220$$

Which provides an interval range (in seconds)

$$intervalSeconds = nEnd - nStart = 7200$$

which is divided by $timeunit \rightarrow seconds(unit)$ to derive per $unit$

$$per = 7200 \iff unit = second \Rightarrow 7200/1$$

$$per = 120 \iff unit = minute \Rightarrow 7200/60$$

$$per = 2 \iff unit = hour \Rightarrow 7200/3600$$

such that if

$$nOccurances = 10$$

and $unit = second$ then the output is 0.001389 occurances per second within $start \rightarrow end$

$$rateOf(nOccurances, start, end, second) \equiv 10/7200 \equiv 0.001389$$

and $unit = minute$ then the output is 0.0833 occurances per minute within $start \rightarrow end$

$$rateOf(nOccurances, start, end, minute) \equiv 10/120 \equiv 0.0833$$

and $unit = hour$ then the output is 5 occurances per hour within $start \rightarrow end$

$$rateOf(nOccurances, start, end, hour) \equiv 10/2 \equiv 5$$

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
  $$start(state, stmt) = currentTime$$

  $$\iff$$

  $$start_{state} = nil$$

  $$\lor$$

  $$inSeconds_{stmt} \leq inSeconds_{start}$$

  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
  $$end(state, stmt) = currentTime$$

  $$\iff$$

  $$end_{state} = nil$$

  $$\lor$$

  $$inSeconds_{stmt} \geq inSeconds_{end}$$

  otherwise
  $$end(state, stmt) = end_{state}$$

- what should *statementCount* be?

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

$$\lor$$

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

- 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 $timeUnit$

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

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.[345]

---

[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

$$
\begin{array}{|l}
\hline
\;RateOfCompletion \underline{\hspace{4cm}} \\
\hline
\; Statements \\
\; S_{completions} : \mathbb{F}_1 \\
\; S_{grouped}, S_{timeunit}, S_{processed} : \mathbb{F} \\
\hline
\; S_{completions} = statements \\
\; S_{grouped} = \{byId : \text{seq}_1\ statement\} \\
\; S_{withRate} = \{byGroup : (\text{seq}_1\ statement, \mathbb{N})\} \\
\; S_{processed} = \{rate : (id, \mathbb{N}, TIMEUNIT)\} \\
\hline
\end{array}
$$

- 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 $TIMEUNIT$

66

13.6.3 Initial System State

```
┌─ InitRateOfCompletion ─────────────────────────────────
│ RateOfCompletion
│ T : TIMEUNIT
├────────────────────────────────────────────────────────
│ S_completions ≠ ∅
│ S_grouped = ∅
│ S_withRate = ∅
│ S_processed = ∅
│ T = {day}
└────────────────────────────────────────────────────────
```

$$S_{completions} \neq \emptyset$$
$$S_{grouped} = \emptyset$$
$$S_{withRate} = \emptyset$$
$$S_{processed} = \emptyset$$
$$T = \{day\}$$

- 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

```
┌─ IsoToUnix ────────────────────────────────────────────
│ convert : 𝔽₁ → ℕ#1
│ c? : 𝔽₁
│ c! : ℕ#1
├────────────────────────────────────────────────────────
│ c! = convert(c?)
└────────────────────────────────────────────────────────
```

$$convert : \mathbb{F}_1 \to \mathbb{N}\#1$$
$$c? : \mathbb{F}_1$$
$$c! : \mathbb{N}\#1$$
$$c! = convert(c?)$$

- 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.

$\begin{array}{l}\underline{\quad CalcRateByUnit\quad}\\ \quad Statement\\ \quad IsoToUnix\\ \quad CountPerGroup\\ \quad unit?:TIMEUNIT\\ \quad s?,s!:\mathbb{F}\\ \quad r:\mathbb{N}\\ \quad rate:(\mathbb{F},TIMEUNIT)\to\mathbb{F}\\ \hline \end{array}$

$unit? = \{second\} \Rightarrow 1 \vee \{minute\} \Rightarrow 60 \vee \{hour\} \Rightarrow 3600 \vee$
$\qquad \{day\} \Rightarrow 86400 \vee \{week\} \Rightarrow 604800 \vee$
$\qquad \{month\} \Rightarrow 2629743 \vee \{year\} \Rightarrow 31556926$
$s? = \{g : seq_1\, statement\}$
$s! = rate(s?, unit?)$
$s! = \{s : (g, r) \mid \forall g_n : g_i..g_j \bullet i \leq n \leq j \bullet \exists s_n : (g_n, r_n) \bullet$
$\qquad r_n = count(g_n) \div ((convert(last\ g_n.timestamp) - convert(head\ g_n.timestamp)) \div unit?)\}$

- 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 $last\ g_n.timestamp - head\ g_n.timestamp$

- the output of the function $rate$ is $s!$, the set of all s_n

13.6.5 Processes Results

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

- 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

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

- 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 $S_{completion} \neq \emptyset$ **do**
    foreach $s \in S_{completion}$ **do**
        $id \leftarrow s.object.id$;
        $ts \leftarrow convert(s.timestamp)$;
        if $id \notin context$ **then**
            do
            $times = [ts]$;
            $context' \leftarrow \{id : times\}$;
            $S'_{completion} \leftarrow S_{completion} \setminus s$;
            recur $context', S'_{completion}$;
        else
            do
            $times' \leftarrow context.id \frown ts$;
            $context' \leftarrow \{id : times'\}$;
            $S'_{completion} \leftarrow S_{completion} \setminus s$;
            recur $context', S'_{completion}$;
        end
    end
end
foreach $k \in context'$ **do**
    $allTs \leftarrow context'.k$;
    $totalDuration \leftarrow max(allTs) - min(allTs)$;
    $totalCount \leftarrow count(allTs)$;
    $rate \leftarrow totalCount \div (totalDuration \div timeUnit)$;
    $subVec = [k, rate, timeUnit]$;
    $ratePerObjTu' \leftarrow ratePerObjTu \frown subVec$;
    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

Rate of Completions
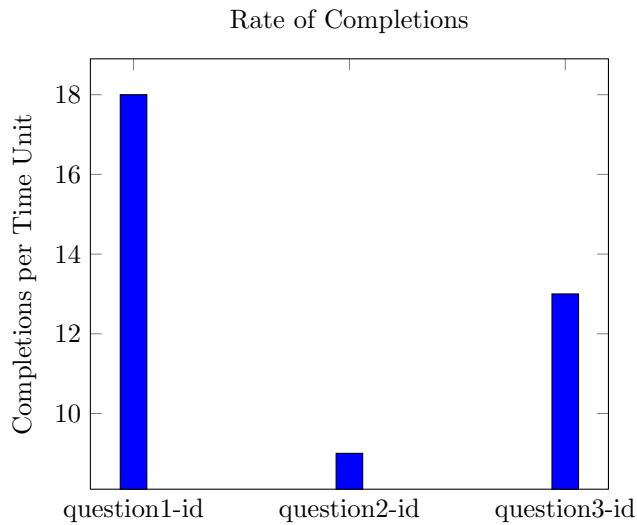
13.11   Prototype Improvement Suggestions

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.[789]

---

[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

```
┌─ TimelineLearnerSuccess ─────────────────────────
│ Statements
│ S_all : 𝔽₁
│ S_completion, S_success, S_processed : 𝔽
├───────────────────────────────────────────────
│ S_all = statements
│ S_completion ⊆ S_all
│ S_success ⊆ S_completion
│ S_processed = {pair : (statement.timestamp, ℕ)}
└───────────────────────────────────────────────
```

- 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

75

14.6.3 Initial System State

```
┌─ InitTimelineLearnerSuccess ─────────────────────────
│ TimelineLearnerSuccess
├──────────────────────────────────────────────────────
│ $S_{all} \neq \emptyset$
│ $S_{completion} = \emptyset$
│ $S_{success} = \emptyset$
│ $S_{processed} = \emptyset$
```

- 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

```
┌─ Completion ─────────────────────────────────────────
│ Statement
│ $completion : STATEMENT \nrightarrow \mathbb{F}$
│ $s? : STATEMENT$
│ $s! : \mathbb{F}$
├──────────────────────────────────────────────────────
│ $s? = statement$
│ $s! = completion(s?)$
│ $completion(s?) = $ **if** $s?.verb.id : COMPLETION$
│                        **then** $s! = s?$
│                        **else** $s! = \emptyset$
```

- 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

```
┌─ FilterForCompletion ────────────────────────────────
│ $\Delta TimelineLearnerSuccess$
│ Completion
│ $completions : \mathbb{F}$
├──────────────────────────────────────────────────────
│ $completions \subseteq S_{all}$
│ $completions' = \{s : STATEMENT \,|\, completion(s) \neq \emptyset\}$
│ $S'_{completion} = S_{completion} \cup completions'$
```

- 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

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

- 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

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

- 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

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

- 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

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

- 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 aruguments 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

$FilterStatements \mathrel{\widehat{=}} FilterForCompletion \mathbin{_9^\circ} FilterForSuccess$

- The schema *FilterStatements* is the sequential composition of operation schemas *FilterForCompletion* and *FilterForSuccess*

- *FilterForCompletion* happens before *FilterForSuccess*

$ProcessedStatements \mathrel{\widehat{=}} FilterStatements \mathbin{_9^\circ} ProcessStatements$

- The schema *ProcessedStatements* is the sequential composition of operation schemas *FilterStatements* and *ProcessStatements*

- *FilterStatements* happens before *ProcessStatements*

14.6.8 Return

$$
\begin{array}{|l}
\hline
\_\,Return \rule{3cm}{0pt} \\
\hline
\Xi TimelineLearnerSuccess \\
ProcessedStatements \\
S_{processed}! : \mathbb{F} \\
\hline
S_{processed}! = S_{processed} \\
\hline
\end{array}
$$

- 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 $S_{all} \neq \emptyset$ **do**
    foreach $s \in S_{all}$ **do**
        if $s.verb.id = COMPLETION$ **then**
            do
            $S'_{completion} \leftarrow s \cup S_{completion};$
            $S'_{all} \leftarrow S_{all} \setminus s$;
            recur $S'_{completion}, S'_{all};$
        else
            do
            $S'_{all} \leftarrow S_{all} \setminus s;$
            recur $S'_{all};$
        end
    end
end
while $S'_{completion} \neq \emptyset$ **do**
    foreach $sc \in S'_{completion}$ **do**
        if $sc.result.success = SUCCESS$ **then**
            do
            $S'_{success} \leftarrow sc \cup S_{success};$
            $S'_{completion} \leftarrow S_{completion} \setminus sc;$
            recur $S'_{success}, S'_{completion};$
        else
            do
            $S'_{completion} \leftarrow S_{completion} \setminus sc;$
            recur $S'_{completion};$
        end
    end
end
foreach $ss \in S'_{success}$ **do**
    $raw? \leftarrow ss.result.score.raw;$
    $max? \leftarrow ss.result.score.max;$
    $min? \leftarrow ss.result.score.min;$
    $scaled \leftarrow scale(raw?, min?, max?);$
    $subVec \leftarrow [ss.timestamp, scaled];$
    $coll' \leftarrow coll \frown subVec;$
    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

Timeline of Learner Success
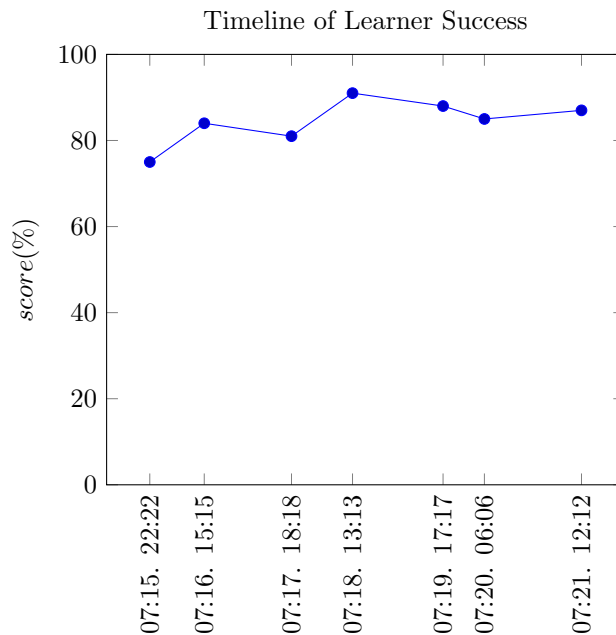
14.11 Prototype Improvement Suggestions

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

$MostDifficultAssessmentQuestions$
Statements
$S_{all} : \mathbb{F}_1$
$S_{incorrect}, S_{grouped}, S_{processed} : \mathbb{F}$

$S_{all} = statements$
$S_{incorrect} \subseteq S_{all}$
$S_{grouped} = \{groups : \text{seq}_1\ statement\}$
$S_{processed} = \{pair : (id, \mathbb{N})\}$

- 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

$InitMostDifficultAssessmentQuestions$
$MostDifficultAssessmentQuestions$

$S_{all} \neq \emptyset$
$S_{incorrect} = \emptyset$
$S_{grouped} = \emptyset$
$S_{processed} = \emptyset$

- 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

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

- 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

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

- 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

$$
\begin{array}{|l}
\hline
\_\,GroupByActivityId \rule[-0.3em]{0pt}{1em} \\[-0.2em]
\hline
\quad Statements \\
\quad g? : \mathbb{F} \\
\quad g! : \mathbb{F} \\
\quad group : \mathbb{F} \to \mathbb{F} \\
\hline
\quad g? = statements \Rightarrow \{g : statement\} \\
\quad g! = group(g?) \\
\quad g! = \{groups : \mathrm{seq}_1\, statement \mid \\
\qquad \mathbf{let}\ \mathrm{seq}_1\, statement_i..statement_j == \mathrm{seq}_1\, s_i..s_j \bullet \\
\qquad \forall s_n : s_i..s_j \bullet i \le n \le j \bullet s_i.object.id = s_j.object.id = s_n.object.id\} \\
\hline
\end{array}
$$

- 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 $\$.object.id$

$$
\begin{array}{|l}
\hline
\_\,CountPerGroup \rule[-0.3em]{0pt}{1em} \\[-0.2em]
\hline
\quad Statement \\
\quad c? : \mathrm{seq}_1\, statement \\
\quad c! : \mathbb{N} \\
\quad count : \mathrm{seq}_1\, statement \to \mathbb{N} \\
\hline
\quad c! = count(c?) \\
\quad c! \ge 1 \\
\quad count(c?) = \forall c_n? : \langle c?_i..c?_j \rangle \bullet i \le n \le j \wedge i = 0 \bullet \\
\qquad\qquad \exists_1 c! : \mathbb{N} \bullet \mathbf{if}\ n = i\ \mathbf{then}\ c! = n + 1\ \mathbf{else}\ c! = j + 1 \\
\hline
\end{array}
$$

- 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

$$
\begin{array}{|l}
\hline \textit{AggregateQuestionStatements} \underline{\hspace{4cm}} \\
\Delta MostDifficultAssessmentQuestions \\
FilterForIncorrect \\
GroupByActivityId \\
CountPerGroup \\
grouped, processed : \mathbb{F} \\
\hline
grouped = \emptyset \\
grouped' = group(S_{incorrect}) \\
S'_{grouped} = S_{grouped} \cup grouped' \\
processed \subseteq S'_{grouped} \\
processed' = \{p : (id, \mathbb{N}) \mid \\
\qquad\qquad \textbf{let } \{\langle processed_i \rangle .. \langle processed_j \rangle\} == \{g_i .. g_j\} \bullet \\
\qquad\qquad i \leq n \leq j \bullet \forall g_n : g_i .. g_j \bullet \exists p_n : p_i .. p_j \bullet \\
\qquad\qquad first\, p_n = head\, g_n.object.id \wedge second\, p_n = count(g_n) \\
S'_{processed} = S_{processed} \cup processed' \\
\hline
\end{array}
$$

- 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

$ProcessedQuestions \mathrel{\widehat{=}} FilterForIncorrect \mathbin{_9^o} AggregateQuestionStatements$

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

- $FilterForIncorrect$ happens before $AggregateQuestionStatements$

15.6.7 Return

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

- 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 $S_{all} \neq \emptyset$ **do**

    foreach $s \in S_{all}$ **do**

        if $s.result.success = INCORRECT$ **then**

            do

            $S'_{incorrect} \leftarrow s \cup S_{incorrect}$;
            $S'_{all} \leftarrow S_{all} \setminus s$;
            recur $S'_{all}, S'_{incorrect}$;

        else

            do

            $S'_{all} \leftarrow S_{all} \setminus s$;
            recur S'_{all}

        end

    end

end
while $S'_{incorrect} \neq \emptyset$ **do**

    foreach $si \in S'_{incorrect}$ **do**

        $id \leftarrow si.object.id$;
        if $id \notin context$ **then**

            do

            $count = 1$;
            $context' \leftarrow \{id : count\}$;
            $S'_{incorrect} \leftarrow S_{incorrect} \setminus si$;
            recur $context', S'_{incorrect}$;

        else

            do

            $count' \leftarrow inc(context.id)$;
            $context' \leftarrow \{id : count'\}$;
            $S'_{incorrect} \leftarrow S_{incorrect} \setminus si$;
            recur $context', S'_{incorrect}$;

        end

    end

end
foreach $id \in context'$ **do**

    $IdToCount \leftarrow [id, context.id]$;

    $display' \leftarrow display \frown IdToCount$;
    recur $display'$

end
return $display'' \leftarrow$ **take(sortBySubArray(**$display'$**),** $displayN$**)**

90

- 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

Most Difficult Assessment Questions
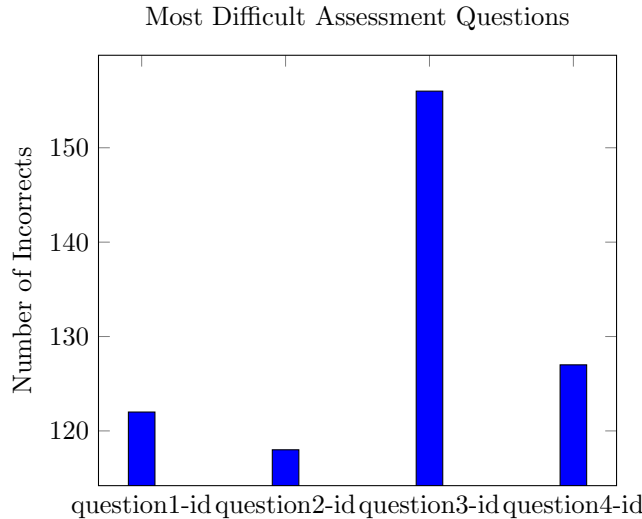
15.11   Prototype Improvement Suggestions

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

$$
\begin{array}{l}
\rule{5mm}{0.4pt}\ FollowedRecommendations \ \rule{40mm}{0.4pt} \\[4pt]
Statements \\
CountPerGroup \\
S_{recommended}, S_{launched} : \mathbb{F}_1 \\
ordered_L, ordered_R, grouped_{launched}, grouped_{recommended}, \\
onlyRecommended, cPerGroup_{launched}, cPerGroup_{recommended}, \\
cPerGroup_{followed}, combined : \text{seq} \\
t_{start}, N_{launched}, N_{recommended}, N_{followed}, P_{followed}, P_{dueto} : \mathbb{N} \\
tr_{start}, tr_{end} : \mathbb{F} \\
unit? : TIMEUNIT \\[4pt]
\rule{60mm}{0.4pt} \\[4pt]
S_{recommended} = statements \\
S_{launched} = statements \\
combined = \langle(tr_{start}, tr_{end}, N_{launched}, N_{recommended}, N_{followed}, P_{followed}, P_{dueto})\rangle \\
count(grouped_{launched}) = count(grouped_{recommended}) \\
count(onlyRecommended) = count(grouped_{launched}) \Rightarrow \\
count(onlyRecommended) = count(grouped_{recommended}) \\
count(cPerGroup_{launched}) = count(cPerGroup_{followed}) = count(cPerGroup_{recommended})
\end{array}
$$

- $S_{recommended}, S_{launched}$ 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

$\underline{\quad InitFollowedRecommendations \underline{\qquad\qquad\qquad\qquad\qquad\qquad}}$
$\quad FollowedRecommendations$

$S_{recommended} \neq \emptyset$
$S_{launched} \neq \emptyset$
$unit? = \{day\}$
$ordered_L = \langle\rangle$
$ordered_R = \langle\rangle$
$grouped_{launched} = \langle\rangle$
$grouped_{recommended} = \langle\rangle$
$onlyRecommended = \langle\rangle$
$cPerGroup_{launched} = \langle\rangle$
$cPerGropu_{recommended} = \langle\rangle$
$cPerGroup_{followed} = \langle\rangle$
$combined = \langle\rangle$
$t_{start} = 0$
$N_{launched} = 0$
$N_{recommended} = 0$
$N_{followed} = 0$
$P_{followed} = 0$
$P_{dueto} = 0$

- $S_{recommended}$ 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

$\text{\textemdash} SortByTimestamp \text{\textemdash}$
$Statement$
$IsoToUnix$
$orderByTimestamp : \mathbb{F}_1 \rightarrow seq_1$
$o? : \mathbb{F}_1$
$o! : seq_1\, statement$
_____
$o? = \{o : statement\}$
$o! = orderByTimestamp(o?)$
$o! = \langle o_i..o_j \rangle \bullet \forall o_n : o_i..o_j \bullet o_n : STATEMENT \wedge i \leq n \leq j \bullet$
$\qquad\qquad convert(o_i.timestamp) \leq convert(o_n.timestamp) \leq convert(o_j.timestamp)$

- 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.

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

- 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.

$\quad$ _GroupByTimeUnit_____

$Statement$
$IsoToUnix$
$WithinRange$
$groupByTimeUnit : (\text{seq}_1, \mathbb{N}, TIMEUNIT) \rightarrow \text{seq}_1$
$g? , g! : \text{seq}_1$
$t_{start}? : \mathbb{N}$

$\rule{4cm}{0.4pt}$

$g? = \langle g?_i .. g?_j \rangle \bullet \forall g?_n : g?_i .. g?_j \bullet i \leq n \leq j \bullet g?_n = statement \land$
$\qquad convert(g?_i.timestamp) \leq convert(g?_n.timestamp) \leq convert(g?_j.timestamp)$
$g! = groupByTimeUnit(g? , t_{start}? \, state? , unit?)$
$g! = \langle g : seq \mid \forall g?_n : g?_i .. g?_j \bullet \exists_1 \langle g_r \rangle : \langle g_q \rangle .. \langle g_s \rangle \bullet q \leq r \leq s \land r = state? \bullet$
$\qquad\qquad \textbf{if } withinRange(convert(g?_n.timestamp), t_{start}, r, unit?) = \{TRUE\}$
$\qquad\qquad\qquad \textbf{then } g? \upharpoonright g?_n \land g?_n \text{ in } \langle g_r \rangle \Rightarrow \langle g_r \rangle = \langle g_{ri}..g_{rn}..g_{rj} \rangle \bullet ri \leq rn \leq rj \bullet g_{rn} = g?_n$
$\qquad\qquad\qquad \textbf{else if } \forall g_n? : g_i? .. g_j? \bullet withinRange(g?_n , t_{start}, r, unit?) = \{FALSE\}$
$\qquad\qquad\qquad\qquad \textbf{then } groupByTimeUnit((g? \upharpoonright \langle g_r \rangle), t_{start}? \, (r+1), unit?) \bullet \langle g_r \rangle = \langle \rangle \lor \neq \langle \rangle \rangle$

- 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

```
┌─ OrderStatements ───────────────────────────────────
│ ΔFollowedRecommendations
│ SortByTimestamp
├─────────────────────────────────────────────────────
│ ordered'_L = orderByTimestamp(S_launched)
│ ordered'_R = orderByTimestamp(S_recommended)
│ t'_start = convert((head ordered'_L).timestamp)
└─────────────────────────────────────────────────────
```

- 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.

```
┌─ GroupByTime ───────────────────────────────────────
│ ΔFollowedRecommendations
│ GroupByTimeUnit
├─────────────────────────────────────────────────────
│ grouped'_launched = groupByTimeUnit(ordered'_L, t'_start, 0, unit?)
│ grouped'_recommended = groupByTimeUnit(ordered'_R, t'_start, 0, unit?)
└─────────────────────────────────────────────────────
```

- 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 $groupByTimeUnit$.

```
┌─ OnlyRecommendedLaunches ───────────────────────────
│ ΔFollowedRecommendations
├─────────────────────────────────────────────────────
│ onlyRecommended' = ⟨o : seq | let grouped'_launched == gl ⇒
│                    ⟨⟨gl_i⟩..⟨gl_j⟩⟩ ⇒ ⟨⟨gl_ii..gl_ij⟩..⟨gl_ji..gl_jj⟩⟩ •
│                    ∀⟨gl_n⟩ : ⟨gl_i⟩..⟨gl_j⟩ • ∃_1⟨o_n⟩ : ⟨o_i⟩..⟨o_j⟩ ⇒ ⟨⟨o_ii..o_ij⟩..⟨o_ji..o_jj⟩⟩ •
│                    ((∀o_in : o_ii..o_ij • o_in.context.statement ≠ ∅ ∧ o_in in gl_i)∧
│                    (∀o_jn : o_ji..o_jj • o_jn.context.statement ≠ ∅ ∧ o_jn in gl_j))∨
│                    ⟨o_n⟩ = ⟨⟩⟩
└─────────────────────────────────────────────────────
```

- 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'*$_{recommended}$.

$\_GetCounts_____$
$\Delta FollowedRecommendations$
$CountPerGroup$

$\rule{6cm}{0.4pt}$

$cPerGroup'_{launched} = \langle c : \mathbb{N} \,|\, \mathbf{let}\ grouped'_{launched} == gl \Rightarrow \langle\langle gl_i\rangle..\langle gl_j\rangle\rangle \bullet$
$\qquad\qquad\qquad \forall\langle gl_n\rangle : \langle gl_i\rangle..\langle gl_j\rangle \bullet \exists_1 c_n : \mathbb{N} \bullet$
$\qquad\qquad\qquad \mathbf{if}\ gl_n = \langle\rangle$
$\qquad\qquad\qquad\qquad \mathbf{then}\ c_n = 0$
$\qquad\qquad\qquad\qquad \mathbf{else}\ c_n = count(\langle gl_n\rangle)\rangle$
$cPerGroup'_{recommended} = \langle c : \mathbb{N} \,|\, \mathbf{let}\ grouped'_{recommended} == gr \Rightarrow \langle\langle gr_i\rangle..\langle gr_j\rangle\rangle \bullet$
$\qquad\qquad\qquad \forall\langle gr_n\rangle : \langle gr_i\rangle..\langle gr_j\rangle \bullet \exists_1 c_n : \mathbb{N} \bullet$
$\qquad\qquad\qquad \mathbf{if}\ gr_n = \langle\rangle$
$\qquad\qquad\qquad\qquad \mathbf{then}\ c_n = 0$
$\qquad\qquad\qquad\qquad \mathbf{else}\ c_n = count(\langle gr_n\rangle)\rangle$
$cPerGroup'_{followed} = \langle c : \mathbb{N} \,|\, \mathbf{let}\ onlyRecommended' == or \Rightarrow \langle\langle or_i\rangle..\langle or_j\rangle\rangle \bullet$
$\qquad\qquad\qquad \forall\langle or_n\rangle : \langle or_i\rangle..\langle or_j\rangle \bullet \exists_1 c_n : \mathbb{N} \bullet$
$\qquad\qquad\qquad \mathbf{if}\ or_n = \langle\rangle$
$\qquad\qquad\qquad\qquad \mathbf{then}\ c_n = 0$
$\qquad\qquad\qquad\qquad \mathbf{else}\ c_n = count(\langle or_n\rangle)\rangle$

- 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*.

$$
\begin{array}{l}
\rule{11cm}{0.4pt}\\
\quad CombineSequences\\
\quad \Delta FollowedRecommendations\\
\rule{4cm}{0.4pt}\\[2pt]
combined' = \langle c : (tr'_{start}, tr'_{end}, N'_{launched}, N'_{recommended}, N'_{followed}, P'_{followed}, P'_{dueto}) \mid\\
\qquad\quad \textbf{let}\ grouped'_{launched} == gl \Rightarrow \langle\langle gl_i\rangle..\langle gl_n\rangle..\langle gl_j\rangle\rangle\\
\qquad\qquad cPerGroup'_{launched} == cl \Rightarrow \langle cl_i..cl_n..cl_j\rangle\\
\qquad\qquad cPerGroup'_{recommended} == cr \Rightarrow \langle cr_i..cr_n..cr_j\rangle\\
\qquad\qquad cPerGroup'_{followed} == cf \Rightarrow \langle cf_i..cf_n..cf_j\rangle\\
\qquad\quad \bullet\ \forall \langle gl_n\rangle : \langle gl_i\rangle..\langle gl_j\rangle \bullet i \le n \le j \bullet\\
\qquad\quad \exists_1 c_n : (tr_{startn}, tr_{endn}, N_{launchedn}, N_{recommendedn}, N_{followedn}, P_{followedn}, P_{dueton}) \bullet\\
\qquad\quad tr_{startn} = (head\ gl_n).timestamp\\
\qquad\quad tr_{endn} = (last\ gl_n).timestamp\\
\qquad\quad N_{launchedn} = cl_n\\
\qquad\quad N_{recommendedn} = cr_n\\
\qquad\quad N_{followedn} = cf_n\\
\qquad\quad P_{followedn} = cf_n \div cr_n\\
\qquad\quad P_{dueton} = cf_n \div cl_n\rangle\\
\rule{11cm}{0.4pt}
\end{array}
$$

- 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:

  - $tr'_{start} \rightsquigarrow tr_{startn}$ which is equal to the timestamp for the first statement within gl_n

  - $tr'_{end} \rightsquigarrow tr_{endn}$ which is equal to the timestamp for the last statement within gl_n.

  - $N'_{launched} \rightsquigarrow N_{launchedn}$ which is equal to the current count of launched statements within the nth time grouping aka cl_n.

  - $N'_{recommended} \rightsquigarrow N_{recommendedn}$ which is equal to the current count of recommended statements within the nth time grouping aka cr_n.

  - $N'_{followed} \rightsquigarrow N_{followedn}$ which is equal to the current count of recommended statements within the nth time grouping aka cf_n.

  - $P'_{followed} \rightsquigarrow P_{followedn}$ which is equal to the result of dividing cf_n by cr_n.

  - $P'_{dueto} \rightsquigarrow P_{dueton}$ which is equal to the result of dividing cf_n by cl_n.

16.6.5 Sequence of Operations

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

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

16.6.6 Return

```
┌─ ReturnFollowedRecommendations ─────────────────────────
│ ΞFollowedRecommendations
│ ProcessFollowedRecommendations
│ combined! : seq
├──────────────
│ combined! = combined′
└─────────────────────────────────────────────────────────
```

- 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**() **$grouped'_{launched}$**;
$C_{recommended} \leftarrow$ **map count**() **$grouped'_{recommended}$**;
$C_{followed} \leftarrow$ **map count**() **$grouped'_{followed}$**;
$combined \leftarrow []$;
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
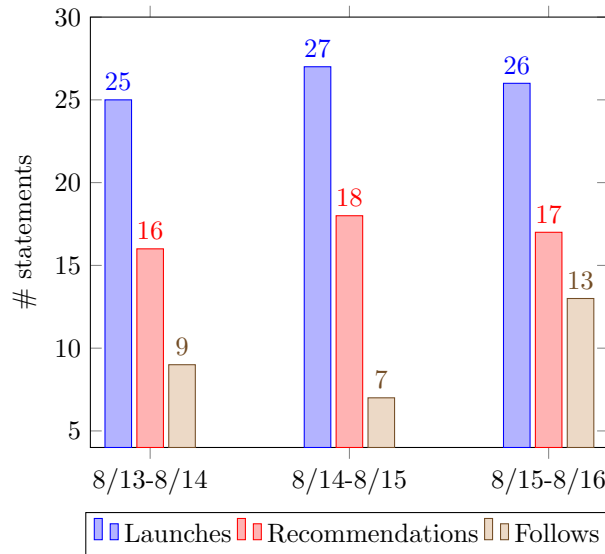
- The percentages described in section 5.9 are not displayed here.

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
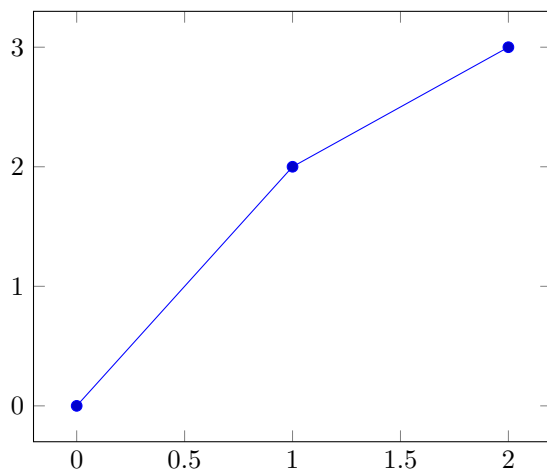
Figure 1: Line Chart
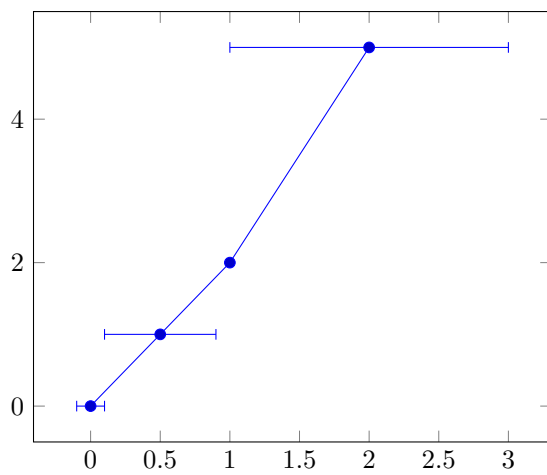
Figure 2: Line Chart with Error
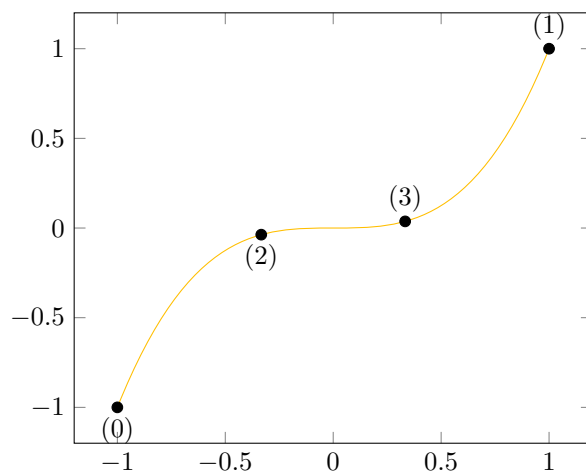
Figure 3: Spline Chart
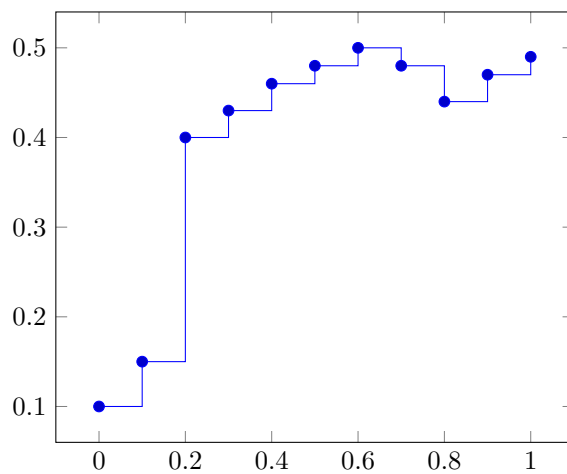
Figure 4: Quiver Chart

110

Grouping Charts
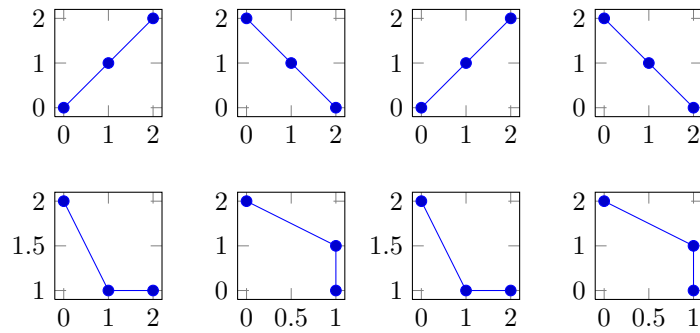
Figure 5: Grouped Line Charts
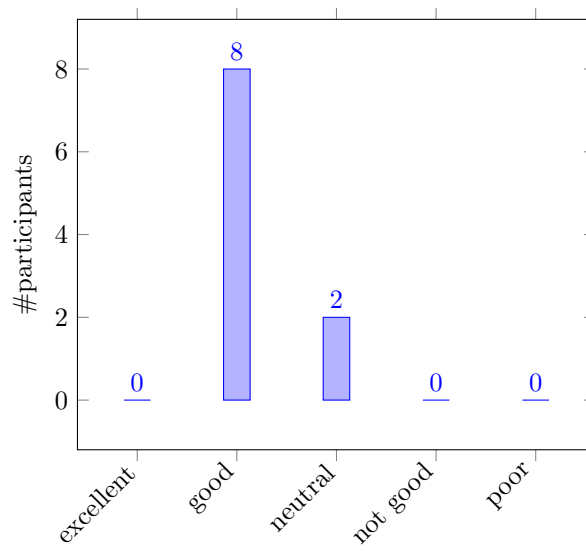
Figure 6: Histogram
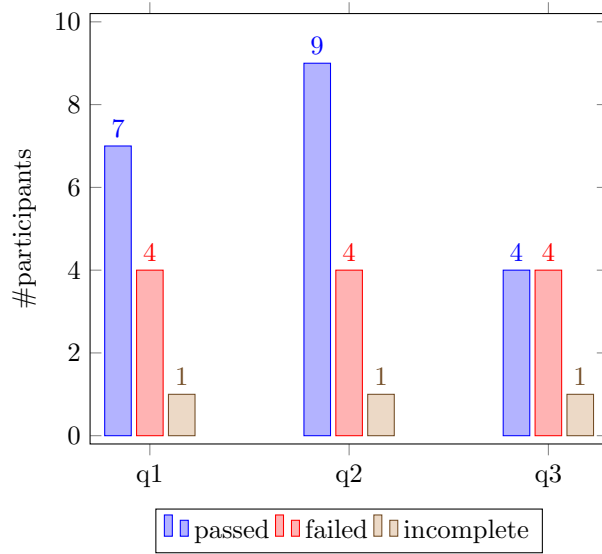
Figure 7: Bar Chart
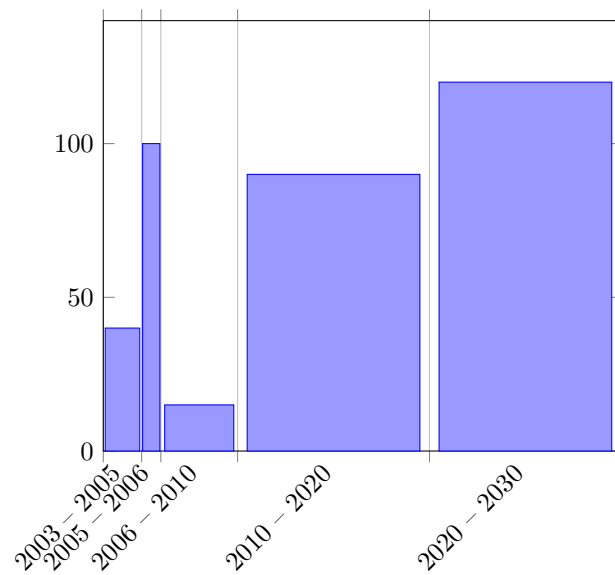
Figure 8: Bar Chart Grouped by Time Range

112
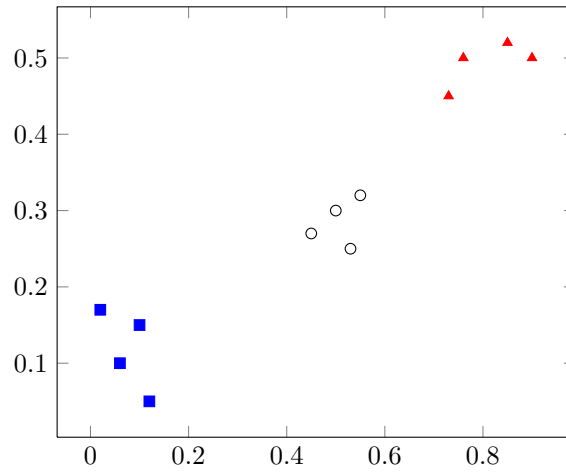
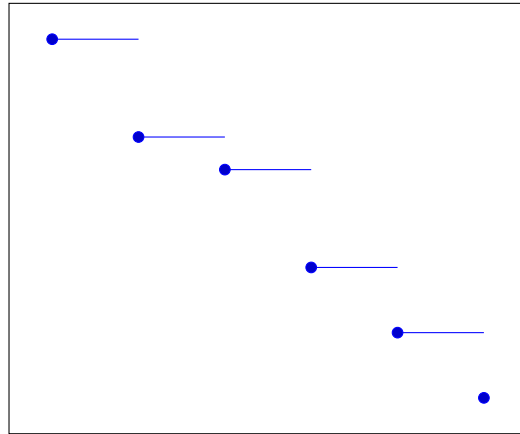Figure 9: Scatter Plot
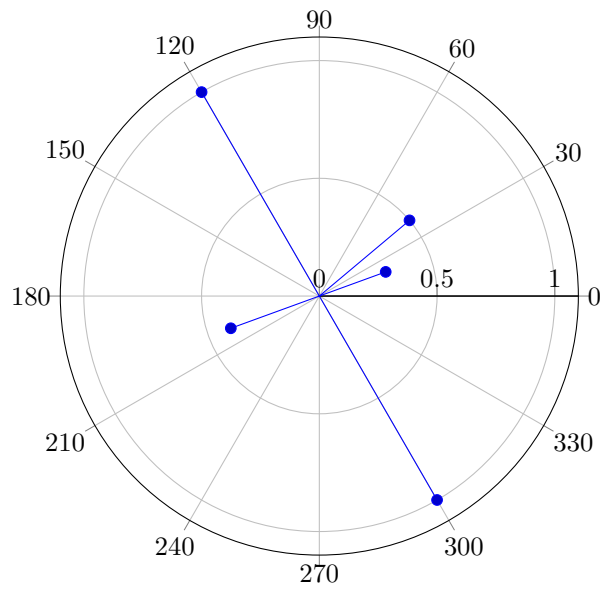
Figure 10: Polar Chart

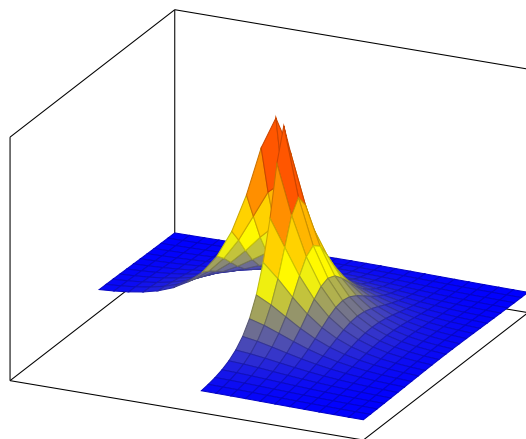# Specialized Charts

Figure 11: Gantt Chart
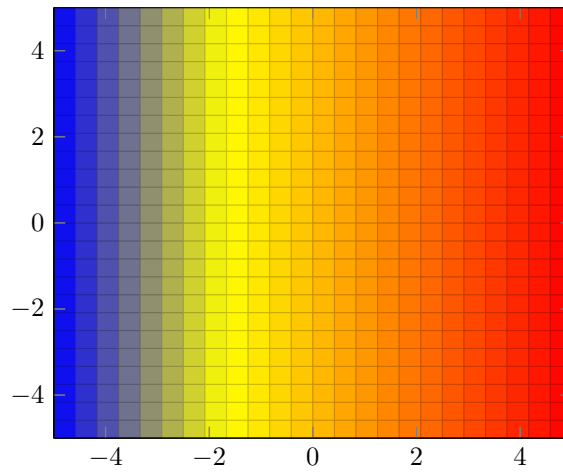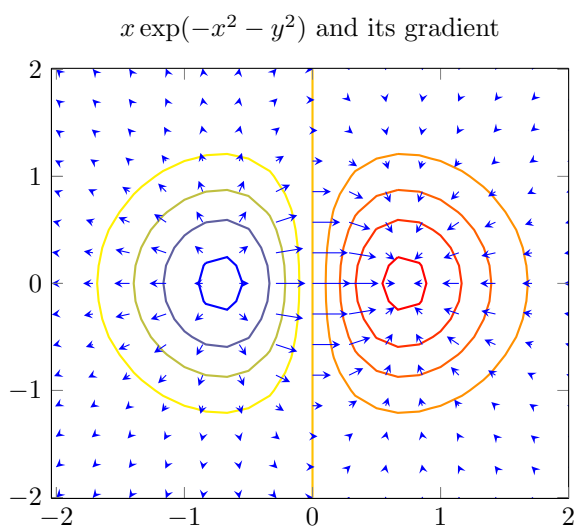

Figure 12: Heat Map

Figure 13: 3D Plot

$x \exp(-x^2 - y^2)$ and its gradient

Figure 14: Gradient Plot