IOSR Journal of Engineering (IOSRJEN) **www.iosrjen.org**
ISSN (e): 2250-3021, ISSN (p): 2278-8719
Vol. 10, Issue 2, February 2020, ||Series -I|| PP 24-32

Conversational Context is a Hierarchy Populated from Sequential Language

Peter Waksman, PhD

Received 20 January 2020; Accepted 10 February 2020

Abstract: This paper discusses the classic question of how language and reality are synchronized, through a discussion of conversational context. Conversational context is formalized as a ledger of entries in a hierarchical format that is created during reading, following principles for conserving context and for duplicating but not overwriting past records. These principles are embodied ina universal read() algorithmthat can convertserial text into formatted dataentries in such a ledger.Thus, whenever word definitions are made using this form of hierarchy, serial text will be readable by these general principles. Not only does this address the classic question, there is an extension of traditional ideas from differential geometry to problems of linguistics; so that the attachment of ledger entries to the topic,moving through the text, is analogous to the attachment ofcoordinate frames to a point moving along a curve.

I. INTRODUCTION

Around the turn of the 20th century, British philosophers and mathematicians such as Bertrand Russell were pre-occupied with the relation between language and reality. Russell and colleagues combined the foundational formalism of Frege’s set theory and the arithmetical approach of Bool’s logical operators to create a “propositional calculus” which gave modern form to old ideas of how**truth** is preserved and propagated through a collection of propositions [1]. But Russell could not figure out what made a proposition true or false *by itself*. One of his students , Ludwig Wittgenstein,postulated in his early writingthat reality and language were synchronized, so that reality can always be described exactly [2]. But he did not identifywhat kinds of mechanismwouldsupport thissynchronization. In his later work, after concludingthat single words did not have single meanings but meanings that changed with context, Wittgenstein wrestled with conversational context in general. Heobserved that a word’s meaning is determined by how the word is used [3] but, again, he made little progress and left behind what is described in various forewords(see [3]) as “one of the most influential works of philosophy of that century” – perhaps because of its combination of breadth of scope and lack of progress.

Trying to understand truth never led anywhere (Fregeadmitted as much [4]) but Wittgenstein’s late focus on conversational context is well justified. Today, when there is strong interest in computer language interfaces -with chatbots such as “Siri” and“Alexa” -the handling of conversational context is of practical significance and financial value(see Phillips [5]). For example, if a customer orders a product through a chatbot, it is important for the vendor to send the correct product. But the question of “truth” doesn’t enter into it. Thus, in a realm of computer language interfaces, what was said is important but not whether it is true or false.Instead of focusing on truth the current focus ought to be the nature of conversational context and how incoming language is stored and interpreted with linguistic structures.

Below, I give preliminary definitions for context structures. These were developed as part of an automated workflow for reading customer order details in dental manufacturing. They apply to a simple language of declarative sentences such as is found in product ordering.We will see that as long as word definitions are made within a‘real’ hierarchical framework, reality is automatically mapped by sequential text. *It is a matter of visiting one topic after the another and storing attributes for the topics as they pass bye, in the sequence.*The main contributions of this paper are two principles governing how an empty hierarchy is populated from a sequence of words:

The principle of **conserved context** says:

Context changes as littleas possible

The principle of record **splitting**[1]says:

What has been heard cannot be un-heard.

[1] Put differently: new memories should not overwrite old ones. This is called “splitting” because the new memory may duplicate default information from the old one but must be split-off as a copy of the original, to avoid data loss. There are some subtleties about writability of duplicated information detailed later.

These principles are sufficient for developing an automated reading capability from a hierarchical vocabulary. A complete Python implementation,theNarwhal Library,can be found at:

https://github.com/peterwaksman/Narwhal/narhwal/

in the "nwcontext.py" and "nwcontextrecord.py" modules. The Narwhal Library serves as a "proof of concept" for this paper's definitions and includesa chatbot development platform.

In what follows, I use the words "language" and "text" interchangeably. The conversion of voice to text is available as a service from companies like Microsoft, Amazon, and Google. So our focus is on detecting, recognizing, and recording linguistic meaning from incoming text – namely reading and recording what has been written.Subtler meanings present in spoken rather than written language are not discussed, nor are mechanisms for generating language.

Two Principles of Automated Reading

I will discuss a data type called ContextFrame for capturing conversational details and will show that a tree of instances of these frames, linked by the context/sub-context relationship,called a**Context Framework,**is a powerful programming construct.A**conversational context** is defined as *a top-down sequence of node instancesin aContext Framework.*

I propose a read() algorithm for converting text intoa conversational context that relies ontwo principles:The**conservation**principle says: current and previous details are processed within the most specificcontext,shared between the two. In other words:*changes in context are as conservative as possible.*This operation, similar to finding a least common multiple, is illustrated in Figure 1:

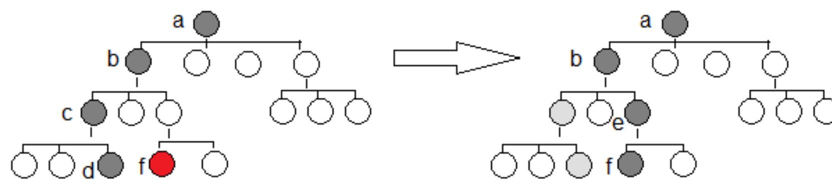

Figure 1Shows "before" and "after" pictures of a conversational context. Contexts and sub-contexts are represented, on the left, as a sequence ofgray dots {a,b,c,d}, from least to most detailed as we go from top to bottom of a tree of ContextFrames.During reading, when a new frame'f' (shown in red) is to be added, the most specificcommon parent 'b' is found, and the new conversational context is represented by the new sequence {a,b,e,f} on the right.

For example:

We read: "I am sitting in the classroom and turn to a fellow classmate to remark that the teacher is late". In this sentence the context includes a classroom so there is no need to specify which teacher is referred to. Assuming a different teacher would broaden the context un-necessarily.

A second principle is the **splitting**principlewhich says**:** *new instances of details cannot overwrite the record of old onesbut, instead, cause the record to split*. This can be illustrated as in Figure 2 but please note that there are some subtleties to be described later that are not illustrated:

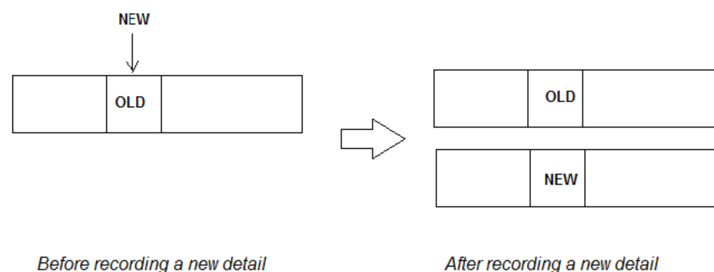

Figure 2Schematic view of the ledger "record" being split. A new detail is not allowed on top of an old one and causes the split.

For example:

We read: "Please go to the nursery and buy some potting soil. And get me some kind of cactus." In this sentence it is understood that the cactus and potting soil come from the same place. The listener must keep track of a 'nursery' and a 'buy' but the 'cactus' does not erase the 'potting soil' and the initial request must be split into two separate requests. Contrast this with "Please go to the nursery and buy potting soil, in a two pound bag." Here, the listener must keep track of a 'nursery' and a 'buy' and a 'potting soil'. But the amount of 'two pounds' fills into the existing request*without overwriting old information*. No record splitting is required.

The definition of **topic***as a ledger entry* and how the topic evolves according to the principles of conservation and splitting are discussed below in the description of a general read() algorithm.

The Geometric Analogy: The moving topic is like a moving point in space

As we read through text, word by word and phrase by phrase, the topic changes gradually and new ledger entries are created or modified. I call this the **moving topic**. Itis helpful to visualize the text as a curve in space, with the moving position of a point on the curve corresponding to the moving topic in the text. The higher order derivatives of the point's motion: the velocity, acceleration, torsion, etc. correspond with attributes that modify the moving topic and modify its modifiers.

In differential geometry, the higher order derivates of a point's motion are associated with standardized 'best fit' objects: the velocity is associated with a *best fit line* at the point; the acceleration is associated with a *best fit circle*; the torsion is associated with a *best fit coil*; etc. [see the Wikipedia definition of "osculating circle"]. In a similar way, the attributes of the moving topic can be derived from *best fit narratives* of increasingly complex structure, such as: 'thing', 'thing has attribute', 'thing has attribute changed by other thing', etc. Below I illustrate the use of the 'thing has attribute' narrative. [For a complete discussion of narrative structures see *The Elements of Narrative* [6]. For a summary of a "goodness of fit" metric for fitting narratives to text, see my *Revised GOF formula* [7].] TheNarwhal Library implements this metricfor general narrative structures.]

In the differential geometry, the moving point and it's first three motion derivatives are grouped as a single coordinate system called a **Frenet frame** for a spatial curve, or a **moving frame** for a more general geometric object. The moving point is the origin of the coordinate system and the motion derivatives (which are vectors) define the axes of the coordinate system. Because the moving frame is a adapted to the local geometry near the point, it is useful for describing the object's invariant shape properties near the point.

Continuing the analogy between points and topics: the word "ContextFrame" is chosen deliberately to emphasize the similarity to moving frames. In this way text becomes a geometric object, with read() being understood as a process for tracing a moving topic and leaving behind a ledger of what has been said.The division of labor between a data format, reading principles, and narrative structures, constitute a geometric approach to studying language.

Frame Instance versus Frame Definition

A frame definition is not the same thing as a frame instance. In geometry, when a frame is attached to a moving point, quantities that are given abstractly in the frame's definition need to be evaluated with particular numeric values in the frame's instance.In the same way, when we attach a ContextFrame to incoming language with a new sequence of ContextFrames, the abstractly defined quantities of the frames will get evaluatedusing the input and following specified procedures. But rather than using numbers as values for the abstractly defined quantities, we will use constants, enumerated values, and word trees defined by the program[2].

I will discuss two data types: the ContextFrame defines the frame in the abstract and the ContextRecord defines a set of constants needed to store a frame instance.Then a universal read() algorithm is described as a merge/split/append operation for writing into an expandable tree of ContextRecords, called a ledger.

ContextFrame

The **ContextFrame** is a structure with fields for parents and children, *so the frame can be a node of a doubly-linked tree*;it also hascontent-specific fieldsthat defineproperties or**modifiers**; andit hasprocedures for setting modifier valuesfrom input text. It also has a field related tokeywords for detecting the frame in incoming text.In Python, this can be written as:

[2]Speaking of trees, note for example that the real numbers are defined using trees. The binary representation of a real number locates it by following branches ('0' or '1') in an infinitely descending binary tree.

```
classContextFrame:
def __init__(self,id,env,modifiers,parts,relations,var=NULLVAR):
self.ID = id# should be a unique string
self.ENV = env# None or another ID
self.PARTS = parts# list of other IDs

self.MODS = modifiers# list of enum vals
self.RELS = relations# list of procedures, one per modifier

self.var = var# keywords that detect this frame
```

The 'ID', 'ENV', and 'PARTS' fields define *constant*parent-child relations, so a collection of ContextFrame*definitions* can be organized as a tree.One ContextFrame nodeis illustrated in Figure 3:

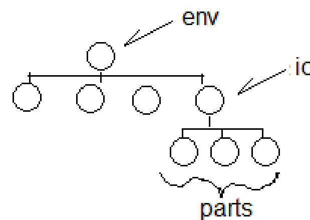

Figure 3 A tree nodeid connected*<u>*from*</u> *an environment context (env)and* <u>*to*</u> *a collection of sub-contexts (parts).

The 'var' field corresponds to a list of keywords. When one is detected in the text, it signals the presence of thisContextFrame.

The 'MODS' is a list of labels of attributes, or modifiers, of this context frame; and the 'RELS' is a list of procedures indexed by the same labels, whose role is to extract values from text and assign those values to the modifiers. When a frame is detected in the text, the corresponding RELS procedures will be invoked to attach the frame there. In this way the abstract frame definition becomes a specific frame instance. When reading text, the sequence of operations is driven by the sequence of id's and the extraction of properties is driven by the procedures, as indexed by the MODS. Since all the other fields of a ContextFrame are constant, it is only the presence of theid and the variable values of the MODS which define the particulars of an attached frame instance and are worth recording.

<u>Initializing an empty Context Framework</u>
Typically a tree of ContextFramesmay be initialized with an array of Python dictionary entries of the form:

id : [env ,parts , mods , rels , var]

Where **id** is a unique string; **env** is 'None' or the id string of another entry; **mods** is a list of labels – one per modifying property of the ContextFrame; **parts** is a list, possibly empty, of id strings of other entries; and **var** is a label associated to a list of keywords that detect this ContextFrame in the incoming text.
Cycles are possible with these definitions, where a sequence of children of parents leads back to the starting parent. These should be avoided. It is also recommended that the ENV of a 'part' should always be the current 'id'. I make these assumptions below.

<u>Example: a teapot</u>

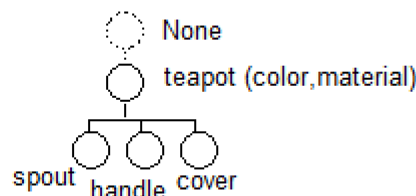

Figure 4Conversational Context for a Teapot

Consider the Context Framework for a simplified teapot, defined by oneContextFrame with id 'teapot': It has two MODS: color and material; and it has three PARTS: a 'spout', a 'handle', and a 'cover'. Although these are

physical parts of the teapot, the intention is that they be considered sub-conversations within a broader ‘teapot’ conversation.This is illustrated in Figure 4.

If we want to talk about the color of the teapot we can remain in the ‘teapot’ frame, but to discuss the shape of the spout or the spout’s hole we need to consider a context that includes the ‘spout’ sub-frame as well as the ‘teapot’ frame.Also, the definition requires the teapot have an ENV. If theENV is set to ‘None’ then this is an abstract teapot floating in space.Amore particular teapot must have a more particular, non-null,ENV.

There is an ambiguity to this formalism when a ‘part’ has just one property. In that case, it is easier to speak of the part’s propertyas a property of the ‘self’. For example, suppose there is a picture of a rose on the teapot. Is this a ‘part’ or a ‘mod’? If discussion of the shape and color of the rose is desired then it should be treated as a ‘part’ because it is a sub-conversation. But practically speaking, if it only has one property – presence or absence – then that might as well be considered a property of the ‘teapot’, withoutneeding to define the rose pattern as a ‘part’.It is unclear if “single part with property” and “single property” are interchangeable. They are implemented differently here.

Pattern Matching of Local Text – the **var**and **RELS**of a ContextFrame

We distinguish between language that detects the frame, versus language that sets values of the modifiers. The ‘var’ field indicates a list of keywords (synonyms or other indicators) of the frame ‘id’ and these are matched to text. Their order in the text defines the sequence of operations. But each procedure in RELS will use its own word trees and have its own opportunity to scan and match the text.

As we proceed through the text, the matching of id keywords is a“0th order” matchingthat describes how the topic and sub-topics change. This is minimal information, since we are simply moving through the topics, detecting them without saying anything about them.To set values for the MODS of one of thesedetected ContextFrames, we use the RELSproceduresand scanwords in the vicinity of the detected var keywords. For example, the expressions: “the teapot is white” or “there is a white teapot” both set the teapot’s color modifier. These are examples of “1st order”narrative pattern matching,using the“thing has attribute” narrative structure. Other narrative structures and rules for forming composite “higher order” narratives can be found in *The Elements of Narrative* (Waksman [10]). Partial matching of narrative patterns is typical, so a “goodness of fit” metric is implemented provisionallyin the *Narwhal* GitHub repository. The RELS procedures can use more than one such narrative pattern and select the best fitting one to extract values, while applying internal acceptance thresholds. Other methods for extracting information locally may be worthwhile, such as natural language understanding based on syntax and grammar.

The higher order derivates of a moving point in geometry are sometimes replaced in the mathematical theory, with “best fit” ideal object having higher “orders of contact” (see the definition of “osculating curve” in Wikipedia). For example the moving frames for a curve in 3D has thetangent line corresponding to the best fit line, the acceleration vector corresponds to a best fit osculating circle, and the torsion vector corresponding to the best fit helix.By analogy here, the RELS procedures will find the best fit narrative pattern, of varying complexity, to extract values from the text.

ContextRecord and contextsplitting

In *Zen and the Art of Motorcycle Maintenance*[8], Robert Pirsig writes about how a new idea becomes divided from an old idea when properties of the new idea force a distinction to be made. A similar principle is involved when we wish to save an instance of a ContextFrame on top of an already saved instance, within a hierarchical sequence of ContextFrame instances. Assuming we should notoverwrite old values with new ones, we must create a duplicateor “split” record where we *can* write the new values (see Figure 2). However, there isa subtlety about duplicating childsub-records of a parentwhen it is split, namely: *the modifier values within duplicated childrenare considered provisional and can beoverwritten later without splitting.*Thusparts of the frame will be attached provisionally to text and modified or finalized later, as the read() continues.

Definition of ContextRecord and ContextLedger

By the conventionsof ContextFrames, the parent-child relations, var keywords, and RELS proceduresare*constants* fixed during initialization. Only the frame id and the values assigned to the MODS are *variable* andneeded for frame attachment. So it is this id and these values that will be recorded in a ledger.This is called a **ContextRecord**. For example, a frame with three MODS will have three variables that can be assigned values. We call these values**details**.

To store a record of details we need another data type, with an id and an array of values indexed by the MODS labels*:*

```
classContextRecord():
def __init__(self, id = '', contextMods = []):
self.id = id# id of corresponding ContextFrame

# details are indexed by mods of the ContextFrame
self.details = { }
for m incontextMods:
self.details[ m ] = ['' , EMPTYDETAIL]

self.children = []
```

The 'id', 'children', and 'contextMods' fields mirror the correspondingfields of a ContextFrame id, PARTS, and MODS.

Here '**EMPTYDETAIL'** means the detail has no assigned value. We also use '**SOFTDETAIL'**to mean there is a value that arose via duplication; and '**HARDDETAIL'**to mean the value is derived directly from incoming text, or has been finalized in some other way.When something has been said/heard it is 'hard'. If it is only assumed then it is 'soft. The splitting principle is embodied in the behavior of these status flags. Thus a HARDDETAIL cannot be overwritten and causes splitting, while a SOFTDETAIL can be "hardened" later orit can be overwritten in the short term. An EMPTYDETAIL is always available for writing. Another way to think about hard and soft details is that soft details are still writeable but hard details are "read-only".

Details are displayed and saved in the form [valuestring, detailStatus] and listed, one per mod, after the id, in a tuple of the form:

(id, [detail1, status1], [detail2, status2], …)

To display and save ContextRecords that are children of other context records, use tabbed indenting, so they stack up below the parent like this:

```
( id,  [value1, status1], [value2, status2], ….)
        (child_idA, [value A1, statusA1], …)
        (child_idA, [value AA1, statusAA1]…)
        (child_idB, [valueB1, statusB1], [valueB2, statusB2],…)
```

The tree of ContextRecords exists in parallel with the tree of ContextFrames. However where a ContextFrame node is singular, the nodein the corresponding position of the record treemay contain multiple records formultiple occurrences of that ContextFrame within the incoming text. Thus aContextRecord tree can be appended to. We call this expandable tree a **context ledger**. *The expandable ledger is a record of what has been said in a particular conversation.*A body of text with a context ledger is equivalent to a geometric object with attached moving frames, like a curve in 3D. What the topic is "about"as defined by the context framework is equivalent to the overall 3D space;what was actually "said" is defined by a ledger and is equivalent toa particular curvein that space along with the frames that have been attached at each point.

Example: ordering dental abutments and crowns

Assume a context tree as illustrated in Figure 5. [A full implementation of a dental context tree is given in the Narwhal\mouthContext\ sub-project.] There is a 'site' of a dental restoration, to be modified by a tooth number, and having PARTS for 'abutment' and 'crown'. In turn these have modifiers for 'type' and 'material'.

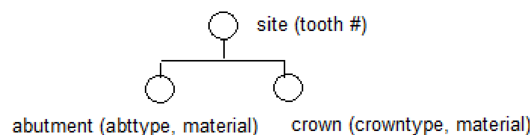

Figure 5 Part of a dental context framework, shows a ContextFrame for the tooth location, with two sub-contexts for possible restorations at the location: an abutment and a crown

Some typical expressions might be:

"I need titanium abutments on tooth #3, 5, and 7"

"I need a titanium abutment and a temp crown on #3"

"I need a titanium abutment on #3 and on #5 I need a zirconia abutment."

The reader should appreciate how these all give similar information but with the different ContextFrames occurring in different orderings. The last example is like the first but with an assumption that

gets overwritten at the end. Here is how these may be recorded in a ledger, following steps that are formalized later in the read() algorithm description. *Please note*: original sentences are displayed in red lettering. Green lettering indicates new or changed values. EMPTYDETAIL, SOFTDETAIL, and HARDDETAIL will be represented, respectively, with 'E', 'S', and 'H'.

Consider the expression:

"I need titanium abutments on tooth #3, 5, and 7"

Step 1: "I need titanium abutments…" generates a record with values for 'abutment'. An empty 'site' record is needed, to create a ledger entry of the form:
('site' , ['', E])
    ('abutment', ['', E] , ['titanium', H])

The 'tooth #' and 'abttype' strings are left empty with EMPTYDETAIL status, while the titanium abutment material is recorded as a HARDDETAIL.

Step 2 : "…on tooth #3" generates a record of the form ('site', ['#3', H]) which can be inserted into the previous in the EMPTYDETAIL slot of the 'site' detail:
('site' , ['#3', H])
    (abutment', ['', E] , ['titanium', H])

Step 3: "…, 5…" generates another detail of the form ['#5', H] and, since the 'site' record already has a HARDDETAIL, a splitting is required and the record becomes:
('site' , ['#3', H])
    ('abutment', ['', E] , ['titanium', H])
('site' , ['#5', H])
    ('abutment', ['', E] , ['titanium', **S]**)

Note that when we split, the first abutment entry is duplicated in the second one but the detail status changes from HARDDETAIL to SOFTDETAIL.

Step 4: "…7" generates another detail of the form ['#7', H] which cannot overwrite the previous 'site' detail and another splitting occurs with another copy of the 'abutment' sub-context information. So we end up with
('site' , ['#3', H])
    ('abutment', ['', E] , ['titanium', H])
('site' , ['#5', H])
    ('abutment', ['', E] , ['titanium', S])
('site' , ['#7', H])
    ('abutment', ['', E] , ['titanium', S])

(Note: *one cannot continue to discuss tooth #5 after mentioning tooth #7.*)

Consider the expression:

"I need a titanium abutment and a temp crown on #3"

Step 1: As before, we create a ledger to record "I need a titanium abutment…."
('site' , ['',E])
    ('abutment', ['', E] , ['titanium', H])

Step 2:"…and a temp crown…". Here, there is nothing to overwrite for the 'site' entry, so there is no splitting, and ('crown', ['temp', HARDDETAIL], ['', EMPTYDETAIL]) is appended:

('site' , ['', E])
    ('abutment', ['', E] , ['titanium', H])
('crown', ['temp', H], ['', E])

Step 3: "…on #3" fills in the available slot in 'site' so we get

('site' , ['#3', H])

('abutment', ['', E] , ['titanium', H])
('crown', ['temp', H], ['', E])

Lastly, consider the expression (which is a bit artificial):
"I need a titanium abutment on #3 and on #5 I need a zirconia abutment."

Steps 1-3 are follow the same logic as the first example above, giving us a ledger that looks like this:

('site' , ['#3', H])
('abutment', ['', E] , ['titanium', H])
('site' , ['#5', H])
('abutment', ['', E] , ['titanium', S])

Step 4: "…I need a zirconia abutment". We go to append this and, because the previous abutment detail has SOFTDETAIL status, we can overwrite it:

('site' , ['#3', H])
('abutment', ['', E] , ['titanium', H])
('site' , ['#5', H])
('abutment', ['', E] , ['zirconia', H])

These variations are all covered by the general purpose algorithm. When the a higher level context changes, or when the reading is done, we can go back and 'harden' any soft values that remain in the ledger, its children, and their children, etc.In any case, later when any one of its parents changes, a soft value will remain in the ledger and no longer be reachable through read operations.

The read() algorithm – merge, split, or append
The following assumes a fully defined and initialized tree of ContextFrames, without cycles.

The read(text) algorithm consist of an outer loop, looping through the incoming text word by word,and matching frame id's; and an inner loop, looping through the mod values of any id that is detected, for setting MODS values. Then, for each such id and each such mod of that id we call

- record = fillRecord(id, text, mod)
- writeRecord(id, record)

The **fillRecord**( ) invokes whatever procedure is stored in the frame's RELS[mod], using whatever pattern matching you like. This may be language and topic specific and will not be discussed further here.The function creates and returns a new record which is to be inserted in the current ledger.

The **writeRecord**() method expands a ledger that initially contains one blank record for the top level id, with no children and with an **activeID** list containing only the top level id. This activeID list will be updated each time writeRecord() is called. The activeID list is a critical algorithm detail representing the current conversational context. As follows:

writeRecord(id, record):
if id is in the activeID
try to insert ("**merge**") the current record for that id.
If OK, return
else (we could not write over a HARDDETAIL),
split the current record, making SOFTDETAIL copies of all thechildrenof the current record.
else (the id is not in the activeID)
find the nearest common ancestor to this id that is in the activeID, and
add a new sequence of empty ContextRecord children, starting just below that common ancestor and following the intermediary nodes to the node just above this id. **Append** the record at the end of this sequence and modify the activeID list to reflect the new chain of id's from the top ledger id,down to the commonancestorid, and then on down to the current id.

To see a detailed implementation of this “merge/split/append” algorithm, please refer to context related modules in the Narwhal GitHub repository.

II. SUMMARY AND CONCLUSIONS

It is surprising that information in the serial form of text is so well suited to setting values inside of hierarchical data structures. The trick is to organize the datahierarchy (equivalently, the dictionary) by topics and sub-topics and to follow the principles of conserved context and record splitting during reading.

I have described a universal read() algorithm,using these two principles, for processing text and producing a ledger of records, which captureshierarchical instances of the contexts where something was said. The read() algorithm relies on a carefully constructed tree of ContextFrames, and procedures that write the text into such structures, one modifier at a time. *It may not be easy* to create a “carefully constructed tree” but it is possible. In such domains as hotel reviews, part ordering, FAQ handling,or other *narrow topic areas*,the possibility of creatingcomprehensive treesis within reach. So also isthe writing of programs that can use that data to adjust as the conversation progresses and, in a some sense, understand what has been said. So it would certainly be worthwhile to find ways to create Context Frameworks automatically by scanning dictionaries and thesauruses, and perhaps using machine learning with samples of the topic.AContext Framework is an excellent data base format, because it is easy to understand and allows natural language retrieval.

The question of how value (GOOD or BAD) is applied to entries in a ContextFramework is very worth pursuing as an elaboration of “value transfer”, as described in Truism 4 (see[6]). This truism refers to transfer of value from one modifier to another within the context’s attributes. But value is also transferred between onesub-context part and another of a ContextFrame. The way that value is subjective, volatile, and transferrable within the Framework is well worth understanding. In particular because it gives insight into how communication fails when communicators are using different versions of the same framework or when they have different value assignment in a more or less common framework.

It would also be worth studying more complex features of language, for example verbs which affect both the subject and object, or the kinds of usage that puzzled Wittgenstein (see his discussion of ‘reporting’ versus ‘ordering’ [3]). I believe suchcomplexities can be handled with a richer set of attribute types within the ContextFrameand with geometric objects that are higher dimensional.

Finally, we note that word trees underlying conversational context are essentially numeric systems and, with the geometric analogy in mind, one speculates that there must be frame bundles and symmetry groups for linguistics in the same way as there are for physics. What, we may ask, are the familiar geometric properties of text? Can one assign a dimension? Are different kinds of text geometrically different? Are there standard objects? How do languages compare?

In this author’s opinion, the notion of conversational context is largely the same as the notion of context in general, independent of language. The merge/split/append algorithm resolves particulars and generalities a bit at a time and in an essentially sequential order; and the way this happens may hold lessons for how categories and particulars are processed in non-linguistic contexts. In particular, when we process sequential experiences, principles like the conservation of context and the splitting of previous records (the non-destructive re-use of memory)must be operating.

REFERENCES

[1]. Russell, Bertrand. The Principles of Mathematics.W.W. Norton Inc. New York. (First published 1903.) ISBN 0 393 00249 7. Chapter II. Also Appendix A and p. 502 etc.

[2]. Wittgenstein, Ludwig. Tractatus Logico-Philosophicus.

[3]. Wittgenstein, Ludwig. Philosophical Investigations.Translated by Anscombe, Hacker, and Schulte. Wiley Blackwell, 4th edition, 2009 p. 31.

[4]. Frege, G., 1918. "The Thought", in his Logical Investigations, Oxford: Blackwell, 1977 – quoted in the Wikipedia definition of “redundancy theory of truth”.

[5]. Phillips, Casey. *Why Conversational Context is the Most Powerful Tool you can give your Chatbot.* Chatbots Life website. Retrieved from https://chatbotslife.com/why-conversational-context-is-the-most-powerful-tool-you-can-give-your-chatbot-19d9d8656697

[6]. Waksman, Peter. *The Elements of Narrative – A formalism for analyzing story structure.* IOSR Journal of Engineering (IOSRJEN). Vol. 06, Issue 10 (Oct. 2016) pp 52-63.

[7]. Waksman, Peter*Revised GOF* in Sphinxmoth Blog. December 18, 2016. Retrieved from https://sphinxmoth.blogspot.com/2016/12/revised-gof-formula.html

[8]. Pirsig, Robert. Zen and the Art of Motorcycle Maintenance. William Morrow and Company (1974).