ANSWER: A Principled Approach to Scientific Knowledge Representation, Extraction, Curation, and Utilization

DARPA ASKE TA1 ANSWER M3 Report[1]

February 1, 2019

Andrew Crapo, Nurali Virani, Varish Mulwad
GE Research

1 Introduction

The work performed by GE Research on DARPA ASKE TA1 during December 2018 and January 2019 has resulted in a much more detailed system design, architecture, and methodology that will enable the functional capabilities of the ANSWER (Augmented Bayesian Networks Integrating Semantics With Extraction and Readability) system. These capabilities include the following.

- Extract RDF triples conforming to a base OWL ontology from code, add curated content to the knowledge graph
- Extract RDF triples conforming to a base OWL ontology from text, including code comments, code documentation, and publications, and add curated content to the knowledge graph
- Generate computable models capturing scientific knowledge in a K-CHAIN computational graph with rich semantics of each model captured in the knowledge graph. Models can be physics-based or data-based and can be aggregated into composite models.
- Do knowledge curation guided by 1) the systems awareness of gaps and weaknesses in the knowledge, including an inability to answer questions, and 2) the human user's directions to the system and answers to questions from the system (human in-the-loop).

The system functionality is illustrated in Figure 1

[1] This work is supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00111990006.

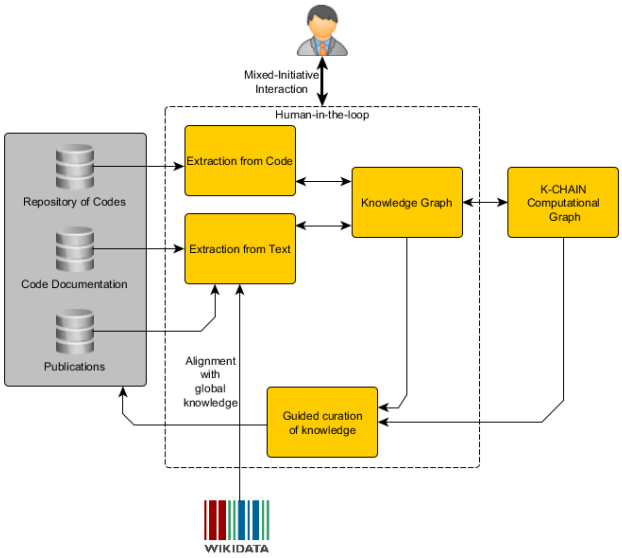


Figure 1: The overall functional architecture of ANSWER system. Knowledge curation uses a variety of sources, such as code, documentation, and relevant publications. The extraction is carried out in the context of current knowledge graph for code or text independently or in a coordinated effort with human-in-the-loop. The ANSWER system allows for mixed initiative interaction with users and creates computational instantiations of quantitative aspects of knowledge using computational graphs in K-CHAIN framework

Several principles have guided our system design. First and foremost, the system must be designed so that the human user is a participant in knowledge extraction and curation and is not limited to the role of querying the knowledge base for answers. We have designed for, and believe it is essential to have, a mixed initiative capability that allows the user to both ask questions and provide unsolicited information, but which also allows the system to ask the user questions and ingest responses. Designing to this principle has far-reaching effects on the system architecture as will be detailed in the following sections.

Another principle guiding our system architecture is locality. It is important that the extraction of knowledge from code be informed by extraction from text, with the most localized information being given the most attention. Thus, a comment in a specific line or section of code is potentially more meaningful and useful than general documentation of the code. Code documentation, in turn, should be considered before general publications. And where there is associated code and text, the text extraction may in turn be aided by the model extracted from the code.

A third principle guiding our work is that knowledge from various sources must be aligned. It is important to know when a concept referenced in multiple sources is the same concept and when it is not. This comes into play on a continuing basis as new knowledge is extracted and reconciled with existing knowledge. The alignment capabilities of OWL, i.e., owl:sameAs, owl:differentFrom, are useful in following this principle.

A fourth principle guiding our design is that provenance must be captured and credibility assessed. Knowledge curation requires having a knowledge of the origin of each piece of knowledge. When the same knowledge is found in multiple sources its credibility is increased. When sources are found which are contradictory, credibility is decreased. The knowledge-based

system must always be able to tell the user not only what is known or inferred, but also from whence it is known and how credible it is believed to be.

While the effect on system architecture might not be great, a fifth principle that is fundamental to the program's successful capture and use of scientific knowledge has to do with the context of each piece of extracted knowledge. It is almost always the case that knowledge captured in both code and text makes a certain number of assumptions implicitly. Humans knowledgeable in a domain intuitively are aware of these assumptions and usually conform to them unconsciously. Using a forward reference to the Speed of Sound use case, the Mach number computation must use the speed of sound in the air through which the aircraft is traveling, not just the speed of sound in air in any arbitrary location or condition. This topic will be addressed in greater detail in the Supporting Ontology section.

The organization of this report is as follows: Section 2 introduces the illustrative use case from the NASA Hypersonics Index that we will be referring to throughout the report and that has greatly informed the work done since the M1 report. Section 3 shows the details of the system architecture and illustrates various blocks of the architecture with examples. Section 4 discusses the supporting domain ontology which can capture scientific knowledge. Finally, section 5 summarizes outcomes and outlines next steps towards M5 deliverable.

2 Speed of Sound as Illustrative Use Case

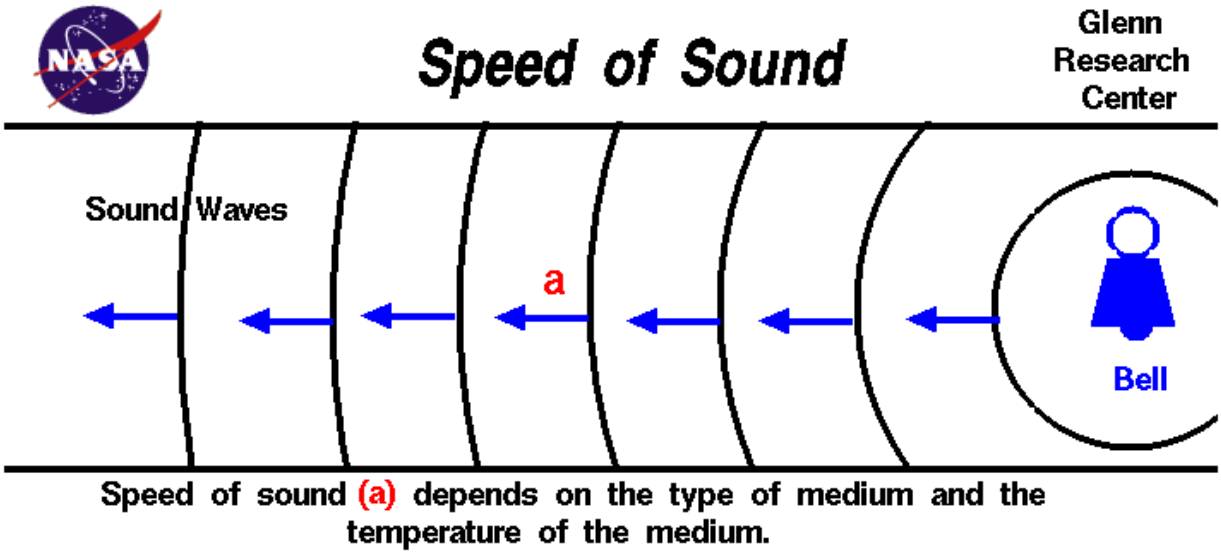


Figure 2: From Speed of Sound Web page, https://www.grc.nasa.gov/www/BGH/sound.html

To allow a more concrete illustration of the system design and the functionality of each of the system's components, we select the NASA Glenn Research Center's "Speed of Sound" Web page.[2] This page discusses the manner of travel of sound through a gas such as air and the properties of the gas upon which it depends. It further differentiates between the speed of sound in a calorically perfect gas and a calorically imperfect gas, with different but related equations for

[2] https://www.grc.nasa.gov/www/BGH/sound.html.

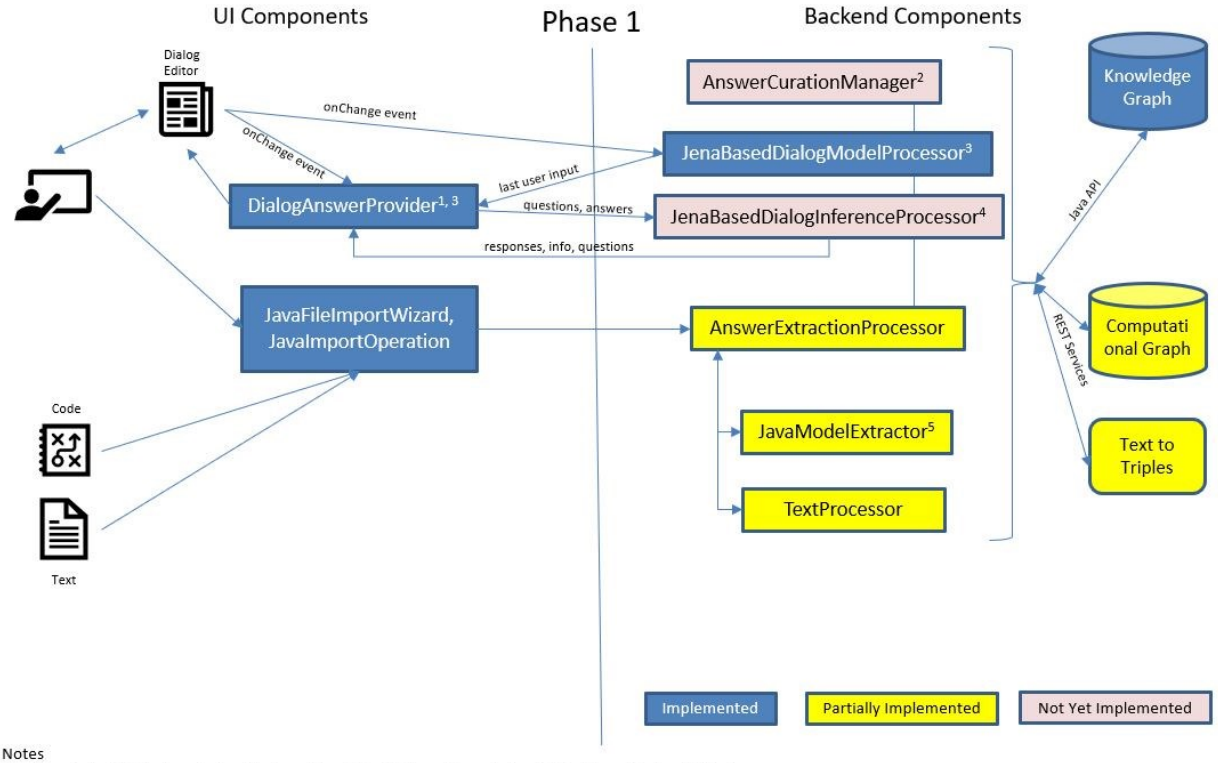
each. A Java applet that computes the speed of sound in air and the Mach number of an aircraft at a given speed and altitude, or the speed of the aircraft given its Mach number and altitude, can be downloaded from the page. The page has enough complexity to provide a variety of examples.

Since this page is related to other pages on the NASA Hypersonics Web site, it can be expanded upon to include related science as our capability increases.

3 ANSWER Architecture

3.1 Overall System Architecture

The overall system architecture is shown in Figure 3.



Notes

1. Uses AutoEditStrategy to be able to put text into Dialog editor window to implement mixed initiative
2. Will use other backend components, orchestrate achievement of all functions
3. JenaBasedDialogModelProcessor extends JenaBasedSadlModelProcessor. The DialogAnswerProvide is called in the UI before the model processor in the backend, but when the UI retrieves the "last cmd" the backend is invoked for validation. This is essential to correct behavior—the backend must process the AST first.
4. Extends JenaBasedSadlInferenceProcessor, adding any special processing unique to Dialog grammar and to K-Chain and DBN invocation
5. User-driven way of initiating code extraction, along with associated text extraction

Figure 3: Overall ANSWER System Architecture

To leverage existing capability, the ANSWER system is built as a set of Eclipse[3] plugins. These plugins and components are broken down into UI components on the left and backend components on the right of Figure 3. The backend components are user-interface independent and can be used with, for example, a Web browser-based user-interface in the future.

[3] Eclipse is an open source software collaboration and innovation environment, see https://www.eclipse.org/.

We have found controlled-English to be an effective way to allow [English-speaking] humans to build and to view and understand semantic models.[4] In particular, the Semantic Application Design Language (SADL) implements a controlled-English grammar with expressivity of OWL 1 plus qualified cardinality constraints from OWL 2, as well as rules. It also supports queries, along with tests, explanations, and other model maintenance aids.[5] The SADL grammar and integrated development environment (IDE) is implemented using Xtext.[6] We have created, and will continue to extend, the Dialog language as an extension of the SADL language. The Dialog grammar enables the user to create a Dialog conversation in a Dialog Editor window and specify a knowledge graph to serve as the domain of discourse. Once created, the Dialog's OWL model, which extends the selected domain model with any new knowledge captured in the conversation, is usable for query and inference and, if saved by the user, is saved both as a file in the Eclipse project with the ".dialog" extension and as an OWL file that contains the domain reference and the new knowledge. The user and Dialog Editor window are shown in the upper left corner of Figure 3. New or modified content in the Dialog Editor is passed both to the UI component DialogAnswerProvider and the backend component JenaBasedDialogModelProcessor. Details of user-system interactions are covered in the next section.

While in a fully-functional system, code and associated text, publications, etc., might be found and extraction initiated by the curation manager through use of a Web bot or some other search mechanism over some set of potential sources, in the current proof-of-concept architecture the user initiates extraction from code and text through the UI component JavaFileImportWizard and the associated JavaImportOperation. The import operation is carried out by the backend component AnswerExtractionProcessor. This component makes use of the TextProcessor and JavaModelExtractor to extract from text and code. Either extractor can use tentative OWL models extracted by the other and will almost certainly use the Knowledge Graph and Text to Triples services. The knowledge graph is accessed via a Java API utilizing Apache Jena[7], while Text to Triples is accessed via a REST service interface[8]. Note that during extraction, the user could be consulted to resolve issues, answer questions, or interact in any way desired by the extractor and supported by the Dialog grammar.

To complete the system overview, the K-CHAIN computational graph (CG) is extended whenever a model containing useful scientific knowledge is identified and validated for inclusion. To be added to the CG, the scientific model needs to either have an equation or have data from which a data-driven model may be trained. Once created, a model may be extended by adding an equation, if it doesn't already have one, and/or data. The CG uses TensorFlow in Python to

[4] See "Toward a Unified English-like Representation of Semantic Models, Data, and Graph Patterns for Subject Matter Experts", A Crapo and A. Moitra, International Journal of Semantic Computing, Vol. 7, No. 3 (2013), pp 215-236. Available at http://sadl.sourceforge.net/S1793351X13500025.pdf.
[5] See http://sadl.sourceforge.net/. SADL Version 3 may be downloaded from https://github.com/crapo/sadlos2/releases.
[6] Xtext is a framework for development of domain-specific languages, see https://www.eclipse.org/Xtext/.
[7] Apache Jena is an open source Java framework for building Semantic Web applications, see https://jena.apache.org/.
[8] REST is a type of web service, see https://en.wikipedia.org/wiki/Representational_state_transfer.

capture computations and creates a language-neutral representation of the model for training, inference, and model composition.

3.2 Human-Computer Interaction: Dialog Mixed Initiative Collaboration Language

Fundamentally, a mixed-initiative human-computer interface requires that the listener that "hears" what the user says to initiate an interaction is not the same thread as the processor that is working and may at any time initiate an interaction from the computer side. Otherwise the conversation would always be controlled entirely from one side. In our case, the Dialog Editor serves as a "chat window" for input from either side and so needs an appropriate level of synchronization to make sure that messages are not intermingled. This architecture for mixed initiative interaction is used because it is inexpensive to implement and integrates well in the Eclipse Xtext environment where other existing controlled-English elements are used. For example, any domain concept used in the Dialog editor window, whether coming from the user or the system, is colored by semantic type and hyperlinked to the content in the knowledge graph's ontology where it is defined, as well as to other places where it is referenced. The design places as much of the system implementation as possible in the backend (right side of Figure 3) so that the UI components can be replaced, if desired in the future, with as little impact as possible to the backend components that implement the rest of the architecture. In other words, we want to demonstrate the utility of mixed initiative in placing the human as eminently in the loop as needed without spending a lot of time designing a state-of-the-art mixed initiative interface.

The key in our proof-of-concept system is that the statements (or questions) that initiate a collaborative exchange are annotated with their location in the document, and the response from the other party is inserted into the document right after that location. With appropriate synchronization so that messages are not intermingled or lost, the result will be a series of interactions, initiating statement followed by the other party's response, regardless of who initiated the interaction. System messages, whether initiation or response, are prefixed with "CM" for "Curation Manager", but this can be modified or perhaps enhanced to identify the specific part of the ANSWER system that is the source of the statement.

To illustrate, consider the very simple ontology that defines a circle and adds a rule to compute the area given the radius.

```
uri "http://sadl.org/model.sadl" alias mdl.

Circle is a class described by radius with values of type decimal,
      described by area with values of type decimal.

Rule AreaOfCircle: if c is a Circle then area of c is PI*radius of c^2.
```

Now suppose that we open a Dialog window, import the simple model above, and ask a question about our domain.

```
uri "http://darpa/test" alias tst.
```

```
import "http://sadl.org/model.sadl".

What is a Circle?
CM: Circle is a class
      described by area with values of type decimal,
      described by radius with values of type decimal.
```

We might also wish to give the system a specific instantiation of a circle and get computed information about it.

```
uri "http://darpa/test" alias tst.

import "http://sadl.org/model.sadl".

UnitCircle is a Circle with radius 1.
What is UnitCircle?
CM: UnitCircle is a Circle with radius 1, with area 3.1415927410125732.
```

The Dialog grammar supports more specific questions, such as “What is the area of UnitCircle?” and will be extended as needed to allow a richer interaction as we discover what kinds of interactions are most useful. Extensions will be informed by the GE ASKE TA2 team.

3.3 Extraction from Code

We chose the NASA Hypersonics Web site as our domain of experimentation and demonstration. For M3 we have focused on the Speed of Sound page, with its downloadable class Mach.java, which extends JavaApplet. From our initial investigation has emerged a more detailed approach to extraction from code. The Java Applets on the NASA Hypersonics Web site in general, and Mach.java in specific, have characteristics that make extraction somewhat difficult. However, these same difficulties allow us to identify challenges earlier and design to overcome them. These characteristics are undoubtedly not unique to this code but will be encountered in many scientific codes in different languages. They include the following.

1. The code has very few comments (since there are so few, and we may create some comments to better demonstrate the important capability of using in-code comments during extraction of scientific models)
2. Top-level input and output variables are fields in the class with global scope and are not identified as to whether they are a model input or output or neither. The main method encapsulating the scientific model has neither input arguments nor a return value. The code snippet below shows some of the global variables and the main method signature.

```
double gama,alt,temp,press,vel,ttot ;
double rgas, rho0, rho, a0, lrat, mach ;
double Q, TT, T, M, GAM, R, A, V, G  ;
int lunits,vparam,tparam,ther,prs, planet ;
int mode ;
…
public void computeMach() {
…
```

3. User-interface components are intermingled with scientific computation. For example, the outputs of the main computation method are passed to GUI widgets, e.g., the computed value *vel* is used in this line of code:

```
in.dn.o4.setText(String.valueOf(filter0(vel))) ;
```

   This may be unusually common in this code because it is a Java Applet, but the intermingling is certainly not unique to applets.
4. Variables are assigned values by computation, and then the same variable is reassigned a value, e.g. in the code snippet below vel already has a value, is assigned a value (the comment suggests in ft/sec), then after a few lines is reassigned a value (the comment suggests in mph, which is apparently the same value it had originally).

```
V = vel = vel * 88. / 60. ;  // v in ft/sec
temp = ttot - (gama -1.0) * vel * vel / (2.0 * gama * rgas) ;
a0 = Math.sqrt(gama*rgas*temp) ;  // feet /sec
mach = vel / a0 ;
a0 = a0 * 60.0 / 88. ;   // mph
vel = vel * 60. / 88. ;  // v in mph
```

5. The execution path through the main computational method is controlled by two flags set in the user-interface, one relating to the target computation output (object speed or Mach number) and the other to the input to be used (altitude or total temperature).
6. The code apparently was once broader in scope. There are crumbs leftover from previous versions. For example, the variable *planet* (see code snippet in #2 above) never appears outside of its declaration.

To deal with these challenges, we have identified a multi-level approach. Our overall approach, as identified in our proposal, remains to use the abstract syntax tree (AST) generated by an existing parser. For Java, we use the Eclipse Java Development Tools (JDT), which exposes its AST nicely. This allows us to "see" the overall structure of the code—the class or classes and the methods in the case of Java. Within a method, if there are arguments we assume that these are inputs, and if there is a return value we assume that it is an output. These assumptions can be tested, or if there are no arguments and/or a returned value, input/output assumptions can be generated, by looking at the sequence of computational statements in the method. If an argument variable first appears in the right-hand side (RHS) of an assignment statement or first appears in another kind of statement, e.g., a comparison, that validates the belief that the variable is an input. If a variable which has global scope first appears in the RHS of an assignment statement or first appears in another kind of statement, then it can be assumed to be an input to the method. For example, *alt* is a field in the Mach class (has global scope) and first appears in the computeMach method in this statement, from which we infer that it is an input.

```
if (alt <= 36152.) {           // Troposphere
```

Similarly, if a variable is returned by a method then that variable can be assumed to be an output. However, in computational methods that do not return values, the situation is more ambiguous. When a variable is set by an assignment statement, that is the variable appears on the left-hand side (LHS) of the assignment, and the variable is not used in any subsequent statements, it can be presumed to be an output of the method. Yet a variable that is used in subsequent statements can still be an output. Either it may be used, after it is set, to compute other outputs, or, as in our case, it may appear in an output that makes the value available in some destination, e.g., *a0* in this statement is placed in a GUI text box.

```
in.dn.o3.setText(String.valueOf(filter0(a0))) ;
```

From the analysis of the AST one may categorize methods, variables, and statements or blocks of statements. In fact, the analysis of blocks of statements appearing within a control statement may be more effective in understanding the scientific meaning of the block of statements than an attempt to understand the control structure. For example, consider these two partial blocks that depend upon the flow control flag *vparam*.

```
if (vparam == 1) {
    temp = ttot / (1.0 + .5 * (gama - 1.0) * mach * mach) ;
    a0 = Math.sqrt(gama*rgas*temp) ;  // feet /sec
    a0 = a0 * 60.0 / 88. ;   // mph
    vel = mach * a0 ;

 if (vparam == 0) {
    V = vel = vel * 88. / 60. ;  // v in ft/sec
    temp = ttot - (gama -1.0) * vel * vel / (2.0 * gama * rgas) ;
    a0 = Math.sqrt(gama*rgas*temp) ;  // feet /sec
    mach = vel / a0 ;
```

It would be difficult, from analyzing the code that sets *vparam*, to determine what it means. But from looking at the input and output variables of the respective code blocks, one can reason that the first computes *vel* given *mach*, and that the second computes *mach* given *vel*.

It is hoped, but not yet proven, that the logic that analyzes code based on these characterizations of inputs and outputs, will be applicable beyond Java source code. To that end we will continue to try to separate the extraction of characteristics from the AST from the reasoning applied to the extracted characteristics. Thus, we hope to apply the same logical analysis to other languages.

Once a block of statements has been determined to be a scientific computation, we go to the next level. Rather than create a new translator to convert the code block into the target language, we opt to reuse existing translators. This architecture should support the addition of other languages by plugging in existing translators, reducing substantially the effort needed. In our case, where the source is Java and the target is Python, we find a promising translator in the github project java2python[9]. While there may be Java source code for which translation at the

[9] See https://github.com/natural/java2python.

class (.java file) level would be useful, such is clearly not the case with our selected code source. Rather, the most useful translation will be, in the case of the top-level computational method *computeMach*, at the block-of-statements level. We can easily wrap any block of Java statements in a pseudo method with the identified inputs and outputs, and wrap that in a simple class container. After translating this to Python, we can extract the desired Python method and use it as the model extracted from code and add it to the K-CHAIN computational graph. In the case of lower-level methods in the *Mach* class, which are called from our identified code blocks, they can be translated separately or be placed in the same pseudo Java class for translation. In either case, the result is a set of equations extracted from the Java code, translated to Python methods, and added to the computational graph with semantic metadata characterizing the model(s) added to the knowledge graph. More about the semantic characterization will be described Section 4.

3.4 Extraction from Text

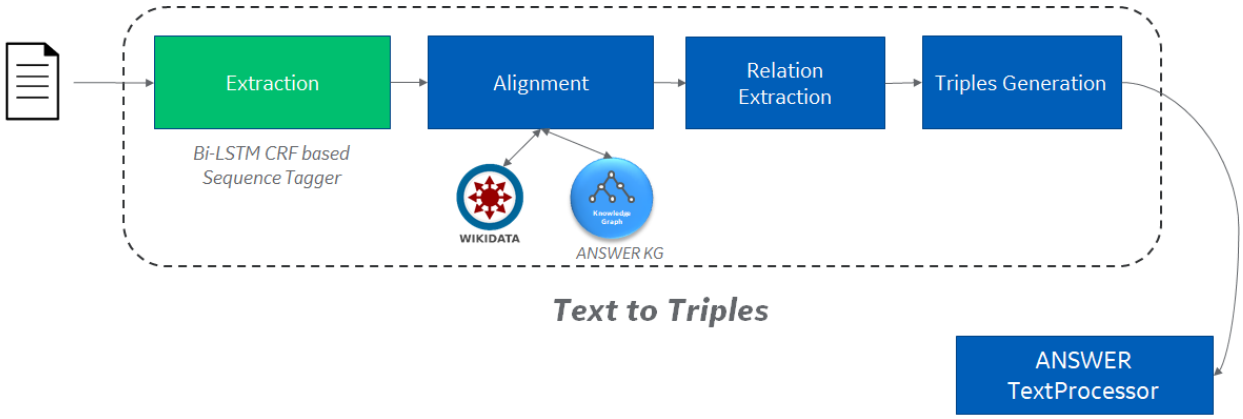


Figure 4 Architecture for the "Text to Triples" module introduced in ANSWER's overall architecture in Figure 3. The green color module has been implemented, while the blue ones are in progress.

Figure 4 shows the overall approach for the "Text to Triples" module. This module is responsible for parsing textual content and extracting triples related to scientific concepts and equations. The Text to Triples module will expose its functionality via REST services.

The text processing begins with the Extraction module which identifies and extracts scientific concepts and equations of interest. The Alignment module further reconciles the extracted concepts with the existing ones in ANSWER's knowledge graph. If no match is found, the alignment module attempts to search for existing concepts in external knowledge graphs such as Wikidata. After the alignment process is complete, the Relation Extraction module identifies relations that may exists between the extracted concepts. Finally, the concepts, equations and relations are represented in the appropriate triple format using classes and properties from the supporting ontology. Text to Triples returns the generated triples to ANSWER's TextProcessor module.

We have currently finished our implementation of the Extraction module that's part of Text to Triples. For rest of the section, we describe the development of the extraction module including

details such as generation of training and test data, training a sequence tagging model, and preliminary evaluations including an independent evaluation over the Speed of Sound webpage.

3.4.1 Extraction

The extraction module treats the problem of identifying scientific concepts and equations in text as a sequence tagging task. Sequence tagging, in natural language processing (NLP), involves tasks such as assigning a grammatical tag to each word (part of speech tagging), grouping words into phrases (chunking) and assigning an entity type to a group of words (named entity recognition). The task of tagging scientific concepts and equations in text is akin to a sequence tagging problem, in which the goal is to tag every token in a sentence as either belonging to a scientific concept or equation. Figure 5 shows an example output of sequence tagging task applied on a sentence from the Speed of Sound webpage. The example shows extracted scientific concepts (example for equations can be found further in this section).

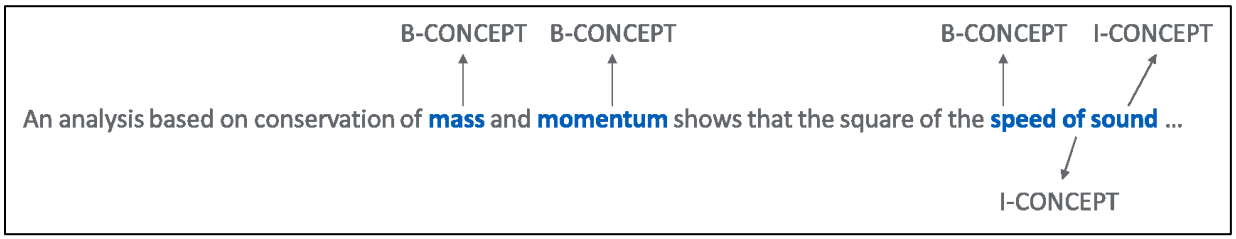


Figure 5 The tags B-CONCEPT and I-CONCEPT indicate that the token is either the beginning of a concept or inside the concept

Generating Training Data: Typical architectures for sequence tagging are supervised in nature and need sufficient training data for the models to be trained. We generate the required training data by automatically annotating all the web pages (except the Speed of Sound ..Interactive** page), listed on NASA's Hypersonic Aerodynamics Index[10] (henceforth referred to as the "NASA corpus"). Annotating the NASA corpus involves two steps. In the first step, we manually create a dictionary of known scientific physical concepts. This dictionary of concepts is then used along with Apache's UIMA ConceptMapper[11] to identify and annotate scientific concepts in pages in the NASA corpus. The identified concepts are used as ground truth annotations for both training and test datasets.

We create the scientific concepts dictionary by extracting all the titles for Wikipedia pages that are classified as "Physical Quantities"[12]. These titles are represented as concepts in a semantic model using SADL. Each concept is assigned a unique uniform resource identifier (URI) along with its possible synonyms. The advantage of representing them in SADL is that it will allow domain experts in the future to tweak and extend the "dictionary" as and when necessary. Please note, the current SADL model is an initial model as part of M3 report. This model will be updated to include relation between the Physical Quantities in here with the concepts in the supporting

[10] Hypersonic Aerodynamics Index: https://www.grc.nasa.gov/WWW/BGH/shorth.html
[11] Tanenblatt, M. A., Coden, A. and Sominsky, I. L. 2010. The ConceptMapper Approach to Named Entity Recognition. In Language Resources and Evaluation Conference (LREC).
[12] List of Wikipedia pages under the "Physical quantities" category: https://en.wikipedia.org/wiki/Category:Physical_quantities

ontology (e.g. relations such as subclass of ScientificConcept and/or UnittedQuanitity). The SADL model file can be found in our GitHub repo: *DARPA-ASKE-TA1/ModelsFromText/models-from-text-sadl-model/PhysicalQuantities.sadl (link)*.

```
Amplitude(alias 'Amplitude')is a top-level class.
Angular_acceleration(alias 'Angular acceleration')is a top-level class.
Angular_diameter_distance(alias 'Angular diameter distance')is a top-level class.
Angular_frequency(alias 'Angular frequency')is a top-level class.
Angular_momentum(alias 'Angular momentum')is a top-level class.
Angular_velocity(alias 'Angular velocity')is a top-level class.
API_gravity(alias 'API gravity')is a top-level class.
Areal_velocity(alias 'Areal velocity')is a top-level class.
Attenuation_coefficient(alias 'Attenuation coefficient')is a top-level class.
```

Figure 6 Physical Quantities extracted from Wikipedia are represented as concepts in a SADL model allowing users to easily modify the base dictionary. SADL will allow us to capture intricate relationships between these concepts including parent-child relation in the future

The OWL file generated by SADL is further translated into the UIMA ConceptMapper dictionaries in XML format. Each of the concepts in the ontology is translated into its canonical form and the variants for the dictionary format. Canonical form of a concept is represented by its URI, while its variants include the aliases. The ConceptMapper pipeline searches for and identifies the presence of canonical forms and variants from the dictionaries in the given text. It's able to provide accurate mappings between the content in the text and terms in the dictionaries, essentially performing a complex dictionary lookup. Details of the ConceptMapper workings can be found it its documentation[13]. The code for generating annotations with the UIMA ConceptMapper pipeline can be found under *DARPA-ASKE-TA1/ModelsFromText/text.extraction/* (link).

Once the annotation process is complete, each sentence is tokenized and assigned the appropriate Inside-Outside-Beginning (IOB) tags (see Figure 5). "I" indicates the token is inside the class (e.g. in our case its either a scientific concept or equation), "O" indicates the token is not of interest (or outside the class) and "B" indicates the token is the beginning of the class. The IOB tags for scientific concepts are assigned based on the ConceptMapper annotations, whereas the ones for the equations are assigned manually. The data is further randomly split with 2/3rd of the corpus into a training set and the remaining into a test set. The training and test files are located under *DARPA-ASKE-TA1/ModelsFromText/sequence-tagger/dataset/nasa-website/annotations/data/* (link).

Training a Sequence Tagging Model: We use the annotated data to train a supervised sequence tagging model. Neural network architectures, in recent times, have been able to match or surpass the state of the art for several NLP tasks including sequence tagging. We utilize the bidirectional Long Short-Term Memory (BI-LSTM) with a Conditional Random Field (CRF) layer (BI-LSTM-CRF)

[13] Apache UIMA ConceptMapper Documentation: https://uima.apache.org/downloads/sandbox/ConceptMapperAnnotatorUserGuide/ConceptMapperAnnotatorUserGuide.html

architecture proposed by Huang et al.[14] which can produce state of the art accuracies for named entity recognition.

We use the BI-LSTM-CRF implementation provided as part of the flair framework[15]. Text data needs to be mapped into a vector space before it can be processed by the neural network and with the advent of deep learning, embeddings (or dense vector representations)[16] have become one of the most popular techniques to do so. More recently, the idea of "Stacking Embeddings" or combining different types of embeddings to produce a single vector has shown promising results in sequence tagging tasks. The idea behind stacking embeddings is to concatenate embedding vectors from different sources to produce a single vector. We map our text data into a dense vector by stacking two different embeddings. We use a pre-trained model of the popular GloVe word embeddings[17] and stack it with Character embeddings trained for our specific task. Our intuition behind using character embeddings is to support the equation tagging task – equations are made of up of more characters than words. The flair framework provides an easy option to specify the different embeddings that one wishes to stack. When character embeddings is provided as one of the options, flair automatically trains it during the downstream task training (in this case, during the sequence tagger training). The code for training the sequence tagger can be found in our GitHub repo: *DARPA-ASKE-TA1/ModelsFromText/sequence-tagger/ner-train.ipynb* (link).

Initial Experiments: The sequence tagging model was trained with 256 hidden states and 1 LSTM layer. The initial learning rate was set to 0.1 and the maximum number of epochs to 150. Flair employs a simple learning rate annealing method in which the learning rate is halved if training loss does not fall for 5 consecutive epochs. It chooses the model that gives the best F-measure in the best epoch as judged over the test data set.

Our sequence tagging model was trained over the training and test set created from the NASA corpus. The training stopped after 95 epochs as the learning rate become too low. Figure 7 shows the preliminary performance of the model.

	Precision	Recall	F-measure	Accuracy
Scientific Concepts	0.9761	0.9109	0.9424	0.9424
Equations	0.9322	0.8871	0.9091	0.9091

Figure 7 Evaluations of the BI-LSTM-CRF Sequence Tagger model over a test data set created from the NASA corpus

The current results look promising. The model performs well over the test data set with high accuracies for identifying both scientific concepts and equations. We choose to describe these results as "initial", since we haven't performed any model optimization experiments yet. Options

[14] Huang, Z., Xu, W. and Yu, K., 2015. Bidirectional LSTM-CRF models for sequence tagging. arXiv preprint arXiv:1508.01991.

[15] https://github.com/zalandoresearch/flair

[16] Goldberg, Y., 2016. A primer on neural network models for natural language processing. Journal of Artificial Intelligence Research, 57, pp.345-420.

[17] Pennington, J., Socher, R. and Manning, C., 2014. Glove: Global vectors for word representation. In Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP) (pp. 1532-1543).

include evaluating with different training rates, different combination of embeddings etc. The current accuracy numbers are also a function of the scientific concepts dictionary that was used to automatically annotate the training and test data sets. Valid concepts that are absent from the dictionary and appear in text do not get counted as missed concepts (nor were they used during training). We will need to perform a human evaluation (albeit on a smaller scale) to estimate how many such concepts may have been missed. We also need to evaluate the bias of the model. The current model has been trained and evaluated over the NASA corpus, which doesn't necessarily give picture of its generalization. Next steps would include evaluating the model over scientific articles from different sources (e.g. different websites and publications). The current model's prediction over the test data set can be found here: DARPA-ASKE-TA1/ModelsFromText/sequence-tagger/resources/taggers/ner-equations-01-23-2019/test.tsv (link). Unfortunately, we were not able to upload the trained model since the size exceeds the maximum file size permitted by GitHub. We will find alternate method to share the model with DARPA.

Speed of Sound Use-case: We use this model and annotate concepts and equations on the Speed of Sound webpage. Recollect, this webpage was neither part of training or the test data set. This experiment helps us to both validate the performance of the model as well as identify challenges and next steps.

```
Equation Text: a^2 = R * T * gamma

Tagged Equation:
a^2 <B-EQUATION> = <I-EQUATION> R <I-EQUATION> * <I-EQUATION> T <I-EQUATION> * <I-
EQUATION> gamma <I-EQUATION>

Equation Text: a^2 = R * T * {1 + (gamma - 1) / ( 1 + (gamma-1) * [(theta/T)^2 *
e^(theta/T) /(e^(theta/T) -1)^2]) }

Tagged Equation:
a^2 <B-EQUATION> = <I-EQUATION> R <I-EQUATION> * <I-EQUATION> T <I-EQUATION> * <I-
EQUATION> {1 <I-EQUATION> + <I-EQUATION> (gamma <I-EQUATION> - <I-EQUATION> 1) <I-
EQUATION> / <I-EQUATION> ( <I-EQUATION> 1 <I-EQUATION> + <I-EQUATION> (gamma-1) <I-
EQUATION> * <I-EQUATION> [(theta/T)^2 <I-EQUATION> * <I-EQUATION> e^(theta/T) <I-
EQUATION> /(e^(theta/T) <I-EQUATION> -1)^2]) <I-EQUATION> } <I-EQUATION>
```

Figure 8 Accurately tagged equations from the Speed of Sound page

Firstly, the model was able to accurately tag both the equations that appear on the page. The <B-EQUATION> tag indicates the start of the equation and every subsequent <I-EQUATION> tag indicates that the preceding token is part of the equation (Figure 8).

```
Because the speed <B-CONCEPT> of <I-CONCEPT> transmission depends on molecular
collisions, the speed <B-CONCEPT> of <I-CONCEPT> sound <I-CONCEPT> depends on the
state of the gas. The speed <B-CONCEPT> of <I-CONCEPT> sound <I-CONCEPT> is a
constant within a given gas and the value of the constant depends on the type of
gas (air, pure oxygen, carbon dioxide, etc.) and the temperature <B-CONCEPT> of the
gas.
```

Figure 9 The model can extract partial concepts (highlighted in yellow)

The model was also able to identify several relevant concepts from the page such as mass, momentum, temperature, and, speed of sound to name a few. We also noticed that it was able to identify partial concepts. For e.g. while the model was not able to extract "speed of transmission", it was only able to go as far as "speed of". Partial concepts like these provide an opportunity for a human collaborator to intervene and give feedback to the system (Figure 9). The complete results of the model annotation of the Speed of Sound webpage can be found in this Python notebook: DARPA-ASKE-TA1/ModelsFromText/sequence-tagger/ner-speed-of-sound.ipynb (link).

3.4.2 Next Steps

We will continue to optimize and evaluate the current sequence tagger model. Next steps in terms of evaluation of the model have been outlined in the previous section. Our next immediate target will be the development of the Alignment module followed by Relation Extraction and Triple Generation.

3.5 Extraction as Coordinated Effort

If the NASA Hypersonics Web site is an indication of what to expect when extracting scientific equations from code, extraction of equations is easy compared to understanding their meaning by understanding the scientific concept represented by each input and output. It seems likely that a human looking at the code would be in the same boat—unless the equation is already part of the person's knowledge, they would easily understand the operations that are being performed but might find it impossible to relate the computations to anything in the real world. Consider, for example, this statement from Mach.java.

```
a0 = Math.sqrt(gama*rgas*temp) ;  // feet /sec
```

The Java AST allows us to see the comment, "feet /sec", and associate it with the statement. If the code extractor were to query the Text to Triples service for the meaning of this comment, one might expect to get back something like what is found if one does a Google Search on exactly that phrase. The second reference is a Wikipedia page which begins with this statement.

> "The **foot per second** (plural **feet per second**) is a unit of both speed (scalar) and velocity (vector quantity, which includes direction).[1] It expresses the distance in feet (**ft**) traveled or displaced, divided by the time in seconds (**s**, or *sec*).[2] The corresponding unit in the International System of Units (SI) is the metre per second."

Or we might expect a good ontology, such as qudt[18], to already include this knowledge. In either case, the units give us a hint that *a0* might be speed. But speed of what we have not yet deciphered from the code.

There are more hints in code comments. In the file header we find this comment.

[18] See http://qudt.org/.

```
"Mach and Speed of Sound Calculator
Interactive Program to solve Standard Atmosphere Equations
for mach and speed of sound
Calorically imperfect gas modeled for hypersonics"
```

If we look at where rgas is set, which can be determined from the AST, we find this statement.

```
// Earth  standard day
   rgas = 1718. ;                /* ft2/sec2 R */
```

The value and units in the comment on the line, along with the preceding comment, might be enough to identify the concept in one of several scientific concept definition repositories. The first hit of a google search on "1718 ft2/sec2 R" has this summary sentence.

> "R is the universal gas constant, 1.995 m2/sec2 K or **1718 ft2/sec2 R**."

In fact, as sparse as comments are in this code, they can be a useful addition to the documentation found in the Speed of Sound Web page. In this text we find the equation.

> a^2 = R * T * gamma

which is preceded by text which reads, in part,

> "the square of the speed of sound **a^2** is equal to the … gas constant **R** times the temperature **T** times the ratio of. specific heats **gamma"**

where the bold font is in the original text. Several inferences are necessary to reconcile the equation from code with the equation from text:

- *gama* is the same as *gamma*
- *rgas* is the same as *R*
- *temp* is the same as *T*
- *a0* is the same as *a*
- *Math.sqrt(a^2)* is the same as *a*
- Multiplication is commutative: *R*T*gam*ma is equivalent to *gamma*R*T*

Given the potential difficulty of drawing all of these inferences reliably, this example demonstrates the utility of the human in the loop. Whether by the suggested inference chain, or because a machine-learning approach finds a similarity between the equation in code and the equation in text, our mixed initiative approach allows the system to ask the user for final verification. Such verification might begin with something like this, where the links would open a window on the respective sources:

CM: Is equation "a0 = Math.sqrt(gama*rgas*temp)" (see code-link) the same as "a^2 = R * T * gamma" (see text-link)?

The coordinated effort that we see as the longer-term objective includes extraction from code, extraction from text, and interaction with the human. The model created by extraction from documentation can provide context to the extraction from code and comments in code. Likewise, extraction from code can inform extraction from associated documentation. An iterative, bootstrapping approach may prove useful in difficult situations.

3.6 Knowledge-Consistent Hybrid AI Network (K-CHAIN) Computational Model

3.6.1 Why use TensorFlow Graph to represent computational models?

TensorFlow uses dataflow, which is a common programming model for parallel computing. In a dataflow graph, the nodes represent units of computation, and the edges represent the data consumed or produced by a computation. As mentioned in the TensorFlow documentation[19], dataflow graphs have several advantages that TensorFlow leverages when executing models including: 1) identifying operations that can execute in parallel, 2) distributed execution on different hardware, 3) hardware-specific compilation for faster execution, and 4) portability. It is important to note that portability implies the dataflow graph generated by our ANSWER framework is a language-independent representation of the model. Thus, we can build a dataflow graph in Python, store it locally, and restore it in a C++/Java program for low-latency inference. Due to developments in the deep learning community, automatic differentiation is also available in TensorFlow, which allows us to encode and compute with differential equations on computation graphs. Additionally, TensorBoard allows visualization of the computational graph and model parameters, which is a helpful feature. Recently TensorFlow Probability has also introduced Bayesian inference to characterize uncertainty in the model. Because of these powerful attributes that aligns with the necessary scalability and the grand vision of the ASKE program, we have chosen to use TensorFlow Graph to represent the computational models. However, it is noted that the curation manager and extraction modules are agnostic of the computational modeling framework. The implementation via RESTful web services provides modularity and enables future interaction with other modeling approaches and packages, such as dynamic Bayesian networks, grounded function networks (GrFNs)[20], etc. as long as RESTful services can be created to with a compatible interface.

3.6.2 How are the computational models created, stored, and utilized?

The computational model creation and evaluation is provided as a web service, which interfaces with the ANSWER curation manager and inference processor. The architecture of the implementation is given in Figure 10. The web service utilizes the K-CHAIN library (a.k.a. kChain), which was created since the submission of the M1 report.

[19] https://www.tensorflow.org/guide/graphs#why_dataflow_graphs
[20] https://delphi.readthedocs.io/en/master/grfn_spec.html

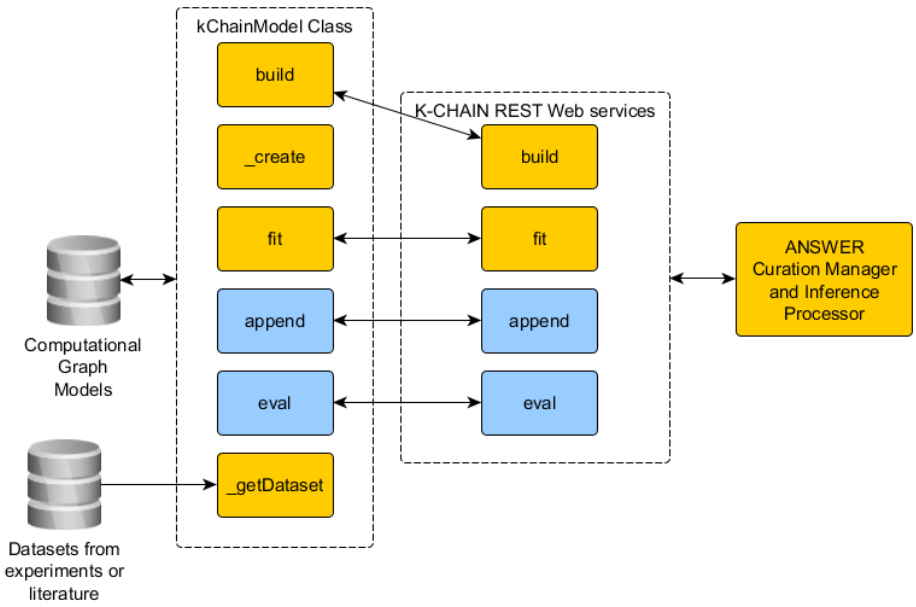


Figure 10: The implementation architecture for computational graph modeling and execution. (The yellow blocks have a working instantiation and blue blocks are being developed). Architecture for the "Computational Graph" module introduced in ANSWER's overall architecture in ***Error! Reference source not found.****.*

The kChain library consists of kChainModel class, which includes various methods to build, fit, and evaluate models, as well as compose disparate computational models. The K-CHAIN web service is utilized to build a model in at least the following two cases:

1) An equation representing a relationship between semantically-meaningful scientific concepts is available, or
2) A dataset is available consisting of variables that are semantically-meaningful scientific concepts and are known to have a "depends" or "causes" or similar relationship.

Here semantically-meaningful scientific concepts refer to those variables that are grounded into the knowledge graph of the domain as objects of type ScientificConcept.

In case of exact knowledge (case 1), if a dependency of speed of sound upon temperature of medium (air/gas) and other properties of gas (molecular weight, specific heat capacities, etc.) is extracted from text and/or code, then a computational model can be directly created as well. On the other hand, an example of the imperfect knowledge (case 2) from the NASA Speed of sound page is as follows: "… the temperature depends on the altitude in a rather complex way". Here the scientific concept of temperature is known to be dependent on the concept of altitude, however the relationship is not available. If through the context of surrounding text or with human help, ANSWER can infer that here temperature refers to 'temperature of the atmosphere' or 'temperature of air', then concepts can be incorporated and aligned with the knowledge graph. Additionally, if a dataset with observations of those semantic concepts becomes available either via experiments or during literature search, then the build service can be utilized to create a data-driven model relating those concepts.

To illustrate use cases of the build service, we show three demonstrations here with a simple example from Newton's Second Law.

Demo 1: Build model without data or equation:
Input:

```
{'inputVariableNames': ['Mass', 'Acceleration'], 'outputVariableNames':
['Force'], 'dataLocation': None, 'equationModel': None, 'modelName':
'Newtons2LawModel'}
```

Output:

```
{"modelType": "NN", "trainedState": 0, "metagraphLocation":
"Models/Newtons2LawModel"}
```

Note that `equationModel` is given as `None` as in this case *F = m * a* is not known to the system a priori and `dataLocation` is `None` as a suitable dataset has not been identified. The service parses the json object and calls kChainModel.build() method. This method internally uses _create method to construct a neural network model as a TensorFlow graph with input variables named using `inputVariableNames` and output variables named using `outputVariableNames`. The resulting graph is stored as a MetaGraph[21] object by TensorFlow and location of that MetaGraph is returned as an output of the _create method. The output informs that the model created is a neural network (NN), which is not yet trained, and the MetaGraph location is provided for future use of the model. Ideally, Curation Manager will use the build service only if `equationModel` or `dataLocation` is available. We will see these two cases next.

Demo 2: Build model using a dataset
If a dataset for Force, *Mass*, and *Acceleration* concepts called 'Force_dataset.csv' is created or becomes available, then the dataset location can be specified to create and fit the model using the build service as follows:

Input:

```
{'inputVariableNames': ['Mass', 'Acceleration'], 'outputVariableNames':
['Force'], 'dataLocation': 'Datasets/Force_dataset.csv', 'equationModel':
None, 'modelName': 'Newtons2LawModel'}
```

Output:

```
{"modelType": "NN", "trainedState": 1, "metagraphLocation":
"Models/Newtons2LawModel"}
```

In this execution after the model is created using _create method or _createNNModel method to be precise, the dataset is retrieved by using the _getDataset method. In the fit method, the model is revived from the MetaGraph and computational nodes necessary for training the model, such as for loss function and optimizers, are appended to the graph and training is performed. Note that in the output, the `trainedState` is now switched to True or 1. The resulting trained model with parameters and weights is saved back as a MetaGraph.

Demo 3: Build model using equations:

[21] MetaGraph in TF: https://www.tensorflow.org/api_guides/python/meta_graph

In lieu of the dataset, a more likely scenario in ASKE effort is to find the equation for that relation in code or in text. In those scenarios, we use the build service with the equation. There are several ways of sharing the equation model to create the computational graph that we are exploring. One of the approaches is illustrated below:

Input:

```
{'inputVariableNames': ['Mass', 'Acceleration'], 'outputVariableNames':
['Force'], 'dataLocation': None, 'equationModel': 'Force = Mass *
Acceleration', 'modelName': 'Newtons2LawModel'}
```

Output:

```
{"modelType": "Physics", "trainedState": 0, "metagraphLocation":
"Models/Newtons2LawModel"}
```

Here the illustration is set as if the equation was extracted from text and all relevant concepts were incorporated or already known in the knowledge graph. If a code snippet was extracted from Java, then java2python translators (as mentioned in extraction from code section above) will provide Python multi-line code snippets, which can be provided as formatted strings in a similar way. These experiments with larger code snippets with NASA Hypersonics Index are also being performed now. The build service calls the build method and then _createEqnModel method to create the computational model for the equation. The output of this service is a TensorFlow model to perform that computation which can be executed using the *eval* service. These models are created by defining new Python methods from equation strings and then wrapping those methods as TensorFlow computations using tensorflow.py_func()[22] functionality. Additionally, we also leverage a new package in TensorFlow called as AutoGraph[23], which provides basic functionality to create TF graph or code from native Python code including conditionals and loops.

In summary for the *build* service, the I/O interaction JSON is given in Table 1. We will also update the service to provide input and output variables as a list of json objects, so that name, type, and default attributes with corresponding variables can all be included without ambiguity. For example: "inputs": [{"name": "a", "type": "double"}, {"name": "b", "type": "double"}]. The degree of fitness is currently a placeholder to report accuracy and other metrics of the fitted model.

Table 1: Summary of input-output JSON interface for build service

Input to the build service	Output returned by the build service

{
"inputVariableNames": ["a", "b"],
"outputVariableNames": ["c"],
"dataLocation": "URIString",
"equationModel": "string",
"modelName": "URIString"
}
	{
	"modelType": "someTypeVal",
	"trainedState": true,
	"metagraphLocation": "somePath",
	"degreeFitness": [1.10, 2.20]
	}

[22] Documentation for py_func: https://www.tensorflow.org/api_docs/python/tf/py_func

[23] Introduction to TensorFlow AutoGraph: https://www.tensorflow.org/guide/autograph?hl=en

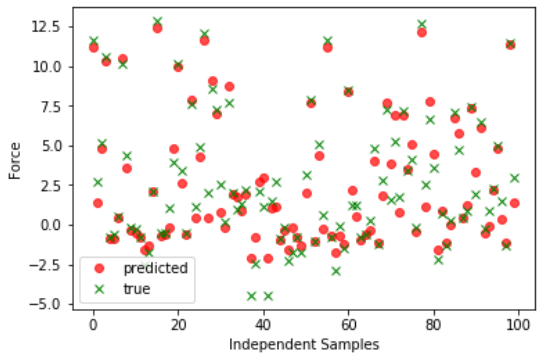
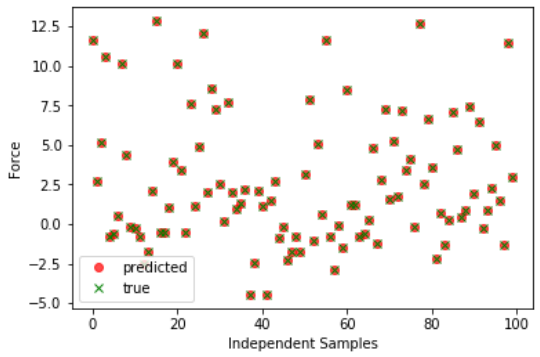
In the K-CHAIN model RESTful web services, the *fit* service is also directly available to update parameters of an existing model with a new dataset. The fit service utilizes the fit method of kChainModel Class. Due to modularity of the implementation, the fit method is internally used in the build method as well to train new models with any available data. The key-value pairs in the JSON object are a subset of pairs given for the build service.

Table 2: Summary of input-output JSON interface for fit service

Input to the fit service	Output returned by the fit service

{
"modelName": "URIString",
"inputVariableNames": ["a", "b"],
"outputVariableNames": ["c"],
"dataLocation": "URIString"
}
	{
	"trainedState": true,
	"metagraphLocation": "somePath",
	"degreeFitness": [1.10, 2.20]
	}

Table 3: Example of eval service for predictions with data-driven and physics models

Predictions from data-driven Force model	Predictions from physics-based Force model
	

In the *eval* service, any existing model irrespective of `modelType`, that is, physics or data-driven or hybrid can be evaluated or executed for inference to create outputs of interest by providing relevant inputs. Unlike other services, this eval service will typically be requested by the ANSWER Inference Processor instead of the ANSWER Curation Manager, to provide quantitative outputs during mixed-initiative dialogues between ANSWER and user. In Demo 2 and Demo 3 above, a data-driven model for *Force* and a physics equation-based model for *Force* from Newtons 2nd law was constructed. In Table 3, we show outputs of running those models using the same inference functionality. Since the dataset was created via simulation, we see that the physics model fits perfectly, whereas the data-driven model has some discrepancy. The *eval* functionality is still being put up as a web service and incorporated as a part of kChainModel class and will be available in a few more days.

We will also add a service and method to append concepts to existing computational models. This *append* service will add new concepts to an existing model and enable us to grow our model over time. In *eval* during session run time, the relevant subgraph will be automatically identified and executed for generating the relevant outputs. For example, an initial model with *position*, *velocity*, *acceleration*, *mass*, and *force* variables can append the concept of *momentum* to produce the final model. If the initial model was ModelType Physics and the concept of

momentum is incorporated with an equation, then the final model is of ModelType Physics. If the KG consists of "momentum depends on mass and velocity" only, then the momentum model will be of type NN or GP, hence final model will be Hybrid. If curation manager uses model "append" with an equation for the concept, then ModelType evolves as follows: Physics -> Physics, Data-driven -> Hybrid, and Hybrid -> Hybrid. If curation manager uses model *append* without an equation for the concept, then ModelType evolves as follows: Physics -> Hybrid, Data-driven -> Data-driven, and Hybrid -> Hybrid. The ability to create hybrid models with data-driven and physics-based components led to the name of Hybrid AI Networks in K-CHAIN. Note that the *append* service also allows one to create meta-models by merging multiple computations over similar variables with conditional statements denoting context. For example, the computation for speed of sound for a calorically perfect gas and for a calorically imperfect gas have the same inputs variables, but the equations are completely different. A manually-created computational graph for speed of sound is shown in Figure 11.

```
mdl = tf.Graph()
with mdl.as_default():
    str1 = "def speedOfSound_1(T, R, gam): \n \t return math.sqrt(R*gam*T) "
    exec(str1)

    str2 = "def speedOfSound_2(T, R, gam): \n \t theta = 3056.0 #in Kelvin \
    \n \t x1 = math.sqrt(R * T * (1 + (gam - 1) / ( 1 + (gam-1) * ((theta/T)**2 \
    * np.exp(theta/T)/(np.exp(theta/T) -1)**2)))) \
    \n \t return x1"
    exec(str2)

    def my_func(T, R, gam, cp_flag):
        if cp_flag:
            y = tf.py_func(speedOfSound_1, [T, R, gam], tf.double, name="speedOfSound")
        else:
            y = tf.py_func(speedOfSound_2, [T, R, gam], tf.double, name="speedOfSound")
        return y

    cp_flag = tf.placeholder(tf.bool, name="caloricallyPerfectGasFlag")
    T = tf.placeholder(tf.double, name="temperature")
    R = tf.placeholder(tf.double, name="gasConstant")
    gam = tf.placeholder(tf.double, name="specificHeatRatio")
    tf_my_func = autograph.to_graph(my_func)
    z = tf_my_func(T, R, gam, cp_flag)

sess = tf.Session(graph=mdl)
yval = sess.run(z, {T:[300.0], R: 286.0, gam: 1.4, cp_flag:True})
```

Figure 11: An example of code to create and execute computational model for speed of sound with conditional flag for calorically perfect gas. Here autograph is used to convert Python conditional to TensorFlow code and Python code snippets are converted to TensorFlow code by tf.py_func.

3.6.3 Next Steps

We will focus on adding the *append* capability to K-CHAIN service and library. Along with the addition of the append capability, we plan to incorporate some initial capability of guided-curation where the ANSWER agent infers that if a model is not trained, then it needs to look for datasets and if a model is data-driven, then it needs to look for physics equation within the page, code, or through Wikidata to refine the computational graph with a physics equation. We will also add TensorFlow Probability models to capture uncertainty in parameters of equations.

3.7 Curation Management

It is important to differentiate between curated knowledge, knowledge that has passed some threshold of assessment of reliability, and tentative knowledge, models that have been extracted from code and/or text and have not yet met the threshold to be added to the curated knowledge. This is one role of the AnswerCurationManager shown in Figure 3. On the knowledge graph side, it is easy to store a model in an OWL ontology and annotate it as pending or curated, or provide it with a level of belief. However, sometimes the curation of knowledge may include exercising of the model and comparing computations with observations, so a pending model may also need to be instantiated in K-CHAIN. The details of how to do that have not yet been explored.

As noted above, the backend ANSWER agent responsible for initiating dialog with the user must be running on as separate thread from the Dialog model processor. Although our design and implementation have not yet resolved exactly what component meets this criterion, it is likely the curation manager. In that case the design and implementation will allow other components such as the AnswerExtractionProcessor to interact with the curation manager to obtain feedback from the user.

As noted in the Introduction, the long-term objective is a curation manager that is aware of gaps in the knowledge base, which include what is learned from TA2 queries that failed to produce the desired results, and models that have weak accuracy or credibility, and is able take direction from the user. From this awareness the curation manager should focus knowledge search over available sources to improve the knowledge store.

4 Supporting Ontology

Before exploring the ontological commitments useful to the capture of scientific knowledge and identified under the ASKE TA1 ANSWER project, we introduce some useful models that have been previously developed for use in SADL and will be used in ASKE.

4.1.1 SADL Constructs Useful to ASKE

4.1.1.1 Unitted Quantities

The SADL grammar contains, with very high precedence so that grouping with parentheses is not needed, a construct called UnitExpression. This allows a numeric value to be followed by a unit designation. If the unit designation contains white space it must be quoted.

The result is that when a number followed by a unit designation occurs in a SADL statement, it is translated into OWL according to the meta-model defined in the SadlImplicitModel (a system-supplied model automatically loaded by all user-defined models in SADL).

```
UnittedQuantity is a class,
      described by ^value with values of type decimal,
      described by unit with values of type string.
```

Note that the "^" before *value* is necessary because *value* is also a keyword in the grammar. This definition generates the OWL meta-model snippet below, which allows both the value and the unit to be captured in an OWL model

```
<owl:Class rdf:ID="UnittedQuantity"/>
<owl:DatatypeProperty rdf:ID="value">
  <rdfs:domain rdf:resource="#UnittedQuantity"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#decimal"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="unit">
  <rdfs:domain rdf:resource="#UnittedQuantity"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>
```

This expressivity is important for capturing scientific knowledge because units are important in the expression of almost all scientific measurements and calculations. Most scientific concepts, e.g., *force*, *mass*, *time*, etc., will be subclasses of *UnittedQuantity* in our knowledge graph. For example, given the domain model

```
PhysicalObject is a class described by weight with values of type Force.
Force is a type of UnittedQuantity.
```

One can then make the statement

```
EmpireStateBuilding is a PhysicalObject with weight 365000 tons.
```

This statement generates the following OWL in the knowledge graph.

```
<tst:PhysicalObject rdf:ID="EmpireStateBuilding">
  <tst:weight>
    <tst:Force>
      <sim:unit rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
      >tons</sim:unit>
      <sim:value rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
      >365000</sim:value>
    </tst:Force>
  </tst:weight>
</tst:PhysicalObject>
```

where *tst* is the prefix for the current model and *sim* the prefix for the *SadlImplicitModel*.

It appears that most equations captured in code are devoid of any unit considerations other than perhaps in comments. Equations in text will often identify the type of an argument, e.g., *mass*, in the surrounding text. The type of the argument implies a set of units that might be associated with any value. Certainly, the values that are assigned to variables as inputs to calculations are usually numerical values without any way to associate units. In the Mach.java code, for example, we have the following declaration of global variables in the calculation of Mach number or velocity.

```
double gama,alt,temp,press,vel ;
double rgas, rho0, rho, a0, lrat, mach ;
```

Similarly, in method definitions within the code, we do not have explicit units.

```
public double CAL_TT (double T, double M, double G, double Q) { … }
```

In other words, the problem of making sure that we have appropriate and consistent units for any calculation is normally left as an exercise for the human applying the knowledge. Given that this is the case, we have chosen the approach that units will be accounted for in the knowledge graph (in the semantic models) but not in the K-CHAIN computational graph. The requestor of a computation in the CG, whether programmatic or human, must make sure that the units of inputs are compatible and assign appropriate units to the output(s) of the computation. UnittedQuantity provides one of the ontological foundations upon which to build this capability and more about our approach will be described in the Rules as Model Interfaces subsection below.

4.1.1.2 Typed Lists

The central idea of a list is that its elements have an unambiguous order. Typed lists were essential to the application of semantic models to software verification and validation[24]. Consider, for example, the modeling of a flight management system. The weight points along the flight path constitute a list of way points. The order matters a great deal to the efficient journey of an airplane from origin to destination.

The vocabulary of OWL, our chosen representation for the knowledge graph, does not include a way of expressing typed lists. The less expressive RDF[25] has a list construct, but since RDFList is used in the serialization of OWL itself, it is not easily used in creating domain models in OWL. Furthermore, RDF lists are not typed, and typing is of significance in error checking.

SADL implements an approach to typed lists that is internally consistent and allows lists to be built monotonically, both of which are missing from other approaches.[26] The SADL grammar uses the *List* keyword, which can be added to any class or data type, to indicate that something is not a single instance of that type but a list of instances of that type. As a simple example, suppose one wished to capture the grades of each student taking a course. One could use a model without lists.

```
CourseGrades is a class,
    described by course with a single value of type Course,
    described by student with a single value of type Person,
```

[24] SADL was extended to create the Verification and Validation tool ASSERT™, see A. Crapo and A. Moitra, "Using OWL Ontologies as a Domain-Specific Language for Capturing Requirements for Formal Analysis and Test Case Generation", in proceedings of the 13th IEEE International Conference on SEMANTIC COMPUTING, Jan 30 - Feb 1, 2019, Newport Beach, California, USA. See also A. Crapo, A. Moitra, C. McMillan and D. Russell, "Requirements Capture and Analysis in ASSERT(TM)," in 2017 IEEE 25th International Requirements Engineering Conference (RE), Lisbon, Portugal, 2017.

[25] Resource Description Framework, see https://www.w3.org/RDF/.

[26] For a quick introduction to typed lists in OWL and SADL, see http://sadl.sourceforge.net/sadl3/SadlConstructs.html#TypedLists. For a detailed discussion, see http://sadl.sourceforge.net/sadl3/Crapo_TypedListsV2.pdf.

```
        described by score with values of type int.
```

This model allows an instance of *CourseGrades* to be the subject of multiple triples with predicate *score* and an *xsd:int* value. But what if we wanted to be able to answer questions like whether a student's scores trended up or down over the duration of the course? Then the following model in which the predicate *score* can have only one value, but that value is a list of values of type *xsd:int,* would be more useful.

```
CourseGrades is a class,
        described by course with a single value of type Course,
        described by student with a single value of type Person,
        described by scores with a single value of type int List.
```

An actual instance of *CourseGrades* can be created with the following statement. As illustrated, the SADL syntax for the actual elements of a list are comma-separated values inside square brackets.

```
ThisExample is a CourseGrades with course Physics101_Fall_2017,
        with student George
        with scores [83, 85, 87, 93].
```

Typed lists will be important to the definition of equations, described in the next section.

4.1.1.3 Equations

Like typed lists, there is no explicit OWL construct for representing equations. Therefore, we build a representation of the important information about equations in an OWL model. The SadlImplicitModel includes a metamodel for two different kinds of equation, one a subclass of the other. Both types include the equation's signature—the name and type of arguments that may be passed in and the types of values returned. They are differentiated by where the details of the computation, the equation body, may be found. In the SADL grammar, an *Equation* has a body which is expressed in the SADL expression grammar. This grammar includes math operations and list operations but does not currently support branching and conditional blocks. The serialization of this body is the value of the *expression*. An *ExternalEquation* in the SADL grammar, on the other hand, does not have a body but is a reference to an external computational entity that matches the given signature. Internally, an *Equation* is uniquely identified by its namespace and name. Only an *ExternalEquation* has an external reference, identified by the value of the property *externalURI*. Optionally, an *ExternalEquation* may also have a *location* URL. Below is the SadlImplicitModel meta-model for equations.

```
^Equation is a class,
        described by expression with a single value of type string.
ExternalEquation is a type of ^Equation,
        described by externalURI with a single value of type anyURI,
        described by location with values of type string.
Argument is a class, described by name with a single value of type string,
        described by ^type with a single value of type anyURI.
arguments describes ^Equation with a single value of type Argument List.
```

```
returnTypes describes ^Equation with a single value of type anyURI List.
```

Note that the "^" before *Equation* and *type* is necessary because those are also keywords in the grammar. Note also that the inputs to an equation are captured as a list of type *Argument*, where each element has a *name* and a *type*. The range of *type* is *xsd:anyURI*[27], a unique identifier of a class or data type. Similarly, the types of the returned values are captured in a list of type *xsd:anyURI*. In this case there is no associated name as there is in the case of an argument. In both cases, arguments and returned values, the order is essential. The typed list construct allows us to capture that order.

A small extension of the SADL equation grammar specifically made for the DARPA ANSWER project allows equations to return multiple values. This allows greater ability to leverage Python, which allows multiple return values, and is motivated by the use of Python by both GE and other ASKE performers.

4.1.1.4 Rules

Rules are another useful construct not directly supported by OWL but implemented by Semantic Web recommendations like SWRL[28]. A SADL rule has a set of conditions in the "given" and/or "if" sections, also known as the rule "body" or "premises". If the conditions are met, the "then" section conclusions, or rule "head", are imposed. In the context of semantic models, it is essential that the rule engine which processes rules is tightly integrated with the OWL reasoner, which infers logical entailments implied by the OWL model and the scenario data.

Many of the patterns normally found in both the premises and conclusions of rules in a semantic model are triple patterns or chains of triple patterns. A simple model and rule was shown in Section 3.2 and is repeated here for convenience.

```
Circle is a class described by radius with values of type decimal,
      described by area with values of type decimal.

 Rule AreaOfCircle: if c is a Circle then area of c is PI*radius of c^2.
```

Note that the SADL grammar of this rule allows patterns to be used in the "then" section which are really part of the premises. The rule is shown below as translated into Jena Rule syntax to illustrate this point. Only the last triple pattern, which assigns the value of variable *?v2* as the *area* of *c*, is an actual conclusion triple pattern. The SADL rule grammar makes the rule much more readable by people.

[AreaOfCircle:

[27] The XSD namespace contains the primitive data types such as int, float, string, etc. See https://www.w3.org/TR/xmlschema11-1/.

[28] SWRL is the acronym for Semantic Web Rule Language, see ???. OWL reasoners sometimes use this OWL-based rule representation. The SADL default reasoner is the Jena RDF rule engine, which has its own representation, Jena Rules. The SADL rule grammar is translated, by a translator associated with the selected reasoner, to the representation utilized by that reasoner. Jena Rules, SWRL, and Prolog are rule languages that have been supported or are supported by SADL.

```
(?c rdf:type http://sadl.org/model.sadl#Circle),
(?c http://sadl.org/model.sadl#radius ?v0),
pow(?v0, 2, ?v1),
product(3.141592653589793, ?v1, ?v2)
-> (?c http://sadl.org/model.sadl#area ?v2)]
```

The use of rules in ANSWER is discussed in detail in Section 4.1.2.1.

4.1.1.5 Queries

SADL supports SPARQL[29] queries as opaque strings. The SADL grammar supports a subset of the expressivity of SPARQL, and provides the advantage of error checking, syntax highlighting, etc. Using the same simple model as in the previous section, the following is a query in SADL syntax.

```
Ask: select c, r, ar
        where circle is a Circle and c has radius r and c has area ar.
```

This query translated to SPARQL is:

```
select ?c ?r ?ar where {?circle <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> <http://sadl.org/model.sadl#Circle> . ?c
<http://sadl.org/model.sadl#radius> ?r . ?c <http://sadl.org/model.sadl#area>
?ar}
```

The various Dialog constructs discussed in Section 3.2 are translated into SPARQL and executed using the SPARQL query engine embedded in SADL (the Jena ARQ query engine) using existing SADL functionality.

4.1.2 Scientific Concept Support

4.1.2.1 Rules as Model Interfaces

In the introduction we mentioned the problem that an essential portion of knowledge surrounding a scientific model often remains implicit—the relationship between the inputs. Besides the speed of sound example, consider Newton's Second Law: *F = ma*, or more appropriately for our domain of hypersonics, since mass of a rocket or airplane is not constant, *F = d(mv)/dt*. That *F* is the net force is often stated in the surrounding text, but that the mass, the velocity, and the net force are all associated with the same physical object is often assumed. Even the standard form of *area = PI * radius^2* doesn't come with an explicit representation that the *radius* and the *area* are of the same circle. Again, it will likely be in a textual description, "the area of a circle is…" This implies (to the human reader) that there is only one circle.

However, note that the rule makes this explicit. In the rule *AreaOfCircle* shown in Section 4.1.1.4, the conditions are that "c is a Circle", the input to the computation is "radius of c", and the conclusion makes explicitly clear that we have computed "area of c", so whatever instance of the *Circle* class is bound to *c*, that instance is the subject of a triple in the inferred semantic model with *area* as property and the computed number as value.

[29] SPARQL is a powerful if somewhat difficult to use graph query language and a W3C recommendation. See ???.

Rules are a nearly perfect interface between the domain ontology, that defines concepts such as *force*, *mass*, *velocity*, *acceleration*, or *circle*, *radius*, and *area*, and the computational graph models that simply take numeric (and possibly other) inputs and compute a numeric (and possibly other) outputs. The rule allows that numeric (and other types of) output to be put back into the correct scientific context.

Another problem that we have raised in the sections above is that of units. Most equations take numeric inputs without units and return numeric results without units. The caller of the computational graph model (in this case the ANSWER backend software) must worry about the compatibility of the units of inputs, and the units of the outputs. Rules also allow this to happen, although a good ontology of units is necessary[30].

An issue that arises is that making units explicit in a rule makes the rule appear complex and "messy". For example, suppose that we have a model of Newton's Second Law in the computational graph, call it nslcm, and an associated interface rule:

```
Rule newtons2ndLaw:
      if o is a PhysicalObject and m is mass of o and
            v is velocity of o and
            fv is nslcm(m,v) and
            pu is unitResolver("*", unit of m, unit of v) and
            fu is unitResolver("/", pu, ^time)
      then there exists (a Force with ^value fv, with unit fu) and
            force of o is the Force.
```

This representation assumes a built-in function *unitResolver* that is able to determine the units of the output of an operation from the units of its inputs. This rule does not address the appropriate unit for *time* which comes from the derivation with respect to *time*. More exploration is needed to solve this issue. Ideally, we will find that the complexity of units can be largely hidden from the user and the rule can be written as:

```
Rule newtons2ndLaw:
      if o is a PhysicalObject and m is mass of o and
            v is velocity of o and
            fv is nslcm(m,v) and
      then there exists (a Force with ^value fv) and
            force of o is the Force.
```

Perhaps unit information can be associated with the equation *nslcm* in the semantic model and applied automatically by rule translation from SADL to the target rule language. The details are, after all, rather rote in nature.

[30] An example of a good ontology that includes units is qudt, see http://qudt.org/.

One might be tempted to try to make a general rule that would find the best computational graph model for a given set of inputs and outputs, e.g., mass and velocity in, force out, and then manage the inputs and outputs. However, that would require inventing the search mechanism to find the best CG model. At least for phase 1, we chose rather to have a rule for each CG model and allow the complex mechanisms of the semantic model inference/rule engine find an appropriate rule and associated equation for the question at hand.

One need that is not necessarily answered by this approach is the selection of a model for a computation when multiple models exist that could be used. The rule engine will not necessarily have a built-in mechanism for favoring one rule over another. One low-cost approach to this is to give each rule a property that reflects the desirability of the interfaced model. This desirability could be computed from some combination of trust in and accuracy of the model, and may be learned over time. Even simple rule systems like Jena can be made to load rules with high desirability first, and, if a solution is not found, then load rules of increasingly lower desirability until a solution is found or no more rule cohorts exist. We have implemented this kind of reasoning in previous versions of SADL, although the current version does not have this capability pending implementation of an improved approach.

4.1.2.2 Derivatives

Differential equations are an indispensable part of scientific modeling in many domains. Derivatives with respect to time show up in the models of almost all non-steady state systems. The speed of sound use case does not use differential equations, so they have not been a main focus of our recent efforts, but they will be an important problem to solve in the next two months.

Derivation can be numerical or symbolic. For example, consider Newtons Second Law, the force *F* is equal to the rate of change of mass *m* times velocity *v* with respect to time.

$$F = d(mv)/dt$$

If we have an equation for *m* and *v* as a function of time, we can do symbolic differentiation. More specifically, the K-Chain computational model can use TensorFlow to accomplish differentiation. In this case we can get a value of *F* for any input value of time *t*, assuming that the context is specific enough to allow us to identify the appropriate equations in terms of *t*. On the other hand, we might have a series of observations of mass and velocity at different points in time (a time-series data set) and one could, for each time step, obtain the change in the product of *m* and *v* over the time interval. If there are *n* observations in our set, we would have *n-1* differentials and so could return *n-1* values of *F*. Providing or identifying this data set to the computational graph, we could get an approximate value of *F* for any time *t* that is within the time window of the data series. Thus, the computational graph can handle both symbolic and numerical differentiation given sufficient information. Our M5 report will illustrate a completed approach to modeling differentiation in the computational graph.

We have looked at several ways of representing derivatives in the semantic model. We are confident that the actual differentiation will takes place in the computational graph, so the

semantic side should only need to know that it happens and how to appropriately handle the units of results. One possible model of derivation is the following:

```
Derivative is a type of ScientificConcept,
      described by derivativeOf with a single value of type ScientificConcept,
      described by withRespectTo with a single value of type class.
```

With that meta-model, scientific concepts related to physical objects might be expressed as follows:

```
Time is a type of UnittedQuantity.
Length is a type of UnittedQuantity.
Position is a type of UnittedQuantity,
      described by x-coordinate with values of type Length,
      described by y-coordinate with values of type Length,
      described by z-coordinate with values of type Length,
      described by ^time with values of type Time.
Mass is a type of UnittedQuantity.

PhysicalObject is a class,
      described by mass with values of type Mass,
      described by position with values of type Position.

Velocity is a type of {UnittedQuantity and Derivative}.
derivativeOf of Position only has values of type Velocity.
withRespectTo of Position always has value Time.
velocity describes PhysicalObject with values of type Velocity.

Acceleration is a type of {UnittedQuantity and Derivative}.
derivativeOf of Velocity only has values of type Acceleration.
withRespectTo of Velocity always has value Time.
acceleration describes PhysicalObject with values of type Acceleration.

Momentum is a type of {UnittedQuantity and Derivative}.
momentum describes Mass with values of type Momentum.
Force is a type of {UnittedQuantity and Derivative}.
derivativeOf of Momentum only has values of type Force.
withRespectTo of Momentum always has value Time.
force describes PhysicalObject with values of type Force.
```

We have not yet completed the integration of the knowledge graph and the computational graph. As we complete this integration we will gather more insight into how best to handle derivatives in the ontology. Since differentiation with respect to time is so prevalent, it may be that this kind of differentiation gets special consideration, just as it does in many texts for human consumption where a dot over a variable indicates that it is differentiated with respect to time.

4.1.2.3 Equation Scripts

While the SADL grammar does allow expressions, it is not nearly expressive enough to represent equations in general. Therefore, we chose to represent equations as *ExternalEquations* with the actual computational instructions captured outside the semantic model. Since Python is a very expressive language and is the language of choice for our computational graph using TensorFlow

and is the language of choice for a number of other ASKE participants, we have chosen Python as the target language for representing equations. However, they are generated, from code, from text, or from user input, an equation serialized as Python is stored as a string using the *expression* property of *Equation* in our meta-model (see Section 4.1.1.3). This equation script is passed to the K-CHAIN computational graph using that REST service's *build* method. Other computational graph implementations, e.g., TA2 providers, can use these Python scripts or translate them into their desired languages.

4.1.2.4 Provenance and Credibility

To create a credible knowledge base for doing scientific modeling, one must capture a great deal of provenance information. For each model, one should be able to answer questions about from what source or sources was the model obtained, how credible and accurate it is believed to be, what data was used to access its accuracy, and for data-driven models, upon what data it was trained. A computed result should be annotated with information about the model that provided the calculation. One might also wish to keep metrics on model usage and even user-feedback. When a question cannot be answered (no model found), this should also be kept as guidance for further knowledge search. We have not yet focused on the meta-model to support provenance and accuracy information, but do not expect it to be particularly challenging.

In ANSWER, when the same knowledge is found in multiple sources its credibility is to be increased. When sources are found which are contradictory, credibility should be decreased. In order to keep track of credibility scores for each nugget of knowledge, we plan to use Beta-Bernoulli distribution from Bayesian statistics. Here Beta distribution captures posterior distribution of the Bernoulli parameter and that parameter denotes the probability of knowledge being credible. The two hyperparameters of Beta distribution completely characterize the continuous distribution and they can be tractably updated sequentially when same knowledge or contradicting knowledge is encountered during curation. This credibility as an attribute of knowledge in the knowledge graph will be added in the ANSWER system.

5 Conclusions

Our ANSWER approach has solidified significantly over the past two months and our capability has increased proportionately. While this report does not have explicit links to all of the components of the system architecture that have been created, they are all available in our github.com ASKE TA repository at https://github.com/GEGlobalResearch/DARPA-ASKE-TA1. The SADL environment, which the Dialog components extend, is available for download at any time from https://github.com/crapo/sadlos2/releases. The Dialog components will soon be available under releases in that repository. As we hope this report reflects, we feel that we are well-positioned to continue to make significant progress in the next two months as we work towards the M5 milestone.

This report is also available on github.com at https://github.com/GEGlobalResearch/DARPA-ASKE-TA1/blob/master/StatusReports/Phase1/DARPA-ASKE-ANSWER-M3-Report.pdf and may be updated there with additional links and material.