GE ASKE TA1 ANSWER M9 Report

Andrew Crapo, Varish Mulwad, Nurali Virani, Narendra Joshi
GE Research
July 31, 2019

This work is supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00111990006. The latest version of this report is available at https://github.com/GEGlobalResearch/DARPA-ASKE-TA1/blob/master/StatusReports/Phase2/M9_ANSWER_Report.pdf.

Contents

Introduction

This report provides an overview of the work done by GE Research during months eight and nine of the DARPA ASKE TA1 ANSWER project. The report is divided into five sections, excluding the Introduction and Conclusions and Next Steps sections. The first four cover the main four thrusts of our research namely, 1) extraction of scientific concepts from code, 2) extraction of scientific concepts from text, 3) capturing scientific knowledge in a K-CHAIN computational graph, and 4) developing a collaborative environment that facilitates keeping the human in the loop. The fifth section addresses the relevance of the research to GE and to DARPA. In some cases, greater detail is provided appendices or in linked documents.

Extraction of Scientific Concepts from Code

Code Extraction Meta-model and Process

The GE TA1 approach to extraction from code is to use an existing parser for code in the given language to generate an abstract syntax tree (AST). This AST becomes an artifact from which to extract structural information about the code. During the beginning of Phase 2, we have refined our meta-model of code structure. This evolved metamodel is shown in Appendix A. Noteworthy improvements include:

- Added structuring of method calls in the code, adding several additional concepts
    - MethodCall class, a subclass of CodeElement
    - MethodCallMapping class with subclasses InputMapping and OutputMapping
- Added ExternalMethod as a subclass of Method to identify the target of a call to a method not in the current code file
- Clarified the meaning of "firstRef"in the Reference class (speaking in a frame-based way) to be the first reference in this CodeBlock
- Refactored rules to infer when a reference to a code variable is an implicit input and when a reference to a code variable is an implicit output. Our chosen Java code repository has lots of top-level class variables that are used in or computed by methods without passing the variable in or explicitly returning values. Identification of implicit inputs and outputs is necessary to be able to extract an interesting code block for conversion to a model for use in the computational graph
- Added "doesComputation" property with domain Method to record observation that a method has computational code in it. Methods that do computation are hypothesized to be of greater potential interest than those that do not.

Note that while we use the AST to characterize the code and identify segments of interest, we do not construct the code proposed for a computational graph model by rewriting the AST. Rather, we use the beginning and ending location of code segments of interest stored in the AST to extract the code directly from the source code file. In the case of implicit inputs or outputs, we will wrap or rewrap the extracted code with a method declaration and/or return statement to make inputs and outputs explicit in the computational graph equivalent to the extracted code. Note that this wrapping may need to happen at the Python rather than the Java level as a code block might have multiple implicit outputs, but Java only supports a single returned value. The JavaModelExtractorJP class has been updated to use the code extraction meta-model to create the OWL model of the code and run inference to infer useful information about code structure and interest portions.

Capturing Assumptions and Constraints

During Phase 1 it became clear to the GE TA1 team that extraction of scientific concepts from code would both be informed by and would need to capture knowledge about the assumptions and constraints that are not explicitly present in the code but that must be satisfied for the knowledge embedded in the code to be correctly applied. In the case of computational

algorithms reduced to code, these assumptions fall into several broad categories. Examples will subsequently be given of each type of assumption.

- Inputs to a computation embodied in code have a context within the modeling domain and these inputs may have assumed or required relationships to each other and/or to the outputs of the computation. We refer to these as semantic constraints. Often semantic constraints of this type are not only not explicitly present in code but may be absent from documentation as well. They are assumed to be known by the user.
- An algorithm embodied in code may be valid over a limited range. The range of inputs for which an output is valid is also referred to as a semantic constraint. These constraints may be absent from the code but are more likely to be present in the documentation or comments as it is less likely that the user will already know them. When the code of interest is not an entire method but some smaller code snippet, these conditions will normally be explicit in the code.
- Numeric inputs to a computation specified in code may be required to be in a particular unit of measure. Similarly, outputs may also be numerical quantities in a particular unit of measure. Providing inputs that are not in the assumed units or incorrectly assigning units to an output is a common source of error.

Understanding and conforming to these kinds of assumptions or constraints is often a task left to the human user of a computer code. For scientific models to be extracted and represented in such a way that an artificial intelligence could make sure that inputs satisfy these assumptions and therefore property utilize a model and understand the meaning of the outputs requires that these assumptions be explicitly captured in a format that facilitates querying, exploration, and inference.

This necessity to capture the assumptions accompanying a piece of code becomes a goal of the extraction process itself. Since code seems to rarely contain this contextual information, we are dependent upon code comments and documentation text to supply relevant information. Observing that, at least in our grounding Java code examples from the NASA hypersonics Web site, the accompanying documentation, although excellent, is mostly silent on contextual assumptions, we expect to rely on the human to help capture assumptions explicitly.

To make assumptions understandable to humans, thereby making their guidance during extraction feasible, we need a relatively transparent representation. All three of the types of assumptions identified above have to do with the inputs and/or the outputs of a computation implemented in a code segment. Code segments in many languages, including our target Java, are often wrapped as a method, in which case the inputs to the computation correspond to the inputs to the method, and the outputs of the computation correspond to the returned values of the method. In many programming languages, method declarations, also known as method signatures, specify the input arguments and their types as well as the type of the returned value or values. These method signatures can also be used to capture the valid inputs and the

expected outputs from models in the computational graph. For example, the following appears in a NASA hypersonics Java code example[1].

```
public double CAL_SOS (double T, double G, double R, double Q) { …}
```

This method computes the speed of sound from the temperature T, the gamma G, the gas constant R, and a thermal constant Q. Each of these values is input as a numerical value of type double and the returned speed of sound is also of type double.

If a human were to write code to call this method, she would need to know what each of the inputs and what the output means in domain terms. This information includes the following.

- This equation is for the speed of sound in a gas, e.g., in atmospheric air
- T is the static temperature of the gas in degrees Rankine
- G is the ratio of the calorically perfect specific heat capacity of the gas at constant pressure to the calorically perfect specific heat capacity of the gas at constant volume
- R is the gas constant of the gas in Imperial units (lb/lbmole)
- Q is a thermal constant, 5500 Rankine
- The value returned is the speed of sound in the gas in ft/sec

As a proposed way of displaying this information to the human or allowing the human to capture this information in a formal, unambiguous manner, we have extended the SADL grammar to support further description of each method argument and of the return value in equation and external equation declarations. We have dubbed these extensions *augmented types*. Using the SADL grammar extensions, the augmented signature for the speed of sound equation above becomes

```
External CAL_SOS(double T (temperature of a Gas {Rankine}),
                 double G (gamma of the Gas),
                 double R (gasConstant of the Gas {"lbm/lbmole"}),
                 double Q (Theta {Rankine}))
             returns double (speedOfSound of the Gas {"ft/sec"})
```

Note that the keyword “External” simply means that the body of this function—the instructions for accomplishing the computation of output from inputs—is not given in this SADL model as SADL expressions, but is defined in another location, e.g., in the Python code created from the Java code and added to the computational graph.

Just as was the case in the above list of what is assumed to be knowledge of the user, the SADL grammar employs the common English usage of indefinite and definite articles to specify

[1] From Java Applet at https://www.grc.nasa.gov/WWW/BGH/sound.html.

unbound and bound variables, respectively. Thus “a Gas” in the augmented type for the first argument T identifies a [new] unbound variable of type Gas. Subsequent references to “the Gas” specify the same variable (as opposed to “another Gas” and “the other Gas”, which would be another, independent variable of type Gas). The simple type (double) along with the augmented type information and the unit information capture all the details present in the preceding list of assumptions.

The SADL grammar for augmented types provides a human-readable representation, but it is not intended to be the representation used for querying, inferencing, and other programmatic uses of the information about the assumptions made by an equation or method. While SADL models can be translated to any desired representation with corresponding expressivity, SADL in most cases is translated to OWL. Therefore, we explore capturing the augmented type information in OWL. To do so we need an OWL metamodel for augmented types.

An exploratory metamodel that supports augmented types is part of the SADL implicit model, a model included automatically as an import to every SADL model. The relevant portions of the SADL implicit model are shown in Appendix B. To better understand translation of semantic constraints to OWL, we will use as an example a simpler equation than the equation whose signature was used above. From the NASA hypersonics website and from the corresponding Java code[2], we find an equation for the temperature of atmospheric air at a given altitude. Note that “some Air” is equivalent to “an Air” but sounds better for a substance rather than an object so SADL supports “some” as an indefinite article. Here the range of “altitude” is of type UnittedQuantity, which is the domain of properties “value” and “unit”. Hence the actual decimal numerical input is the “value of altitude of some Air”. The “^” in front of “value” in the signature is necessary because “value” is a keyword in the grammar and the “^” indicates that this is a user-defined semantic concept, not a keyword. Likewise, the returned decimal value must be placed in an instance of a UnittedQuanitity, which is the “temperature of the Air”. The input value must be in “ft” and the computed return value is in degrees F. All of this is represented in this augmented signature.

```
Equation lowerStratosphereTemperature(decimal alt (^value of altitude of some Air
                                    and alt > 36152 and alt <= 82345 {ft}))
    returns decimal (^value of temperature of the Air {F}) :
          return -70.
```

The relational and unit assumptions stated in this augmented signature can be represented in nine triple patterns (see TriplePattern in Appendix B). If “some Air” is represented with the variable “v0” and a triple pattern is captured as “TriplePattern(<s>, <p>, <o>)”, where <s>, <p>, and <o> are the subject, predicate, and object of the triple pattern respectively, these eight patterns are

1. TriplePattern (v0, rdf:type, Air)
2. TriplePattern (v1, rdf:type, UnittedQuantity)

[2] The documentation of Earth Atmosphere Model is found at https://www.grc.nasa.gov/WWW/BGH/atmos.html. The Java Applet code implementing the computations is found at https://www.grc.nasa.gov/WWW/BGH/atmosi.html and is also found in the code to compute Mach number at https://www.grc.nasa.gov/WWW/K-12/airplane/mach.html.

3. TriplePattern (v0, altitude, v1)
4. TriplePattern (v1, value, alt)
5. TriplePattern(v1, unit, “ft”)
6. TriplePattern (v2, rdf:type, UnittedQuantity)
7. TriplePattern (v0, temperature, v2)
8. TriplePattern (v2, value, v3)
9. TriplePattern (v2, unit, “F”)

where v3 is the returned decimal value computed by the equation.

This example combines the semantic constraints relating the domain-specific meaning of the returned value to the meaning of the input with the functional constraints on the input value—this equation is only applicable if the input “alt” is in the numeric interval (36152, 82345]. These functional constraints are captured as two function patterns (see FunctionPattern in Appendix B). Using the syntax “FunctionPattern(<functionName>, <arg1>, <arg2>, …)”, the two constraints are

1. FunctionPattern(greaterThan, alt, 36152)
2. FunctionPattern(lessThanOrEqual, alt, 82345)

The conversion of augmented type constraints to OWL is a bit more subtle than evident from the TriplePatterns and FunctionPatterns of this example because of the identification and scoping of variables. The variables in this example, both explicit such as “alt” and implicit such as “v0” created from “some Air”, must be unique for each equation. Equation “lowerStratosphereTemperature” is one of three equations for computing atmospheric temperature at different altitudes, each of which use the argument name “alt” but with different constraints. For a complete discussion of augmented types and their translation to OWL using the meta-model of Appendix B, see “Using Graph Patterns to Augment Integration of Models and Data into a Semantic Framework”[3].

The W3C Shapes Constraint Language (SHACL)[4] provides an alternate mechanism for capturing semantic constraints which may be applicable to constraints of the type described above but in a W3C-compliant manner. With encouragement and support from Clare Paul, Materials and Ontology Engineer at the Air Force Research Lab, Wright-Patterson AFB, Dayton, Ohio, we are beginning to explore the expressivity of SHACL in this regard. The example below has been supplied by Clare based on the semantic constraints for lower stratosphere temperature as a function of altitude. Using SHACL requires that one create SHACL NodeShape instances that have as targetClass the class to be constrained. For this example, one might create the class Altitude, which is the range of the “altitude” property and is the targetClass of the AltitudeShape instance of NodeShape. Note that this example uses the more sophisticated QUDT ontology rather than our simple UnittedQuantity. The QUDT equivalent of “value” is “numericValue” and the equivalent of “unit” is “hasUnit”. The AltitudeShape constrains Altitude

[3] https://github.com/GEGlobalResearch/DARPA-ASKE-TA1/blob/master/SupportingFiles/UsingGraphPatternToAugmentIntegrationOfModelsIntoSemanticFrameworkV2.pdf
[4] https://www.w3.org/TR/shacl/

to have only one value of numericValue of type Literal, and to have only one value of hasUnit of type LengthUnit. Expressed in turtle syntax, this AltitudeShape instance is the following.

```
tbd-shp-aat:AltitudeShape
  rdf:type sh:NodeShape ;
  sh:property [
      rdf:type sh:PropertyShape ;
      sh:path qudt:numericValue ;
      sh:maxCount 1 ;
      sh:minCount 1 ;
      sh:nodeKind sh:Literal ;
    ] ;
  sh:property [
      sh:path qudt:hasUnit ;
      sh:class qudt-v1:LengthUnit ;
      sh:maxCount 1 ;
      sh:minCount 1 ;
    ] ;
  sh:severity sh:Warning ;
  sh:targetClass tbd-sch-aat:Altitude ;
.
```

Note that the information in the AltitudeShape class could also have been represented as property restrictions (qualified cardinality and all-values-from restrictions).

One might further create the NodeShape instance LowerStratosphereShape with targetClass Altitude to express the condition that the numericValue of Altitude be in the interval (36152,82345]. Expressed in turtle syntax, this NodeShape instance is the following.

```
tbd-shp-aat:LowerStratosphereShape
  rdf:type sh:NodeShape ;
  sh:property [
      rdf:type sh:PropertyShape ;
      sh:path qudt:numericValue ;
      sh:maxInclusive "82345"^^xsd:double ;
      sh:minExclusive "36152"^^xsd:double ;
      sh:name "located in lower stratosphere atmospheric region" ;
      sh:nodeKind sh:Literal ;
    ] ;
  sh:severity sh:Info ;
  sh:targetClass tbd-sch-aat:Altitude ;
.
```

The semantic constraint on the numericValue of Altitude can now be captured explicitly in the SPARQLRule construct of SHACL by setting the sh:condition to the instance LowerStratosphereShape. The SPARQLRule explicitly relates the computed temperature and the input altitude to the variable “$air”. The sh:condition with object LowerStratosphereShape, which in turn has targetClass Altitude, identifies the pre-bound variable “$this” as the instance of Altitude for which the rule is applied. Once again in turtle syntax, the SPARQLRule is as follows.

```
tbd-shp-aat:determine_air_temperature_in_lowerstratosphere
  rdf:type sh:SPARQLRule ;
```

```
  sh:condition tbd-shp-aat:LowerStratosphereShape ;
  sh:construct """CONSTRUCT {
      $air tbd-sch-aat:hasTemperature _:temperature .
    _:temperature rdf:type tbd-sch-aat:Temperature .
    _:temperature qudt:numericValue \"-70.0\"^^xsd:double .
    _:temperature qudt:hasUnit unit-v1:DegreeFahrenheit .
    $air rdfs:comment \"lower stratosphere\"^^xsd:string .
}
WHERE {
      $air  tbd-sch-aat:hasAltitude $this .
}     """ ;
  sh:prefixes <https://example.org/ge/shapes/airaltitudetemperature> ;
.
```

In this case the SPARQLRule's sh:construct property value is a SPARQL CONSTRUCT statement that computes and assigns the numericValue and hasUnit properties of a new instance of Temperature.

While we have captured the semantic constraints in SHACL, we have not, so far, accomplished all our objectives. We have not:

1. indicated that the numerical interval limiting the allowed for values of the numericValue of Altitude for LowerStratosphere is in feet,
2. captured the semantic constraints in a general representation that is easily queryable and amenable to inference since the value of the sh:construct property of the SPARQLRule is a string rather than structured triples. That means that we would have to parse the CONSTRUCT statement string and extract triple patterns from the parse tree and insert them into a triple store before we would be able to query or reason about the triple patterns in the statement.

We expect to soon finish this exploration of SHACL. If we are successful in fully representing semantic constraints, both TriplePatterns and FunctionPatterns, in a way that allows us to query over a knowledge graph and draw inferences, then we will lean toward using the SHACL representation as we would prefer to be standards-based.

Extraction of Scientific Concepts from Text

In Phase 2, the focus of the text extraction module so far has been on extending and improving the capability to extract the context associated with equations. This context can be found in the text surrounding the equation and comments that appear in related codebases. As part of this process, the arguments and return variable are mapped to scientific concepts that appear in the text (which in turn are linked to entities from existing knowledge graphs such as Wikidata). The process would also include the identification of units (e.g. mph, kg, etc.) that appear in text and leveraging them to identify relevant associated concepts. This context provides two benefits: i) better semantics to ground the equations; the semantics in turn is useful for several other tasks such as model comparison, model combination etc. and ii) guiding model extraction from code.

In the spirit of reuse and collaboration, we evaluated the technology stacks of other participants in the DARPA ASKE program to identify modules that would provide the same or

similar functionality. We found the "Text Reading" module from AutoMATES (University of Arizona) most promising. AutoMATES's text reading module in turn relies on grobid-quantities, an open-source package that identifies quantities and their units. We evaluated grobid-quantities and found that it doesn't provide the functionality we wish to implement. grobid-quantities was unable to identify quantities and units from code comments. grobid-quantities works best when the quantity appears with an associated value in text (e.g. "A solution of 1.18 g of the compound was obtained"), which is not always the case with comments. Either the units or quantity is mentioned, with the comments themselves being short phrases, instead of complete sentences. We are currently in the process of setting up and evaluating the AutoMATES's text reading module for identifying the relationship/link between equation variables and their associated concept (e.g. *m* represents *mass* in equation xyz).

Additionally, we have made improvements to the text extraction services to make it easier for others to install, deploy and test. Key properties and their values have been moved to a configuration file and a script to start all relevant services has been made available. The text extraction services depend on a subset of Wikidata loaded into elastic search, which the users are expected to stand up on their own. Going forward, we will automate it, by making a docker image available for the relevant elastic search services. These changes can be found in the "development" branch of the ANSWER system codebase on GitHub: https://github.com/GEGlobalResearch/DARPA-ASKE-TA1/tree/development.

Making Knowledge Useful through K-CHAIN

Knowledge-Consistent Hybrid AI Networks are used as a means to get executable computational instantiation of curated knowledge. Here we can allow modifications at the semantic level through the Curation Manager and update computational graphs representing the computational model. In Phase 1 we focused on automatically building and evaluating these computational graphs and created the K-CHAIN services of build and eval to demonstrate this capability and enable Curation Manager to use it. In Phase 2, our focus has been to enable a service to append computational models to existing computational graphs as and when these new models have been extracted from code or text. This capability will enable the computational graph to evolve with updated knowledge over time. An example of evolutions that we are considering is given in Figure 1. There are a few key challenges that we are facing in appending and evolving computational graphs. In this section, we will discuss these challenges, describe our progress so far, give examples of outputs, and outline next steps.

When a new computational model needs to be added to an existing computational graph (Figure 2(a)), if we rely on TensorFlow itself, then a disjoint computational subgraph is created (Figure 2(b)). In other words, the computational graph does not realize that a some input or output variables of the new model already exists in current the computational graph even though those nodes have the same name. To address this issue, we created a dictionary of nodes that records all variable or placeholder nodes added to the computational graph along with the graph node reference that TensorFlow uses to uniquely represent the node. When a new computational model is to be appended, we check the variables in the new model against dictionary keys. If there is a match, then we use the graph node reference for that variable

node, so that a new variable is not created again by TensorFlow. The resulting behavior for Option 1 shown in Figure 1 is demonstrated with the example in Figure 2(c). As a next step, this approach of matching nodes to reuse existing graph nodes will be extended to declare a match only if name and other properties, such as units and/or semantic type information also matches. This will enable us to capture and obey semantic constraints as the model evolves over time.

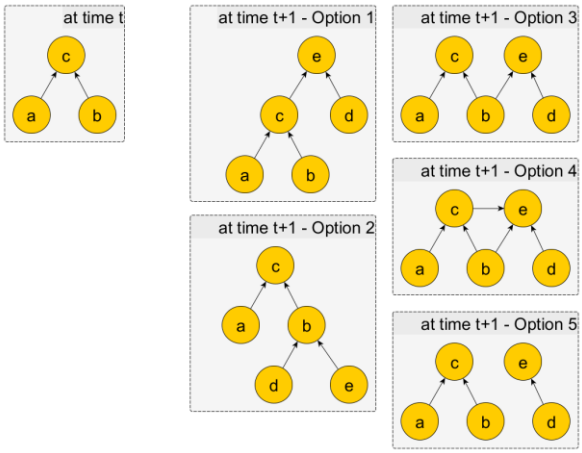

Figure 1: Computational graph append options as new computational model fragments are extracted from text or code. *The computational graph at time t includes computational models that have been curated until time t. As a new knowledge fragment is extracted and given to K-CHAIN for curation, the computational graph evolves. The evolution depends on whether the inputs and outputs are already a part of existing models. See Options 1-5 for different cases that we are considering.*

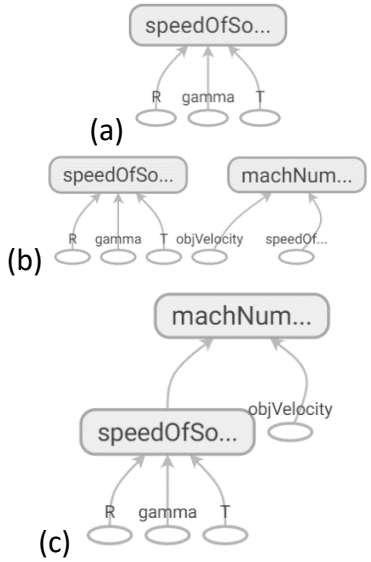

Figure 2: Example of computational graph evolution. *(a) Initial computational graph at time t created from speed of sound model extracted from text, (b) Trying to append the Mach number computation gives two disjoint graphs at time t+1, (c) append operation with node dictionary gives the correct model at time t+1 (Option 1 of Figure 1).*

Inference with these models in our K-CHAIN library can use default values or report missing variables that are required to successfully execute the computation. Below are a few cases to demonstrate the capability as well as highlight a few challenges.

Case 1: If we want to compute MachNumber, given temperature T = 300K and object velocity objVelocity = 1000m/s, then we get the following output:

```
output: 2.885309
default values used: {'R': 286.0, 'gamma': 1.4}
```

Here default values for R and gamma for air medium was provided during mode build step.

Case 2: If we want to compute MachNumber, given speed of sound speedOfSound = 350m/s and object velocity objVelocity = 1000m/s, then we get the following output:

```
output: 2.857143
default values used: {}
```

Case 3: If we want to compute MachNumber, given speed of sound speedOfSound = 350m/s, then we get the following output:

```
Please provide value for: objVelocity
output: None
default values used: {}
```

The incomplete set of inputs led to unsuccessful inference. However, the information "`Please provide value for …`" is helpful and can be passed by the ANSWER agent back to the user, so that the human can provide the key missing value.

Case 4: If we want to compute MachNumber, given object velocity objVelocity = 1000m/s, then we get the following output:

```
Please provide value for: T
output: None
default values used: {}
```

The incomplete set of inputs led to unsuccessful inference. In contrast to Case 3, here the missing variable is identified as temperature T in lieu of speedOfSound, as temperature is an independent variable in the model that does not have a default value. If this information is provided to the user through the ANSWER agent, then the user will not realize that inference can also be successful accomplished by providing a value for speedOfSound. Addressing this issue is currently a low priority but will be under consideration.

Case 1 mentions that default values were provided during the model build step. However, the new knowledge fragment that has been extracted might be a default value for an existing variable in the graph, for example, gamma = 1.4 for air. This case is considered as a special case of the append functionality where values for optional properties, such as default value, units, semantic types, etc. needs to be added instead of a new computation. In the append service, if the model code from text2python or java2python is empty, then this special case is activated and the dictionary of default values and node the dictionary with node properties can be updated.

When we consider computational graph evolution, Options 1, 3, 4, and 5 from Figure 1 can be handled with the solution approach discussed above. However, Option 2 has unique challenges. Option 2 considers the case where output of a new knowledge fragment is input to an existing model. This case is common with knowledge extraction from multiple sources. For example, the equation for Mach number with speed of sound as an input may be extracted first from code or text and then the equation for speed of sound may be found by the Curation Manager later. The challenge arises from the fact that a reference to the computation output is created for

each computation and cannot be assigned directly as we could for model inputs. The current solution to this problem is suboptimal and will be further explored in the next month. This solution identifies that an output of the new model already exists as an input in a previous model and it rebuilds the previous model with the reference for the input node the same as the new model output. The results are shown in Figure 3.

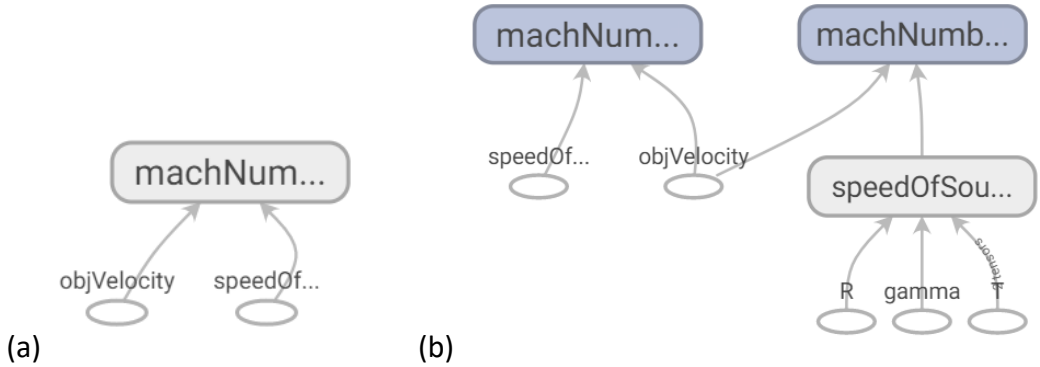

Figure 3: Example model evolution under scenario of Option 2 from Figure 1.*(a) Mach number model is extracted first, (b) The appended model where the previous model is rebuilt with an updated reference for the input variable to be an output of the new model fragment for speed of sound.*

Note that the previous model still exists as a part of the graph, however, the inference examples shown in Cases 1 to Case 4 are still applicable for the resulting graph. Deletion of nodes is not supported in native TensorFlow, so we will explore the GraphSurgeon package from nVidia to modify graphs. We will also consider rebuilding the whole graph with updated and consistent references.

Keeping the Human in the Loop through Collaboration

Our human-in-the-loop efforts during the first part of Phase 2 have focused on two objectives. One is the SADL grammar of augmented types, discussed in an earlier section of this report, which facilitates human-computer collaboration around the assumptions and constraints that accompany pieces of computational scientific knowledge. Representing this knowledge formally, so that it can be queried and reasoned over, and succinctly in a human-friendly format so that the human collaborator can understand quickly and easily the assumptions and constraints of a scientific model, or help to capture them during the extraction process, is an important element of collaboration and trust. Each semantic concept referenced in the augmented type constraints is hyperlinked to the concept definition in this or an imported ontology. An equation declaration is hyperlinked to all locations where the equation is referenced. This hyperlinking across semantic models makes it easy for human users to explore the meaning of concepts used in assumptions, and equation usage. The augmented types grammar, as part of the SADL language, is inherited by the Dialog grammar. Thus, all the linking that occurs in semantic models also occurs in a Dialog conversation window.

The second focus has been on the environment in which the Dialog grammar is used for mixed-initiative conversation between the human and the ANSWER and KApEESH systems. During Phase 1, a prototype, Eclipse-based environment was created. At the start of Phase 2 we took a step back and, informed by our experience to that point, tried to create a requirements document for this Dialog environment. The primary question was whether to improve upon the existing Eclipse-based environment or to move to a Web browser-based Dialog environment, and if so, at what cost.

The Dialog collaboration is envisioned to occur in a simple "chat window" which can be implemented in a text editor. In specific, an Xtext editor is envisioned so that Xtext functionality can be used 1) to constrain the dialog content to that supported by the Dialog grammar to avoid ambiguity (with the controlled grammar errors are detected, underlined, and explained with markers in the margin), 2) to enable content assist to provide possible statements or continuations of partially completed statements, and 3) to enable hyperlinking of all domain concepts appearing in the dialog to their definitions and to additional references in imported models (imported directly or indirectly) and to additional references in this or other dialog files. From these objectives we derived three requirements to be met by the target conversational system.

1. It must be possible to identify which party has contributed each statement in the dialog.
2. It must be possible to have multiple topics and to differentiate between topics.
3. It must be possible to navigate programmatically between statements in a dialog topic in order to obtain the full sequence of topic statements for analysis by the AI in determining whether to/how to respond in the course of a conversation.

A more thorough discussion of these requirements and possible ways to meet them may be found in the full requirements document[5].

As much as we would like to move to a browser-based environment so that it is more accessible and can more easily be integrated with existing browser-based text extraction user-interfaces, the effort in doing so at this time seems to be incompatible with the limited resources and short timeline of the ASKE project. Therefore, we intend to improve upon the Eclipse-based environment, guided by the requirements document and with the objective of achieving as much of the desired functionality as possible during the remainder of Phase 2.

Technology Relevance to GE and Value to DARPA

Department of Defense procurements that provide a differentiated edge are based on developing new technology, maturing this technology through the Engineering and Maturation Development programs and then providing products through the procurement process. At the very highest level, continued differentiation of capabilities will require evolution of technologies faster than the world can copy and in the world of ever-increasing complexity of systems this will get harder to do.

[5] https://github.com/GEGlobalResearch/DARPA-ASKE-TA1/blob/master/SupportingFiles/A%20Mixed%20Initiative%20Dialog%20Framework.pdf

The time from conception to fielding a new capability such as the F135 aircraft has exceeded decades. Over the last 50 years, each of the DOD contractors has developed a great knowledge base of what works and what does not, in their field of expertise, and why. This knowledge base is really the differentiator in the new world with the advent of near-peers. This deep experience is perhaps the only thing the near-peer does not possess. The ASKE program is one element in pulling DOD-relevant value out of a large body of knowledge held by these contractors. While historians, writers and textbook authors have documented and pass on to the next generation an enormous body of literature, there is still a lot of finer details on lessons learn that are not passed on as these are retained as corporate memory in the form of internal documents and code. The next generation of engineers and scientists need access to this knowledge so that they do not repeat the mistakes of the past. Many of these lessons are retained in design notebooks, reports and presentations held very closely as intellectual property. Ingestion of this knowledge and making it available to the next generation of designers without burdening them with sifting through an enormous store of relatively raw data will enable these designers and engineers to access this corporate memory and develop designs benefiting from valuable lessons learned. Figure 1 below depicts this value.

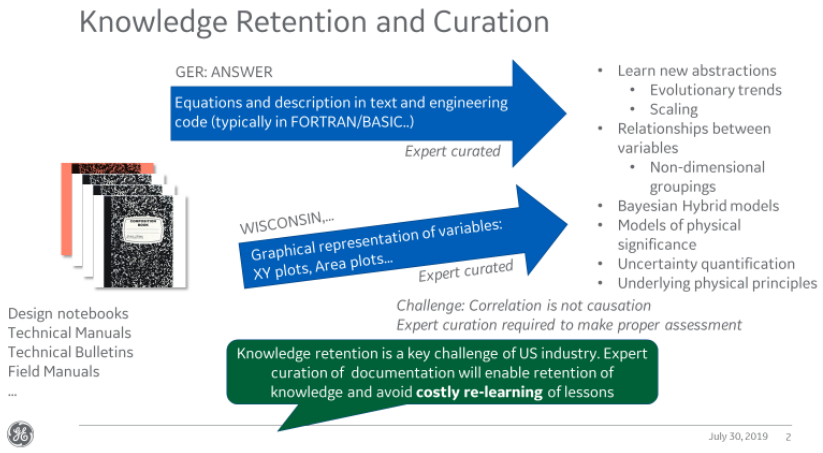

Figure 4: Value of knowlede retention and curation

For example, designing a new jet engine involves setting requirements on thrust, operating envelope, and other boundary conditions. A preliminary design is developed and then analytically and experimentally tested to see if it meets the requirements. If it does not, the components or the engine are redesigned and retested. These design and test cycles may eventually produce a design that meets all requirements but at the cost of significant time and effort. The capabilities envisioned by the ASKE program put expert-curated features from all prior designs at the fingertips of the designer. With the ability to quickly look at prior experience, including test results, manufacturing and field experience data, the designer can eliminate a number of design redo cycles, saving the engineering and manufacturing program time and expense and/or allowing more design options to be explored, potentially leading to better designs.

Even greater value will be created from the ASKE program when it is used as a front end to a more comprehensive tool suite whose roadmap includes solving the inverse design problem. Ingestion of design analysis, manufacturing and test data from multiple products, such as various jet engines, can provide ASKE with sufficient information to understand, for example, scaling with non-dimensionalized numbers such as the Mach number used in the ANSWER program's hypersonics domain.

Conclusion and Next Steps

During this reporting period for months eight and nine of the ANSWER project, we have pushed forward on the solution to some problems and reevaluated our approach to other problems where there is more uncertainty about the best solution. The latter has included looking at what other ASKE providers are doing in text extraction to see what should be leveraged as an alternative to our Phase 1 approach. It has also focused on establishing requirements for the collaboration interface and considering the option of moving to a browser-based interface. The challenges with appending models to the TensorFlow computational graph has been an area of concern where we wish to identify more completely the risks and how to retire them.

On the other hand, we are convinced that capturing the semantic constraints around the inputs and outputs of computational models is a critical need that will not go away. Although there may be room for improvement, our augmented types controlled-English grammar makes these semantic constraints explicit in a human readable format. How best to translate these constraints into models amenable to query and inference is an ongoing topic of research. Similarly, we are comfortable with our code extraction methodology of 1) using existing parsers to create abstract syntax trees (ASTs) from code, 2) transforming these ASTs into semantic models of the code over which we can query and reason to identify portions of the code that are candidates for extraction, 3) performing curation in collaboration with humans in the loop, 4) extracting the relevant code segments from the original code and translating them to the target language, and 5) adding the curated, extracted, translated models to the computational graph.

Finally, we have business commitment to the importance of this activity and have initiated an internal project to develop closely related capabilities that are not within the scope of the ASKE project.

Appendix A: Code Extraction Meta-Model

Note: the latest version of this model can be found at https://github.com/GEGlobalResearch/DARPA-ASKE-TA1/blob/master/Ontology/M5/ExtractedModels/CodeExtractionModel.sadl.

```
uri "http://sadl.org/CodeExtractionModel.sadl" alias cem.

// This is the code extraction meta-model
CodeElement is a class described by beginsAt with a single value of type int,
        described by endsAt with a single value of type int.

CodeBlock is a type of CodeElement,
        described by serialization with a single value of type string,
        described by comment with values of type Comment,
        described by containedIn with values of type CodeBlock.

{Class, Method, ConditionalBlock, LoopBlock} are types of CodeBlock.

cmArguments describes Method with a single value of type CodeVariable List.
cmReturnTypes describes Method with a single value of type string List.
cmSemanticReturnTypes describes Method with a single value of type string List.
doesComputation describes Method with a single value of type boolean.
calls describes Method with values of type MethodCall.
ExternalMethod is a type of Method.

// The reference to a CodeVariable can be its definition (Defined),
//      an assignment or reassignment (Reassigned), or just a reference
//      in the right-hand side of an assignment or a conditional (Used)
Usage is a class, must be one of {Defined, Used, Reassigned}.

Reference  is a type of CodeElement
        described by firstRef (note "first reference in this CodeBlock")
                with a single value of type boolean
        described by codeBlock with a single value of type CodeBlock
        described by usage with values of type Usage
        described by cem:input (note "CodeVariable is an input to codeBlock CodeBlock")
                with a single value of type boolean
        described by output (note "CodeVariable is an output of codeBlock CodeBlock")
                with a single value of type boolean
        described by isImplicit (note "the input or output of this reference is implicit
(inferred), not explicit")
                with a single value of type boolean
        described by setterArgument (note "is this variable input to a setter?") with a single
value of type boolean
        described by comment with values of type Comment.

MethodCall is a type of CodeElement
        described by codeBlock with a single value of type CodeBlock
        described by inputMapping with values of type InputMapping,
        described by returnedMapping with values of type OutputMapping.
MethodCallMapping is a class,
        described by callingVariable with a single value of type CodeVariable,
        described by calledVariable with a single value of type CodeVariable.
{InputMapping, OutputMapping} are types of MethodCallMapping.

Comment (note "CodeBlock and Reference can have a Comment") is a type of CodeElement
        described by commentContent with a single value of type string.
```

```
// what about Constant also? Note something maybe an input and then gets reassigned
// Constant could be defined in terms of being set by equations that only involve Constants
// Constants could also relate variables used in different equations as being same
CodeVariable  is a type of CodeElement,
       described by varName with a single value of type string,
       described by varType with a single value of type string,
       described by semanticVarType with a single value of type string,
       described by quantityKind (note "this should be qudt:QuantityKind") with a single value
of type ScientificConcept,
       described by reference with values of type Reference.

{ClassField, MethodArgument, MethodVariable} are types of CodeVariable.

//External findFirstLocation (CodeVariable cv) returns int: "http://ToBeImplemented".

Rule Transitive
if inst is a cls and
   cls is a type of CodeVariable
then inst is a CodeVariable.

Rule Transitive2
if inst is a cls and
   cls is a type of CodeBlock
then inst is a CodeBlock.

Rule FindFirstRef
if c is a CodeVariable and
   ref is reference of c and
   ref has codeBlock cb and
   l is beginsAt of ref and
   minLoc = min(c, reference, r, r, codeBlock, cb, r, beginsAt) and
   l = minLoc
then firstRef of ref is true
//     and print(c, " at ", minLoc, " is first reference.")
.

Rule ImplicitInput
if cb is a CodeBlock and
   ref has codeBlock cb and
   ref has firstRef true and
   ref has usage Used
   and cv has reference ref
   and ref has beginsAt loc
then input of ref is true and isImplicit of ref is true
//     and print(cb, cv, loc, " implicit input")
.

Rule ImplicitOutput
if cb is a CodeBlock and
   ref has codeBlock cb and
   ref has firstRef true and
   ref has usage Reassigned
   and cv has reference ref
   and noValue(cv, reference, ref2, ref2, codeBlock, cb, ref2, usage, Defined)
   and ref has beginsAt loc
then output of ref is true and isImplicit of ref is true
//     and print(cb, cv, loc, " implicit output")
.
```

Appendix B: Augmented Types Meta-Model from SADL Implicit Model

Note: this is only a portion of the SadlImplicitModel.sadl file used in ANSWER ontology development. The complete model can be found at https://github.com/GEGlobalResearch/DARPA-ASKE-TA1/blob/master/Ontology/M5/ImplicitModel/SadlImplicitModel.sadl.

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

DataDescriptor is a class, described by localDescriptorName with a single value of type
string,
        described by dataType (note "the simple data type, e.g., float") with a single value of
type anyURI,
        described by augmentedType (note "ties the DataDescriptor to the semantic domain
model") with values of type AugmentedType.
dataType of DataDescriptor has at most 1 value.
descriptorVariable of DataDescriptor has at most 1 value.

Language is a class, must be one of {Java, Python, Text, OtherLanguage}.
Script is a class, described by language with a single value of type Language,
        described by script with a single value of type string.
^Equation is a class,
        described by expression with values of type Script.
arguments describes ^Equation with a single value of type DataDescriptor List.
returnTypes describes ^Equation with a single value of type DataDescriptor List.

ExternalEquation is a type of ^Equation,
        described by externalURI with a single value of type anyURI,
        described by externalURL with values of type string.

AugmentedType is a class.
augTypeUnits describes AugmentedType with a single value of type string List.
SemanticType (note "allows direct specification of the semantic type of an argument") is a
type of AugmentedType,
        described by semType with a single value of type class.
GraphPattern is a class.
{TriplePattern, FunctionPattern} are types of GraphPattern.
gpSubject describes TriplePattern.
gpPredicate describes TriplePattern.
gpObject describes TriplePattern.
builtin describes FunctionPattern with a single value of type ^Equation.
GPAtom is a class.
{GPVariable, GPLiteralValue, GPResource} are types of GPAtom.
gpVariableName describes GPVariable with a single value of type string.
gpLiteralValue describes GPLiteralValue with values of type data.
argValues (note "values of arguments to the built-in") describes FunctionPattern with a single
value of type GPAtom List.

SemanticConstraint (note "used to identify necessary patterns in semantic domain terms") is a
type of AugmentedType,
        described by constraints with a single value of type GraphPattern List.
ThisArgument (note "allows reference to self within an Argument's constraints") is a
DataDescriptor.
```

```
anyDataType (note "union of all relevant data types") is a type of {decimal or boolean or
string or date or dateTime or anyURI}.
DataTableRow is a class,
       described by rowValues with a single value of type anyDataType List.
DataTable is a class,
       described by columnDescriptors with a single value of type DataDescriptor List,
       described by dataContent with a single value of type DataTableRow List,
       described by dataLocation with a single value of type anyURI.

^Rule is a class.
NamedQuery is a class.


/****** The following content comes from the Dialog implicit model fragment provider ******/
// these properties used by TA2
model describes DataTable with values of type ^Equation.
^data describes ^Equation with values of type DataTable.
```