Using Graph Patterns to Augment Integration of Models and Data into a Semantic Framework

A. W. Crapo, GE Global Research, crapo@ge.com

Last revised July 30, 2019

Introduction

A great deal of scientific knowledge may be found in books, journal articles, conference reports, and perhaps most explicitly in computer code encapsulating scientific calculations. To some extent, all of these repositories have, as their target audience, a human with some level of skill in the domain. For documents, the skill is required to correctly interpret the meaning of the text. For computer programs, the skill is in providing correct inputs—inputs that are consistent in ways that may be left implicit to some degree, or that conform with documentation that must be read by the user. For data, the relationship of each data element to the domain and the relationship of data elements to each other is at best explained in documentation not present in the data and at worst is simply assumed to be decipherable by the data consumer.

This becomes a problem when such computational models and data are to be integrated into the semantic framework of an artificial intelligence that is then expected to make use of the models, chain them together so that the outputs of one model are the inputs to another model, use data appropriately as model inputs or for model validation, etc. In other words, all of the implicit knowledge that a subject matter expert uses to properly invoke the computational models must now be explicitly captured in the knowledge base of the artificial intelligence.

An Example

As a concrete example, consider a set of equations that capture the science of flight. These equations are taken from the NASA Glenn Research Center Web site.[1] For example, the Mach number is the ratio of the speed of an object in flight to the speed of sound. The referenced Web page illustration of the equation is shown in Figure 1.

[1] See https://www.grc.nasa.gov/www/k-12/airplane/mach.html and linked pages.

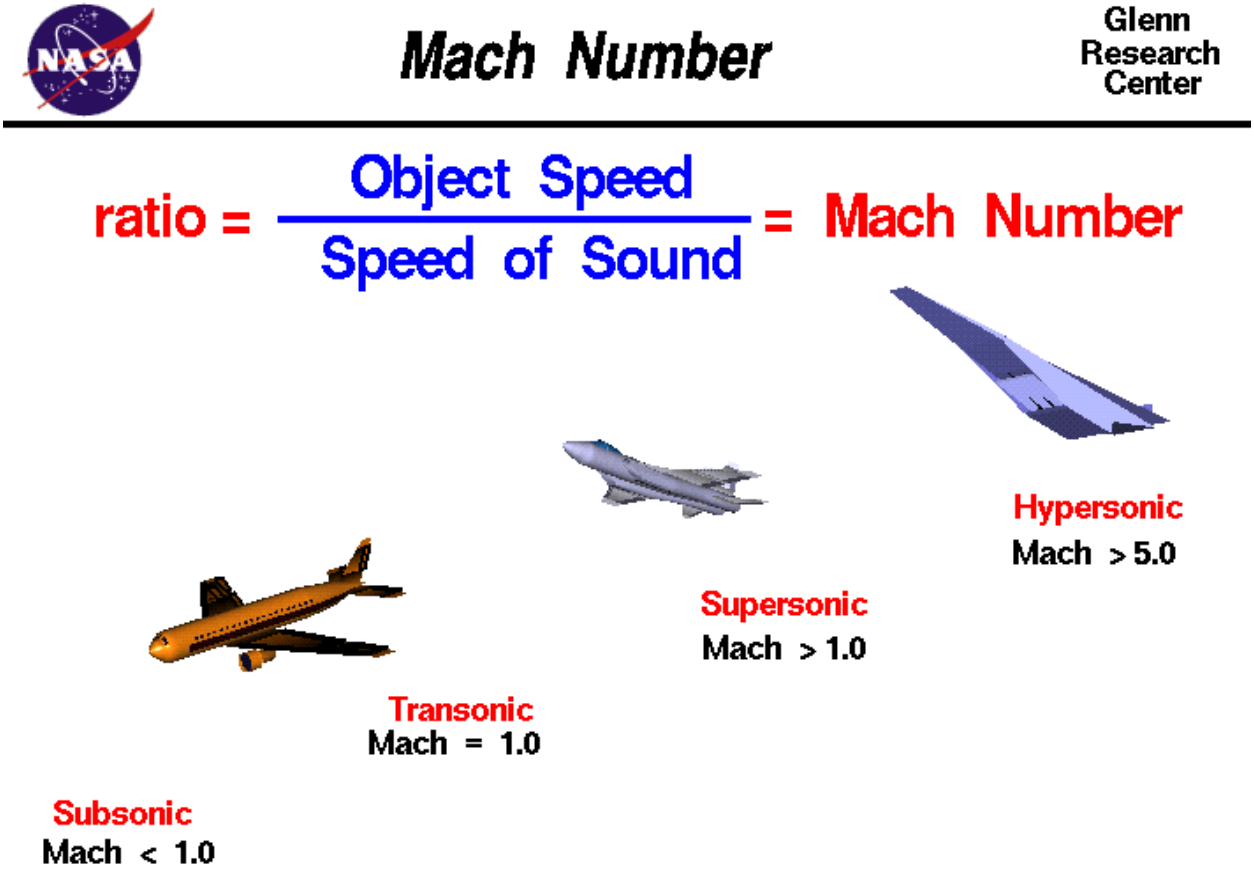

Figure 1: Mach Number from NASA Glenn Research Center, https://www.grc.nasa.gov/www/k-12/airplane/Images/mach.gif

Either a careful reading of the text around this equation, or a pre-existing mental model on the part of the reader, is required to understand clearly that the speed of sound in the denominator must be the speed of sound in the air through which the object is passing. Similarly, the fact that the speeds in the numerator and the denominator must be in the same units is not explicitly specified in the equation nor is it explicitly stated in the text.

The equation for the speed of sound is given on the linked Speed of Sound page[2], which is shown in Figure 2.

[2] See https://www.grc.nasa.gov/www/k-12/airplane/sound.html.

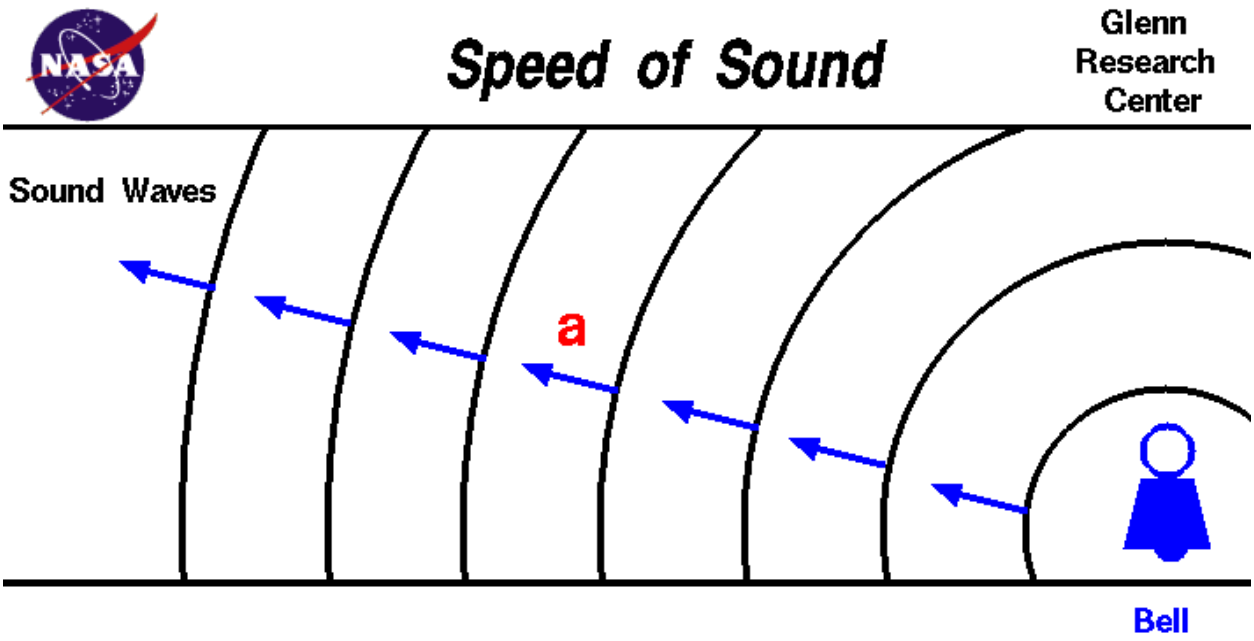

Figure 2: Speed of Sound illustration, from NASA Glenn Research Center Web site

The ratio of specific heats, γ in the equation above, may also be computed. If the speed of the object is large, then the ratio of specific heats is given by:

gam = 1 + (gamma - 1) / (1 + (gamma-1) * [(theta/T)^2 * e^(theta/T) /(e^(theta/T) -1)^2])

where

gamma (γ) is the ratio of specific heats for a calorically perfect gas, 1.4 for air,

theta is a thermal constant, 5,500 degrees Rankine or 3056 degrees Kelvin

T is the static temperature.

However, static temperature can be computed from altitude for a "standard Earth day" as follows.[3] If the altitude (*h*) is less than 36,152 feet (troposphere), then the temperature in degrees F is given by:

T = 59 - .00356 *h*

If the altitude *h* is between 36,152 and 82,345 feet, the temperature is:

[3] Earth Atmosphere Model, see https://www.grc.nasa.gov/www/k-12/airplane/atmos.html.

T = -70

If the altitude *h* is greater than 82,345 feet, then the temperature is:

T = -205.05 + .00164 *h*

To properly use these equations, an artificial intelligence must know how the inputs and outputs of each equation are related in domain terms and must have explicit knowledge of the units of inputs and outputs of each equation and ensure that they are aligned when using multiple equations. This task is handled for a human user on the NASA Glenn Research Center by wrapping the computation models in a Java Applet, which requires that the user choose the units. The required units implied by this choice are shown in the applet interface, as illustrated in Figure 2. The applet code takes care of the task of linking the various computational models together in a consistent manner. Appropriate assumptions are baked into the implementation.

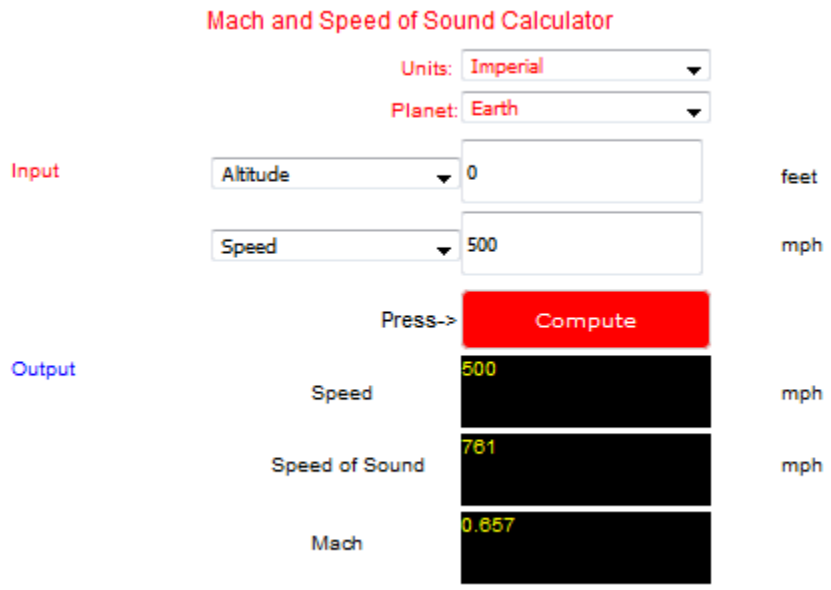

Figure 3: Applet interface to Mach number calculation, from NASA Glenn Research Center page

Graph Models of the Domain

There are multiple formal modeling methodologies that one might use to capture knowledge of a domain. Predicate logic is a very powerful formalism that is often used. If predicate logic is restricted to predicates of arity 2 or less, it corresponds exactly with the graph methods used by the Web Ontology Language (OWL). This is the case because a predicate of arity 2 is an edge in a directed graph. When using graph representations, predicates of arity greater than 2 must be represented by creating additional, mediating concepts that map the arguments of the higher-arity predicate together.

For our example above, we might represent the domain with the following graphs.

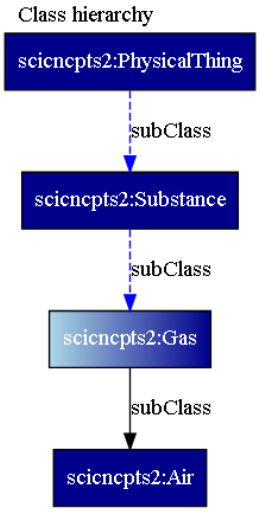

Figure 4: Class hierarchy of Air

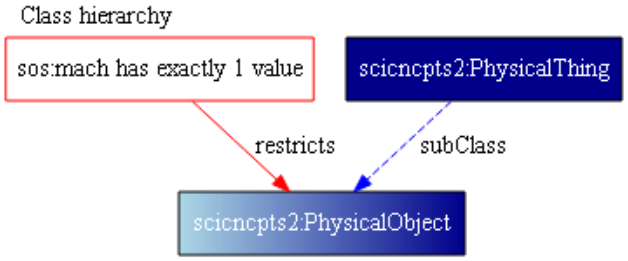

Figure 5: Class hierarchy of PhysicalObject

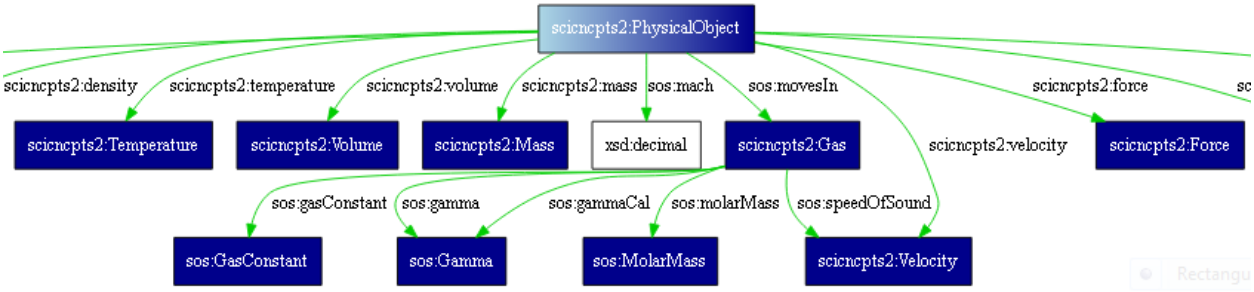

Figure 6: Portion of domain and range graph for PhysicalObject

This captures explicitly several important relationships.

1. *Air* is a type of *Gas*. Note that we model *Air* as a class, not an instance, because different instances of *Air* have different properties—*temperature*, *density*, etc.
2. *PhysicalObject* is constrained to have a single value of the property *mach*. (Note that if we were capturing a temporal model, this would be a single value an any given instant in time.)
3. *PhysicalObject* is in the domain of a number of important properties
    a. *velocity*
    b. *force* (important for thrust, flight models)
    c. *movesIn*, with range *Gas*
    d. *mach*, range decimal
    e. *mass* (important for thrust, flight models)
    f. *temperature* inherited from *PhysicalThing*; *Gas* also inherits
4. *Gas* is in the domain of several important properties as well
    a. *gasConstant*, which shows up in the equations above
    b. *gamma*, the calorically perfect gas ratio of specific heats

c. *gammaCal*, the calorically imperfect gas ratio of specific heats
d. *molarMass*, used to compute the *gasConstant* (not shown in above equations, for simplicity)
e. *speedOfSound*, an important property in the Mach equation above

The domain model captures explicitly the relationships between the various classes whose property values are inputs and outputs of the equations above. The question at hand is how one might best associate this domain information with the computational models represented by these equations.

A Solution: Augmented Types

Most programming languages have some concept of built-in types, e.g., integer, float, string, Boolean, etc. These types are used to specify the type of a variable, the signature of a method call, and the type of value or values returned. In an object-oriented language, classes may be defined that align with the classes in the domain model and these classes may then be used as types. However, there are important differences between the expressivity of most object-oriented languages and the expressivity of a graph-based ontology language such as OWL. Not least among these is that in most object-oriented languages, properties are represented as fields in a class and have no independent existence. By contrast, properties in graph ontology languages are first-class citizens and can have restrictions on value type, cardinality, etc., based on the class of the thing having the property. This means that more than one class can be in the domain of the same property, and that a property may have restrictions on cardinality, type of value, etc., which are different for different classes but is still identifiably the same property. So even in the case of object-oriented languages, the useful parts of the code may be methods whose types are the built-in or primitive types. Such is the case, for example, in the Java applets on the NASA Glenn Research Center Hypersonics Web site. The only non-built-in classes used, besides the applet itself, are user-interface classes.

When integrating computational models whose inputs and outputs are typed according to the built-in data types of the language into a semantic framework, both the built-in data type and the semantic domain type are important. In fact, these two types are not sufficient. It is also necessary to capture explicitly how the inputs and outputs of the computational model relate to each other in the domain model. Just as the graphs above illustrated these relationships, graph patterns can be associated with a computational model as a means of making these relationships explicit. For any single computational model, the extent of the graph pattern needed is the domain subgraph that connects all inputs and outputs together.

Consider some examples using the equations above. These examples will be illustrated in the Semantic Application Design Language (SADL), but the language could be different. The information content—(1) the primitive data types of the inputs and outputs, (2) the domain concept of which the input or output is a value, and (3) how the inputs and outputs are related in domain terms—is the essential point. The following is a declaration of the signatures for four equations described above, speed of sound, gamma of a calorically imperfect gas, temperature of the atmosphere as a function of altitude, and Mach number for a flying object, in terms of the language built-in types of inputs and returned values. Note that the keyword *External* is short for external equation, meaning that the body of the equation, the computational instructions, reside somewhere else, e.g., in a backend computational graph.

```
1. External CAL_SOS(double T, double G, double R, double Q) returns double.
```

```
2. External CAL_GAM(double T, double G, double Q) returns. double

3. External tempFromAltitude (double alt) returns double.

4. External computeMach(double alt, double R, double G, double Q,
                        double vel) returns double.
```

Now we will add the domain model graph patterns necessary to capture the explicit context of the equation, along with the units supported by the model.

```
5. External CAL_SOS(double T (temperature of a Gas {Kelvin, Rankine}),
                    double G (gamma of the Gas),
                    double R (gasConstant of the Gas {"g/mole", "lbm/lbmole"}),
                    double Q (a Theta {Kelvin, Rankine}),
                    string us (a UnitSystem {Metric, Imperial}))
                     returns double (speedOfSound of the Gas {"m/sec", "ft/sec"}).

6. External CAL_GAM(double T (temperature of a Gas {Kelvin, Rankine}),
                    double G (gamma of the Gas),
                    double Q (a Theta {Kelvin, Rankine}),
                    string us (a UnitSystem {Metric, Imperial}))
                    returns double (gammaCal of the Gas).

7. External tempFromAltitude (double alt (altitude of some Air {ft}))
                    returns double (temperature of the Air {Fahrenheit}).

8. External computeMach(double alt (altitude of a PhysicalObject and the
                                    PhysicalObject movesIn some Air {ft, m}),
                    double R (gasConstant of the Air {"lbm/lbmole", "g/mole"}),
                    double G (gamma of the Air),
                    double Q (a Theta {Kelvin, Rankine}),
                    double vel (velocity of the PhysicalObject {"mph", "km/hr"}),
                    string us (a UnitSystem {Metric, Imperial}))
                    returns double (mach of the PhysicalObject).
```

In Equation 5 the graph pattern need only relate properties whose values are being input and output to a particular instance of the class Gas. In this syntax, the use of “a Gas” in the first argument and “the Gas in the subsequent arguments and in the return type indicates explicitly that all refer to the same *Gas*. Thus, the use in this structured-English language is consistent with meaning in standard English usage. The same is true in Equation 6. In Equation 7, the computational model being described is actually specific to *Air*, not just any subclass of *Gas*. This is important to capture explicitly.

In Equation 8, which is illustrative of a more complex equation that includes properties of more than one concept, the inputs and outputs are related via a graph pattern going up to the node representing the instance of a *PhysicalObject*, identified as “a PhysicalObject in the first argument and “the PhysicalObject” in subsequent arguments and in the return value. Note that the first reference to *Air* is “some Air”. This is an allowed alternative to “an Air” as it is more natural-sounding for a substance. Note also that an equation might have made reference to more than one *PhysicalObject*, in which case the identity of each different object would be made explicitly clear, in the SADL language, by using “a

PhysicalObject", "the Physical Object", etc., for the first, and "a second PhysicalObject" or "another PhysicalObject", "the second PhysicalObject" or "the other PhysicalObject" for the second, etc. (This usage of indefinite and definite articles is covered by invention *Concept Level Rules (Crules)*, US Patent reference number 317709-US-1, and by the paper *Concept-level Rules for Capturing Domain Knowledge*, A. Moitra, A. Crapo, R. Palla.. 12th IEEE International Conference on Semantic Computing, Jan 31-Feb 2, 2018. https://ieeexplore.ieee.org/abstract/document/8334469/

The other important information to capture is the constraints on units of inputs and outputs. Equation 7 shows the case of a computational model that only works for a single set of units: "ft" as the unit of input and "[degrees] Rankine" as the unit of output. The other equations are modified slightly from those shown in Equations 1, 2, and 4, to take an additional argument showing whether the unit system is Metric or Imperial. For example, in Equation 5, if the *UnitSystem* is *Metric*, the unit of the first argument must be *Kelvin*, the unit of the 3rd argument must be "g/mole", the unit of the 4th argument *Kelvin*, and the unit of the returned value is "m/sec". Similarly, if the *UnitSystem* is Imperial, the units are respectively *Rankine*, "lbm/lbmole", *Rankine*, and "ft/sec". Note that the 2nd argument is unitless. The presence of the unit system argument implies that the encoding of the equation has computations for both systems of units with appropriate flow control within.

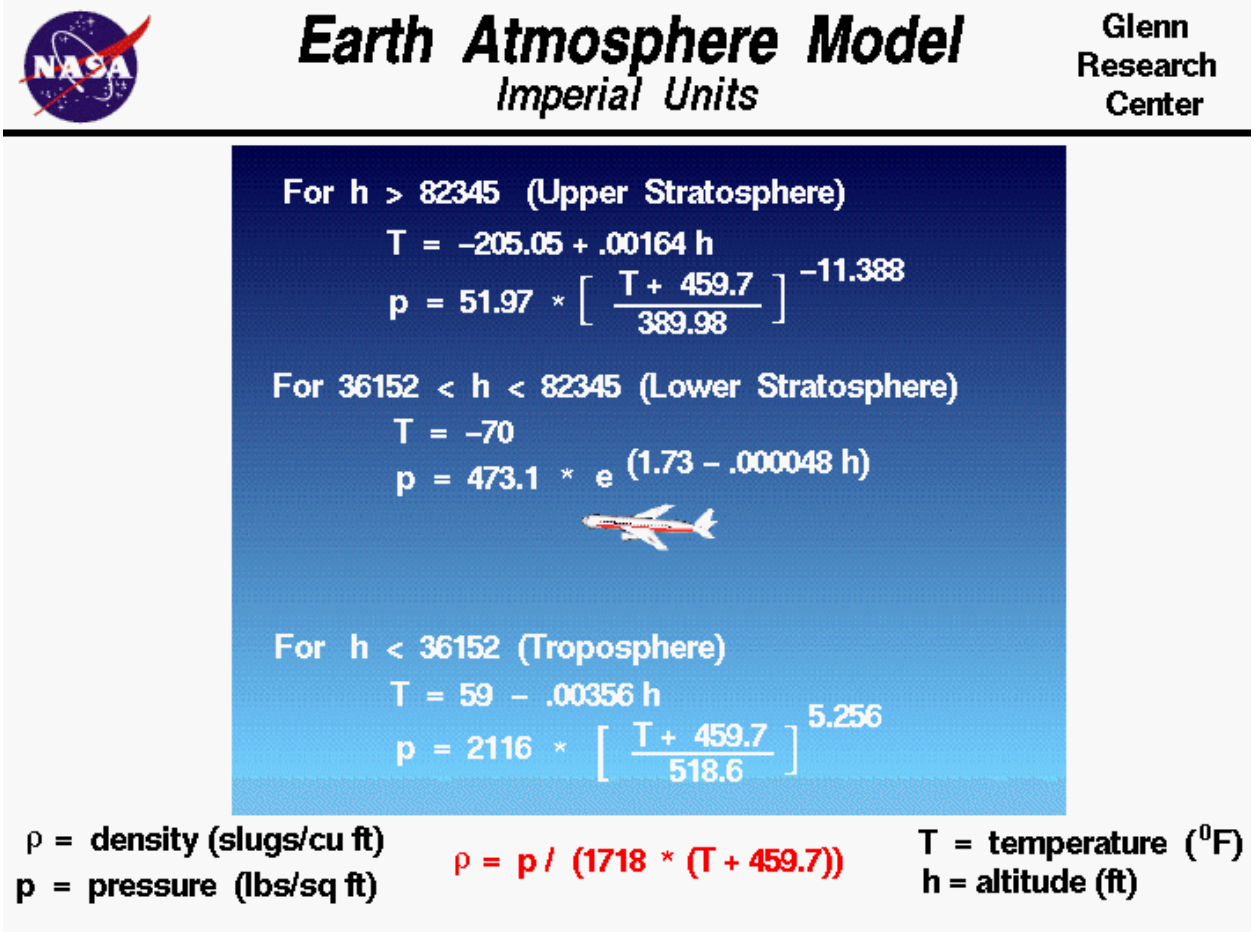

Figure 7: Earth Atmosphere Model from NASA Web Site (https://www.grc.nasa.gov/www/k-12/airplane/atmos.html)

Returning to a previous example, consider the equations for the static temperature of air as a function of altitude. The equations for temperature are shown in Figure 7. Expressed in SADL syntax they are:

```
9. Equation troposphereTemperature(decimal alt
          (^value of altitude of some Air and alt <= 36152 {ft}))
        returns decimal (^value of temperature of the Air {F}):
          return 59 - .00356 * alt.

10. 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.

11. Equation upperStratosphereTemperature(decimal alt
          (^value of altitude of some Air and alt > 82345 {ft}))
        returns decimal (^value of temperature of the Air {F}) :
          return -205.05+.00164 * alt.
```

These equations illustrate functional constraints, namely the range of altitude values for which each equation is valid, as well as the relationships of the input and output. Note the use of property chains. The class UnittedQuantity is in the domain of the property “value” (escaped with a “^” in the model because it is a keyword in the grammar) and in the domain of the property “unit”. As both altitude and temperature have ranges which are of type UnittedQuantity, the property chains “value of altitude of some Air” and “value of temperature of the Air” tie the input and output together through the air at that altitude and temperature.

Extending the Solution to Tabular Data

In order to apply the same approach to tabular data, data with columns with headers providing information about the data in that column, and rows containing the actual data, we create a common ontology for a *DataDescriptor* and use this common definition for both equations and data tables. Here is a snippet of the SadlImplicitModel.sadl file used by all ontologies created in SADL.

```
DataDescriptor is a class,
   described by descriptorName 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.

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.

^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.

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.
```

Given these definitions, a tabular data set in the hypersonics domain might have header information like this.

```
Data1 is a table
  [double alt (alias "Alt") (altitude of a PhysicalObject {"ft"}),
   double u0 (velocity of the PhysicalObject and the PhysicalObject movesIn some Air {"mph"}),
   double tt (staticTemperature of the Air {"R"})]
  with data located at "http://datasource/statictemperatureobservations/data1".
```

By capturing explicitly the semantic context of each equation argument or returned value, and by capturing explicitly the semantic context of each data table element, an artificial intelligence can be expected to appropriately use equations and data in scientific endeavors.

Representing Augmented Types in OWL

It is desirable for the constraints captured in the augmented type information shown to be translated into OWL so that there is a standards-based representation and so that we can more easily query and infer over the model to find equations matching given semantic constraints. The representation of triple patterns in OWL or RDF is not a new concept. Rules and queries both have triple patterns as essential parts with variables used to connect triple patterns together to form complex graph patterns. The Semantic Web Rule Language (SWRL) represents rules in OWL. [1] The SPIN language supported triple patterns as well as functions. [2] The newer Shapes Constraint Language (SHACL), which largely replaces SPIN, also has some capability to capture triple patterns. [3]

The variables that connect triple patterns and function patterns together are a necessary part of the capture of semantic constraints. Sometimes these variables will reference the variable which is an argument of the equation or the column title in a table. Sometimes they will be created from class references with indefinite articles, e.g. “a PhysicalObject”. In some cases they will be created by expanding nested expressions, as is the case when a property of subject is nested inside an equation call as an argument. In any case, the OWL representation must take care not to create an Individual in the OWL graph with the name of the variable as the localname. The reason is because of scoping. Scoping in the Xtext implementation of the SADL grammar recognizes that “alt” in Equation 9 is not the same “alt” as in Equation 10 and not the same “alt” as in Equation 11. The variable for any equation is scoped only within that equation’s arguments, return values, and constraints. Another equation in the same namespace might use the same argument name but it might have a different type and different semantic constraints.

OWL offers no such equation-level scoping unless each equation were in a separate namespace. Therefore, we must create a variable for “alt” in each equation which is different from the variable for “alt” in any other equation in the namespace. While some triple pattern representations in OWL use blank nodes for these equation-scoped variables, with a property capturing their name, we take the approach of creating unique variable names for each variable reference within the namespace. This facilitates the use of the variable in multiple triples and/or function patterns. Regardless of whether the variable has a user-defined name, e.g., is an argument to the equation signature, or has a name generated by the translation, the variable is given that name as the value of the property “localDescriptorName”.

While it would be possible to create sequential unique variable names in a namespace with a counter as a way to obtain uniqueness, this has the disadvantage that the content of the OWL semantic constraints would depend upon other equations in the namespace and upon their order, which means that test cases would be affected by any change in the input SADL. Another approach, which eliminates this problem, is to prepend the equation name to the variable name and start the variable index counter anew for each equation. This eliminates the dependency of the output on the number and order of equations in the input SADL file.

The model for semantic constraints, in SADL syntax, is as follows.

```
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 specifiedUnits (note "the array of possible units")
    with a single value of type string List,
  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.

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 anyURI List.

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

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.

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.
```

Example Constraints in OWL

To illustrate, consider Equation 9, troposphereTemperature. We need to capture the constraints as graph patterns for the input "alt" and for the return value. In order to do type checking, we also need to capture the types of the variables created in the Owl model.

For the first argument "alt", the constraint should contain:

1. a triple pattern with variable troposphereTemperature _v0 as subject, rdf:type as predicate, and class Air as object to represent the "some Air"
2. a triple pattern with variable troposphereTemperature _v1 as subject, rdf:type as predicate, class UnittedQuantity as object
3. triple pattern with variable troposphereTemperature _v0 as subject, "altitude" as predicate, and variable troposphereTemperature _v1 as object to represent "altitude of some Air"
4. a triple pattern with variable troposphereTemperature _v2 as subject, rdf:type as predicate, xsd:decimal as object
5. a triple pattern with variable troposphereTemperature _v2 as subject, localDescriptorName as predicate, xsd:string "alt" as object (troposphereTemperature _v2 represents the argument "alt", but we can't actually give it that name as other equations in the scope can have that name and might have different constraints)
6. a triple pattern with variable troposphereTemperature _v1 as subject, "value" as predicate, and a variable troposphereTemperature _v2 as object
7. a function pattern with built-in "le" (less than or equal), with first argument troposphereTemperature _v2, with second argument the literal 36152

Although the returned value is not named in the equation, we create a named variable (troposphereTemperature _v4) in the OWL to represent the returned value so that we can use it in multiple triple and/or functional patterns. Then we have these constraints:

1. a triple pattern with variable troposphereTemperature _v3 as subject, rdf:type as predicate, class UnittedQuantity as object
2. a triple pattern with variable troposphereTemperature _v0 as subject, "temperature" as predicate, and variable troposphereTemperature _v3 as object
3. a triple pattern with variable troposphereTemperature _v4 as subject, rdf:type as predicate, xsd:decimal as object
4. a triple pattern with troposphereTemperature_ v3 as subject, "value" as predicate, and troposphereTemperature _v4 as object

References

[1] H. e. al, "SWRL: A Semantic Web Rule Language Combining OWL and RuleML," 21 May 2004. [Online]. Available: https://www.w3.org/Submission/SWRL/. [Accessed 18 July 2019].

[2] H. Knublauch, "SPIN - SPARQL Syntax," 12 September 2013. [Online]. Available: https://spinrdf.org/sp.html. [Accessed 18 July 2019].

[3] H. Knublauch and D. Kontokostas, "Shapes Constraint Language (SHACL," 20 July 2017. [Online]. Available: https://www.w3.org/TR/shacl/. [Accessed 18 July 2019].