ASKE TA2 Project KApEESH M9 Report

Alfredo Gabaldon, Natarajan C. Kumar, Andrew Crapo, Vijay Kumar.
GE Research

This work is supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00111990007.

In Phase 2 we have started working toward lifting the simplifying assumptions made for the initial proof of concept. We have in particular been working on incorporating model assumptions and representing and reasoning about units. We also started working on an approach for checking model validity through the assumptions and leveraging reasoning about the assumptions to present explanations when a model turns out to be invalid and using the assumptions to hypothesize alternative models and launch targeted knowledge extraction. We are also working on the mixed initiative dialog interface which needs to incorporate units and augmented types, but also to let the user change the dialog context by modifying previous statements to reissue the query in a different set up. Finally, we report our work on extending the DBN framework.

Incorporating model context and units

Although the initial prototype form Phase 1 uses knowledge about the semantic types of the inputs and outputs of models represented in the knowledge graph, it does not leverage any other context and does not handle unit information. We have been working on remediating this limitation by means of SADL's "Augmented Types"[1]---an extension that lets one describe the arguments of a function (equation) and the output in a way that makes explicit the objects that are the sources of the input/output quantities. In other words, it lets one talk about the inputs and outputs explicitly as properties of the underlying objects. This is crucial for a machine to understand how quantities are related in a model, and hence to understand what the model actually means. If a knowledge graph truly captures the domain knowledge around a model, then the input and output variables must be connected via concepts and properties in the graph.

In addition to making underlying objects explicit, we are also able to capture and make explicit assumptions in the model, at least those assumptions relative to the input quantities. For example, a model that has atmospheric altitude as an input can now not only be described as having an input, say denoted by *alt*, linked to the concept *Altitude* in the knowledge graph (meaning that the input is of that type) but one may also capture that the particular model assumes that *alt < 361256 ft.* That is, we can explicitly capture that the model, as extracted, works under the assumption that the altitude input has a value under 36125 ft and should not be used with altitudes outside that range nor composed with models that make inconsistent assumptions.

[1] Augmented types

Along with the variables, semantic types of the variables, and the objects whose properties they describe, extended SADL lets us capture units information. The equation for static temperature in the troposphere can now be expressed as:

```
Equation st_temp_eq_tropo(double alt (altitude of some Air
                                                    and alt < 36152 {ft}))
        returns double (staticTemperature of the Air {F}) :
        return 59 - 0.00356 * alt .
```

stating that the equation takes altitude (denoted by a "local variable" *alt* which in knowledge graph terms is an instance of the *Altitude* concept) in feet (ft) as input and outputs the static temperature in Farenheit (F). The hardcoded constants 59 and 0.00356 have been empirically derived by atmospheric scientists and the equation expression reflects how we find it on the Hypersonics NASA page. Instead of hardcoding them in the return expression, we could alternatively define them in separate equations and use them as arguments in this equation. This is how we captured another constant, the heat capacity ratio (denoted by gamma) and use it in the speed of sound equation as input. We first define gamma by the equation:

```
Equation gamma_const() returns double (gamma of some Air): return 1.4.
```

Then define speed of sound by an equation that takes gamma, the gas constant R, and the static temperature as arguments:

```
Equation sos_eq(double ts0 (staticTemperature of some Air {F,Rankine}),
                            double g (gamma of the Air),
                            double R (rgas of the Air))
        returns double (speedOfSound of the Air {"ft/sec"}):
        return sqrt(g * R * ts0) .
```

At the knowledge level, the system will check if the units of the output of one model need to be converted before being input into another model. Conceptually, we can handle unit conversion in the DBN by adding unit conversion nodes, that is, nodes that take one input of a given type and units, say temperature in Rankine, and have output of the same type in another unit, say temperature in Kelvin. Although this works in principle, adding conversion nodes to the DBN is probably computationally expensive, so we currently plan to rely on unit conversion libraries such as Pint[2] and have the DBN execution framework invoke conversion when necessary based on unit information passed on from the knowledge layer down to the execution layer.

These changes to incorporate assumptions and units in equations require modifying many of the backend knowledge services we developed in Phase 1, as we expected. In particular we are working on modifying the knowledge graph procedures that build model compositions and the micro-services that create DBN specifications from those models.

---

[2] https://pint.readthedocs.io/en/latest/index.html

Checking assumptions in model composition

A composite model is essentially a set of equations that together can be used to compute a desired output from the given input variables. Models are composed at the knowledge level and the system needs to supplying the necessary information to the lower layers to ensure units are used consistently. At the knowledge level we can also perform some checks on the assumptions made by the equations in the model. Two types of test can be performed at this level:

a) A query specific test checks that the given context and input values in the query do not violate the assumptions made by the candidate model. If they do, the model cannot be used to answer the query, even though the model may be valid. A simple example is a query that has altitude of 35,000ft as a given and a model that assumes altitude values of less than 36152ft.
b) The second type of test is query independent and checks if the collection of assumptions in the model is consistent. The set of assumptions may use intermediate variables whose values are unknown at model composition time, so we treat the assumptions as a set of constraints and test if they are self-consistent. If they are not, the model is invalid. For example, a valid model cannot contain one equation that assumes altitude is less that 36152 and another that assumes altitude greater than 36152.

These checks do not guarantee that model execution will be successful. For example, downstream computation of intermediate variable values may turn out to invalidate an assumption. However, catching invalid assumptions at composition time at the knowledge layer level should save time and computational resources. It should also simplify the task of identifying and explaining to the user the reason why a model candidate doesn't work.

Both (a) and (b) checks are useful to eliminate unsuitable candidate models. But they are also useful for explanation and hypothesis generation. We go over some details on this in the next sections.

Post-execution checking for explanation

Query specific input checks (type (a)) are easy to perform as we only need to check the query given values against any assumptions on the input variables. Values that satisfy input variable assumptions can violate assumptions of intermediate variables downstream. However, the values of intermediate variables are not available until the model is executed, so checking those assumptions is not possible at composition time. However, it is useful to check those assumptions **post-execution** in order to generate explanations. After a successful run, the BDN execution framework returns computed values for all nodes in the network, intermediate and output nodes. When the run fails, it is often because a variable goes out of range. We plan to modify the DBN framework to return additional results in case of failure in order to better explain to the user what the problem was and help modify or create an alternative model. In particular, we would like to modify the DBN to return the values of any successful intermediate nodes even if computing the final desired output failed. Whether the run is successful or

unsuccessful, the intermediate values are ingested into the knowledge graph together with query and composite model used metadata, and other execution results (see M3 report for details). We can then query the knowledge graph to retrieve intermediate values and the model assumptions, check if any assumptions failed, and include that information in an explanation of the results.

Consistency checking

For assumption consistency checking in the knowledge layer (type (b) check), we are working on implementing a new assumption reasoning module based on SWI-Prolog[3] and Constraint Handling Rules (CHR).[4] Initially we are targeting an assumption language that consists of Boolean conditions (e.g., "Object is moving through Air") and less-than-or-equal (leq) constraints (e.g., altitude < 36152) and combinations of these two types of assumption. These are the types of constraints that we have encountered in the hypersonics domain so far. CHR, however, is a general constraint language that will let us accommodate more types of assumptions later on or use different constraints and solvers in other domains. Solvers for leq, Boolean, linear equations, interval constraints, and others are publicly available.[5] We will reuse those wherever possible and develop new ones as necessary.
One important feature of SWI-Prolog and the CHR library is that they can be used in combination with the SWI-Prolog's Semantic Web library. This means we can combine knowledge graph reasoning with CHR constraint reasoning in one language. This will also be advantageous for the task of generating explanations when assumptions are violated or inconsistent.

Another advantage of using CHR is that we are free to define what happens when a set of assumptions is found to be inconsistent. We will exploit this flexibility to present the user with the reasons why a candidate model does not apply or is invalid by showing the assumptions that clashed and pointing out the model components that led to the clashing assumptions. Since a clash can have a long derivation behind it, we need to keep track of the assumptions and equations involved in each inference step that led to the clash. In case of a large number of steps, we will present only the equations and original assumptions involved to avoid overwhelming the user.

Using assumptions to generate hypothesis and trigger knowledge extraction

Model assumptions can be used to hypothesize new models or to launch extraction for specific knowledge that may be missing. Consider a set of input and output variables (could be from a query specified by the user but ignoring any given values) and suppose that the system finds valid (in the sense of passing check (b)) models M_1, M_2, …, M_k with consistent assumptions A_1, A_2, …, A_k, respectively. Each set of assumptions A_i can be seen as the conjunction of conditions assumed by each equation in model M_i. Moreover, we can look at the disjunction of the

[3] https://www.swi-prolog.org/
[4] https://dtai.cs.kuleuven.be/CHR/
[5] https://dtai.cs.kuleuven.be/CHR/applications.shtml

assumption sets, A_1 or A_2 or … or A_k, as a logical theory representing the possibilities covered by the available models. If all possibilities are covered (similar to a complete case-statement in a programming language), the negation of the logical theory, ¬(A_1 or A_2 or … or A_k), will be unsatisfiable. But if it is not, we can use it to infer cases that are not covered by the models. Let us illustrate how this work with an example.

Consider three models M_1, M_2, and M_3 that respectively make the assumptions:

A_1: a < 10,
A_2: a >= 10 & a < 20,
A_3: a >= 30.

To check if there are situations that fall outside the three models, we consider the negation of the disjunction of the assumptions, ¬(A_1 or A_2 or A_3): a >= 10 & (a < 10 or a >= 20) & a < 30. Simplifying it, we have a new set of assumptions: A_4: a >= 20 & a < 30. The new set A_4 describes a situation that none of the models covers. This suggests the possible existence of another model that works under these conditions. The system can proceed in several ways: it can try to modify M_2 by replacing equations that make the a<20 assumption. Or it could test whether M_3 works for values in that range. The system can also engage the knowledge extraction services to look for equations that apply under the A_4 assumptions.

Note that this "case analysis" not only works on sets of inequalities. It works for any Boolean combination of constraints as long as it is possible to determine complementary conditions. For example, the condition may be in terms of a finite domain variable, say *medium*=[air,CO2], where the domain of *medium* is the set {air, CO2, water}, in which case the complementary assumption would be *medium*=[water].

This logical analysis could alternatively suggest that there are unstated assumptions in the models. Hence human user input is important in this process as well to decide if an assumption should be added to the current knowledge base or if the system should look for new models or extract new knowledge.

Dialog interface

Incorporating additional context around queries, assumptions in equations and units, imposes additional functionality requirements on the mixed-initiative dialog interface. In particular, the dialog interface needs to accommodate all the extensions to the SADL semantic language with augmented types so that the user can use the new expressivity in queries and any other interaction through the interface with TA1 and TA2 functionality. More details on the dialog interface requirements and potential approaches to meet those requirements are summarized in a separate document.[6]

[6] A mixed-initiative dialog framework.

Extensions to the DBN execution framework

In this section we describe work to implement sensitivity analysis in the DBN execution framework, with the intention of using it as a tool to aid the user understand how the inputs and outputs of a model are related, and to answer some types of "why" and "how" queries.

Sensitivity analysis using Dynamic Bayesian Networks

Sensitivity analysis is an important aspect of engineering problems, where a multitude of inputs influence the output through the formulation of a model that relates the inputs and outputs. The information about the relative sensitivity of the different inputs to the outputs can help us answer a number of further queries, such as: is this parameter important? In what regions of the design space is that particular input important? which parameter is affecting the output almost as a noise? etc.

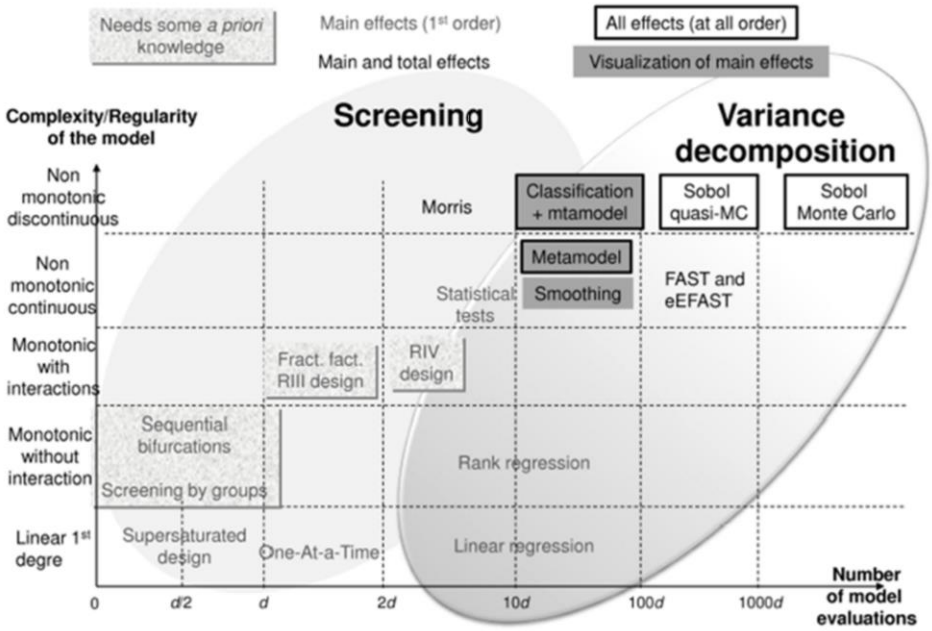


Figure 1 Schematic of the different methods of sensitivity analysis in the space of complexity and model evaluations (from *[1]*)

There are a number of methods available in the literature for performing sensitivity analysis [1]. These include screening methods, such as simple scatter plotting and regression analysis, local sensitivity methods, such as local derivatives and one parameter at a time trending, and global methods, such as analysis of variance (ANOVA) with polynomial regression and variance-based methods. Figure 1 shows a schematic of the different methods based on the nonlinearity and dimensionality of the problem being solved. We will not get into the details of the different types of methods, we encourage the reader to refer to [1] for more detailed insight into sensitivity analysis methods. In our current work we focus on variance based global sensitivity analysis (VGSA).

Variance decomposition

When we approximate a system using models (either directly through equations as we acquire the scientific knowledge in the ASKE program or data-driven models fitted directly to data),

VGSA helps us understand how the uncertainty in the overall system response (i.e., output) can be divided among the different uncertain input variables and their mutual interactions. Figure 2 shows the decomposition of output variance schematically to be composed of the variance in the output due to the different input factors.

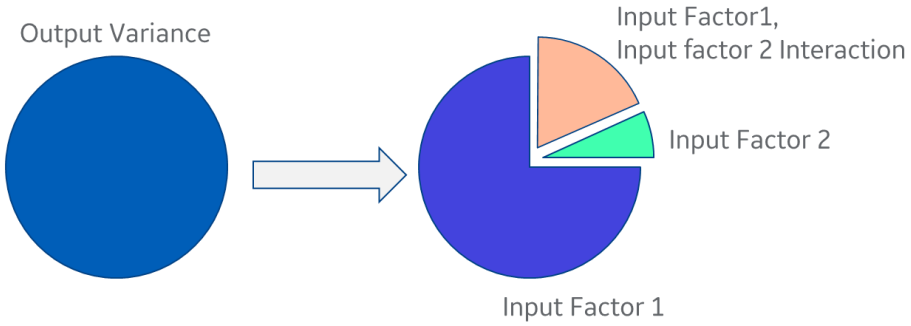


Figure 2 Decomposition of output variance

Sobol [2] devised certain indices (referred to as Sobol indices) that can help us rank the effect of the different inputs on the variance of the output. Sobol indices are defined as:

$$S_i = \frac{Var[E(Y|X_i)]}{Var(Y)}$$

where Y is the output response and X_i, i = 1, 2, 3…, n_p, are the inputs, where n_p is the number of dimensions of the input space. Var[.] and E[.] are the variance and expectation operations, respectively. This formulation of Sobol indices can be recursively expanded for two-way and higher order interaction indices as follows:

$$S_{ij\dots n_p} = \frac{Var\left[E\left(Y\middle|X_i, X_j, \dots, X_{n_p}\right)\right]}{Var(Y)}$$

We can also compute the total effect of a particular input X_i by computing the sum of all the partial sensitivities involving the input X_i. For example, we can write the total sensitivity index as:

$$S_{T_i} = S_1 + S_{12} + S_{13} + \cdots + S_{123} + S_{124} + \cdots + S_{123\dots n_p}$$

The Sobol indices are good descriptors of the global sensitivity of the model to its input parameters since they do not assume any linearity or monotonicity of the model. The expressions for Sobol indices can be estimated directly with a model that is relatively inexpensive to evaluate, but as the model becomes more complex with high dimensions, it becomes very expensive to evaluate. In that case, a simplified model (such as a surrogate model) and sampling-based techniques can help evaluate the indices very quickly.
Dynamic Bayesian Networks (DBN), which is our execution framework for KApEESH, utilizes a particle filter-based sampling methodology to evaluate the models (with the inherent uncertainties of the input variables). So it is natural to use the samples directly to compute

Sobol sensitivity indices to get an insight into the effect of the different inputs on the quantities of interest. Since the overall problem can include several intermediate responses, we can easily analyze the sensitivities of those quantities as well since they are also sampled during a DBN execution run.

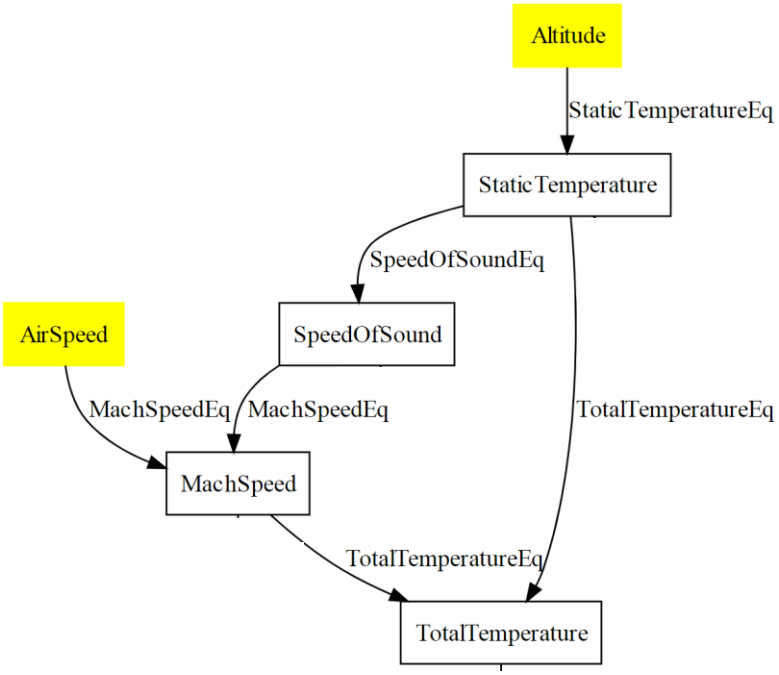
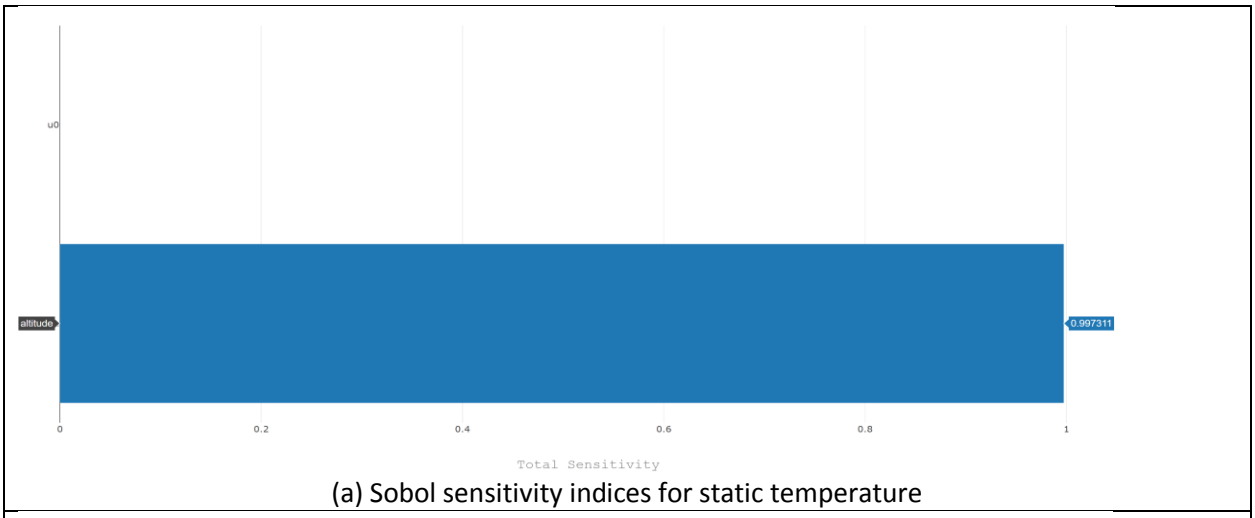
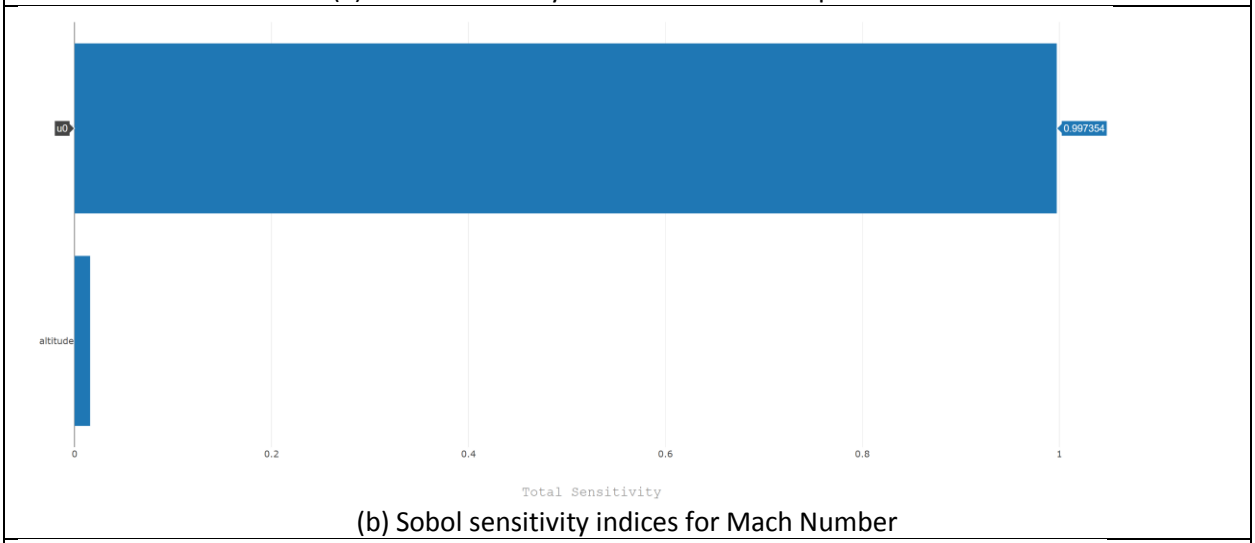


Figure 3 Example network of the total temperature model with altitude and airspeed as inputs.

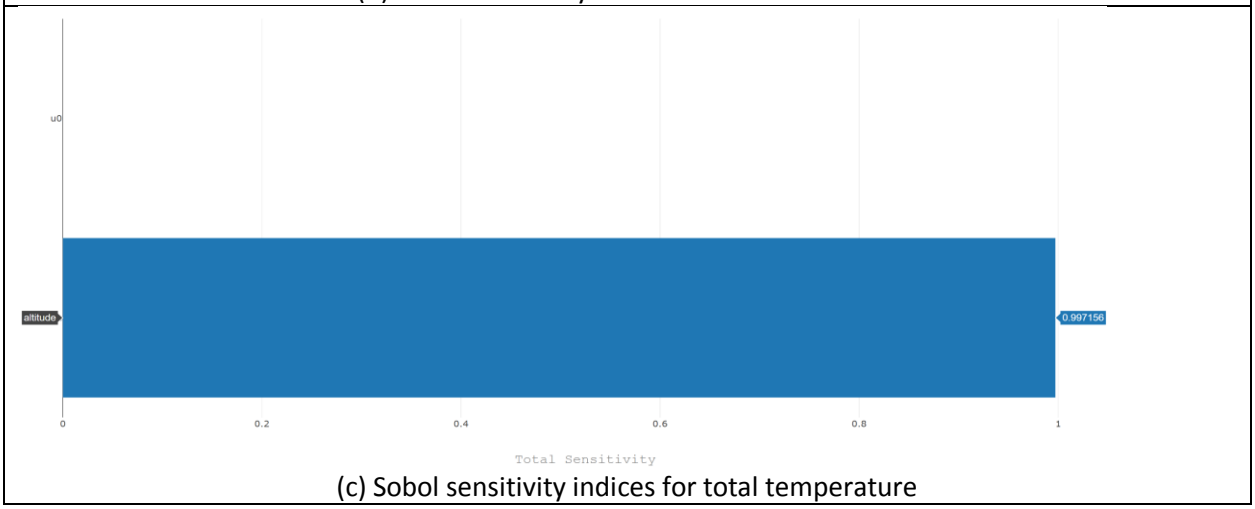
We demonstrate the Sobol sensitivity indices computed for the simplified freestream flow models from the overall hypersonic aerodynamics scientific domain of applicability for KApEESH as shown in Figure 3. The network shows two main inputs: altitude and free stream velocity (u), which are connected through various models to generate static temperature, speed of sound, Mach number and total temperature. We computed the Sobol indices for these quantities with altitude and velocity as inputs. As can be clearly seen in Figure 4, altitude influences the value of static temperature and speed of sound quantities almost to a 100% level, since they are computed from simple equations that do not have velocity as an input at all. Conversely, the Mach number shows a small effect of altitude and remaining variance due to velocity in Figure 4.



(a) Sobol sensitivity indices for static temperature



(b) Sobol sensitivity indices for Mach Number



(c) Sobol sensitivity indices for total temperature

Figure 4 Sobol sensitivity indices plots for static temperature, Mach number and total temperature.

Sensitivity analyses can be utilized to possibly answer some “Why” and “How” types of queries. For example, if the user wants to know how much of the change in the output is due to a specific input, we can present the user with the Sobol index (either main or total effect).

Queries such as Why is a specific input needed? or Why do we need all these inputs? can also be easily addressed using the Sobol indices of the sensitivity analysis.

Returning partial results

When a model execution fails, the DBN framework currently does the standard thing and returns a simple message stating some reason or error. But for the purpose of explaining results to the user, it is much more useful to have the DBN return whatever partial results it was able to compute. For instance, using the Hypersonics examples, if the model fails to compute a final output for Total Temperature because values went out of range or some other reason, but the DBN was able to compute values for some of the intermediate nodes, say for Speed of Sound, Mach, etc, the DBN can return those partial results back. We can then ingest them into the knowledge graph just as when the model executions are successful, and use the results to generate better explanations for the user of what went wrong. Simply pinpointing where in the model things went awry is useful and can help the user make changes to the model. But further, we can use the partial results to check if assumptions were violated. For example, if the equation for Total Temperature used in the model assumes that Mach<5 (which as mentioned earlier we cannot check at composition time) then after the execution attempt fails we can use the partial results and show the user that the DBN computed a value for Mach of 5.5, which violates the Total Temperature equation assumption. The user can then intervene by modifying the assumption, adding a new equation, and so on. Alternatively, if the knowledge graph was missing the assumption that the equation only works when Mach<5, showing the partial results could help the user realize that the equation is for Mach<5 cases and the assumption needs to be added, that an equation for higher Mach values needs to be added or searched for and extracted.

Returning and capturing partial results is also useful to avoid re-computation. In the previous example, after the execution fails the user may just modify the model by replacing an equation and relaunch the execution. The DBN already features an execution restart capability that can use the partial results from a previous execution and restart from there.

Unit handling in the DBN framework

In order to incorporate units into the overall framework, we are considering two options. The knowledge graph will have unit information for the different variables that form the composite models. This unit information will be included in the DBN json specification of the model that is used to invoke model execution via the DBN micro-service. The computation within the DBN framework can deal with units by:

a. Uniformly inserting a unit conversion node throughout the network for all the quantities that have a unit associated with them. As mentioned previously, we can utilize a library like Pint and perform general purpose unit conversion. If the unit conversion is not needed, then the unit conversion node does not do any transformation. This has the potential drawback that computational performance may suffer due to applying unit conversion at all the nodes in case of large networks. Utilizing lightweight modules such as Pint will help with scalability.

b. An alternative idea is to be more intrusive in the execution framework and add a pre-processing step where we go through the network specification as generated by the knowledge graph, identify the nodes that require unit conversion, and then insert a conversion node only for those nodes that require it. In case of large networks, the pre-processing overhead may be offset by saving unnecessary calls to the unit conversion library. This also brings up the point regarding large models that require large networks to evaluate – it is possible to divide the network into smaller sub-networks and combine the results together.

We are currently working on implementing the unit conversion capability in our platform from both the knowledge graph representation and execution framework aspects.

Conclusions

During this last two months we have made concrete progress toward incorporating model assumptions and handling unit information and conversion. We are exploring using CHRs for reasoning about assumptions to detect is a candidate model is valid and can be used to answer a particular query. We are also exploring using assumption reasoning for hypothesis generation. However, we are still in an exploratory phase on the use of CHRs. An alternative approach, which the ANSWER team is exploring, is the use of the SHACL language, a Semantic Web language for expressing constraints.

Beyond the scope of ASKE, GE has committed resources for an internal project to develop GE relevant use cases where ASKE technology can be applied. We hope this will help support our ASKE projects and the overall program.

References

[1] B. Ioos and P. Lemaitre, "A review on global sensitivity analysis methods," in *Uncertainty Management in Simulation-Optimization of Complex Systems: Algorithms and Applications*, Springer, 2015.

[2] I. Sobol, "Global sensitivity indices for nonlinear mathematical models and their Monte Carlo estimates," *Mathematics and computers in simulation,* vol. 55, no. 1-3, pp. 271-280, 2001.